更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

HZ BOOKS
华章科技

Git领域的集大成之作，在广度、深度和实战性上均史无前例

国内顶级Git专家亲自撰写，Git官方维护者等数位专家联袂推荐

實戰



蒋鑫 著

Got Git: The Definitive Guide of Git

# Git 权威指南

机械工业出版社
China Machine Press

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

2009年9月，我出版了一本针对日本读者的Git专著，当Linus收到我赠送的签名本时，他对我说："除了截图和命令行示例外，其他我什么也看不懂"（Linus不懂日文）。因为同样的原因，虽然我不能了解蒋鑫这本书的全部内容，但是我可以看出这本书涵盖了非常广泛的主题，并且可以看出蒋鑫对这本书的用心。我非常高兴能够看到这本书的出版，感谢向世界传播Git。

**——Junio C Hamano**
Git维护者（2005年7月至今）

仔细拜读了本书前三篇共20章的内容，感觉这本书极好。作者在软件版本控制系统方面有超过10年的经验，对版本控制系统有非常深入的认识。尤为难得的是，本书文笔很流畅，虽然是技术书籍，但是作者娓娓道来，阅读体验很好。Git的学习门槛较高，包括我们公司在内的很多企业都将版本控制系统转向了Git，强烈推荐大家看一看。

**——范凯（Robbin）**
CSDN平台开发总监 / ITeye（www.iteye.com）创始人

这是我读过的最好的关于Git的书。将复杂的Git解释得清晰而透彻绝非易事，蒋鑫做到了，更让人惊喜的是，他还分享了大量的经验总结。我几年来累积下来的诸多疑惑都在读罢该书后一一得以解开。如果你正在使用，或者打算使用Git，本书当然是必备的。你也可以抱着Subversion或CVS不放，不过，如果哪一天有人拿起这本书敲你的头时可别怪我没提醒过你。

**——许晓斌（Juven Xu）**
资深Maven专家 / 著有畅销书《Maven实战》

Git是当前开源社区最流行的版本控制系统，代表了版本控制的未来。每一位有志于从事软件开发的学习者都应该学习和掌握这一工具，它不但可用于追踪编程者的思考过程，还是打开开源软件世界的金钥匙，更可以通过版本控制掌握与他人协同工作的技能。本书是一本耐读的原创著作，因为其涵盖的内容之广足以让每一位程序员在成长的不同阶段都可以从中汲取丰富营养。

**—— 冯钤**
清华大学计算机科学与技术系 长江学者特聘教授

目前市面上关于Git的书不在少数，但迄今为止，国内外还没有一本书能在广度和深度上达到本书的水平，也许在今后相当长的一段时间内也不会有。如果你是一位软件开发者，强烈建议你学习并使用Git；如果你要学习Git，本书无疑是你最佳的选择。

**——徐继哲**
哲思社区（http://www.zeuux.org/）创始人

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

封面设计：陈子平

上架指导：计算机 程序设计
ISBN 978-7-111-34967-9
9 787111 349679

定价：89.00元

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

實戰




# Git权威指南

蒋鑫 著

机械工业出版社
China Machine Press

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

本书是Git领域的集大成之作，是一本关于Git的百科全书，在广度、深度和实战性上让同类作品望尘莫及。作者是国内顶尖的版本控制专家和咨询顾问之一，本书得到了Git官方维护者Junio C Hamano和ITeye创始人范凯（Robbin）先生等数位专家的高度认可和极力推荐，权威性毋庸置疑。

全书一共9篇，共41章和4个附录，内容几乎涵盖了Git的所有方面。第1篇介绍了版本控制工具的演变历史、Git的各种优点，以及它在3种主流操作系统中的安装与配置。第2篇和第3篇既是本书的基础，又是本书的核心，不仅介绍了Git的操作和使用，而且还讲解了Git的原理。第2篇详细讲解了个人用户如何使用Git，包括Git初始化、日常操作、暂存区、对象、重置、检出、恢复进度、历史变更、克隆、库管理等；第3篇详细讲解了Git协议和团队如何使用Git，包括Git支持的协议、冲突解决、里程碑、分支、远程版本库和补丁文件交互等。第4篇全面介绍了Git的协同模型，即它在实际工作中的使用模式，包括各种经典的Git协同模型、Topgit协同模型、子模组协同模型、子树合并、Android多版本库协同、Git与SVN协同模型等。第5篇介绍了Git服务器的架设，首先讲解了HTTP协议、Git协议、SSH协议的使用，然后讲解了Gitolite、Gitosis、Gerrit等服务器的架设方法，最后还讲解了Git版本库的托管。第6篇介绍了版本库的迁移，包括如何从CVS、SVN、Hg等版本库迁移到Git，以及Git版本库整理等方面的内容。第7篇讲解了Git的其他应用，包括etckeeper、Gistore等的安装、配置和使用，以及补丁中的二进制文件和云存储等内容。第8篇介绍了Git的跨平台操作，以及它的钩子和模板、稀疏检出和浅克隆、嫁接和替换等重要特性。第9篇是附录，详细给出了Git的命令索引，以及CVS、SVN和Hg与Git的比较与命令对照，方便读者查阅。

**封底无防伪标均为盗版**
**版权所有，侵权必究**
**本书法律顾问　北京市展达律师事务所**

**图书在版编目（CIP）数据**

Git权威指南 / 蒋鑫著. —北京：机械工业出版社，2011.6

ISBN 978-7-111-34967-9

I. G…　II. 蒋…　III. 软件工具—程序设计　IV. TP311.56

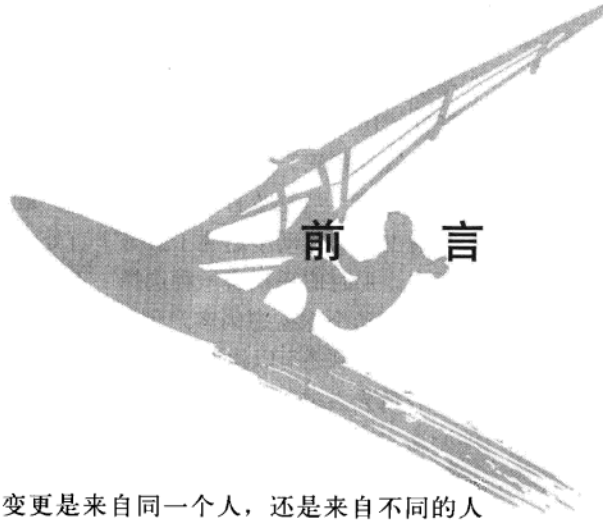
中国版本图书馆CIP数据核字（2011）第103019号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：陈佳媛
北京京北印刷有限公司印刷
2011年6月第1版第1次印刷
186mm×240mm • 39印张
标准书号：ISBN 978-7-111-34967-9
定价：89.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 前 言

版本控制是管理数据变更的艺术，无论数据变更是来自同一个人，还是来自不同的人（一个团队）。版本控制系统不但要忠实地记录数据的每一次变更，还要能够帮助还原任何一次历史变更，以及实现团队的协同工作等。Git 就是版本控制系统中的佼佼者。

我对版本控制系统的兴趣源自于我的个人知识管理实践，其核心就是撰写可维护的文档，并保存于版本控制系统中。可维护文档的格式可以是 DocBook、FreeMind、reStructuredText 等。我甚至还对 FreeMind 加以改造以便让其文档格式更适合于版本控制系统，这就是我的第一个开源实践：托管于 SourceForge 上的 FreeMind-MMX 项目[①]。文档书写格式的问题解决之后，就是文档的存储问题了。通过版本控制系统，很自然地就可以实现对文档历史版本的保存，但是如何避免因为版本控制系统瘫痪而导致数据丢失呢？ Git 用其崭新的分布式的版本控制设计提供了最好的解决方案。使用 Git，我的知识库不再只有唯一的版本库与之对应，而是可以通过克隆操作分发到不同的磁盘或主机上，克隆的版本库之间通过推送（PUSH）和拉回（PULL）等操作进行同步，数据安全得到了极大的提升。在版本控制系统的忠实呵护下，我的知识库中关于 Git 的 FreeMind 脑图在日积月累中变得越来越翔实，越来越清晰，最终成为本书的雏形。

版本控制能决定项目的成败，甚至是公司的生死，此言不虚。我在推广开源项目管理工具和为企业提供咨询服务的过程中看到，有很多团队因为版本控制系统管理的混乱导致项目延期、修正的 Bug 重现、客户的问题不能在代码中定位……无论他们使用的是什么版本控制系统（开源的或是商业的）都是如此。这是因为传统的集中式版本控制系统不能有效地管理分支和进行分支间合并。集中管理的版本库只有唯一的分支命名空间，需要专人管理，从而造成分支创建的不自由；分支间的合并要么因为缺乏追踪导致重复合并、引发严重冲突，要

① *http://freemind-mmx.sourceforge.net/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


么因为版本控制系统本身蹩脚的设计导致分支合并时效率低下和陷阱重重。Git 凭借其灵活的设计让项目摆脱分支管理的梦魇。

我的公司也经历过代码管理的生死考验。因为公司的开发模式主要是基于开源软件的二次开发，所以最早在使用 SVN（Subversion）做版本控制时，很自然地使用了 SVN 卖主分支模型来管理代码。随着增加和修改的代码越来越多，我们开发的软件与开源软件上游的偏离也越来越远，当上游有新版本发布时，最早可能只用几个小时就可以将改动迁移过去，但是如果对上游的改动多达几十甚至上百处时，迁移的过程就会异常痛苦，基本上和重新做一遍差不多。那时似乎只有一种选择：不再与上游合并，不再追踪上游的改动，而这与公司的价值观“发动全球智慧为客户创造价值”相违背。迷茫之中，分布式版本控制系统飘然而至，原来版本控制还可以这么做。

我最先尝试的分布式版本控制系统是 Hg（Mercurial），当发现 Hg 和 MQ（Hg 的一个插件）这一对宝贝儿的时候，我如获至宝。逐渐地，公司的版本库都迁移到了 Hg 上。但随着新的开发人员的加入，问题又出现了，一个人使用 Hg 和 MQ 很好，但多个人使用时则会出现难以协同的问题。于是我们大胆地采用了 Git，并在实践中结合 Topgit 等工具进行代码的管理。再一次，也许是最后一次，我们的代码库迁移到了 Git。

最早认识分布式版本控制，源自于我们看到了众多开源项目的版本控制系统大迁移，这场迁移还在进行中。

- MoinMoin 是我们关注的一个开源的维基软件，2006 年，它的代码库从 SVN 迁移到了 Hg[①]。
- Mailman 同样是我们关注的一个开源邮件列表软件。2007 年，它的代码库从 SVN 迁移到了 Bazaar[②]。
- Linux 采用 Git 作为版本控制系统（一点都不奇怪，因为 Git 就是 Linus Torvalds 开发的）。
- Android 是目前最为流行的开源项目之一，因为潜在市场巨大，已经吸引了越来越多的开发者进入这个市场，而 Android 就是用 Git 维护的。

当开源软件纷纷倒向分布式版本控制系统大旗（尤其是 Git）的时候，很多商业公司也在行动了，尤其是涉及异地团队协同和 Android 核心代码定制开发的公司。对于那些因保守而不敢向 Git 靠拢的公司，Git 也可以派上用场，因为 Git 可以与现在大多数公司部署的 SVN 很好地协同，即公司的服务器是 SVN，开发者的客户端则使用 Git。相信随着 Git 的普及，以及公司在代码管理观念上的改进，会有更多的公司拥抱 Git。

① *http://moinmo.in/NewVCS*
② *http://wiki.list.org/display/DEV/Home*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 本书的组织

本书共分为 9 篇，前 8 篇是正文，一共 41 章，第 9 篇是附录。

第 1 篇讲解了 Git 的相关概念，以及安装和配置的方法，共 3 章。第 1 章介绍了版本控制的历史。第 2 章用十几个小例子介绍了 Git 的一些闪亮特性，期待这些特性能够让你爱上 Git。第 3 章则介绍了 Git 在三种主要操作系统平台上的安装和使用。在本书的写作过程中，我 70% 的时间使用的是 Debian Linux 操作系统，Linux 用户可以毫无障碍地完成本书列举的所有实践操作。在 2010 年年底，当得知有出版社愿意出版这本书后，我向妻子阿巧预支了未来的部分稿费购买了我的第一台 MacBook Pro，于是本书就有了较为翔实的如何在 Mac OS X 下安装和使用 Git 的内容，以及在本书第 22 章中介绍的关于 Topgit 在 Mac OS X 上的部署和改进相关的内容。在本书的编辑和校对过程中因为要使用 Word 格式的文稿，所以本书后期的很多工作是在运行于 VirtualBox 下的 Windows 虚拟机中完成的，即使是使用运行于资源受限的虚拟机中的 Cygwin，Git 依然完美地完成了工作。

第 2 篇和第 3 篇详细讲解了 Git 的使用方法，是本书的基础和核心，大约占据了全书 40% 的篇幅。这两篇的内容架构方式是我在进行 SVN 培训时就已经形成的习惯，即以“独奏”指代一个人的版本控制所要讲述的知识点，以“和声”指代团队版本控制涉及的话题。在第 2 篇“Git 独奏”中，本书将 Git 的设计原理穿插在各章之中讲解，因为唯有了解真相（Git 原理），才有可能自由（掌握 Git）。在第 3 篇“Git 和声”中，本书讲解了团队版本控制必须掌握的里程碑和分支等概念，以及如何解决合并中遇到的冲突。

第 4 篇细致地讲解了 Git 在实际工作中的使用模式。除了传统的集中式和分布式使用模式之外，第 22 章还介绍了 Topgit 在定制开发中的应用，这也是我公司在使用 Git 时采用的最主要的模式。这一章还讲解了我对 Topgit 所做的部分改进，相关的具体介绍最早出现在我公司的博客上[①]。第 23 ～ 25 章介绍了多版本库协同的不同方法，其中第 25 章介绍的一个独辟蹊径的解决方案是由 Android 项目引入的名为 repo 的工具实现的，我对其进行改造后可以让这个工具脱离 Gerrit 代码审核服务器，直接操作 Git 服务器。第 26 章介绍了 git-svn 这一工具，该工具不但可以实现从 SVN 版本库到 Git 版本库的迁移，还可以实现以 Git 作为客户端向 SVN 提交。

第 5 篇介绍了 Git 服务器的架设。本篇是全书最早开始撰写的部分，这是因为我给客户做的 Git 培训讲义的相关内容不够详细，于是应客户要求针对 Gitolite 等服务器的架设撰写了详细的管理员手册，即本书的第 30 章。第 32 章介绍了 Android 项目在 Git 管理上的又一大创造，即 Gerrit，它实现了一个独特的集中式 Git 版本库管理模型。

第 6 篇讲解了 Git 版本库的迁移。其中第 34 章详细介绍了从 CVS 版本库到 Git 版本库

① *http://blog.ossxp.com/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的迁移，其迁移过程也可以作为从 CVS 到 SVN 迁移的借鉴。本篇还介绍了从 SVN 和 Hg 版本库到 Git 的迁移。对于其他类型的版本库，介绍了一个通用的需要编程来实现的方法。在本篇的最后还介绍了一个 Git 版本库整理的利器，可以理解为一个 Git 库转换为另外一个 Git 库的方法。

第 7 篇是关于 Git 的其他应用，其主要内容介绍了我在 etckeeper 启发下开发的一款备份工具 Gistore，该工具可以运行于 Linux 和 Mac OS X 下。

第 8 篇是 Git 杂谈。其中第 40 章的内容可供跨平台的项目组借鉴。第 41 章介绍了一些在前面没有涉及的 Git 的相关功能和特性。

第 9 篇是附录。首先介绍了完整的 Git 命令索引，然后分别介绍了 CVS、SVN、Hg 与 Git 之间的比较和命令对照，对于有其他版本控制系统使用经验的用户而言，这一部分内容颇具参考价值。

## 适用读者

本书适合所有翻开它的人，因为我知道这本书在书店里一定是放在计算机图书专柜。本书尤其适合以下几类读者阅读。

### 1. 被数据同步困扰的“电脑人”

困扰“电脑人”的一个常见问题是，有太多的数据需要长久保存，有太多的电脑设备需要数据同步。可能有的人会说：“像 Dropbox 一样的网盘可以帮助我呀”。是的，云存储就是在技术逐渐成熟之后应运而生的产品，但是依然解决不了如下几个问题：多个设备上同时修改造成的冲突；冗余数据传输造成的带宽瓶颈；没有实现真正的、完全的历史变更数据备份。具体请参见本书第 7 篇第 39 章的内容。

Git 可以在数据同步方面做得更好，甚至只需借助小小的 U 盘就可以实现多台电脑的数据同步，并且支持自动的冲突解决。只要阅读本书第 1 篇和第 2 篇，就能轻易掌握相关的操作，实现数据的版本控制和同步。

### 2. 学习计算机课程的学生

我非常后悔没有在学习编程的第一天就开始使用版本控制，在学校时写的很多小程序和函数库都丢失了。直到使用了 CVS 和 SVN 对个人数据进行版本控制之后，才开始把每一天的变更历史都保留了下来。Git 在这方面可以比 CVS 和 SVN 等做得更好。

在阅读完本书的前 3 篇掌握了 Git 的基础知识之后，可以阅读第 5 篇第 33 章的内容，通过 Github 或类似的服务提供商建立自己的版本库托管，为自己的数据找一个安全的家。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 3. 程序员

使用 Git 会让程序员有更多的时间休息，因为可以更快地完成工作。分布式版本控制让每一个程序员都能在本地拥有一个完整的版本库，所以几乎所有操作都能够脱离网络执行而不受带宽的限制。加之使用了智能协议，版本库间的同步不但减少了数据传输量，还能显示完成进度。

Git 帮助程序员打开了进入开源世界的大门，进而开阔视野，提升水平，增加择业的砝码。看看使用 Git 作为版本控制的开源软件吧：Linux kernel、Android、Debian、Fedora、GNOME、KDevelop、jQuery、Prototype、PostgreSQL、Ruby on Rails……不胜枚举。还有，不要忘了所有的 SVN 版本库都可以通过 Git 方式更好地访问。

作为一个程序员，必须具备团队协同能力，本书第 3 篇应该作为学习的重点。

## 4.Android 程序员

如果你是谷歌 Android 项目的参与者，尤其是驱动开发和核心开发的参与者，必然会接触 Git、repo 和 Gerrit。对于只是偶尔参考一下 Android 核心代码的 Android 应用开发人员而言，也需要对 repo 有深入的理解，这样才不至于每次为同步代码而耗费一天的时间。

repo 是 Android 为了解决 Git 多版本库管理问题而设计的工具，在本书第 4 篇第 25 章有详细介绍。

Gerrit 是谷歌为了避免因分布式开发造成项目分裂而开发的工具，打造了 Android 独具一格的集中式管理模式，在本书第 5 篇第 32 章有详细介绍。

即使是非 Android 项目，也可以使用这两款工具为自己的项目服务。我还为 repo 写了几个新的子命令以实现脱离 Gerrit 提交，让 repo 拥有更广泛的应用领域。

## 5. 定制开发程序员

当一个公司的软件产品需要针对不同的用户进行定制开发时，就需要在一个版本库中建立大量的特性分支，使用 SVN 的分支管理远不如使用 Git 的分支管理那么自然和方便。还有一个应用领域就是对第三方代码进行维护。当使用 SVN 进行版本控制时，最自然的选择是卖主分支，但随着定制开发的逐渐深入，与上游的偏离也会越大，于是与上游代码的合并也将越来越令人痛苦。

第 4 篇第 22 章介绍 Topgit 这一杀手级的工具，这是这个领域最佳的解决方案。

## 6.SVN 用户

商业软件的研发团队因为需要精细的代码授权，所以不会轻易更换现有的 SVN 版本控制系统，这种情况下 Git 依然大有作为。无论是出差在外，或是在家办公，或是开发团队分处异地，都会遇到 SVN 版本控制服务器无法访问或速度较慢的情况。这时 git-svn 这一工具

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

会将 Git 和 SVN 完美地结合在一起，既严格遵守 SVN 的授权规定，又可以自如地进行本地提交，当能够连接到 SVN 服务器时，可以在悠闲地喝着绿茶的同时，等待一次性批量提交的完成。

我有几个项目（pySvnManager、Freemind-MMX）托管在 SourceForge 的 SVN 服务器上，现在都是先通过 git-svn 将其转化为本地的 Git 库，然后再使用的。以这样的方式访问历史数据、比较代码或提交代码，再也不会因为网速太慢而望眼欲穿了。

本书第 4 篇第 26 章详细介绍了 Git 和 SVN 的互操作。

### 7. 管理员

Git 在很大程度上减轻了管理员的负担：分支的创建和删除不再需要管理员统一管理，因为作为分布式版本控制系统，每一个克隆就是一个分支，每一个克隆都拥有独立的分支命名空间；管理员也不再需要为版本库的备份操心，因为每一个项目成员都拥有一个备份；管理员也不必担心有人在服务器上篡改版本库，因为 Git 版本库的每一个对象（提交和文件等）都使用 SHA1 哈希值进行完整性校验，任何对历史数据的篡改都会因为对后续提交产生的连锁反应而原形毕露。

本书第 7 篇第 37 章介绍了一款我开发的基于 Git 的备份工具，它使得 Linux 系统的数据备份易如反掌。本书第 5 篇介绍的 Git 服务器搭建，以及第 6 篇介绍的版本库迁移方面的知识会为版本控制管理员的日常维护工作提供指引。

### 8. 开发经理

作为开发经理，你一定要对代码分支有深刻的理解，不知本书第 18 章中的“代码管理之殇”是否能引起你的共鸣。为了能在各种情况下恰当地管理开发团队，第 4 篇“Git 协同模型”是项目经理应该关注的重点。你的团队是否存在着跨平台开发，或者潜在着跨平台开发的可能？本书第 8 篇第 40 章也是开发经理应当关注的内容。

## 排版约定

本书使用的排版格式约定如下：

1. 命令输出及示例代码

执行一条 Git 命令及其输出的示例如下：

```
$ git --version
git version 1.7.4
```

2. 提示符（$）

命令前面的 $ 符号代表命令提示符。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


3. 等宽字体（Constant width）

用于标示屏幕输出的字符、示例代码，以及正文中出现的命令、参数、文件名和函数名等。

4. 等宽粗体（**Constant width bold**）

用于表示由用户手工输入的内容。

5. 占位符（*<Constant width>*）

用尖括号扩起来的内容，表示命令中或代码中的占位符，读者应当用实际值将其替换。

## 在线资源

- 官方网站：*http://www.ossxp.com/doc/gotgit/*

  在本书的官方网站上，大家可以了解到与本书相关的最新信息，查看本书的勘误，以及下载与本书相关的资源。官网是以 Git 方式维护的，人人都可以参与其中。

- 新浪微博：*http://weibo.com/GotGit*

  欢迎大家通过新浪微博与作者交流，也欢迎大家通过新浪微博将你们的宝贵意见和建议反馈给作者。

## 致谢

感谢 Linus Torvalds、Junio C Hamano 和 Git 项目的所有贡献者，是他们带给我们崭新的版本控制体验。

本书能够出版要感谢机械工业出版社华章公司，华章公司对中文原创计算机图书的信任让中国的每一个计算机从业者都有可能圆自己出书的梦想。作为一个新人，拿着一个新的选题，遇到同样充满激情的编辑，我无疑是幸运的。这个充满激情的编辑，就是华章公司的杨福川编辑。甚至没有向我索要样章，在看过目录之后就“冒险”和我签约，他的激情让我不敢懈怠。同样要感谢王晓菲编辑，她的耐心和细致让我吃惊，也正是因为她的工作本书的行文才能更加流畅，本书也才能够更快问世。还有张少波编辑，感谢她在接到我的电话后帮我分析选题并推荐给杨福川编辑。

本书的部分内容是由我的 Git 培训讲义扩展而来的，在此感谢朝歌数码的蒋宗贵，是他的鼓励和鞭策让我完善了本书中的与服务器架设的相关章节。还要感谢王彦宁，正是通过她的团队我才认识了 Android，才有了本书关于 repo 和 Gerrit 的相关章节。

感谢群英汇的同事们，尤其要感谢王胜，正是因为我们在使用 Topgit 0.7 版本时遇到了严重的冲突，才使我下定决心研究 Git。

感谢上海爱立信研发中心的高级技术专家蔡煜，他对全书尤其是 git-svn 和 Gitolite 相关章节做了重点评审，他的意见和建议修正了本书的很多不当之处。因为时间的关系，他的一些非常好的观点没有机会在这一版中体现，争取在改版时弥补遗憾。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


中国科学院软件研究所的张先轶、比蒙科技的宋伯润和杨致伟、摩博科技的熊军、共致开源的秦红胜，以及王胜等人为本书的技术审校提供了帮助，感谢他们的宝贵意见和建议。来自中国台湾的 PyLabs 团队纠正了本书在对 Hg 的认识上的偏颇，让本书附录中的相关内容更加准确和客观，在此向他们表示感谢。

因为写书亏欠家人很多，直到最近才发现女儿小雪是多么希望拥有一台儿童自行车。感谢妻子阿巧对我的耐心和为家庭的付出。感谢岳父、岳母这几年来对小雪和我们整个家庭的照顾，让我没有后顾之忧。还要感谢我的父母和妹妹，他们对我事业的支持和鼓励是我前进的动力。在我写作本书的同时，老爸正在富春江畔代表哈尔滨电机厂监督发电机组的制造，而且也在写一本监造手册方面的书，抱歉老爸，我先完成了。 :)

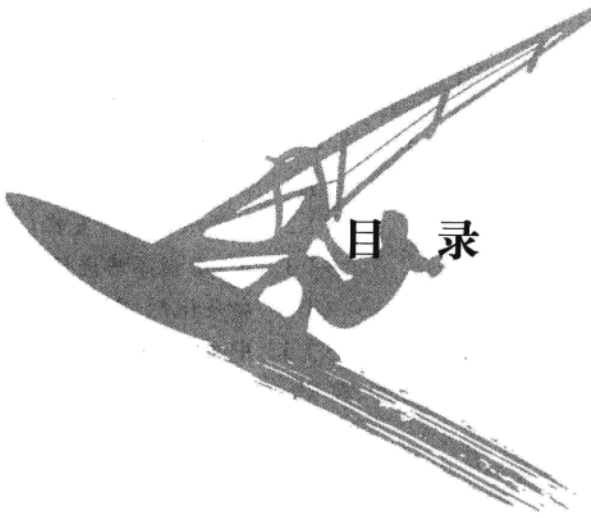
蒋鑫（http://www.ossxp.com/）

2011 年 4 月



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 目 录

前 言

## 第 1 篇 初识 Git

### 第 1 章 版本控制的前世和今生 / 2

1.1 黑暗的史前时代 / 2
1.2 CVS——开启版本控制大爆发 / 5
1.3 SVN——集中式版本控制集大成者 / 7
1.4 Git——Linus 的第二个伟大作品 / 9

### 第 2 章 爱上 Git 的理由 / 11

2.1 每日工作备份 / 11
2.2 异地协同工作 / 12
2.3 现场版本控制 / 13
2.4 避免引入辅助目录 / 15
2.5 重写提交说明 / 15
2.6 想吃后悔药 / 16
2.7 更好用的提交列表 / 17
2.8 更好的差异比较 / 18
2.9 工作进度保存 / 18
2.10 代理 SVN 提交实现移动式办公 / 19
2.11 无处不在的分页器 / 20

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


2.12 快 / 21

## 第 3 章 Git 的安装和使用 / 22

3.1 在 Linux 下安装和使用 Git/ 22
- 3.1.1 包管理器方式安装 / 22
- 3.1.2 从源代码进行安装 / 23
- 3.1.3 从 Git 版本库进行安装 / 23
- 3.1.4 命令补齐 / 25
- 3.1.5 中文支持 / 25

3.2 在 Mac OS X 下安装和使用 Git/ 26
- 3.2.1 以二进制发布包的方式安装 / 26
- 3.2.2 安装 Xcode/ 27
- 3.2.3 使用 Homebrew 安装 Git/ 29
- 3.2.4 从 Git 源码进行安装 / 29
- 3.2.5 命令补齐 / 30
- 3.2.6 其他辅助工具的安装 / 30
- 3.2.7 中文支持 / 31

3.3 在 Windows 下安装和使用 Git（Cygwin 篇）/ 31
- 3.3.1 安装 Cygwin/ 32
- 3.3.2 安装 Git/ 36
- 3.3.3 Cygwin 的配置和使用 / 37
- 3.3.4 Cygwin 下 Git 的中文支持 / 40
- 3.3.5 Cygwin 下 Git 访问 SSH 服务 / 41

3.4 Windows 下安装和使用 Git（msysGit 篇）/ 45
- 3.4.1 安装 msysGit/ 46
- 3.4.2 msysGit 的配置和使用 / 48
- 3.4.3 msysGit 中 shell 环境的中文支持 / 49
- 3.4.4 msysGit 中 Git 的中文支持 / 50
- 3.4.5 使用 SSH 协议 / 51
- 3.4.6 TortoiseGit 的安装和使用 / 52
- 3.4.7 TortoiseGit 的中文支持 / 55

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


# 第 2 篇 Git 独奏


## 第 4 章 Git 初始化 / 58

4.1 创建版本库及第一次提交 / 58
4.2 思考：为什么工作区根目录下有一个 .git 目录 / 60
4.3 思考：git config 命令的各参数有何区别 / 63
4.4 思考：是谁完成的提交 / 65
4.5 思考：随意设置提交者姓名，是否太不安全 / 67
4.6 思考：命令别名是干什么的 / 68
4.7 备份本章的工作成果 / 69

## 第 5 章 Git 暂存区 / 70

5.1 修改不能直接提交吗 / 70
5.2 理解 Git 暂存区（stage） / 76
5.3 Git Diff 魔法 / 78
5.4 不要使用 git commit -a/ 81
5.5 搁置问题，暂存状态 / 82

## 第 6 章 Git 对象 / 83

6.1 Git 对象库探秘 / 83
6.2 思考：SHA1 哈希值到底是什么，是如何生成的 / 88
6.3 思考：为什么不用顺序的数字来表示提交 / 90

## 第 7 章 Git 重置 / 93

7.1 分支游标 master 探秘 / 93
7.2 用 reflog 挽救错误的重置 / 95
7.3 深入了解 git reset 命令 / 96

## 第 8 章 Git 检出 / 99

8.1 HEAD 的重置即检出 / 99


更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


8.2 挽救分离头指针 / 102
8.3 深入了解 git checkout 命令 / 103

**第 9 章 恢复进度 / 105**

9.1 继续暂存区未完成的实践 / 105
9.2 使用 git stash/ 108
9.3 探秘 git stash/ 109

**第 10 章 Git 基本操作 / 114**

10.1 先来合个影 / 114
10.2 删除文件 / 114
10.2.1 本地删除不是真的删除 / 115
10.2.2 执行 git rm 命令删除文件 / 116
10.2.3 命令 git add -u 快速标记删除 / 117
10.3 恢复删除的文件 / 118
10.4 移动文件 / 119
10.5 一个显示版本号的 Hello World/ 120
10.6 使用 git add -i 选择性添加 / 122
10.7 Hello World 引发的新问题 / 124
10.8 文件忽略 / 125
10.9 文件归档 / 129

**第 11 章 历史穿梭 / 130**

11.1 图形工具：gitk/ 130
11.2 图形工具：gitg/ 131
11.3 图形工具：qgit/ 135
11.4 命令行工具 / 140
11.4.1 版本表示法：git rev-parse/ 141
11.4.2 版本范围表示法：git rev-list/ 144
11.4.3 浏览日志：git log/ 146
11.4.4 差异比较：git diff/ 150

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


11.4.5 文件追溯：git blame/ 151
11.4.6 二分查找：git bisect/ 152
11.4.7 获取历史版本 / 156

**第 12 章 改变历史 / 157**

12.1 悔棋 / 157
12.2 多步悔棋 / 159
12.3 回到未来 / 161
12.3.1 时间旅行一 / 162
12.3.2 时间旅行二 / 167
12.3.3 时间旅行三 / 171
12.4 丢弃历史 / 174
12.5 反转提交 / 177

**第 13 章 Git 克隆 / 179**

13.1 鸡蛋不装在一个篮子里 / 179
13.2 对等工作区 / 180
13.3 克隆生成裸版本库 / 183
13.4 创建生成裸版本库 / 184

**第 14 章 Git 库管理 / 187**

14.1 对象和引用哪里去了 / 187
14.2 暂存区操作引入的临时对象 / 189
14.3 重置操作引入的对象 / 191
14.4 Git 管家：git-gc/ 193
14.5 Git 管家的自动执行 / 196

## 第 3 篇 Git 和声

**第 15 章 Git 协议与工作协同 / 200**

15.1 Git 支持的协议 / 200

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


15.2 多用户协同的本地模拟 / 202
15.3 强制非快进式推送 / 203
15.4 合并后推送 / 207
15.5 禁止非快进式推送 / 208

**第 16 章 冲突解决 / 210**

16.1 拉回操作中的合并 / 210
16.2 合并一：自动合并 / 212
16.2.1 修改不同的文件 / 212
16.2.2 修改相同文件的不同区域 / 214
16.2.3 同时更改文件名和文件内容 / 215
16.3 合并二：逻辑冲突 / 217
16.4 合并三：冲突解决 / 218
16.4.1 手工编辑完成冲突解决 / 221
16.4.2 图形工具完成冲突解决 / 221
16.5 合并四：树冲突 / 225
16.5.1 手工操作解决树冲突 / 227
16.5.2 交互式解决树冲突 / 228
16.6 合并策略 / 230
16.7 合并相关的设置 / 231

**第 17 章 Git 里程碑 / 233**

17.1 显示里程碑 / 234
17.2 创建里程碑 / 236
17.2.1 轻量级里程碑 / 237
17.2.2 带说明的里程碑 / 238
17.2.3 带签名的里程碑 / 239
17.3 删除里程碑 / 242
17.4 不要随意更改里程碑 / 243
17.5 共享里程碑 / 243
17.6 删除远程版本库的里程碑 / 246
17.7 里程碑命名规范 / 247

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


**第 18 章 Git 分支 / 253**

18.1 代码管理之殇 / 253
18.1.1 发布分支 / 253
18.1.2 特性分支 / 256
18.1.3 卖主分支 / 257
18.2 分支命令概述 / 258
18.3 “Hello World”开发计划 / 259
18.4 基于特性分支的开发 / 260
18.4.1 创建分支 user1/getopt/ 261
18.4.2 创建分支 user2/i18n/ 262
18.4.3 开发者 user1 完成功能开发 / 263
18.4.4 将 user1/getopt 分支合并到主线 / 264
18.5 基于发布分支的开发 / 265
18.5.1 创建发布分支 / 266
18.5.2 开发者 user1 工作在发布分支 / 267
18.5.3 开发者 user2 工作在发布分支 / 268
18.5.4 开发者 user2 合并推送 / 270
18.5.5 发布分支的提交合并到主线 / 271
18.6 分支变基 / 275
18.6.1 完成 user2/i18n 特性分支的开发 / 275
18.6.2 分支 user2/i18n 变基 / 277

**第 19 章 远程版本库 / 284**

19.1 远程分支 / 284
19.2 分支追踪 / 287
19.3 远程版本库 / 290
19.4 PUSH 和 PULL 操作与远程版本库 / 292
19.5 里程碑和远程版本库 / 294
19.6 分支和里程碑的安全性 / 294



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


**第 20 章 补丁文件交互 / 296**

20.1 创建补丁 / 296
20.2 应用补丁 / 297
20.3 StGit 和 Quilt/ 300
20.3.1 StGit/ 300
20.3.2 Quilt/ 304

# 第 4 篇 Git 协同模型

**第 21 章 经典 Git 协同模型 / 308**

21.1 集中式协同模型 / 308
21.1.1 传统集中式协同模型 / 309
21.1.2 Gerrit 特殊的集中式协同模型 / 310
21.2 金字塔式协同模型 / 311
21.2.1 贡献者开放只读版本库 / 312
21.2.2 以补丁方式贡献代码 / 313

**第 22 章 Topgit 协同模型 / 314**

22.1 作者版本控制系统的三个里程碑 / 314
22.2 Topgit 原理 / 316
22.3 Topgit 的安装 / 317
22.4 Topgit 的使用 / 319
22.5 用 Topgit 方式改造 Topgit/ 330
22.6 Topgit 使用中的注意事项 / 334

**第 23 章 子模组协同模型 / 336**

23.1 创建子模组 / 336
23.2 克隆带子模组的版本库 / 339
23.3 在子模组中修改和子模组的更新 / 340
23.4 隐性子模组 / 343

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


23.5 子模组的管理问题 / 345

**第 24 章 子树合并 / 347**

24.1 引入外部版本库 / 347
24.2 子目录方式合并外部版本库 / 349
24.3 利用子树合并跟踪上游改动 / 351
24.4 子树拆分 / 353
24.5 git-subtree 插件 / 353

**第 25 章 Android 式多版本库协同 / 356**

25.1 关于 repo/ 357
25.2 安装 repo/ 357
25.3 repo 和清单库的初始化 / 359
25.4 清单库和清单文件 / 360
25.5 同步项目 / 361
25.6 建立 Android 代码库本地镜像 / 363
25.7 repo 的命令集 / 365
25.8 repo 命令的工作流 / 370
25.9 好东西不能 Android 独享 / 371
25.9.1 repo+Gerrit 模式 / 371
25.9.2 repo 无审核模式 / 371
25.9.3 改进的 repo 无审核模式 / 372

**第 26 章 Git 和 SVN 协同模型 / 378**

26.1 使用 git-svn 的一般流程 / 380
26.2 git-svn 的奥秘 / 386
26.2.1 Git 库配置文件的扩展及分支映射 / 387
26.2.2 Git 工作分支和 Subversion 如何对应 / 388
26.2.3 其他辅助文件 / 390
26.3 多样的 git-svn 克隆模式 / 390
26.4 共享 git-svn 的克隆库 / 393

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


26.5 git-svn 的局限 / 394

## 第 5 篇 搭建 Git 服务器

### 第 27 章 使用 HTTP 协议 / 398

27.1 哑传输协议 / 398
27.2 智能 HTTP 协议 / 400
27.3 Gitweb 服务器 / 401
27.3.1 Gitweb 的安装 / 402
27.3.2 Gitweb 的配置 / 403
27.3.3 版本库的 Gitweb 相关设置 / 404
27.3.4 即时 Gitweb 服务 / 405

### 第 28 章 使用 Git 协议 / 406

28.1 Git 协议语法格式 / 406
28.2 Git 服务软件 / 406
28.3 以 inetd 方式配置运行 / 406
28.4 以 runit 方式配置运行 / 407

### 第 29 章 使用 SSH 协议 / 409

29.1 SSH 协议语法格式 / 409
29.2 服务架设方式比较 / 409
29.3 关于 SSH 公钥认证 / 411
29.4 关于 SSH 主机别名 / 411

### 第 30 章 Gitolite 服务架设 / 413

30.1 安装 Gitolite/ 414
30.1.1 服务器端创建专用账号 / 414
30.1.2 Gitolite 的安装 / 升级 / 415
30.1.3 关于 SSH 主机别名 / 417

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


30.1.4 其他的安装方法 / 418
30.2 管理 Gitolite/ 419
30.2.1 管理员克隆 gitolite-admin 管理库 / 419
30.2.2 增加新用户 / 420
30.2.3 更改授权 / 422
30.3 Gitolite 授权详解 / 423
30.3.1 授权文件的基本语法 / 423
30.3.2 定义用户组和版本库组 / 424
30.3.3 版本库 ACL/ 424
30.3.4 Gitolite 授权机制 / 426
30.4 版本库授权案例 / 427
30.4.1 对整个版本库进行授权 / 427
30.4.2 通配符版本库的授权 / 428
30.4.3 用户自己的版本库空间 / 429
30.4.4 对引用的授权：传统模式 / 430
30.4.5 对引用的授权：扩展模式 / 430
30.4.6 对引用的授权：禁用规则的使用 / 431
30.4.7 用户分支 / 431
30.4.8 对路径的写授权 / 432
30.5 创建新版本库 / 432
30.5.1 在配置文件中出现的版本库，即时生成 / 433
30.5.2 通配符版本库，管理员通过推送创建 / 434
30.5.3 直接在服务器端创建 / 435
30.6 对 Gitolite 的改进 / 435
30.7 Gitolite 功能拓展 / 436
30.7.1 版本库镜像 / 436
30.7.2 Gitweb 和 Git daemon 支持 / 438
30.7.3 其他功能拓展和参考 / 439

**第 31 章 Gitosis 服务架设 / 441**

31.1 安装 Gitosis/ 442
31.1.1 Gitosis 的安装 / 442

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


31.1.2 服务器端创建专用账号 / 442
31.1.3 Gitosis 服务初始化 / 443
31.2 管理 Gitosis/ 443
31.2.1 管理员克隆 gitolit-admin 管理库 / 443
31.2.2 增加新用户 / 444
31.2.3 更改授权 / 446
31.3 Gitosis 授权详解 / 447
31.3.1 Gitosis 默认设置 / 447
31.3.2 管理版本库 gitosis-admin/ 447
31.3.3 定义用户组和授权 / 448
31.3.4 Gitweb 整合 / 449
31.4 创建新版本库 / 449
31.5 轻量级管理的 Git 服务 / 450

**第 32 章 Gerrit 代码审核服务器 / 452**

32.1 Gerrit 的实现原理 / 452
32.2 架设 Gerrit 的服务器 / 456
32.3 Gerrit 的配置文件 / 461
32.4 Gerrit 的数据库访问 / 462
32.5 立即注册为 Gerrit 管理员 / 464
32.6 管理员访问 SSH 的管理接口 / 467
32.7 创建新项目 / 468
32.8 从已有的 Git 库创建项目 / 472
32.9 定义评审工作流 / 473
32.10 Gerrit 评审工作流实战 / 477
32.10.1 开发者在本地版本库中工作 / 477
32.10.2 开发者向审核服务器提交 / 478
32.10.3 审核评审任务 / 478
32.10.4 评审任务没有通过测试 / 480
32.10.5 重新提交新的补丁集 / 482
32.10.6 新修订集通过评审 / 483
32.10.7 从远程版本库更新 / 485

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


32.11 更多 Gerrit 参考 / 486

第 33 章 Git 版本库托管 / 487

33.1 Github/ 487
33.2 Gitorious/ 489

第 6 篇 迁移到 Git

第 34 章 CVS 版本库到 Git 的迁移 / 492

34.1 安装 cvs2svn（含 cvs2git）/ 492
34.1.1 Linux 下 cvs2svn 的安装 / 492
34.1.2 Mac OS X 下 cvs2svn 的安装 / 493
34.2 版本库转换的准备工作 / 494
34.2.1 版本库转换注意事项 / 494
34.2.2 文件名乱码问题 / 494
34.2.3 提交说明乱码问题 / 494
34.3 版本库转换 / 496
34.3.1 配置文件解说 / 496
34.3.2 运行 cvs2git 完成转换 / 500
34.4 迁移后的版本库检查 / 501

第 35 章 更多版本控制系统的迁移 / 502

35.1 SVN 版本库到 Git 的迁移 / 502
35.2 Hg 版本库到 Git 的迁移 / 503
35.3 通用版本库迁移 / 505
35.4 Git 版本库整理 / 511
35.4.1 环境变量过滤器 / 513
35.4.2 树过滤器 / 513
35.4.3 暂存区过滤器 / 513
35.4.4 父节点过滤器 / 514

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

35.4.5 提交说明过滤器 / 514
35.4.6 提交过滤器 / 515
35.4.7 里程碑名字过滤器 / 516
35.4.8 子目录过滤器 / 516

## 第 7 篇 Git 的其他应用

### 第 36 章 etckeeper/ 518

36.1 安装 etckeeper/ 518
36.2 配置 etckeeper/ 519
36.3 使用 etckeeper/ 519

### 第 37 章 Gistore/ 520

37.1 Gistore 的安装 / 520
37.1.1 软件依赖 / 520
37.1.2 从源码安装 Gistore/ 521
37.1.3 用 easy_install 安装 / 521
37.2 Gistore 的使用 / 522
37.2.1 创建并初始化备份库 / 522
37.2.2 Gistore 的配置文件 / 523
37.2.3 Gistore 的备份项管理 / 524
37.2.4 执行备份任务 / 525
37.2.5 查看备份日志 / 525
37.2.6 查看及恢复备份数据 / 527
37.2.7 备份回滚及设置 / 528
37.2.8 注册备份任务别名 / 529
37.2.9 自动备份：crontab/ 529
37.3 Gistore 双机备份 / 529

### 第 38 章 补丁中的二进制文件 / 531

38.1 Git 版本库中二进制文件变更的支持 / 531

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


38.2 对非 Git 版本库中二进制文件变更的支持 / 535
38.3 其他工具对 Git 扩展补丁文件的支持 / 536

第 39 章 云存储 / 538

39.1 现有云存储的问题 / 538
39.2 Git 式云存储畅想 / 539

第 8 篇 Git 杂谈

第 40 章 跨平台操作 Git/ 542

40.1 字符集问题 / 542
40.2 文件名大小写问题 / 543
40.3 换行符问题 / 545

第 41 章 Git 的其他特性 / 549

41.1 属性 / 549
41.1.1 属性定义 / 549
41.1.2 属性文件及优先级 / 550
41.1.3 常用属性介绍 / 552
41.2 钩子和模板 / 557
41.2.1 Git 钩子 / 557
41.2.2 Git 模板 / 562
41.3 稀疏检出和浅克隆 / 563
41.3.1 稀疏检出 / 563
41.3.2 浅克隆 / 566
41.4 嫁接和替换 / 568
41.4.1 提交嫁接 / 568
41.4.2 提交替换 / 568
41.5 Git 评注 / 570
41.5.1 评注的奥秘 / 570

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


41.5.2 评注相关命令 / 573
41.5.3 评注相关配置 / 574

## 第 9 篇 附录

### 附录 A Git 命令索引 / 576

A.1 常用的 Git 命令 / 576
A.2 对象库操作相关命令 / 578
A.3 引用操作相关命令 / 578
A.4 版本库管理相关命令 / 579
A.5 数据传输相关命令 / 579
A.6 邮件相关命令 / 580
A.7 协议相关命令 / 580
A.8 版本库转换和交互相关命令 / 581
A.9 合并相关的辅助命令 / 581
A.10 杂项 / 582

### 附录 B Git 与 CVS 面对面 / 583

B.1 面对面访谈录 / 583
B.2 Git 和 CVS 命令对照 / 585

### 附录 C Git 与 SVN 面对面 / 587

C.1 面对面访谈录 / 587
C.2 Git 和 SVN 命令对照 / 589

### 附录 D Git 与 Hg 面对面 / 592

D.1 面对面访谈录 / 592
D.2 Git 和 Hg 命令对照 / 593



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第 1 篇 初识 Git

Git 是一款分布式版本控制系统，有别于 CVS 和 SVN 等集中式版本控制系统，Git 可以让研发团队更加高效地协同工作，从而提高生产率。使用 Git，开发人员的工作不会因为频繁地遭遇提交冲突而中断，管理人员也无须为数据的备份而担心。经过 Linux 这样的庞大项目的考验之后，Git 被证明可以胜任任何规模的团队，即便团队成员分布于世界各地。

Git 是开源社区奉献给每一个人的宝贝，用好它不仅可以实现个人的知识积累、保护好自己的数据，而且还能与他人分享自己的成果。这在其他的很多版本控制系统中是不可想象的。你会为个人的版本控制而花费高昂的费用去购买商业版本控制工具吗？你会去使用必须搭建额外的服务器才能使用的版本控制系统吗？你会把“鸡蛋”放在具有单点故障、服务器软硬件有可能崩溃的唯一的“篮子”里吗？如果你不会，那么选择 Git，一定是最明智的选择。

本篇我们首先用一章的内容来回顾一下版本控制的历史，并以此向版本控制的前辈 CVS 和 SVN 致敬。第 2 章会通过一些典型的版本控制实例向您展示 Git 独特的魅力，让您爱上 Git。在本篇的最后一章会介绍 Git 在 Linux、Mac OS X 及 Windows 下的安装和使用，这是我们进一步研究 Git 的基础。



在这里有必要纠正一下 Git 的发音。一种错误是按照单个字母来发音，另外一种更为普遍的错误是把整个单词读作“技特”，实际上 Git 中字母 G 的发音与下列单词中的 G 类似：GOD、GIVES、GREAT、GIFT。因此 Git 正确的发音应该听起来像是“歌易特”。本书的英文名为《Got Git》，当面对这样的书名时您还会把 Git 读错吗？

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 1 章　版本控制的前世和今生

除了茫然未知的宇宙，几乎任何事物都是从无到有，从简陋到完善。随着时间车轮的滚滚向前，历史被抛在身后逐渐远去，如同我们的现代社会，世界大同，到处都是忙碌和喧嚣，再也看不到已经远去的刀耕火种、男耕女织的慢生活岁月。

版本控制系统是一个另类。虽然其历史并不短暂，也有几十年，但是它的演进进程却一直在社会的各个角落重复着，而且惊人的相似。有的人从未使用甚至从未听说过版本控制系统，他和他的团队就像停留在黑暗的史前时代，任由数据自生自灭。有的人使用着有几十年历史的 CVS 或其改良版 Subversion，让时间空耗在网络连接的等待中。以 Git 为代表的分布式版本控制系统已经风靡整个开源社区，正等待你的靠近。

## 1.1　黑暗的史前时代

谈及远古，人们总爱以“黑暗”来形容。黑暗实际上指的是秩序和工具的匮乏，而不是自然。如果以自然环境而论，由于工业化和城市化对环境的破坏，现今才是最黑暗的年代。对于软件开发来说也是如此，在 C 语言一统天下的日子里我们的选择很简单，如今面临 Java、.Net 和脚本语言时，我们的选择变得复杂起来，但是从工具和秩序上讲，过去的年代是黑暗的。

回顾一下我经历的版本控制的“史前时代”吧。在大学里，代码分散地拷贝在各个软盘中，最终我被搞糊涂，不知道哪个软盘中的代码是最优的，因为最新并非最优，失败的重构会毁掉原来尚能运作的代码。在我工作的第一年，代码的管理并未得到改善，还是以简单的目录拷贝进行数据的备份，三四个程序员利用文件服务器的共享目录进行协同，公共类库和头文件在操作过程中相互覆盖，痛苦不堪。很明显，那时我尚不知道版本控制系统为何物。我的版本控制史前时代一直延续到 2000 年，那时 CVS 已经诞生了 14 年，而我在那时对 CVS 还一无所知。

实际上，即便是在 CVS 出现之前的“史前时代”，也已经有了非常好用的用于源码比较和打补丁的工具：`diff` 和 `patch`，它们今天生命力依然顽强。大名鼎鼎的 Linus Torvalds 先生（Linux 之父）也对这两个工具偏爱有加，在 1991 ～ 2002 年之间，Linus 一直顽固地使用 `diff`、`patch` 和 `tar` 包管理着 Linux 的代码，虽然不断有人提醒他有 CVS 的存在[①]。

那么来看看 `diff` 和 `patch`，熟悉它们将对理解版本控制系统（差异存储）和使用版本

① Linus Torvalds 于 2007 年 5 月 3 日在 Google 的演讲：*http://www.youtube.com/watch?v=4XpnKHJAok8*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




控制系统（代码比较和冲突解决）都有莫大的好处。

### 1. 用 diff 命令比较两个文本文件或目录的差异

先来构造两个文件：

| 文件 hello | 文件 world |
| --- | --- |
| 应该杜绝文章中的错别子[①]。 | 应该杜绝文章中的错别字。 |
| 但是无论使用<br>* 全拼，双拼<br>* 还是五笔 | 但是无论使用<br>* 全拼，双拼<br>* 还是五笔 |
| 是人就有可能犯错，软件更是如此。 | 是人就有可能犯错，软件更是如此。 |
| 犯了错，就要扣工资！ | 改正的成本可能会很高。 |
| 改正的成本可能会很高。 | 但是“只要眼球足够多，所有 Bug 都好捉”，这就是开源的哲学之一。 |

对这两个文件执行 diff 命令，并通过输出重定向，将差异保存在 diff.txt 文件中。

```
$ diff -u hello world > diff.txt
```

上面执行 diff 命令的 -u 参数很重要，使得差异输出中带有上下文。打开文件 diff.txt，会看到其中的差异比较结果。为了说明方便，为每一行增添了行号。

```
 1 --- hello       2010-09-21 17:45:33.551610940 +0800
 2 +++ world       2010-09-21 17:44:46.343610465 +0800
 3 @@ -1,4 +1,4 @@
 4 - 应该杜绝文章中的错别子。
 5 + 应该杜绝文章中的错别字。
 6
 7   但是无论使用
 8   * 全拼，双拼
 9 @@ -6,6 +6,7 @@
10
11   是人就有可能犯错，软件更是如此。
12
13 - 犯了错，就要扣工资！
14 -
15   改正的成本可能会很高。
16 +
```

① 此处是故意将“字”写成“子”，以便两个文件进行差异比较。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
17 +但是“只要眼球足够多，所有 Bug 都好捉”，
18 +这就是开源的哲学之一。
```

上面的差异文件，可以这么理解：

- 第 1 行和第 2 行分别记录了原始文件和目标文件的文件名及时间戳。以三个减号（---）开始的行标识的是原始文件，以三个加号（+++）开始的行标识的是目标文件。
- 在比较内容中，以减号（-）开始的行是只出现在原始文件中的行，例如：第 4、13、14 行。
- 在比较内容中，以加号（+）开始的行是只出现在目标文件中的行，例如：第 5 行和 16-18 行。
- 在比较内容中，以空格开始的行，是在原始文件和目标文件中都出现的行，例如：第 6-8、10-12 和第 15 行。这些行是用作差异比较的上下文。
- 第 3-8 行是第一个差异小节。每个差异小节以一行差异定位语句开始。第 3 行就是一条差异定位语句，其前后分别用两个 @ 进行标识。
- 第 3 行定位语句中 -1,4 的含义是：本差异小节的内容相当于原始文件的从第 1 行开始的 4 行。而第 4、6、7、8 行是原始文件中的内容，加起来刚好是 4 行。
- 第 3 行定位语句中 +1,4 的含义是：本差异小节的内容相当于目标文件的从第 1 行开始的 4 行。而第 5、6、7、8 行是目标文件中的内容，加起来刚好是 4 行。
- 因为命令 diff 是用于行比较的，所以即使改正了一个字，也显示为一整行的修改（参见差异文件第 4、5 行）。Git 对 diff 进行了扩展，并且还提供一种逐词比较的差异比较方法，参见本书第 2 篇的第 11.4.4 小节。
- 第 9-18 行是第二个差异小节。第 9 行是一条差异定位语句。
- 第 9 行定位语句中 -6,6 的含义是：本差异小节的内容相当于原始文件的从第 6 行开始的 6 行。第 10-15 行是原始文件中的内容，加起来刚好是 6 行。
- 第 9 行定位语句中 +6,7 的含义是：本差异小节的内容相当于目标文件的从第 6 行开始的 7 行。而第 10-12、15-18 行是目标文件中的内容，加起来刚好是 7 行。

### 2. 命令 patch 相当于 diff 的反向操作

有了 hello 和 diff.txt 文件，可以放心地将 world 文件删除或用 hello 文件将 world 文件覆盖。用下面的命令可以还原 world 文件：

```
$ cp hello world
$ patch world < diff.txt
```

也可以保留 world 和 diff.txt 文件，删除 hello 文件或用 word 文件将 hello 文件覆盖。用下面的命令可以恢复 hello 文件：

```
$ cp world hello
$ patch -R hello < diff.txt
```

命令 diff 和 patch 还可以对目录进行比较操作，这也就是 Linus 在 1991 ～ 2002 年用于维护 Linux 不同版本间差异的办法。在没有版本控制系统的情况下，可以用此命令记录

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




并保存改动前后的差异，还可以将差异文件注入版本控制系统（如果有的话）。

标准的 diff 和 patch 命令存在一个局限，就是不能对二进制文件进行处理。对二进制文件的修改或添加会在差异文件中缺失，进而丢失对二进制文件的改动或添加。Git 对差异文件格式提供了扩展支持，支持二进制文件的比较，解决了这个问题。这一点可以参考本书第 7 篇第 38 章的相关内容。

## 1.2 CVS——开启版本控制大爆发

CVS（Concurrent Versions System）[①]诞生于 1985 年，是由荷兰阿姆斯特丹 VU 大学的 Dick Grune 教授实现的。当时 Dick Grune 和两个学生共同开发一个项目，但是三个人的工作时间无法协调到一起，迫切需要一个记录和协同开发的工具软件。于是 Dick Grune 通过脚本语言对 RCS（一个针对单独文件的版本管理工具）进行封装，设计出有史以来第一个被大规模使用的版本控制工具。Dick 教授的网站上介绍了 CVS 的这段早期历史。[②]

“在 1985 年的一个糟糕的秋日里，我在校汽车站等车回家，脑海里一直纠结着一件事——如何处理 RCS 文件、用户文件（工作区）和 Entries 文件的复杂关系，有的文件可能会缺失、冲突、删除，等等。我的头有些晕了，于是决定画一个大表，将复杂的关联画在其中，看看出来的结果是什么样的……”

1986 年 Dick 通过新闻组发布了 CVS，1989 年 Brian Berliner 用 C 语言将 CVS 进行了重写。

从 CVS 的历史可以看出，CVS 不是设计出来的，而是被实际需要“逼”出来的，因此根据“实用为上”的原则，借用了已有的针对单一文件的版本管理工具 RCS。CVS 采用客户端 / 服务器架构设计，版本库位于服务器端，实际上就是一个 RCS 文件容器。每一个 RCS 文件以“,v”作为文件名后缀，用于保存对应文件的每一次更改历史。RCS 文件中只保留一个版本的完全拷贝，其他历次更改仅将差异存储其中，使得存储变得非常有效率。我在 2008 年设计了一个 SVN 管理后台 pySvnManager[③]，实际上也采用了 RCS 作为 SVN 授权文件的变更记录的“数据库”。

图 1-1 展示了 CVS 版本控制系统的工作原理，可以看到作为 RCS 文件容器的 CVS 版本库和工作区目录结构的一一对应关系。

CVS 这种实现方式的最大好处就是简单。把版本库中任意一个目录拿出来就可以成为另外一个版本库。如果将版本库中的一个 RCS 文件重命名，工作区检出的文件名也会相应地改变。这种低成本的服务器管理模式成为很多 CVS 粉丝至今不愿离开 CVS 的原因。

CVS 的出现让软件工程师认识到了原来还可以这样工作。CVS 成功地为后来的版本控

① *http://www.nongnu.org/cvs/*
② *http://dickgrune.com/Programs/CVS.orig/*
③ *http://pysvnmanager.sourceforge.net*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


制系统确立了标准，像提交说明（commit log）、检入（checkin）、检出（checkout）、里程碑（tag）、分支（branch）等概念在 CVS 中早就已经确立。CVS 的命令行格式也被后来的版本控制系统竞相模仿。

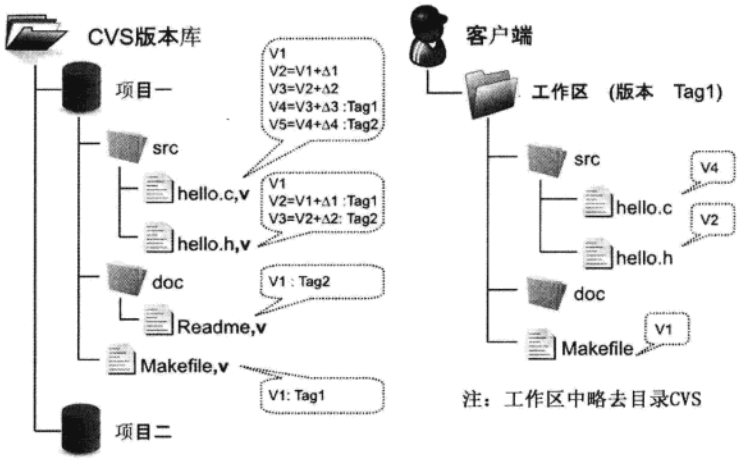



图 1-1 CVS 版本控制系统示意图

在 2001 年，我正为使用 CVS 激动不已的时候，公司领导要求采用和美国研发部门同样的版本控制解决方案。于是，我的项目组率先进行了从 CVS 到该商业版本控制工具的迁移[①]。虽然商业版本控制工具有更漂亮的界面及更好的产品整合性，但是就版本控制本身而言，商业版本控制工具存在着如下缺陷。

- 采用黑盒子式的版本库设计。让人捉摸不透的版本库设计，最主要的目的可能就是阻止用户再迁移到其他平台。
- 缺乏版本库整理工具。如果有一个文件（如记录核弹引爆密码的文件）检入到版本库中，就无法再彻底移除它。
- 商业版本控制工具很难为个人提供版本控制解决方案，除非个人愿意花费高昂的许可证费用。
- 商业版本控制工具注定是小众软件，新员工的培训成本不可忽视。

而上述商业版本控制系统的缺点恰恰是 CVS 及其他开源版本控制系统的优点。但在经历了最初的成功之后，CVS 也尽显疲态：

- 服务器端松散的 RCS 文件导致在建立里程碑或分支时效率不高，服务器端文件越多，速度越慢。
- 分支和里程碑不可见，因为它们被分散地记录在服务器端的各个 RCS 文件中。
- 合并困难重重，因为缺乏对合并的追踪，从而导致重复合并，引发严重冲突。

① 于是就有了这篇文章：*http://www.worldhello.net/doc/cvs_vs_starteam*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




- 缺乏对原子提交的支持，会导致客户端向服务器端提交不完整的数据。
- 不能优化存储内容相同但文件名不同的文件，因为在服务器端每个文件都是单独进行差异存储的。
- 不能对文件和目录的重命名进行版本控制，虽然直接在服务器端修改 RCS 文件名可以让改名后的文件保存历史，但是这样做实际上会破坏历史。

CVS 的成功导致了版本控制系统的大爆发，各式各样的版本控制系统如雨后春笋般诞生了。新的版本控制系统或多或少地解决了 CVS 版本控制系统存在的问题。在这些版本控制系统中，最典型的就是 Subversion（SVN）。

## 1.3 SVN——集中式版本控制集大成者

Subversion[①]，由于其命令行工具名为 svn，因此通常被简称为 SVN。SVN 由 CollabNet 公司于 2000 年资助并开始开发，目的是创建一个更好用的版本控制系统以取代 CVS。SVN 的前期开发使用 CVS 做版本控制，到了 2001 年，SVN 已经可以用于自己的版本控制了[②]。

我开始真正关注 SVN 是在 2005 年，那时 SVN 正经历着后端存储上的变革，即从 BDB（简单的关系型数据库）到 FSFS（文件数据库）的转变。相对于 BDB 而言，FSFS 具有稳定、免维护和实现的可视性高等优点，于是我马上就被 SVN 吸引了。图 1-2 展示了 SVN 版本控制系统的工作原理。

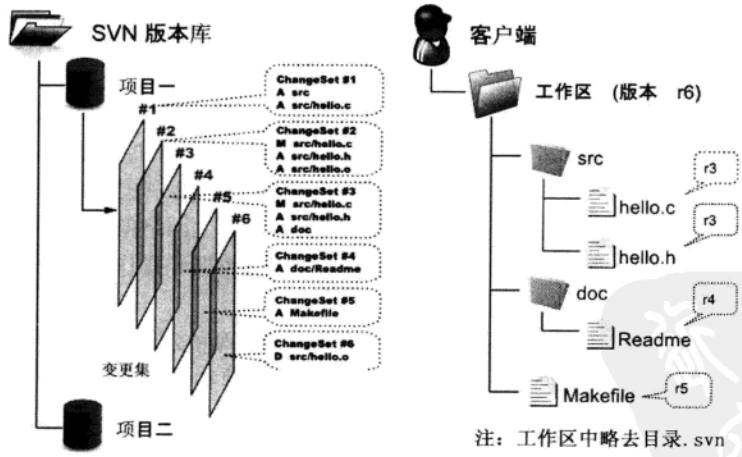



图 1-2 SVN 版本控制系统示意图

SVN 的每一次提交，都会在服务器端的 db/revs 和 db/revprops 目录下各创建一个

① *http://subversion.apache.org/*

② *http://svnbook.red-bean.com/en/1.5/svn.intro.whatis.html#svn.intro.history*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


以顺序数字编号命名的文件。其中，db/revs 目录下的文件（即变更集文件）记录了与上一个提交之间的差异（字母 A 表示新增，M 表示修改，D 表示删除）。在 db/revprops 目录下的同名文件（没有在图 1-2 中体现）则保存着提交日志、作者、提交时间等信息。这样设计的好处有：

- 拥有全局版本号。每提交一次，SVN 的版本号就会自动加一。这为 SVN 的使用提供了极大的便利。回想 CVS 时代，每个文件都拥有各自独立的版本号（RCS 版本号），要想获得全局版本号，只能通过手工不断地建立里程碑来实现。
- 实现了原子提交。SVN 不会像 CVS 那样出现文件的部分内容被提交而其余的内容没有被提交的情况。
- 文件名不受限制。因为服务器端不再需要建立和客户端文件相似的文件名，于是，文件的命名就不再受服务器操作系统的字符集和大小写的限制。
- 文件和目录重命名也得到了支持。

SVN 最具有特色的功能是轻量级拷贝，例如将目录 trunk 拷贝为 branches/v1.x 只相当于在 db/revs 目录中的变更集文件中用特定的语法做了一下标注，无须真正的文件拷贝。SVN 使用轻量级拷贝的功能，轻松地解决了 CVS 存在的里程碑和分支的创建速度慢又不可见的问题，使用 SVN 创建里程碑和分支只在眨眼之间。

SVN 在版本库授权上也有改进，不再像 CVS 那样依赖操作系统本身对版本库目录和文件进行授权，而是采用授权文件的方式来实现。

SVN 还有一个创举，就是在工作区跟踪目录下（.svn 目录）为当前目录中的每一个文件都保存一份冗余的原始拷贝。这样做的好处是部分命令不再需要网络连接，例如文件修改的差异比较，以及错误更改的回退等。

正是由于这些闪亮的功能特性，才使得 SVN 在 CVS 之后诞生的诸多版本控制系统中脱颖而出，成为开源社区一时的新宠，也成为当时各个企业进行版本控制的最佳选择之一。

但是，相对于 CVS，SVN 在本质上并没有突破，都属于集中式版本控制系统。即一个项目只有唯一的一个版本库与之对应，所有的项目成员都通过网络向该服务器进行提交。这样的设计除了容易出现单点故障以外，在查看日志和提交数据等操作时的延迟，会让基于广域网协同工作的团队抓狂。

除了集中式版本控制系统固有的问题外，SVN 的里程碑和分支的设计也被证明是一个错误，虽然这个错误的设计使得 SVN 拥有了快速创建里程碑和分支的能力，但是这个错误的设计也导致了如下的更多问题。

- 项目文件在版本库中必须按照一定的目录结构进行部署，否则就可能无法建立里程碑和分支。

  我在项目咨询过程中见过很多团队，直接在版本库的根目录下创建项目文件。这样的版本库布局，在需要创建里程碑和分支时就无从下手了，因为根目录是不能拷贝到子目录中的。所以 SVN 的用户在创建版本库时必须遵守一个古怪的约定：先创建三个顶级目录 /trunk、/tags 和 /branches。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




- 创建里程碑和分支会破坏精心设计的授权。
  SVN 的授权是基于目录的，分支和里程碑也被视为目录（和其他目录没有分别）。因此每次创建分支或里程碑时，就要将针对 /trunk 目录及其子目录的授权在新建的分支或里程碑上重建。随着分支和里程碑数量的增多，授权愈加复杂，维护也愈加困难。
- 分支太随意从而导致混乱。SVN 的分支创建非常随意：可以基于 /trunk 目录创建分支，也可以基于其他任何目录创建分支，因此 SVN 很难画出一个有意义的分支图。再加上一次提交可以同时包含针对不同分支的文件变更，使得事情变得更糟。
- 虽然 SVN 在 1.5 版之后拥有了合并追踪功能，但这个功能会因为混乱的分支管理而被抵消。

2009 年年底，SVN 由 CollabNet 公司交由 Apache 社区管理，至此 SVN 成为了 Apache 组织的一个子项目[①]。这对 SVN 到底意味着什么？是开发的停滞？还是新的开始？结果如何我们将拭目以待。

## 1.4 Git——Linus 的第二个伟大作品

Linux 之父 Linus 是坚定的 CVS 反对者，他也同样地反对 SVN。这就是为什么在 1991-2002 这十余年间，Linus 宁可以手工修补文件的方式维护代码，也迟迟不愿使用 CVS 的原因。我想在当时要想劝说 Linus 使用 CVS 只有一个办法：把 CVS 服务器请进 Linus 的卧室，并对外配以千兆带宽。

2002 年至 2005 年，Linus 顶着开源社区精英们口诛笔伐的压力，选择了一个商业版本控制系统 BitKeeper 作为 Linux 内核的代码管理工具[②]。BitKeeper 不同于 CVS 和 SVN 等集中式版本控制工具，而是一款分布式版本控制工具。

分布式版本控制系统最大的反传统之处在于，可以不需要集中式的版本库，每个人都工作在通过克隆建立的本地版本库中。也就是说每个人都拥有一个完整的版本库，查看提交日志、提交、创建里程碑和分支、合并分支、回退等所有操作都直接在本地完成而不需要网络连接。每个人都是本地版本库的主人，不再有谁能提交谁不能提交的限制，加上多样的协同工作模型（版本库间推送、拉回，以及补丁文件传送等）让开源项目的参与度有爆发式增长。

2005 年发生的一件事最终导致了 Git 的诞生。在 2005 年 4 月，Andrew Tridgell（即大名鼎鼎的 Samba 的作者）试图对 BitKeeper 进行反向工程，以开发一个能与 BitKeeper 交互的开源工具。这激怒了 BitKeeper 软件的所有者 BitMover 公司，要求收回对 Linux 社区免费使用 BitKeeper 的授权[③]。迫不得已，Linus 选择了自己开发一个分布式版本控制工具以替代 BitKeeper。以下是 Git 诞生过程中的大事记[④]：

① *http://en.wikipedia.org/wiki/Apache_Subversion*
② *http://en.wikipedia.org/wiki/BitKeeper*
③ *http://en.wikipedia.org/wiki/Andrew_Tridgell*
④ *http://en.wikipedia.org/wiki/Git_%28software%29*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 2005 年 4 月 3 日，开始开发 Git。
- 2005 年 4 月 6 日，项目发布。
- 2005 年 4 月 7 日，Git 就可以作为自身的版本控制工具了。
- 2005 年 4 月 18 日，发生第一个多分支合并。
- 2005 年 4 月 29 日，Git 的性能就已经达到了 Linus 的预期。
- 2005 年 6 月 16 日，Linux 内核 2.6.12 发布，那时 Git 已经在维护 Linux 核心的源代码了。

Linus 以一个文件系统专家和内核设计者的视角对 Git 进行了设计，其独特的设计让 Git 拥有非凡的性能和最为优化的存储能力。完成原型设计后，在 2005 年 7 月 26 日，Linus 功成身退，将 Git 的维护交给另外一个 Git 的主要贡献者 Junio C Hamano[①]，直到现在。

最初的 Git 除了一些核心命令以外，其他的都用脚本语言开发，而且每个功能都作为一条独立的命令，例如克隆操作用 `git-clone` ，提交操作用 `git-commit` 。这导致 Git 拥有庞大的命令集，使用习惯也和其他版本控制系统格格不入。随着 Git 的开发者和使用者的增加，Git 也在逐渐演变，例如到 1.5.4 版本时，将一百多个独立的命令封装为一个 `git` 命令，使它看起来更像是一个独立的工具，也使 Git 更贴近于普通用户的使用习惯。

经过短短几年的发展，众多的开源项目都纷纷从 SVN 或其他版本控制系统迁移到 Git。虽然版本控制系统的迁移过程是痛苦的，但是因为迁移到 Git 会带来开发效率的极大提升，以及巨大的效益，所以很快就会忘记迁移的痛苦过程，而且很快就会适应新的工作模式。在 Git 的官方网站上列出了几个使用 Git 的重量级项目，每一个都是人们耳熟能详的，除了 Git 和 Linux 内核外，还有 Perl、Eclipse、Gnome、KDE、Qt、Ruby on Rails、Android、PostgreSQL、Debian、X.org，当然还有 GitHub 的上百万个项目。

Git 虽然是在 Linux 下开发的，但现在已经可以跨平台运行在所有主流的操作系统上，包括 Linux、Mac OS X 和 Windows 等。可以说每一个使用计算机的用户都可以分享 Git 带来的便利和快乐。



① *http://marc.info/?l=git&m=112243466603239*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 2 章　爱上 Git 的理由

本章将通过一些典型应用展示 Git 作为版本控制系统的独特用法，不熟悉版本控制系统的读者可以通过这些示例对版本控制拥有感性的认识。如果是有经验的读者，示例中 Git 和 SVN 的对照可以让您体会到 Git 的神奇和强大。本章将列举 Git 的一些闪亮特性，期待能够让您爱上 Git。

## 2.1　每日工作备份

当我开始撰写本书时才明白写书真的是一个辛苦活。如何让辛苦的工作不会因为笔记本硬盘的意外损坏而丢失？如何防范灾害而不让一个篮子里的鸡蛋都毁于一旦？下面就介绍一下我在写本书时是如何使用 Git 进行文稿备份的，请看图 2-1。

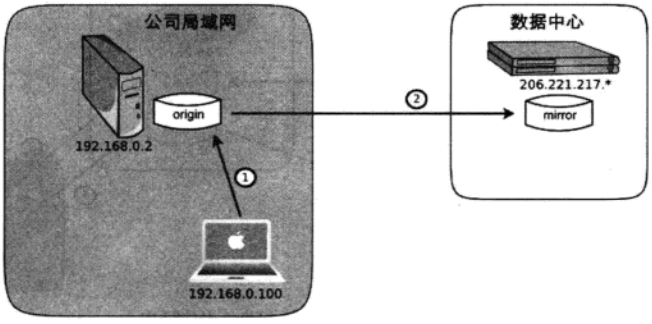



图 2-1　利用 Git 做数据的备份

如图 2-1 所示，我的笔记本在公司局域网里的 IP 地址是 192.168.0.100，公司的 Git 服务器的 IP 地址是 192.168.0.2。公司使用动态 IP 上网因而没有固定的外网 IP，但是公司在数据中心有托管服务器，拥有固定的 IP 地址，其中一台服务器用作 Git 服务器镜像。

我的写书习惯大概是这样：在写完一个小节或是画完一张图后，我会执行下面的命令提交一次。每一天平均提交 3-5 次。提交是在本地完成的，因此在图中没有表示出来。

```
$ git add -u     # 如果创建了新文件，可以执行 git add -i 命令。
$ git commit
```

下班后，我会执行一次推送操作，将我在本地 Git 版本库中的提交同步到公司的 Git 服务器上。相当于图 2-1 中的步骤① 。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git push
```

因为公司的Git服务器和异地数据中心的Git服务器建立了镜像，所以每当我向公司内网服务器推送的时候，就会自动触发从内网服务器到外网Git服务器的镜像操作。这相当于图2-1中的步骤②，步骤②是自动执行的，无须人工干预。图2-1中标记为mirror的版本库就是Git镜像版本库，该版本库只向用户提供只读访问服务，而不能对其进行写操作（推送）。

从图2-1中可以看出，我的每日工作保存有三个拷贝，一个在笔记本中，一个在公司内网的服务器上，还有一个在外网的镜像版本库中。鸡蛋分别装在了三个篮子里。

关于如何架设可以实时镜像的Git服务器，会在本书第5篇第30章中详细介绍。

## 2.2 异地协同工作

为了能够加快写书的进度，熬夜是必须的，这就出现了在公司和家里两地工作同步的问题。图2-2用于说明我是如何解决两地工作同步问题的。

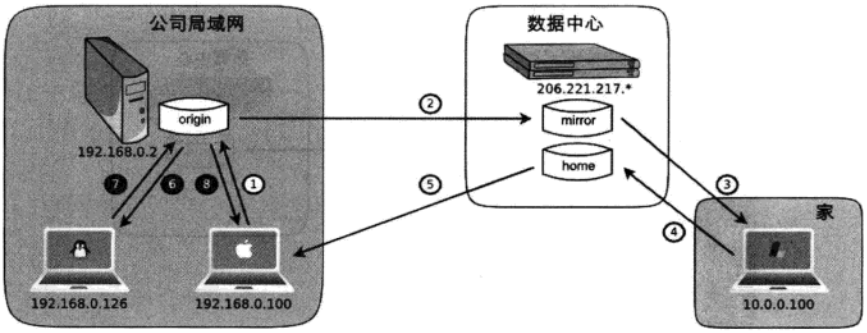



图2-2 利用Git实现异地工作协同

我在家里的电脑IP地址是10.0.0.100（家里也有一个小局域网）。如果在家里有时间工作的话，首先要做的就是图2-2中步骤③的操作：将mirror版本库中的数据同步到本地。只需要一条命令就好了：

```
$ git pull mirror master
```

然后在家里的电脑上继续编写书稿并提交。当准备完成一天的工作后，就执行下面的命令，相当于图2-2中步骤④的操作：将在家中的提交推送到标记为home的版本库中。

```
$ git push home
```

为什么还要再引入另外一个名为home的版本库呢？使用mirror版本库不好么？不要忘了mirror版本库只是一个镜像库，不能提供写操作。

当一早到公司，开始动笔写书之前，先要执行图2-2中步骤⑤的操作，从home版本库

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




将家里做的提交同步到公司的电脑中。

```
$ git pull home master
```

公司的小崔是我这本书的忠实读者，每当有新章节完成，他都会执行图 2-2 中步骤⑥的工作，从公司内网服务器获取我最新的文稿。

```
$ git pull
```

一旦发现文字错误，小崔会直接在文稿中修改，然后推送到公司的服务器上（图 2-2 中步骤⑦）。当然他的这个推送也会自动同步到外网的 mirror 版本库。

```
$ git push
```

而我只要执行 git pull 操作就可以获得小崔对文稿的修订（图 2-2 中的步骤⑧）。采用这种工作方式，文稿竟然分布在 5 台电脑上拥有 6 个拷贝，真可谓狡兔三窟。不，比狡兔还要多三窟。

在本节中，出现在 Git 命令中的 mirror 和 home 是和工作区关联的远程版本库。关于如何注册和使用远程版本库，请参见本书第 3 篇第 19 章中的内容。

## 2.3 现场版本控制

所谓现场版本控制，就是在客户现场或在产品部署的现场进行源代码的修改，并在修改过程中进行版本控制，以便在完成修改后能够将修改结果甚至修改过程一并带走，并能够将修改结果合并至项目对应的代码库中。

### 1. SVN 的解决方案

如果使用 SVN 进行版本控制，首先要将服务器上部署的产品代码目录变成 SVN 工作区，这个过程不仅不简单，而且会显得很繁琐，最后将改动结果导出也非常不方便，具体操作过程如下。

（1）在其他位置建立一个 SVN 版本库。

```
$ svnadmin create /path/to/repos/project1
```

（2）在需要版本控制的目录下检出刚刚建立的空版本库。

```
$ svn checkout file:///path/to/repos/project1 .
```

（3）执行添加文件操作，然后执行提交操作。这个提交将是版本库中编号为 1 的提交。

```
$ svn add *
$ svn ci -m "initialized"
```

（4）开始在工作区中修改文件并提交。

```
$ svn ci
```

（5）如果对修改结果满意，可以通过创建补丁文件的方式将工作成果保存并带走。但是

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


SVN 很难针对每次提交逐一创建补丁，一般用下面的命令与最早的提交进行比较，以创建出一个大补丁文件。

```
$ svn diff -r1 > hacks.patch
```

上面用 SVN 将工作成果导出的过程存在一个致命的缺陷，就是 SVN 的补丁文件不支持二进制文件，因此采用补丁文件的方式有可能丢失数据，如新增或修改的图形文件会丢失。更为稳妥但也更为复杂的方式可能要用到 svnadmin 命令将版本库导出。命令如下：

```
$ svnadmin dump --incremental -r2:HEAD \
 /path/to/repos/project1/ > hacks.dump
```

将 svnadmin 命令创建的导出文件恢复到版本库中也非常具有挑战性，这里就不再详细说明了。还是来看看 Git 在这种情况下的表现吧。

### 2. Git 的解决方案

与 SVN 将产品部署目录转化为 SVN 的工作区相比，Git 要更为简单，而且使用 Git 导出提交历史也更为简单和实用，具体操作过程如下。

（1）创建现场版本库。直接在需要版本控制的目录下执行 Git 版本库初始化命令。

```
$ git init
```

（2）添加文件并提交。

```
$ git add -A
$ git commit -m "initialized"
```

（3）为初始提交建立一个里程碑：“v1”。

```
$ git tag v1
```

（4）开始在工作区中工作——修改文件并提交。

```
$ git commit -a
```

（5）当对修改结果满意并想将工作成果保存带走时，可以通过下面的命令将从 v1 开始的历次提交逐一导出为补丁文件。转换的补丁文件都包含一个数字前缀，并提取提交日志信息作为文件名，而且补丁文件还提供对二进制文件的支持。下面命令的输出摘自本书第 3 篇·第 20 章中的实例。

```
$ git format-patch v1..HEAD
0001-Fix-typo-help-to-help.patch
0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
```

（6）通过邮件将补丁文件发出。当然，也可以通过其他方式将补丁文件带走。

```
$ git send-email *.patch
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





Git 创建的补丁文件使用了 Git 扩展格式，因此在导入时为了避免数据遗漏，要使用 Git 提供的命令而不能使用 GNU 的 patch 命令。即使要导入的不是 Git 版本库，也可以使用 Git 命令，具体操作请参见本书第 7 篇第 38 章中的相关内容。

## 2.4 避免引入辅助目录

很多版本控制系统都要在工作区中引入辅助目录或文件，如 SVN 要在工作区的每一个子目录下都创建 .svn 目录，CVS 要在工作区的每一个子目录下都创建 CVS 目录。

这些辅助目录如果出现在服务器上，尤其是 Web 服务器上是非常危险的，因为这些辅助目录下的 Entries 文件会暴露出目录下的文件列表，让管理员精心配置的禁止目录浏览的努力全部白费。

还有，SVN 的 .svn 辅助目录下还存在文件的原始拷贝，在文件搜索时结果会加倍。如果你曾经在 SVN 的工作区用过 grep 命令进行内容查找，就会明白我指的是什么。

Git 没有这个问题，不会在子目录下引入讨厌的辅助目录或文件（.gitignore 和 .gitattributes 文件不算）。当然，Git 还是要在工作区的顶级目录下创建名为 .git 的目录（版本库目录），不过，如果你认为唯一的一个 .git 目录也过于碍眼，可以将其放到工作区之外的任意目录。一旦这么做了，在执行 Git 命令时，就要通过命令行（--git-dir）或环境变量 GIT_DIR 为工作区指定版本库目录，甚至还要指定工作区目录。

Git 还专门提供了一个 git grep 命令，这样在工作区根目录下执行查找时，目录 .git 也不会对搜索造成影响。

关于辅助目录的详细讨论请参见本书第 2 篇第 4.2 节中的内容。

## 2.5 重写提交说明

很多人可能像我一样，在敲下回车之后才发现提交说明中有错别字，或忘记了写关联的 Bug ID，此时就需要重写提交说明。

### 1. SVN 的解决方案

在默认情况下，SVN 的提交说明是禁止更改的，因为 SVN 的提交说明属于不受版本控制的属性，一旦修改就不可恢复。**我建议 SVN 的管理员只有在配置了版本库更改的外发邮件通知之后，再开放提交说明更改的功能。**我发布于 SourceForge 上的 pySvnManager 项目提供了 SVN 版本库图形化的钩子管理，会简化管理员的配置工作。

即使 SVN 管理员启用了允许更改提交说明的设置，修改提交说明也还是挺复杂的，看看下面的命令：

```
$ svn ps --revprop -r <REV> svn:log "new log message..."
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 2. Git 的解决方案

Git 修改提交说明很简单，而且提交说明的修改也是被追踪的。Git 修改最新提交的提交说明最为简单，使用一条名为修补提交的命令即可。

```
$ git commit --amend
```

这个命令如果不带 -m 参数，会进入提交说明编辑界面，修改原来的提交说明，直到满意为止。

如果要修改某个历史提交的提交说明，Git 也可以实现，但要用到另外一个命令：变基命令。例如，要修改 <commit-id> 所标识的提交的提交说明，执行下面的命令，并在弹出的变基索引文件中修改相应提交前面的动作的关键字。

```
$ git rebase -i <commit-id>^
```

关于如何使用交互式变基操作更改历史提交的提交说明，请参见本书第 2 篇第 12 章中的内容。

## 2.6 想吃后悔药

假如提交的数据中不小心包含了一个不应该检入的虚拟机文件——大约有 1 个 GB！这时候，您会多么希望这个世界上有后悔药卖啊。

### 1. SVN 的解决方案

SVN 遇到这个问题该怎么办呢？删除错误加入的大文件后再提交，这样的操作是不能解决问题的。虽然表面上去掉了这个文件，但是它依然存在于历史中。

管理员可能是受影响最大的人，因为他要负责管理服务器的磁盘空间占用及版本库的备份。实际上这个问题也只有管理员才能解决，所以你必须向管理员坦白，让他帮你在服务器端彻底删除错误引入的大文件。我要告诉你的是，对于管理员，这并不是一个简单的活。

（1）如果 SVN 管理员没有历史备份，只能重新用 svnadmin dump 导出整个版本库。

（2）再用 svndumpfilter 命令过滤掉不应检入的大文件。

（3）然后用 svnadmin load 重建版本库。

上面的操作描述中省略了一些窍门，如果要把窍门讲清楚，这本书就不是讲 Git，而是讲 SVN 了，故本书中不进行深入探讨。

### 2. Git 的解决方案

如果你用 Git，一切就会变得非常简单，而且你也不必去乞求管理员，因为使用 Git，每个人都是管理员。

如果是最新的提交引入了不该提交的大文件：winxp.img，操作起来会非常简单，还是用修补提交命令。

```
$ git rm --cached winxp.img
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
$ git commit --amend
```

如果是历史版本，例如是在 <commit-id> 所标识的提交中引入的文件，则需要使用变基操作。

```
$ git rebase -i <commit-id>^
```

执行交互式变基操作抛弃历史提交，版本库还不能立即瘦身，具体原因和解决方案请参见本书第 2 篇第 14 章中的内容。除了使用变基操作，Git 还有更多的武器可以实现版本库的整理操作，具体请参见本书第 6 篇第 35.4 节的内容。

## 2.7 更好用的提交列表

正确的版本控制系统的使用方法是，一次提交只干一件事：或是完成了一个新功能，或是修改了一个 Bug、或是写完了一节的内容，或是添加了一幅图片，就执行一次提交。不要在下班时才想起来要提交，那样的话版本控制系统就被降格为文件备份系统了。

但有时在同一个工作区中可能要同时做两件事情：一个是尚未完成的新功能，另外一个是解决刚刚发现的 Bug。很多版本控制系统没有提交列表的概念，要么需要在命令行中指定要提交的文件，要么默认把所有修改内容全部提交，破坏了一次提交只干一件事的原则。

### 1. SVN 的解决方案

SVN 1.5 开始提供变更列表（change list）的功能，是通过引入一个新的命令 `svn changelist` 来实现的。但是我从来就没有真正用过，因为：

- 定义一个变更列表太麻烦。例如，没有一个快捷命令将当前所有改动的文件加入列表，也没有快捷命令将工作区中的新文件全部加入列表。
- 一个文件不能同时属于两个变更列表。两次变更不许有文件交叉，这样的限制不合理。
- 变更列表是一次性的，提交之后自动消失。这样的设计没有问题，但是相比定义列表时的繁琐，以及提交时必须指定列表的繁琐，使用变更列表未免得不偿失。
- 因为 Subversion 的提交不能撤销，如果在提交时忘了提供变更列表名称以针对特定的变更列表进行提交，错误的提交内容将无法补救。

总之，SVN 的变更列表尚不如鸡肋，食之无味，弃之不可惜。

### 2. Git 的解决方案

Git 通过提交暂存区实现对提交内容的定制，非常完美地实现了对工作区的修改内容进行筛选提交：

- 执行 `git add` 命令将修改内容加入提交暂存区。执行 `git add -u` 命令可以将所有修改过的文件加入暂存区，执行 `git add -A` 命令可以将本地删除文件和新增文件都登记到提交暂存区，执行 `git add -p` 命令甚至可以对一个文件内的修改进行有选择性的添加。
- 一个修改后的文件被登记到提交暂存区后，可以继续修改，继续修改的内容不会被提

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


交，除非对此文件再执行一次 git add 命令。即一个修改的文件可以拥有两个版本，在提交暂存区中有一个版本，在工作区中有另外一个版本。

- 执行 git commit 命令提交，无须设定变更列表，直接将登记在暂存区中的内容提交。
- Git 支持撤销提交，而且可以撤销任意多次。

只要使用 Git，就会时刻与隐形的提交列表打交道。本书第 2 篇第 5 章会详细介绍 Git 的这一特性，相信你会爱上 Git 的这个特性。

## 2.8 更好的差异比较

Git 对差异比较进行了扩展，支持对二进制文件的差异比较，这是对 GNU 的 diff 和 patch 命令的重要补充。而且 Git 的差异比较除了支持基于行的差异比较外，还支持在一行内逐字比较的方式，当向 git diff 命令传递 --word-diff 参数时，就会进行逐字比较。

上面介绍了工作区的文件修改可能会有两个不同的版本：一个在提交暂存区，一个在工作区。因此，在执行 git diff 命令时会遇到令 Git 新手费解的现象。

- 修改后的文件在执行 git diff 命令时会看到修改造成的差异。
- 修改后的文件通过 git add 命令提交到暂存区后，再执行 git diff 命令将看不到该文件的差异。
- 继续对此文件进行修改，再次执行 git diff 命令会看到新的修改显示在差异中，而看不到旧的修改。
- 执行 git diff --cached 命令才可以看到添加到暂存区中的文件所做出的修改。

Git 差异比较的命令充满了魔法，本书第 5.3 节会带您破解 Git 的 diff 魔法。一旦您习惯了，就会非常喜欢 git diff 的这个行为。

## 2.9 工作进度保存

如果在工作区的修改尚未完成时忽然有一个紧急的任务，需要从一个干净的工作区开始新的工作，或要切换到别的分支进行工作，那么如何保存当前尚未完成的工作进度呢？

### 1. SVN 的解决方案

如果版本库规模不大，最好重新检出一个新的工作区，在新的工作区进行工作。否则，可以执行下面的操作。

```
$ svn diff > /path/to/saved/patch.file
$ svn revert -R
$ svn switch <new_branch>
```

在新的分支中工作完毕后，再切换回当前分支，将补丁文件重新应用到工作区。

```
$ svn switch <original_branch>
$ patch -p1 < /path/to/saved/patch.file
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




但是切记 SVN 的补丁文件不支持二进制文件，这种操作方法可能会丢失对二进制文件的更改！

2. Git 的解决方案

Git 提供了一个可以保存和恢复工作进度的命令 `git stash`。这个命令非常方便地解决了这个难题。

在切换到新的工作分支之前执行 `git stash` 保存工作进度，工作区就会变得非常干净，然后就可以切换到新的分支中了。

```
$ git stash
$ git checkout <new_branch>
```

新的工作分支修改完毕后，再切换回当前分支，调用 `git stash pop` 命令则可恢复之前保存的工作进度。

```
$ git checkout <orignal_branch>
$ git stash pop
```

本书第 2 篇第 9 章会为您揭开 `git stash` 命令的奥秘。

## 2.10 代理 SVN 提交实现移动式办公

使用像 SVN 一样的集中式版本控制系统，要求使用者和版本控制服务器之间要有网络连接，如果因为出差在外或在家办公访问不到版本控制服务器就无法提交。Git 属于分布式版本控制系统，不存在这样的问题。

当版本控制服务器无法实现从 SVN 到 Git 的迁移时，仍然可以使用 Git 进行工作。在这种情况下，Git 作为客户端来操作 SVN 服务器，实现在移动办公状态下的版本提交（当然是在本地 Git 库中提交）。当能够连通 SVN 服务器时，一次性将移动办公状态下的本地提交同步到 SVN 服务器。整个过程对于 SVN 来说是透明的，没有人知道你是使用 Git 在进行提交。

使用 Git 来操作 SVN 版本控制服务器的一般工作流程为：

（1）访问 SVN 服务器，将 SVN 版本库克隆为一个本地的 Git 库，一个货真价实的 Git 库，不过其中包含针对 SVN 的扩展。

```
$ git svn clone <svn_repos_url>
```

（2）使用 Git 命令操作本地克隆的版本库，例如提交就使用 `git commit` 命令。

（3）当能够通过网络连接到 SVN 服务器，并想将本地提交同步到 SVN 服务器时，先获取 SVN 服务器上最新的提交，然后执行变基操作，最后再将本地提交推送给 SVN 服务器。

```
$ git svn fetch
$ git svn rebase
$ git svn dcommit
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


本书第 4 篇第 26 章会详细介绍这一话题。

## 2.11 无处不在的分页器

虽然拥有图形化的客户端，但 Git 的主要操作还是以命令行的方式完成的。使用命令行方式的好处一个是快，另外一个是可以防止鼠标手的出现。Git 的命令行加入了大量的人性化设计，包括命令补全、彩色字符输出[①]等，不过最具特色的还是无处不在的分页器。

在操作其他版本控制系统的命令行时，如果命令的输出超过了一屏，为了能够逐屏显示，需要在命令的后面加上一个管道符号将输出交给一个分页器。例如：

```
$ svn log | less
```

而 Git 则不用如此麻烦，因为每个 Git 命令自动带有一个分页器，默认使用 less 命令（less -FRSX）进行分页。当一屏显示不下时启动分页器，这个分页器支持带颜色的字符输出，对于太长的行则采用截断方式处理。因为 less 分页器在翻屏时使用了 vi 风格的热键，如果您不熟悉 vi 的话，可能会遇到麻烦。下面是分页器中常用的热键：

- 字母 q：退出分页器。
- 字母 h：显示分页器帮助。
- 按空格下翻一页，按字母 b 上翻一页。
- 字母 d 和 u：分别代表向下翻动半页和向上翻动半页。
- 字母 j 和 k：分别代表向上翻一行和向下翻一行。
- 如果行太长被截断，可以用左箭头和右箭头使窗口内容左右滚动。
- 输入 /pattern：向下寻找和 pattern 匹配的内容。
- 输入 ?pattern：向上寻找和 pattern 匹配的内容。
- 字母 n 或 N：代表向前或向后继续寻找。
- 字母 g：跳到第一行；字母 G：跳到最后一行；输入数字再加字母 g：则跳转到对应的行。
- 输入 !<command>：可以执行 Shell 命令。

如果不习惯分页器的长行截断模式而希望能够自动换行，可以通过设置 LESS 环境变量来实现。LESS 环境变量设置如下：

```
$ export LESS=FRX
```

或者使用 Git 的方式，通过定义 Git 配置变量来改变分页器的默认行为。例如设置 core.pager 配置变量如下：

```
$ git config --global core.pager 'less -+$LESS -FRX'
```

① 须通过 Git 配置变量启用，如运行命令：git config --global color.ui true

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



## 2.12 快

您有项目托管在 sourceforge.net 的 CVS 或 SVN 服务器上么？是否会因为公司的 SVN 服务器部署在另外一个城市而需要经过互联网才能访问？

使用传统的集中式版本控制服务器时，如果遇到上面的情况，而且网络带宽没有保证，那么使用起来时一定会因为速度慢而让人痛苦不堪。Git 作为分布式版本控制系统彻底解决了这个问题，几乎所有的操作都在本地进行，而且速度还不是一般的快。

还有很多其他的分布式版本控制系统，如 Hg 和 Bazaar 等，与这些分布式版本控制系统相比，Git 在速度上也有优势，这得益于 Git 独特的版本库设计。第 2 篇的相关章节会向您展示 Git 独特的版本库设计。

其他很多的版本控制系统，当输入检出、更新或克隆等命令后，只能双手合十，然后望眼欲穿，因为整个操作过程不知道什么时候才能够完成。而 Git 在版本库克隆及与版本库同步的时候，能够实时地显示完成的进度，这不但是非常人性化的设计，更体现了 Git 的智能。Git 的智能协议源自于会话过程中在客户端和服务器端各自启用了一个会话的角色，用于按需传输和获取进度。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 3 章 Git 的安装和使用

Git 源自 Linux，现在已经可以部署在所有的主流平台之上，包括 Linux、Mac OS X 和 Windows。我们在开始 Git 之旅之前，首先要做的就是安装 Git。

## 3.1 在 Linux 下安装和使用 Git

Git 诞生于 Linux 平台并作为版本控制系统率先服务于 Linux 内核，因此在 Linux 上安装 Git 是非常方便的。可以通过两种不同的方式在 Linux 上安装 Git：一种方法是通过 Linux 发行版的包管理器安装已经编译好的二进制格式的 Git 软件包，另外一种方式就是从 Git 源码开始安装。

### 3.1.1 包管理器方式安装

用 Linux 发行版的包管理器安装 Git 最为简单，而且会自动配置好命令补齐等功能，但安装的 Git 可能不是最新的版本。还有一点要注意，Git 软件包在有的 Linux 发行版中可能不叫 git，而叫 git-core。这是因为有一款名为 GNU 交互工具[①]（GNU Interactive Tools）的 GNU 软件，在 Git 之前就在一些 Linux 发行版（Deian lenny）中占用了 git 这个名称。为了以示区分，作为版本控制系统的 Git，其软件包在这些平台就被命名为 git-core。不过，因为作为版本控制系统的 Git 太有名了，所以一些 Linux 发行版在最新的版本中将 GNU Interactive Tools 软件包的名称由 git 改为了 gnuit，将 git-core 改为了 git。因此在下面介绍的在不同的 Linux 发行版中安装 Git 时，会看到有 git 和 git-core 两个不同的名称。

- Ubuntu 10.10（maverick）或更新的版本、Debian（squeeze）或更新的版本：

```
$ sudo aptitude install git
$ sudo aptitude install git-doc git-svn git-email git-gui gitk
```

其中 git 软件包包含了大部分 Git 命令，是必装的软件包。

软件包 git-svn、git-email、git-gui、gitk 本来也是 Git 软件包的一部分，但是因为有着不一样的软件包依赖（如更多的 perl 模组和 tk 等），所以单独作为软件包发布。

软件包 git-doc 则包含了 Git 的 HTML 格式文档，可以选择安装。如果安装了 Git 的 HTML 格式的文档，则可以通过执行 `git help -w <sub-command>` 命令来自动用 Web 浏览器打开相关子命令 `<sub-command>` 的 HTML 帮助。

- Ubuntu 10.04（lucid）或更老的版本、Debian（lenny）或更老的版本：

① *http://www.gnu.org/software/gnuit/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




在老版本的 Debian 中，软件包 git 实际上是指 GNU Interactive Tools，而非作为版本控制系统的 Git。作为版本控制系统的 Git 在软件包 git-core 中。

```
$ sudo aptitude install git-core
$ sudo aptitude install git-doc git-svn git-email git-gui gitk
```

❑ RHEL、Fedora、CentOS：

```
$ yum install git
$ yum install git-svn git-email git-gui gitk
```

在其他发行版中安装 Git 的过程和上面介绍的方法类似。Git 软件包在这些发行版里或称为 git，或称为 git-core。

### 3.1.2 从源代码进行安装

访问 Git 的官方网站：*http://git-scm.com/*。下载 Git 源码包，例如：`git-1.7.4.1.tar.bz2`[①]。安装过程如下：

（1）展开源码包，并进入到相应的目录中。

```
$ tar -jxvf git-1.7.4.1.tar.bz2
$ cd git-1.7.4.1/
```

（2）安装方法写在 `INSTALL` 文件中，参照其中的指示即可完成安装。下面的命令将 Git 安装在 `/usr/local/bin` 中。

```
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

（3）安装 Git 文档（可选）。

编译的文档主要是 HTML 格式的文档，方便通过 `git help -w <sub-command>` 命令查看。实际上，即使不安装 Git 文档，也可以使用 man 手册查看 Git 帮助，使用命令 `git help <sub-command>` 或 `git <sub-command> --help` 即可。

编译文档依赖 asciidoc，因此需要先安装 asciidoc（如果尚未安装的话），然后编译文档。在编译文档时要花费很多时间，要有耐心。

```
$ make prefix=/usr/local doc info
$ sudo make prefix=/usr/local install-doc install-html install-info
```

安装完毕之后，就可以在 `/usr/local/bin` 下找到 `git` 命令。

### 3.1.3 从 Git 版本库进行安装

如果在本地克隆一个 Git 项目的版本库，就可以用版本库同步的方式获取最新版本的

① 在你下载的时候可能已经有了更新的版本。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Git，这样在下载不同版本的 Git 源代码时实际上采用了增量方式，非常节省时间和空间。当然使用这种方法的前提是已经用其他方法安装好了 Git，具体操作过程如下。

（1）克隆 Git 项目的版本库到本地。

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git
```

（2）如果本地已经克隆过一个 Git 项目的版本库，直接在工作区中更新，以获得最新版本的 Git。

```
$ git fetch
```

（3）执行清理工作，避免前一次编译的遗留文件对编译造成影响。注意，下面的操作将丢弃本地对 Git 代码的改动。

```
$ git clean -fdx
$ git reset --hard
```

（4）查看 Git 的里程碑，选择最新的版本进行安装，例如 v1.7.4.1 。

```
$ git tag
...
v1.7.4.1
```

（5）检出该版本的代码。

```
$ git checkout v1.7.4.1
```

（6）执行安装。例如，安装到 /usr/local 目录下。

```
$ make prefix=/usr/local all doc info
$ sudo make prefix=/usr/local install \
  install-doc install-html install-info
```

我在撰写本书的过程中，就是通过 Git 版本库的方式安装的，在 /opt/git 目录下安装了多个不同版本的 Git，以测试 Git 的兼容性。可以使用类似下面的脚本来批量安装不同版本的 Git。

```
#!/bin/sh

for ver in      \
    v1.5.0      \
    v1.7.3.5    \
    v1.7.4.1    \
; do
    echo "Begin install Git $ver.";
    git reset --hard
    git clean -fdx
    git checkout $ver || {
        echo "Checkout git $ver failed."; exit 1
    }
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
    make prefix=/opt/git/$ver all && \
    sudo make prefix=/opt/git/$ver install || {
        echo "Install git $ver failed."; exit 1
    }
    echo "Installed Git $ver."
done
```

### 3.1.4 命令补齐

Linux 的 shell 环境（bash）通过 bash-completion 软件包提供命令补齐功能，在录入命令参数时按一次或两次 TAB 键可实现参数的自动补齐或提示。例如输入 `git com` 后按下 TAB 键，会自动补齐为 `git commit` 。

如果通过包管理器方式安装 Git，一般都已经为 Git 配置好了自动补齐，但是如果是以源码编译的方式安装 Git，就需要为命令补齐多做些工作，具体操作过程如下。

（1）将 Git 源码包中的命令补齐脚本复制到 bash-completion 对应的目录中。

```
$ cp contrib/completion/git-completion.bash \
     /etc/bash_completion.d/
```

（2）重新加载自动补齐脚本，使之在当前的 shell 中生效。

```
$ . /etc/bash_completion
```

（3）为了能够在终端开启时自动加载 `bash_completion` 脚本，需要在系统配置文件 `/etc/profile` [①]及本地配置文件 `~/.bashrc` [②]中添加下面的内容。

```
if [ -f /etc/bash_completion ]; then
  . /etc/bash_completion
fi
```

### 3.1.5 中文支持

Git 的本地化做得并不完善，命令的输出及命令的帮助还只能输出英文，也许在未来的版本中会使用 gettext 实现本地化，就像目前对 git-gui 命令所做的那样。

中文用户最关心的问题还有：是否可以在提交说明中使用中文？是否可以使用中文文件名或目录名？是否可以使用中文来命名分支或里程碑？简单地说，可以在提交说明中使用中文，但是需要对 Git 进行设置。至于用中文来命名文件、目录和引用，只有在使用 UTF-8 字符集的环境下（Linux、Mac OS X、Windows 下的 Cygwin）才可以，否则应尽量避免使用。

#### 1. UTF-8 字符集

Linux 平台的中文用户一般会使用 UTF-8 字符集，Git 在 UTF-8 字符集下可以工作得非常好：

- 在提交时，可以在提交说明中输入中文。

① 配置文件 /etc/profile 及 ~/.bash_profile、~/.profile 等作用于交互式登录 shell。

② 配置文件 ~/.bashrc 作用于交互式非登录 shell，如 screen 或 byobu 中建立的新的 shell 窗口。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 显示提交历史，能够正常显示提交说明中的中文字符。
- 可以添加名称为中文的文件，并可以在同样使用UTF-8字符集的Linux环境中克隆和检出。
- 可以创建带有中文字符的里程碑名称。

但是，在默认设置下，中文文件名在工作区状态输出、查看历史更改概要，以及在补丁文件中，文件名中的中文不能正确地显示，而是显示为八进制的字符编码，如下：

```
$ git status -s
?? "\350\257\264\346\230\216.txt"
$ printf "\350\257\264\346\230\216.txt\n"
说明.txt
```

通过将Git配置变量 `core.quotepath` 设置为 `false`，就可以解决中文文件名在这些Git命令输出中的显示问题。

```
$ git config --global core.quotepath false
$ git status -s
?? 说明.txt
```

#### 2. GBK字符集

如果Linux平台采用非UTF-8的字符集，例如，用zh_CN.GBK字符集编码（有人这么做吗？），就要另外再做些工作了。

- 将显示提交说明所使用的字符集设置为 `gbk`，这样使用 `git log` 查看提交说明时才能够正确显示其中的中文。

```
$ git config --global i18n.logOutputEncoding gbk
```

- 设置录入提交说明时所使用的字符集，以便在commit对象中正确标注字符集。
  Git在提交时并不会对提交说明进行从GBK字符集到UTF-8的转换，但是可以在提交说明中标注所使用的字符集，因此在非UTF-8字符集的平台中录入中文时需要用下面的指令设置录入提交说明的字符集，以便在commit对象中嵌入正确的编码说明。

```
$ git config --global i18n.commitEncoding gbk
```

## 3.2 在Mac OS X下安装和使用Git

Mac OS X被称为最人性化的操作系统之一，工作在Mac上是件非常惬意的事情，工作中又怎能没有Git呢？

### 3.2.1 以二进制发布包的方式安装

Git在Mac OS X中也有好几种安装方法。最简单的方式是安装 `.dmg` 格式的安装包。

访问git-osx-installer的官方网站：*http://code.google.com/p/git-osx-installer/*，下载Git安装包。安装包带有 `.dmg` 扩展名，是苹果磁盘镜像（Apple Disk Image）格式的软件发布包。从官方网站上下载文件名格式为 `git-<version>-<arch>-leopard.dmg` 的安装包

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




文件，例如：git-1.7.4-x86_64-leopard.dmg 是 64 位的安装包，git-1.7.4-i386-leopard.dmg 是 32 位的安装包。建议选择 64 位的软件包，因为 Mac OS X 10.6（雪豹）完美地兼容 32 位和 64 位（开机按住键盘数字 3 和 2 进入 32 位系统，按住 6 和 4 进入 64 位系统），即使内核的架构是 32 位的，也可以放心地运行 64 位的软件包。

苹果的 .dmg 格式的软件包实际上是一个磁盘映像，安装起来非常方便，点击该文件就可以直接挂载到 Finder 中，打开后如图 3-1 所示。

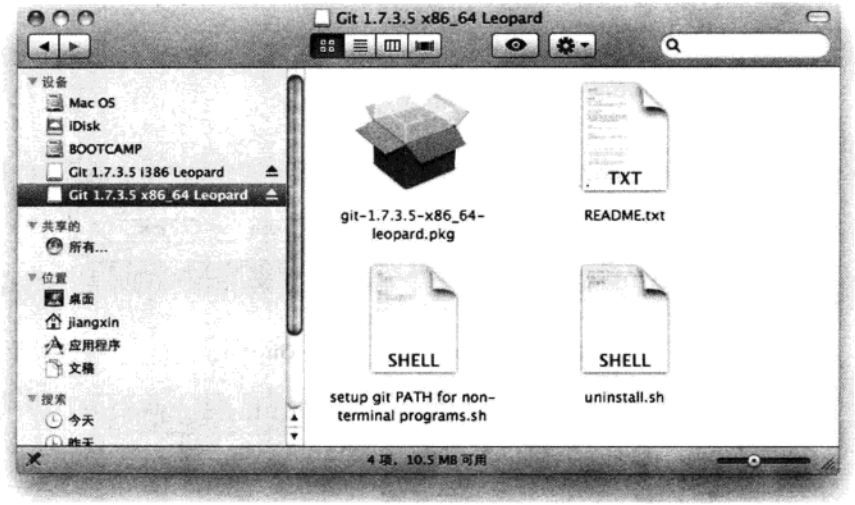



图 3-1 在 Mac OS X 下打开 .dmg 格式磁盘镜像

图 3-1 中显示为拆包图标的文件（扩展名为 .pkg）就是 Git 的安装程序，另外的两个脚本程序，一个用于应用的卸载（uninstall.sh），另外一个带有长文件名的脚本在 Git 安装后再执行，为非终端应用注册 Git 的安装路径，这是因为 Git 部署在标准的系统路径之外 /usr/local/git/bin 。

点击扩展名为 .pkg 的安装程序，开始 Git 的安装（以安装 Git1.7.3.5 版本为例），根据提示按步骤完成安装，如图 3-2 所示。

安装完毕后，Git 会被安装到 /usr/local/git/bin/ 目录下。重启终端程序，这样 /etc/paths.d/git 文件为 PATH 环境变量中添加的新路径注册才能生效，然后就可以在终端直接运行 git 命令了。

### 3.2.2 安装 Xcode

Mac OS X 基于 Unix 内核开发，因此也可以很方便地通过源码编译的方式进行安装，但是默认安装的 Mac OS X 缺乏相应的开发工具，需要安装苹果提供的 Xcode 软件包。在 Mac 随机附送的光盘（Mac OS X Install DVD）的可选安装文件夹下就有 Xcode 的安装包（如图 3-3 所示），通过随机光盘安装 Xcode 可以省去网络下载的麻烦，要知道 Xcode 的大小在 3GB 以上。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

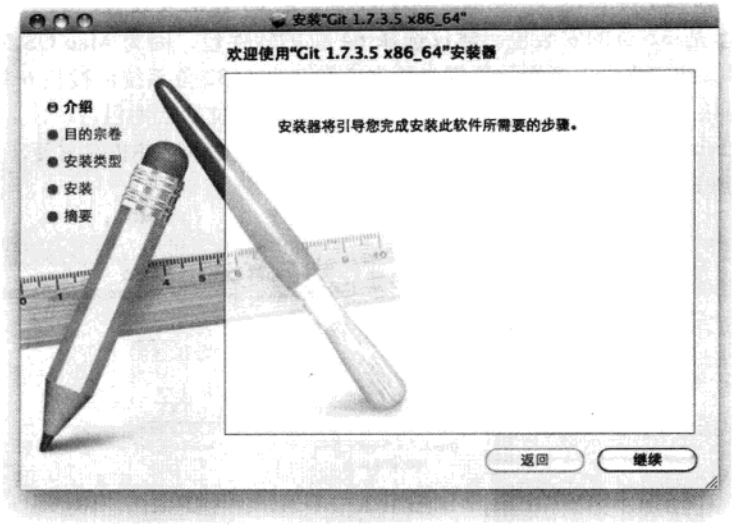




图 3-2 在 Mac OS X 下安装 Git

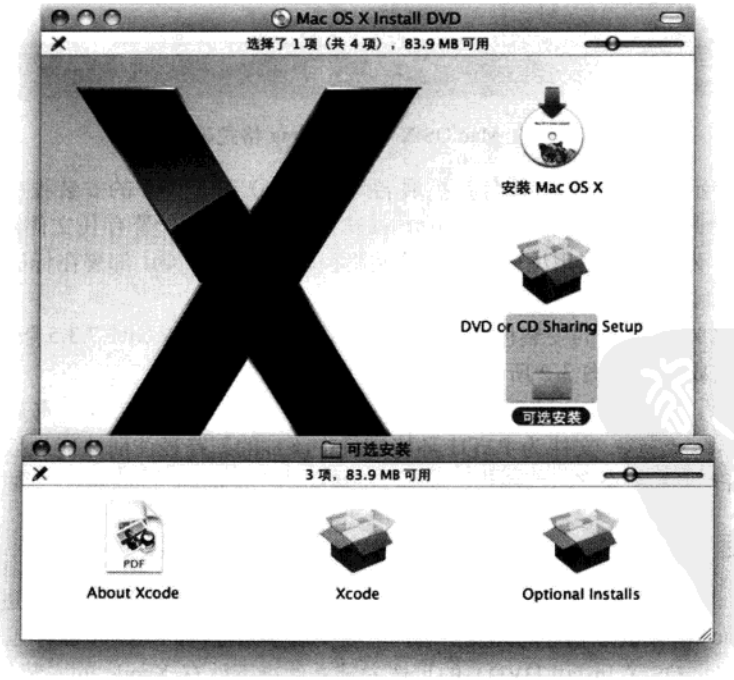



图 3-3 在 Mac OS X 下安装 Xcode

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



### 3.2.3 使用 Homebrew 安装 Git

Mac OS X 有好几个包管理器，用于管理一些开源软件在 Mac OS X 上的安装和升级。有传统的 MacPorts[①]、Fink[②]，还有更为简单易用的 Homebrew[③]。下面就介绍一下如何通过 Homebrew 包管理器来以源码包编译的方式安装 Git。

Homebrew 用 Ruby 语言开发，支持千余种开源软件在 Mac OS X 中的部署和管理。Homebrew 项目托管在 Github 上，网址为：*https://github.com/mxcl/homebrew*。

首先是安装 Homebrew，执行下面的命令：

```
$ ruby -e \
   "$(curl -fsSL https://gist.github.com/raw/323731/install_homebrew.rb)"
```

安装完成后，Homebrew 的主程序安装在 `/usr/local/bin/brew` 中，在目录 `/usr/local/Library/Formula/` 下保存了 Homebrew 支持的所有软件的安装指引文件。

执行下面的命令，通过 Homebrew 安装 Git。

```
$ brew install git
```

使用 Homebrew 方式安装，Git 被安装在 `/usr/local/Cellar/git/<version>/` 中，可执行程序自动在 `/usr/local/bin` 目录下创建符号连接，可以直接在终端程序中访问。

通过 `brew list` 命令可以查看安装的开源软件包。

```
$ brew list
git
```

也可以查看某个软件包安装的详细路径和安装内容。

```
$ brew list git
/usr/local/Cellar/git/1.7.4.1/bin/gitk
...
```

### 3.2.4 从 Git 源码进行安装

如果需要安装历史版本的 Git 或是尚在开发中的未发布版本的 Git，就需要从源码安装或通过克隆 Git 源码库的方式进行安装。既然 Homebrew 就是通过源码编译方式安装 Git 的，那么也应该可以直接从源码进行安装，但是使用 Homebrew 安装 Git 和直接通过 Git 源码安装并不完全等同，例如 Homebrew 安装 Git 文档是通过下载已经编译好的 Git 文档包进行安装，而非从头对文档进行编译。

直接通过源码安装 Git 软件及文档，遇到的主要问题就是对文档的编译，因为 Xcode 没

① *http://www.macports.org/*
② *http://www.finkproject.org/*
③ *http://mxcl.github.com/homebrew/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


有提供 Git 文档编译所需要的相关工具。但是，这些工具可以通过 Homebrew 进行安装。安装工具软件的过程可能会遇到一些小麻烦，不过大多可以通过参考命令输出予以解决。

```
$ brew install asciidoc
$ brew install docbook2x
$ brew install xmlto
```

当编译源码及文档的工具部署完成后，就可以通过源码编译 Git。

```
$ make prefix=/usr/local all doc info
$ sudo make prefix=/usr/local install \
  install-doc install-html install-info
```

### 3.2.5 命令补齐

Git 通过 bash-completion 软件包实现命令自动补齐，在 Mac OS X 下可以通过 Homebrew 进行安装。

```
$ brew search completion
bash-completion
$ brew install bash-completion
...
Add the following lines to your ~/.bash_profile file:
if [ -f 'brew --prefix'/etc/bash_completion ]; then
  . 'brew --prefix'/etc/bash_completion
fi
...
```

根据 bash-completion 在安装过程中的提示，修改配置文件 ~/.bash_profile 和 ~/.bashrc，并在其中加入如下内容，以便在终端加载时自动启用命令补齐。

```
if [ -f 'brew --prefix'/etc/bash_completion ]; then
  . 'brew --prefix'/etc/bash_completion
fi
```

将 Git 的命令补齐脚本拷贝到 bash-completion 对应的目录中。

```
$ cp contrib/completion/git-completion.bash \
      'brew --prefix'/etc/bash_completion.d/
```

不用重启终端程序，只需要运行下面的命令即可立即在当前的 shell 中加载命令补齐。

```
  'brew --prefix'/etc/bash_completion
```

### 3.2.6 其他辅助工具的安装

本书中还会用到一些常用的 GNU 或其他开源软件，在 Mac OS X 下也可以通过 Homebrew 进行安装。这些软件包括：

- gnupg：数字签名和加密工具，在为 Git 版本库建立签名里程碑时会用到。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




- md5sha1sum：生成 MD5 或 SHA1 摘要，在研究 Git 版本库中的对象时会用到。
- cvs2svn：CVS 版本库迁移到 SVN 或 Git 的工具，在版本库迁移时会用到。
- stgit：Git 的补丁和提交管理工具。
- quilt：一种补丁管理工具，在介绍 StGit 时会用到。

在 Mac OS X 下能够使用到的 Git 图形工具除了 Git 软件包自带的 `gitk` 和 `git gui` 之外，还有 GitX。下载地址为：

- GitX 的原始版本：*http://gitx.frim.nl/*。
- GitX 的一个分支版本，提供增强的功能：*https://github.com/brotherbard/gitx/downloads*。

Git 的图形工具一般需要在本地克隆版本库的工作区中执行，为了能与 Mac OS X 更好地整合，可以通过插件实现与 Finder 的整合。在 git-osx-installer 的官方网站：*http://code.google.com/p/git-osx-installer/* 上有两个以 `OpenInGitGui-` 和 `OpenInGitX-` 为前缀的软件包，可以分别实现与 `git gui` 和 `gitx` 的整合：在 Finder 中进入工作区目录，点击对应插件的图标，启动 `git gui` 或 `gitx`。

### 3.2.7 中文支持

由于 Mac OS X 采用 Unix 内核，在中文支持上与 Linux 相近，具体内容请参照第 3.1.5 节介绍的在 Linux 下安装 Git 的相关内容。

## 3.3 在 Windows 下安装和使用 Git（Cygwin 篇）

在 Windows 下安装和使用 Git 有两种不同的方案：通过安装 msysGit[①]或 Cygwin[②]来使用 Git。在这两种不同的方案下，Git 的使用与 Linux 环境下完全一致。还可以通过 msysGit 的图形界面软件 TortoiseGit[③]（也就是在 CVS 和 SVN 时代就已经广为人知的 Tortoise 系列软件的 Git 版本）来使用 Git。TortoiseGit 可与资源管理器整合，从而提供 Git 操作的图形化界面。

首先介绍通过 Cygwin 来使用 Git，并不是因为这是最便捷的方法。如果需要在 Windows 中快速安装和使用 Git，下一节介绍的 msysGit 才是最便捷的方法。之所以将 Cygwin 放在前面介绍是因为本书在介绍 Git 原理和其他 Git 相关的软件时用到了大量的开源工具，在 Cygwin 下很容易获得这些开源工具，而 msysGit 的 MSYS[④]（Minimal SYStem，最简系统）则不能满足我们的需求。因此，我建议 Windows 平台下的读者在跟随本书学习 Git 的过程中首选 Cygwin，当对 Git 有了一定的了解后，无论是 msysGit 还是 TortoiseGit，您都会应对自如。

① *http://code.google.com/p/msysgit/*
② *http://www.cygwin.com/*
③ *http://code.google.com/p/tortoisegit/*
④ *http://www.mingw.org/wiki/msys*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Cygwin 是一款伟大的软件，通过一个小小的 DLL（cygwin1.dll）建立了 Linux 与 Windows 之间的系统调用和 API 之间的转换，使得 Linux 下的绝大多数软件能向 Windows 迁移。Cygwin 通过 cygwin1.dll 所建立的中间层与 VMWare[①]、VirtualBox[②] 等虚拟机软件完全不同，不会独占系统资源。像 VMWare 等虚拟机，只要启动一个虚拟机（操作系统），即使不在其中执行任何命令，同样也会占用大量的内存和 CPU 时间等系统资源。

Cygwin 还提供了一个强大易用的包管理工具（setup.exe），使得几千个开源软件包能在 Cygwin 下便捷地安装和升级，Git 便是其中的一员。

我对 Cygwin 有着深厚的感情，Cygwin 让我能在 Windows 平台下用 Linux 的方式更有效率地工作，使用 Linux 风格的控制台替换 Windows 黑乎乎的、冷冰冰的、由 cmd.exe 提供的命令行。Cygwin 帮助我逐渐摆脱对 Windows 的依赖，当我完全转换到 Linux 平台时，没有感到一丝的障碍。

### 3.3.1 安装 Cygwin

Cygwin 的安装非常简单，先在其官方网站 *http://www.cygwin.com/* 下载安装程序——一个只有几百 KB 的 setup.exe 文件，然后即可开始安装。

安装过程中会让用户选择安装模式：通过网络安装、下载后安装或者通过本地软件包缓存（安装时自动在本地目录下建立的软件包缓存）安装。

（1）如果是第一次安装 Cygwin，因为本地尚没有软件包缓存，当然只能选择从网络安装，如图 3-4 所示。

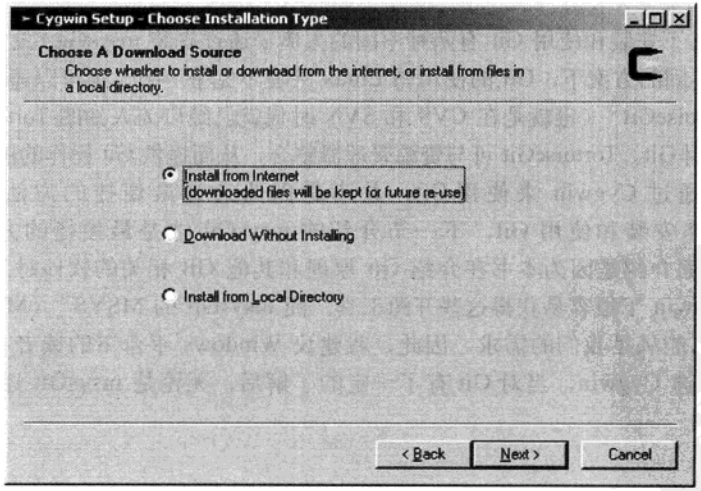



图 3-4 选择安装模式

① *http://www.vmware.com/*

② *http://www.virtualbox.org/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




（2）选择安装目录，默认为 `C:\cygwin`，如图 3-5 所示。这个目录将作为 Cygwin shell 环境的根目录（根卷），Windows 的各个盘符将挂载在根卷的一个特殊目录之下。

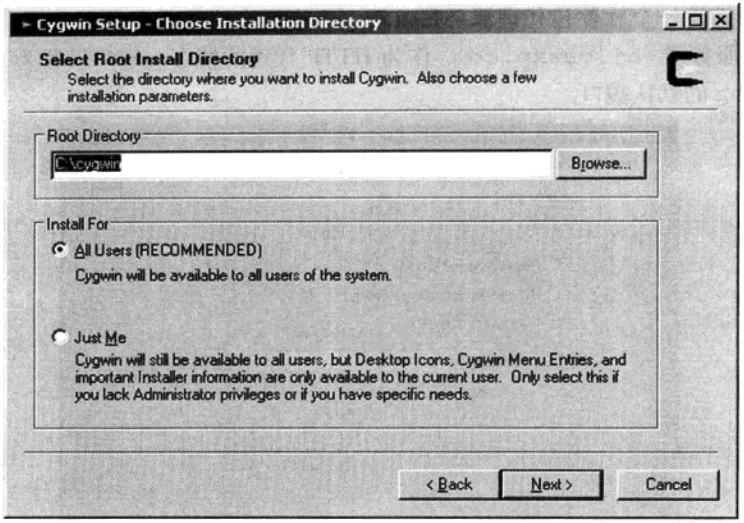



图 3-5 选择安装目录

（3）设置本地软件包缓存目录，默认为 `setup.exe` 所处的目录，如图 3-6 所示。

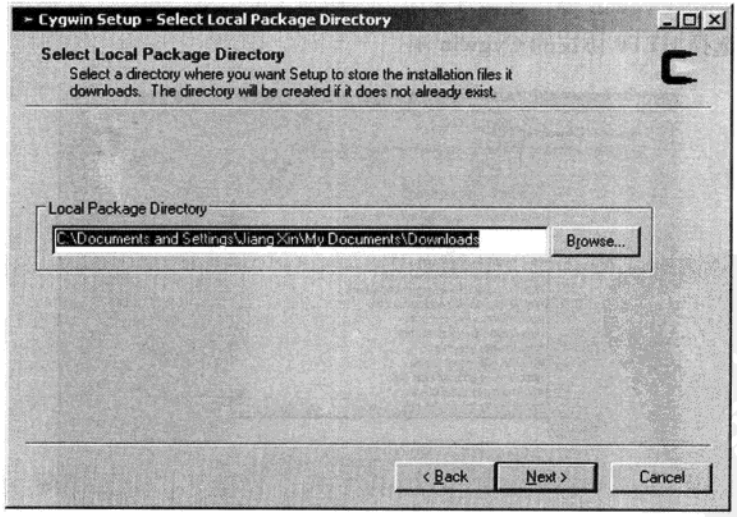



图 3-6 选择本地软件包缓存目录

（4）设置网络连接方式是否使用代理等，如图 3-7 所示。默认会选择第一项：“直接网

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


络连接”。如果一个团队有很多人要使用 Cygwin，架设一个能够提供软件包缓存的 HTTP 代理服务器会节省大量的网络带宽和大量的时间。在 Debian/Ubuntu 下用 apt-cacher-ng[①]就可以非常简单地搭建一个软件包代理服务器。图 3-7 显示的就是我在公司内网中安装 Cygwin 时使用内网的服务器 `bj.ossxp.com` 作为 HTTP 代理的情形，端口设置为 9999，因为这是 apt-cacher-ng 的默认端口。

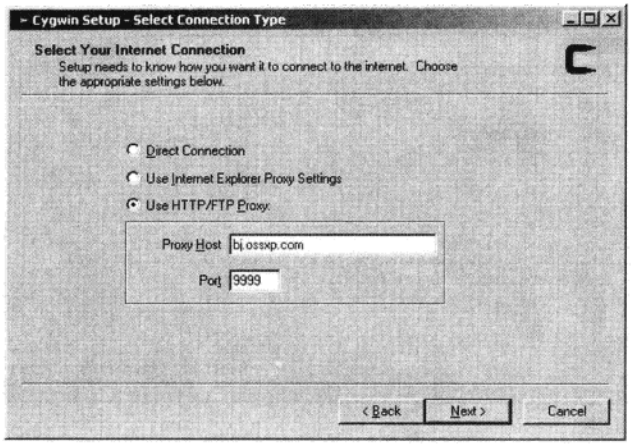



图 3-7 是否使用代理下载 Cygwin 软件包

（5）选择一个 Cygwin 源，如图 3-8 所示。如果在上一个步骤中选择使用 HTTP 代理服务器，就必须选择 HTTP 协议的 Cygwin 源。

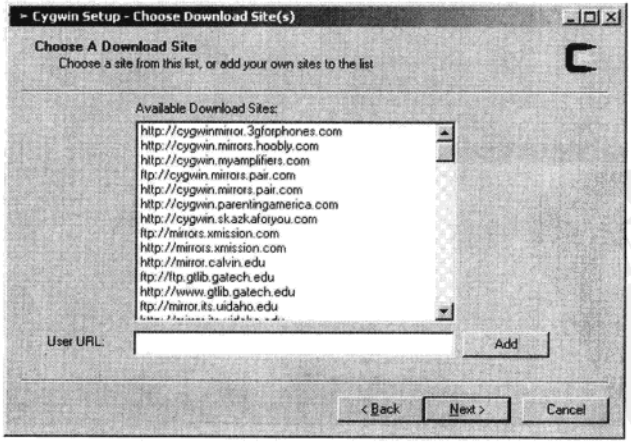



图 3-8 选择 Cygwin 源

① *http://www.unix-ag.uni-kl.de/~bloch/acng/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




（6）从所选的 Cygwin 源下载软件包索引文件，然后显示软件包管理器界面，如图 3-9 所示。

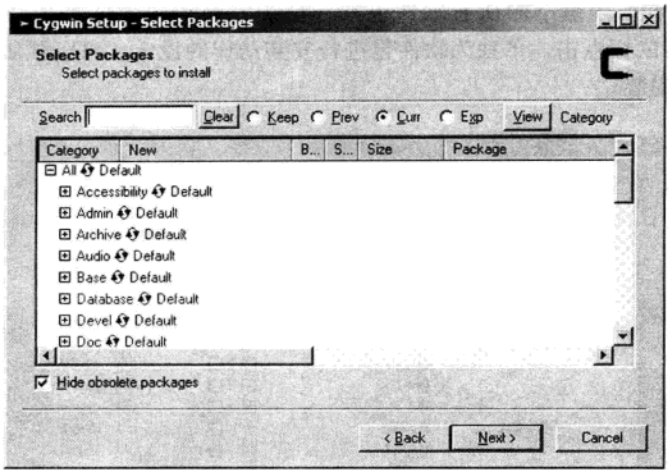



图 3-9 Cygwin 软件包管理器

Cygwin 的软件包管理器非常强大，而且易于使用（如果习惯了其界面）。软件包归类于各个分组中，点击分组前的加号就可以展开分组。在展开的 Admin 分组中，如图 3-10 所示（这个截图不是首次安装 Cygwin 的截图），有的软件包（如 `libattr1`）已经安装过了，因为没有新版本而标记为“Keep”（保持），没有安装过并且不准备安装的软件包则标记为“Skip”（跳过）。

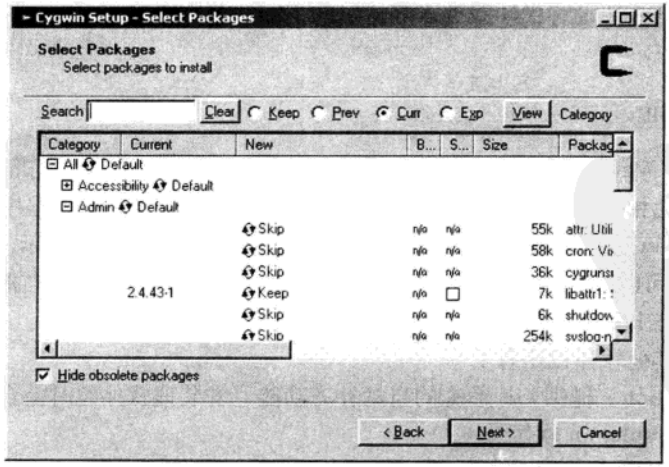



图 3-10 Cygwin 软件包管理器展开分组

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（7）点击分组名称后面的动作名称（文字“Default”），会进行软件包安装动作的切换。例如图 3-11，将 Admin 分组的安装动作由“Default”（默认）切换为“Install”（安装），会看到 Admin 分组下的所有软件包都标记为安装（显示具体要安装的软件包版本号）。也可以通过鼠标点击，单独为软件包进行安装动作的设定：可以强制重新安装、安装旧版本，或者不安装。

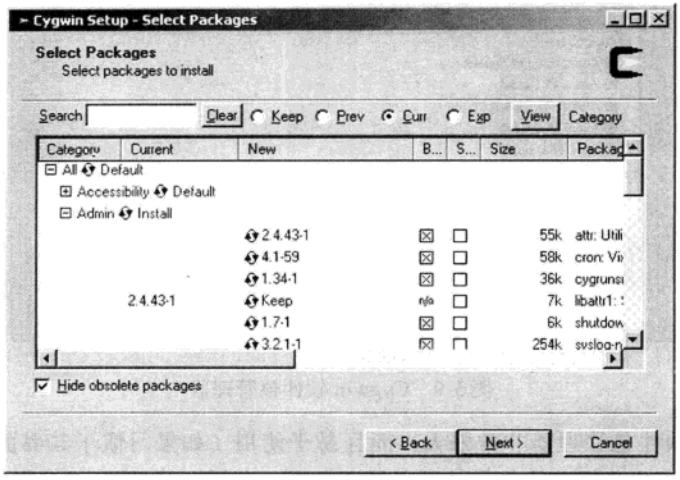



图 3-11 安装某一分组下的所有软件

（8）当通过软件包管理器对要安装的软件包定制完毕后，点击下一步，开始下载软件包、安装软件包和软件包后处理（postinstall），直至完成安装。根据选择的软件包的多少、网络情况，以及是否架设有代理服务器等情况的不同，首次安装 Cygwin 的时间可能从几分钟到几个小时不等。

### 3.3.2 安装 Git

默认安装的 Cygwin 没有安装 Git 软件包。如果在首次安装过程中忘记通过包管理器选择安装 Git 或其他相关软件包，可以在安装后再次运行 Cygwin 的安装程序 `setup.exe` 。当再次进入 Cygwin 包管理器界面时，在搜索框中输入 git，如图 3-12 所示。

从图 3-12 中可以看出在 Cygwin 中包含了很多与 Git 相关的软件包，把这些 Git 相关的软件包全部都安装上吧，如图 3-13 所示。

需要安装的其他软件包还有：

- git-completion：提供 Git 命令的自动补齐功能。安装该软件包时会自动安装其所依赖的 bash-completion 软件包。
- openssh：SSH 客户端，为访问 SSH 协议的版本库提供支持。
- vim：Git 默认的编辑器。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



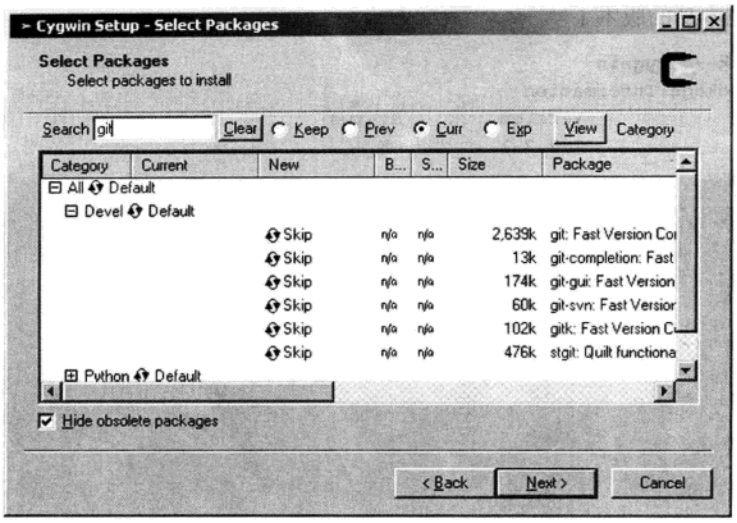




图 3-12 Cygwin 软件包管理器中搜索 git

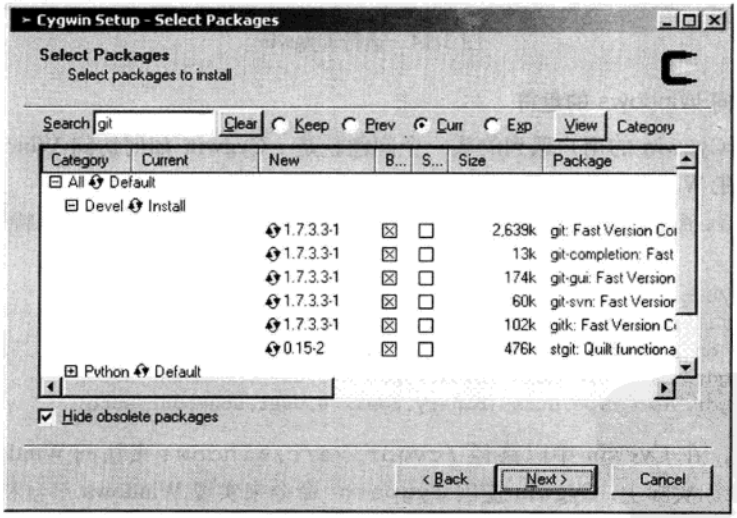



图 3-13 Cygwin 软件包管理器中安装 git

### 3.3.3 Cygwin 的配置和使用

运行 Cygwin 后会进入 shell 环境并见到熟悉的 Linux 提示符，如图 3-14 所示。

可以通过执行 `cygcheck` 命令来查看 Cygwin 中安装的软件包的版本。例如，查看

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Cygwin 软件包本身的版本：

```
$ cygcheck -c cygwin
Cygwin Package Information
Package              Version        Status
cygwin               1.7.7-1        OK
```

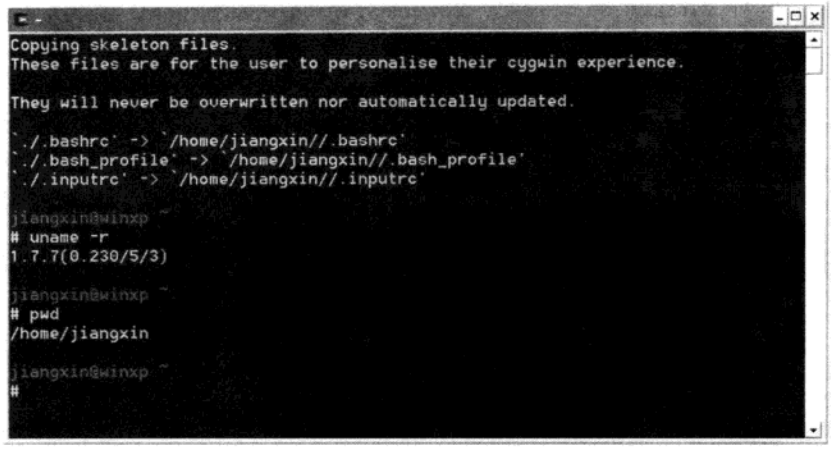



图 3-14 运行 Cygwin

### 1. 如何访问 Windows 的盘符

刚刚接触 Cygwin 的用户遇到的头一个问题就是：Cygwin 如何访问 Windows 的各个磁盘目录，以及在 Windows 平台下如何访问 Cygwin 中的目录。

执行 mount 命令后可以看到 Windows 下的盘符被映射到 /cygdrive 特殊目录下。

```
$ mount
C:/cygwin/bin on /usr/bin type ntfs (binary,auto)
C:/cygwin/lib on /usr/lib type ntfs (binary,auto)
C:/cygwin on / type ntfs (binary,auto)
C: on /cygdrive/c type ntfs (binary,posix=0,user,noumount,auto)
D: on /cygdrive/d type ntfs (binary,posix=0,user,noumount,auto)
```

也就是说，在 Cygwin 中以路径 /cygdrive/c/Windows 来访问 Windows 下的 C:\Windows 目录。实际上，Cygwin 提供 cygpath 命令来实现 Windows 平台和 Cygwin 之间目录名称的变换，如下所示：

```
$ cygpath -u C:\\Windows
/cygdrive/c/Windows

$ cygpath -w ~/
C:\cygwin\home\jiangxin\
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




从上面的示例也可以看出，Cygwin 下的用户主目录（即 /home/jiangxin/ ）相当于 Windows 下的 C:\cygwin\home\jiangxin\ 目录。

### 2. 用户主目录不一致的问题

如果某些其他软件（如 msysGit）为 Windows 设置了 HOME 环境变量，会影响到 Cygwin 中用户主目录的设置，甚至会造成在 Cygwin 中不同的命令有不同的用户主目录的情况。例如：Cygwin 下 Git 的用户主目录被设置为 /cygdrive/c/Documents and Settings/jiangxin，而 SSH 客户端软件的主目录被设置为 /home/jiangxin，这会给用户造成困惑。

之所以出现这种情况，是因为 Cygwin 确定用户主目录有几个不同的依据，要按照顺序确定主目录：首先查看系统的 HOME 环境变量，其次查看 /etc/passwd 中为用户设置的主目录。有的软件遵照这个原则，而有些 Cygwin 应用如 SSH，却没有使用 HOME 环境变量而是直接使用 /etc/passwd 中的设置。要想避免在同一个 Cygwin 环境下有两个不同的用户主目录设置，可以采用下面两种方法。

- 方法 1：修改 Cygwin 启动的批处理文件（如：C:\cygwin\Cygwin.bat ），在批处理的开头添加如下的一行代码，就可以防止其他软件在 Windows 引入的 HOME 环境变量被带入到 Cygwin 中。

```
set HOME=
```

- 方法 2：如果希望使用 HOME 环境变量指向的主目录，则可通过手工编辑 /etc/passwd 文件，将其中的用户主目录修改成 HOME 环境变量所指向的目录[①]。

### 3. 命令行补齐忽略文件名大小写

Windows 的文件系统忽略文件名的大小写，在 Cygwin 下最好对命令行补齐进行相关设置，以忽略大小写，这样使用起来更方便。

编辑文件 ~/.inputrc，在其中添加设置“set completion-ignore-case on”，或者取消已有的相关设置前面的井（#）号注释符。修改完毕后，再重新进入 Cygwin，这样就可以实现命令行补齐对文件名大小写的忽略。

### 4. 忽略文件权限的可执行位

Linux、UNIX、Mac OS X 通过文件权限中的可执行位判断文件是否可执行，而 Windows 是通过文件扩展名进行判断的。Git 提供对类 UNIX 系统文件权限的支持，在版本库中建立对可执行文件的追踪。对于 Windows 平台，Git 的这个特性用处不大，甚至有害。因为，虽然 Cygwin 可以模拟 Linux 下的文件授权并对文件的可执行位提供支持，但为支持文件权限而调用 Cygwin 的 stat() 函数和 lstat() 函数会比调用 Windows 自身的 Win32 API 要慢一半[②]，

① *http://www.cygwin.com/cygwin-ug-net/ntsec.html*

② git-config(1) 用户手册中关于 core.ignoreCygwinFSTricks 的介绍。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


而且非跨平台的项目也没有必要对文件权限位进行跟踪，甚至 Windows 下的其他工具及操作可能会破坏文件的可执行位，从而导致 Cygwin 下的 Git 认为文件的权限更改需要重新提交。通过下面的配置可以禁止 Git 对文件权限的跟踪：

```
$ git config --system core.fileMode false
```

在此模式下，当已添加到版本库中的文件的权限的可执行位改变时，该文件不会显示有改动。版本库中新增的文件，无论文件本身是否设置为可执行，都以 100644 的权限（忽略可执行位）进行添加。

关于 Cygwin 的更多内容，请参见网址 *http://www.cygwin.com/cygwin-ug-net/* 中的信息。

### 3.3.4 Cygwin 下 Git 的中文支持

Cygwin 的当前版本是 1.7.x，对中文的支持非常好。无需任何配置就可以在 Cygwin 的窗口内输入中文，执行 `ls` 命令就可以显示中文的文件名。这与六七年前的 Cygwin 1.5.x 完全不一样了，老版本的 Cygwin 还需要进行一些配置才能在控制台输入中文和显示中文，但是最新版本的 Cygwin 已经完全不需要了。后面要介绍的 msysGit 的 shell 环境仍然需要进行相关的配置（同老版本 Cygwin）才能够正常显示和输入中文。

Cygwin 默认使用 UTF-8 字符集，并能巧妙地与 Windows 系统的字符集进行转换。在 Cygwin 下可执行 `locale` 命令查看 Cygwin 下正在使用的字符集：

```
$ locale
LANG=C.UTF-8
LC_CTYPE="C.UTF-8"
LC_NUMERIC="C.UTF-8"
LC_TIME="C.UTF-8"
LC_COLLATE="C.UTF-8"
LC_MONETARY="C.UTF-8"
LC_MESSAGES="C.UTF-8"
LC_ALL=
```

正因为如此，Cygwin 下的 Git 对中文的支持也非常出色。虽然中文的 Windows 本身使用 GBK 字符集，但是在 Cygwin 下，Git 就如同工作在采用 UTF-8 字符集的 Linux 下，对中文的支持非常的好：

- ❑ 在提交时，可以在提交说明中输入中文。
- ❑ 显示提交历史时，能够正常显示提交说明中的中文字符。
- ❑ 可以添加文件名称为中文的文件，并可以在使用 UTF-8 字符集的 Linux 环境中克隆和检出。
- ❑ 可以创建带有中文字符的里程碑名称。

但是和 Linux 平台一样，在默认设置下，文件名称中包含中文的文件，在工作区状态输出、查看历史更改概要，以及在补丁文件中，文件名中的中文不能正确地显示，而是用若干八进制字符编码来显示，如下：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
$ git status -s
?? "\350\257\264\346\230\216.txt"
$ printf "\350\257\264\346\230\216.txt"
说明 .txt
```

配置变量 core.quotepath 设置为 false 就可以解决中文文件名在这些 Git 命令输出中的显示问题。

```
$ git config --global core.quotepath false
$ git status -s
?? 说明 .txt
```

### 3.3.5 Cygwin 下 Git 访问 SSH 服务

在本书第 5 篇第 29 章中介绍的以公钥认证的方式访问 Git 服务，是实现 Git 写操作的最重要的服务。以公钥认证方式访问 SSH 协议的 Git 服务器时无须输入口令，而且更安全。使用公钥认证就涉及如何创建公钥 / 私钥对，以及在 SSH 连接时应该选择哪一个私钥的问题（如果建立有多个私钥）。

Cygwin 下的 openssh 软件包提供的 ssh 命令和 Linux 下的 ssh 命令没有什么区别，也提供了 ssh-keygen 命令来管理 SSH 公钥 / 私钥对。但是，Cygwin 当前的 openssh（版本号：5.7p1-1）有一个 Bug，使用 Git 克隆采用 SSH 协议的版本库时偶尔会中断，从而无法完成版本库的克隆。示例如下：

```
$ git clone git@bj.ossxp.com:ossxp/gitbook.git
Cloning into gitbook...
remote: Counting objects: 3486, done.
remote: Compressing objects: 100% (1759/1759), done.
fatal: The remote end hung up unexpectedly MiB | 3.03 MiB/s
fatal: early EOFs:  75% (2615/3486), 13.97 MiB | 3.03 MiB/s
fatal: index-pack failed
```

如果您也遇到同样的问题，建议使用 PuTTY 提供的 plink.exe 作为 SSH 客户端，替代 Cygwin 自带的 ssh 命令。

#### 1. 安装 PuTTY

PuTTY 是 Windows 下的一个开源软件，提供 SSH 客户端服务，还包括公钥管理的相关工具。访问 PuTTY 的主页（*http://www.chiark.greenend.org.uk/~sgtatham/putty/*），下载并安装 PuTTY。安装完毕会发现 PuTTY 软件包包含了多个可执行程序，下面几个命令用于与 Git 的整合。

- Plink：即 plink.exe，是命令行的 SSH 客户端，用于替代 ssh 命令。默认安装于 C:\Program Files\PuTTY\plink.exe 路径中。
- PuTTYgen：用于管理 PuTTY 格式的私钥，也可以用于将 openssh 格式的私钥转换为 PuTTY 格式的私钥。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


❑ Pageant：SSH 认证代理，运行于后台，负责为 SSH 连接提供私钥访问服务。

### 2. PuTTY 格式的私钥

PuTTY 使用专有格式的私钥文件（扩展名为 .ppk），而不能直接使用 openssh 格式的私钥。也就是用 openssh 的 ssh-keygen 命令创建的私钥不能直接被 PuTTY 拿过来使用，必需经过转换，程序 PuTTYgen 可以实现私钥格式的转换。

运行 PuTTYgen 程序，如图 3-15 所示。




图 3-15 运行 PuTTYgen 程序

PuTTYgen 既可以重新创建私钥文件，又可以通过点击加载按钮（load）读取 openssh 格式的私钥文件，从而可以将其转换为 PuTTY 格式的私钥。点击加载按钮，会弹出文件选择对话框，选择 openssh 格式的私钥文件（如文件 id_rsa），如果转换成功，会显示如图 3-16 的界面。

然后点击“Save private key”（保存私钥），就可以将私钥保存为 PuTTY 的 .ppk 格式，例如将私钥保存到文件 ~/.ssh/jiangxin-cygwin.ppk 中。

### 3. Git 使用 Pageant 进行公钥认证

Git 在使用命令行工具 Plink（plink.exe）作为 SSH 客户端访问 SSH 协议的版本库服

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




务器时，如何选择公钥呢？使用 Pageant 是一个非常好的选择。Pageant 是 PuTTY 软件包中的代理软件，为各个 PuTTY 应用提供私钥请求，当 Plink 连接 SSH 服务器需要请求公钥认证时，Pageant 就会给 Plink 提供相应的私钥。

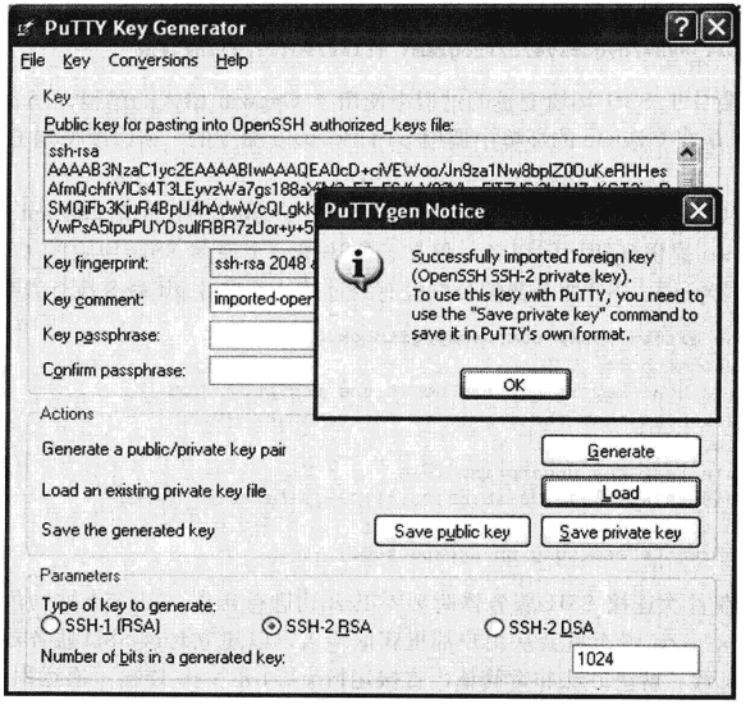



图 3-16 PuTTYgen 完成私钥加载

运行 Pageant，启动后会在托盘区显示一个图标，在后台运行。使用鼠标右键单击 Pageant 的图标，就会弹出相关的菜单，如图 3-17 所示。




图 3-17 Pageant 的弹出菜单

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


点击弹出菜单中的“Add Key”（添加私钥）按钮，会弹出文件选择框，选择扩展名为.ppk 的 PuTTY 格式的公钥，即完成了 Pageant 的私钥准备工作。

接下来，还需要对 Git 进行设置，设置 Git 使用 plink.exe 作为 SSH 客户端，而不是默认的 ssh 命令。通过设置 GIT_SSH 环境变量即可实现：

```
$ export GIT_SSH=/cygdrive/c/Program\ Files/PuTTY/plink.exe
```

上面在设置 GIT_SSH 环境变量的过程中使用了 Cygwin 格式的路径，而非 Windows 格式的，因为 Git 是在 Cygwin 的环境中调用 plink.exe 命令的，所以要使用 Cygwin 能够理解的路径。

这样就可以用 Git 访问 SSH 协议的 Git 服务器了。运行在后台的 Pageant 会在需要的时候为 plink.exe 提供私钥访问服务，但在首次连接一个使用 SSH 协议的 Git 服务器的时候，很可能会因为远程 SSH 服务器的公钥没有经过确认而导致 git 命令执行失败。例如：

```
$ git clone git@bj.ossxp.com:ossxp/gitbook.git
Cloning into gitbook...
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 49:eb:04:30:70:ab:b3:28:42:03:19:fe:82:f8:1a:00
Connection abandoned.
fatal: The remote end hung up unexpectedly
```

这是因为在首次连接 SSH 服务器时要对其公钥进行确认（以防止被钓鱼），而运行于 Git 下的 plink.exe 没有机会从用户那里获取输入，以建立对该 SSH 服务器公钥的信任，因此 Git 访问失败。解决办法非常简单，直接运行 plink.exe 连接一次远程 SSH 服务器，并对公钥确认进行应答即可。操作如下：

```
$ /cygdrive/c/Program\ Files/PuTTY/plink.exe git@bj.ossxp.com
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 49:eb:04:30:70:ab:b3:28:42:03:19:fe:82:f8:1a:00
If you trust this host, enter "y" to add the key to
PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without
adding the key to the cache, enter "n".
If you do not trust this host, press Return to abandon the
connection.
Store key in cache? (y/n)
```

输入“y”，将公钥保存在信任链中，以后再和该主机连接时就不会弹出该确认应答了。当然，Git 命令也就可以成功执行了。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




#### 4. 使用自定义 SSH 脚本取代 Pageant

使用 Pageant 时还要在它每次启动时手动选择私钥文件，这比较麻烦。实际上，可以创建一个脚本对 plink.exe 进行封装，在封装的脚本中使用 -i 参数指定私钥文件。

例如，创建脚本 ~/bin/ssh-jiangxin，文件内容如下：

```
#!/bin/sh

/cygdrive/c/Program\ Files/PuTTY/plink.exe -T -i \
    c:/cygwin/home/jiangxin/.ssh/jiangxin-cygwin.ppk $*
```

设置该脚本为可执行脚本。

```
$ chmod a+x ~/bin/ssh-jiangxin
```

使用下面的命令通过该脚本与远程 SSH 服务器连接：

```
$ ~/bin/ssh-jiangxin git@bj.ossxp.com
Using username "git".
hello jiangxin, the gitolite version here is v1.5.5-9-g4c11bd8
the gitolite config gives you the following access:
     R          gistore-bj.ossxp.com/.*$
     R          gistore-ossxp.com/.*$
  C  R  W       ossxp/.*$
     R  W       test/repo1
     R  W       test/repo2
     R  W       test/repo3
    @R @W       test/repo4
 @C @R  W       users/jiangxin/.+$
```

设置 GIT_SSH 变量，使之指向新建立的脚本，然后就可以脱离 Pageant 来连接 SSH 协议的 Git 库了。

```
$ export GIT_SSH=~/bin/ssh-jiangxin
```

## 3.4 Windows 下安装和使用 Git（msysGit 篇）

运行在 Cygwin 下的 Git 不直接使用 Windows 的系统调用，而是通过二传手 cygwin1.dll 来进行的。虽然 Cygwin 中的 Git 能够在 Windows 下的 cmd.exe 命令窗口中运行良好，但 Cygwin 下的 Git 并不能被看作是 Windows 下的原生程序。相比 Cygwin 下的 Git，msysGit 才是原生的 Windows 程序，msysGit 下运行的 Git 是直接通过 Windows 的系统调用来运行的。

msysGit 名字前面的四个字母来源于 MSYS[①] 项目。MSYS 项目源自于 MinGW[②]

① *http://www.mingw.org/wiki/msys*

② *http://www.mingw.org/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


(Minimalist GNU for Windows，最简 GNU 工具集)，通过增加一个 bash 提供的 shell 环境及其他相关的工具软件组成了一个最简系统（Minimal SYStem），简称 MSYS。利用 MinGW 提供的工具和 Git 针对 MinGW 的一个分支版本，在 Windows 平台为 Git 编译出一个原生应用，结合 MSYS 就组成了 msysGit。

### 3.4.1 安装 msysGit

安装 msysGit 非常简单，访问 msysGit 的项目主页（*http://code.google.com/p/msysgit/*），下载 msysGit。最简单的方式是下载名为 `Git-<VERSION>-preview<DATE>.exe` 的软件包，如 `Git-1.7.3.1-preview20101002.exe` 。如果您有时间和耐心，想观察 Git 是如何在 Windows 上被编译为原生应用的，也可以下载带 `msysGit-fullinstall-` 前缀的软件包。

点击安装程序（如 `Git-1.7.3.1-preview20101002.exe` ）开始安装，如图 3-18 所示。




图 3-18 启动 msysGit 安装

默认安装到 `C:\Program Files\Git` 目录中，如图 3-19 所示。

在安装过程中会询问是否修改环境变量，如图 3-20 所示。默认选择“Use Git Bash Only”，即只在 msysGit 提供的 shell 环境（类似 Cygwin）中使用 Git，不修改环境变量。注意，如果选择最后一项，会将 msysGit 所有的可执行程序全部加入 Windows 的 PATH 路径中，有的命令会覆盖 Windows 相同文件名的程序（如 `find.exe` 和 `sort.exe`）。而且，如果选择最后一项，还会为 Windows 添加 HOME 环境变量，如果安装有 Cygwin，Cygwin 就会受到 msysGit 引入的 HOME 环境变量的影响（参见前面 3.3.3 节的相关讨论）。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



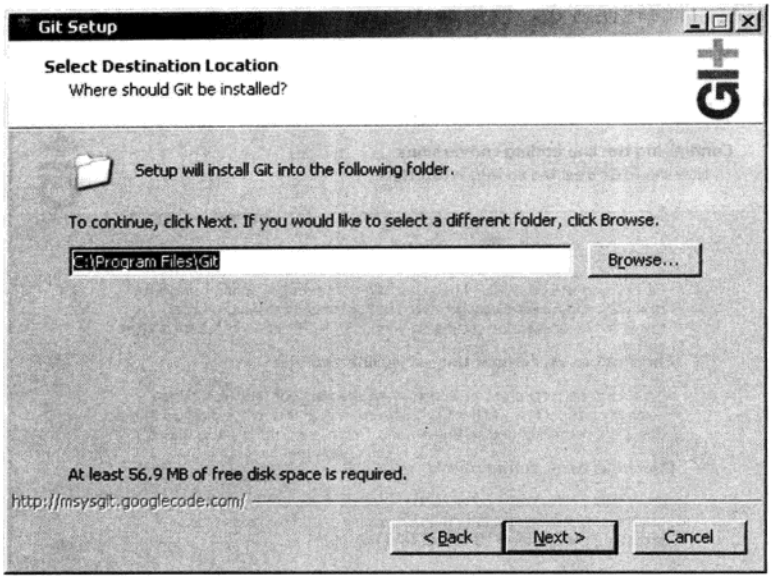




图 3-19 选择 msysGit 的安装目录

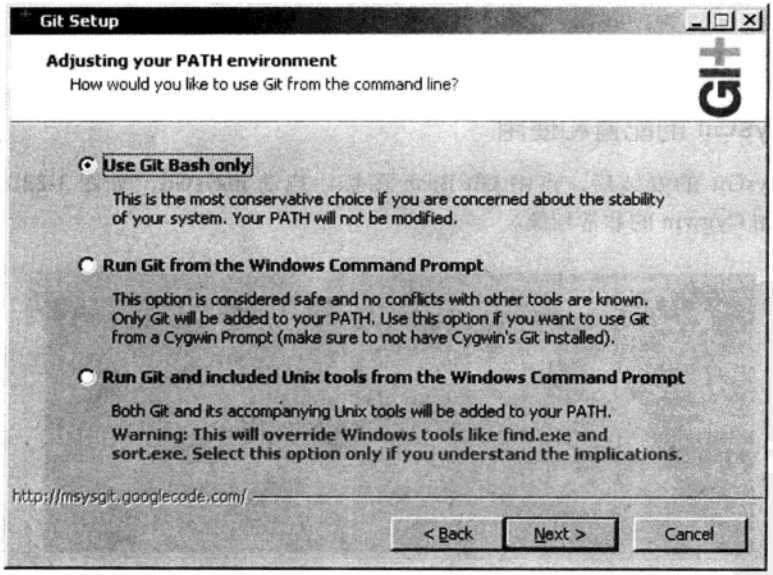



图 3-20 是否修改系统的环境变量

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


还会询问换行符的转换方式，使用默认设置就可以了，如图 3-21 所示。关于换行符转换的内容，请参见本书第 8 篇的相关章节。

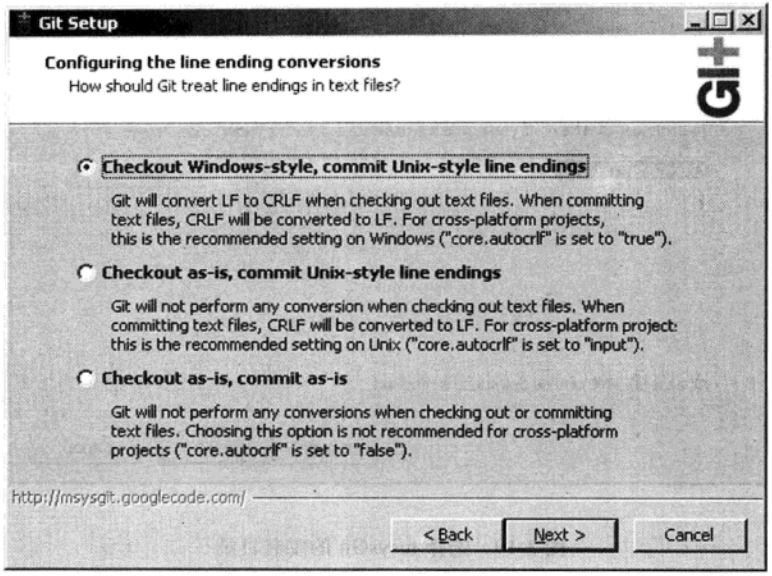



图 3-21 换行符转换方式

根据提示完成 msysGit 的安装。

## 3.4.2 msysGit 的配置和使用

完成 msysGit 的安装后，点击 Git Bash 图标，启动 msysGit，如图 3-22。会发现 Git Bash 的界面和 Cygwin 的非常相像。

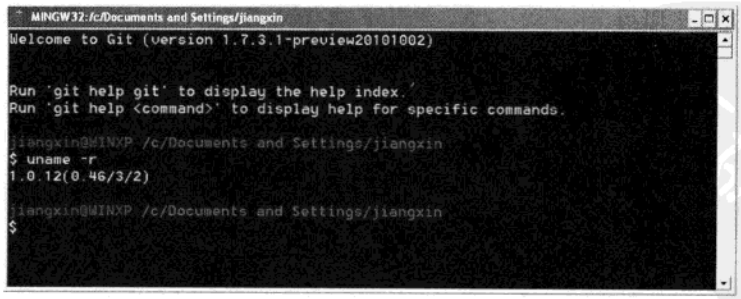



图 3-22 启动 Git Bash

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



#### 1. 如何访问 Windows 的盘符

在 msysGit 下访问 Windows 的各个盘符要比 Cygwin 简单，直接通过“/c”即可访问 Windows 的 C 盘，用“/d”即可访问 Windows 的 D 盘。

```
$ ls -ld /c/Windows
drwxr-xr-x  233 jiangxin Administ         0 Jan 31 00:44 /c/Windows
```

msysGit 的根目录实际上就是 msysGit 的安装目录，如“C:\Program Files\Git”。

#### 2. 命令行补齐和忽略文件大小写

msysGit 默认已经安装并启用了 Git 的命令行补齐功能，是通过在文件 /etc/profile 中加载相应的脚本实现的。

```
. /etc/git-completion.bash
```

msysGit 还支持在命令行补齐时忽略文件名的大小写，这是因为 msysGit 已经在配置文件 /etc/inputrc 中包含了下列的设置：

```
set completion-ignore-case on
```

### 3.4.3 msysGit 中 shell 环境的中文支持

在介绍 Cygwin 时曾经提到过，msysGit 的 shell 环境的中文支持情况与老版本的 Cygwin[①]类似，需要配置才能够录入中文和显示中文。

#### 1. 中文录入问题

默认安装的 msysGit 的 shell 环境中无法输入中文。为了能在 shell 界面中输入中文，需要修改配置文件 /etc/inputrc，增加或修改相关的配置如下：

```
# disable/enable 8bit input
set meta-flag on
set input-meta on
set output-meta on
set convert-meta off
```

关闭 Git Bash 再重启，就可以在 msysGit 的 shell 环境中输入中文了。

```
$ echo 您好
您好
```

#### 2. 分页器中文输出问题

对 /etc/inputrc 进行正确的配置之后就能够在 shell 环境下输入中文了，但是执行下面的命令时会显示乱码。这显然是 less 分页器命令导致的问题。

```
$ echo 您好 | less
<C4><FA><BA><C3>
```

① MSYS 是源自于 Cygwin1.3 的轻量级分支（参考 *http://www.mingw.org/*）。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


通过管道符调用分页器命令 less 后，原本的中文输出变成了乱码显示。因为 Git 使用了大量的 less 命令作为分页器，这导致 Git 的很多命令的输出都出现了中文乱码的问题。之所以 less 命令会导致出现乱码，是因为该命令没有把中文当作正常的字符，可以通过设置 LESSCHARSET 环境变量将 UTF-8 编码字符视为正常的字符，于是中文就能正常显示了。下面的操作可以使 less 分页器中的中文正常显示：

```
$ export LESSCHARSET=utf-8
$ echo 您好 | less
您好
```

编辑配置文件 /etc/profile，将对环境变量 LESSCHARSET 的设置加入其中，以便于 msysGit 的 shell 环境启动时就加载。

```
declare -x LESSCHARSET=utf-8
```

### 3. ls 命令显示中文文件名

最常用的用于显示目录和文件名列表的命令 ls 在显示中文文件名的时候也有问题。下面的命令创建了一个中文名称的文件，文件内容中的中文在显示时没有问题，但是文件名却显示为一串问号。

```
$ echo 您好 > 您好.txt

$ cat \*.txt
您好

$ ls \*.txt
????.txt
```

实际上，只要在 ls 命令后添加参数 --show-control-chars 即可正确显示中文：

```
$ ls --show-control-chars *.txt
您好.txt
```

为方便起见，可以为 ls 命令设置一个别名，这样就不必在输入 ls 命令时输入长长的参数了。

```
$ alias ls="ls --show-control-chars"

$ ls \*.txt
您好.txt
```

将上面的 alias 命令添加到配置文件 /etc/profile 中，可实现在每次运行 Git Bash 时自动加载。

## 3.4.4 msysGit 中 Git 的中文支持

非常遗憾的是，msysGit 中的 Git 对中文支持不如 Cygwin 中的 Git，msysGit 中的 Git

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




对中文的支持情况与前面讨论过的使用了 GBK 字符集的 Linux 环境下的 Git 相当。

（1）使用未经配置的 msysGit 提交，如果提交说明中包含中文，从 Linux 平台或其他 UTF-8 字符集平台上查看提交说明时会显示为乱码。

（2）同样，从 Linux 平台或其他使用 UTF-8 字符集平台进行的提交，若提交说明包含中文，在未经配置的 msysGit 中也会显示为乱码。

（3）如果使用 msysGit 向版本库中添加带有中文文件名的文件，在 Linux（或其他 UTF-8）平台检出文件名时会显示为乱码，反之亦然。

（4）不能创建带有中文字符的引用（里程碑和分支等）。

如果希望版本库中出现使用中文文件名的文件，最好不要使用 msysGit，而应该使用 Cygwin 下的 Git。如果只是想在提交说明中使用中文，对 msysGit 进行一定的设置后还是可以实现的。

为了解决提交说明显示为乱码的问题，msysGit 要为 Git 设置参数 `i18n.logOutputEncoding`，将提交说明的输出编码设置为 gbk：

```
$ git config --system i18n.logOutputEncoding gbk
```

Git 在提交时并不会对提交说明进行从 GBK 字符集到 UTF-8 字符集的转换，但是可以在提交说明中标注所使用的字符集。因此，如果在非 UTF-8 字符集的平台中录入中文，需要用下面的指令设置录入提交说明的字符集，以便在 commit 对象中嵌入正确的编码说明。

```
$ git config --system i18n.commitEncoding gbk
```

同样，为了让带有中文文件名的文件在工作区状态输出、查看历史更改概要，以及在补丁文件中能够正常显示，要为 Git 设置 `core.quotepath` 配置变量，将其设置为 `false`。但是要注意，如果在 msysGit 中添加文件名包含中文的文件，就只能在 msysGit 环境中正确显示，而在其他环境（如 Linux、Mac OS X、Cygwin）中文件名会显示为乱码。

```
$ git config --system core.quotepath false
$ git status -s
?? 说明.txt
```

**注意** 如果同时安装了 Cygwin 和 msysGit（可能配置了相同的用户主目录），或者因为中文支持问题而需要单独为 TortoiseGit 准备一套 msysGit 时，为了保证不同的 msysGit 之间，以及与 Cygwin 之间的配置互不影响，需要在配置 Git 环境时使用 `--system` 参数。这是因为不同的 msysGit 安装及 Cygwin 的系统级配置文件的位置不同，但是用户级配置文件的位置却可能重合。

### 3.4.5 使用 SSH 协议

msysGit 软件包包含的 ssh 命令和 Linux 下的 ssh 命令没有什么区别，也提供 ssh-keygen 命令管理 SSH 公钥 / 私钥对。在使用 msysGit 的 ssh 命令时，没有遇到 Cygwin 中的 ssh 命

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


令（版本号：5.7p1-1）不稳定的问题，即 msysGit 下的 ssh 命令可以非常稳定地工作。

如果需要与 Windows 更好地整合，希望使用图形化工具管理公钥，也可以使用 PuTTY 提供的 **plink.exe** 作为 SSH 客户端。关于如何使用 PuTTY，请参见 3.3.5 节中 Cygwin 和 PuTTY 整合的相关内容。

### 3.4.6 TortoiseGit 的安装和使用

TortoiseGit 可以让 Git 与 Windows 资源管理器进行整合，为 Git 提供了图形化操作界面。像其他 Tortoise 系列产品（TortoiseCVS、TortoiseSVN）一样，在资源管理器中显示的 Git 工作区目录和文件的图标附加了标识版本控制状态的图像，这样可以非常直观地看到哪些文件被更改了并需要提交。通过扩展后的右键菜单，可以非常方便地在资源管理器中操作 Git 版本库。

TortoiseGit 是对 msysGit 命令行的封装，因此需要先安装 msysGit。安装 TortoiseGit 非常简单，访问网站 *http://code.google.com/p/tortoisegit/*，下载安装包，然后根据提示完成安装。

安装过程中会询问要使用的 SSH 客户端，如图 3-23，默认使用内置的 TortoisePLink（来自 PuTTY 项目）作为 SSH 客户端。

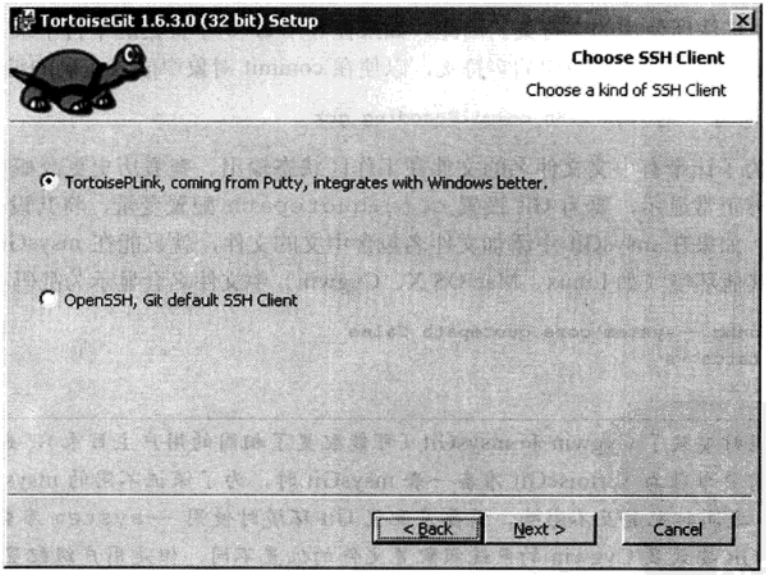



图 3-23 选择 SSH 客户端

TortoisePLink 和 TortoiseGit 的整合性更好，可以直接通过对话框设置 SSH 私钥（PuTTY 格式），而无须再到字符界面去配置 SSH 私钥和其他配置文件。如果安装过程中选择了 OpenSSH，可以在安装完之后通过 TortoiseGit 的设置对话框重新选择 TortoisePLink 作为默认 SSH 客户端，如图 3-24 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

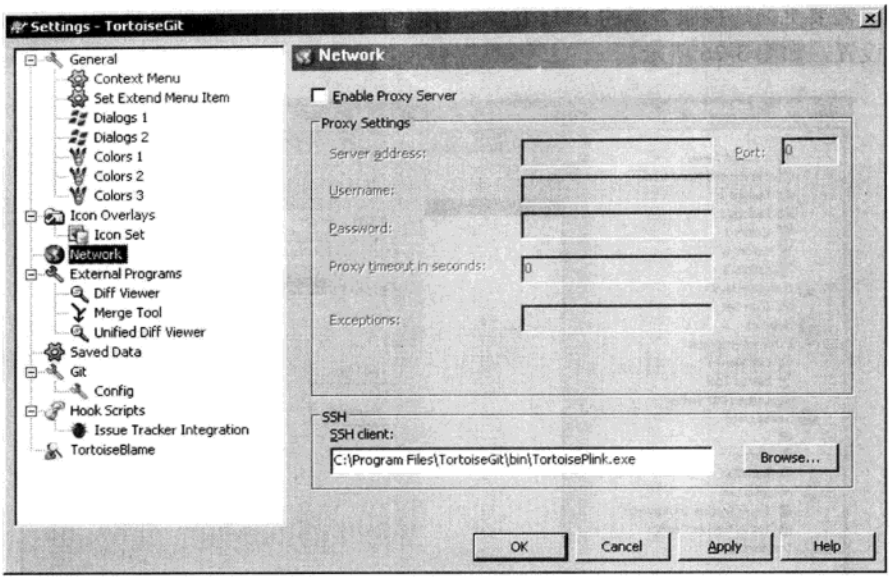




图 3-24 更改默认 SSH 客户端

当使用 TortoisePLink 作为默认的 SSH 客户端后，在执行克隆操作时，可以在 TortoiseGit 操作界面中选择一个 PuTTY 格式的私钥文件进行认证，如图 3-25 所示。

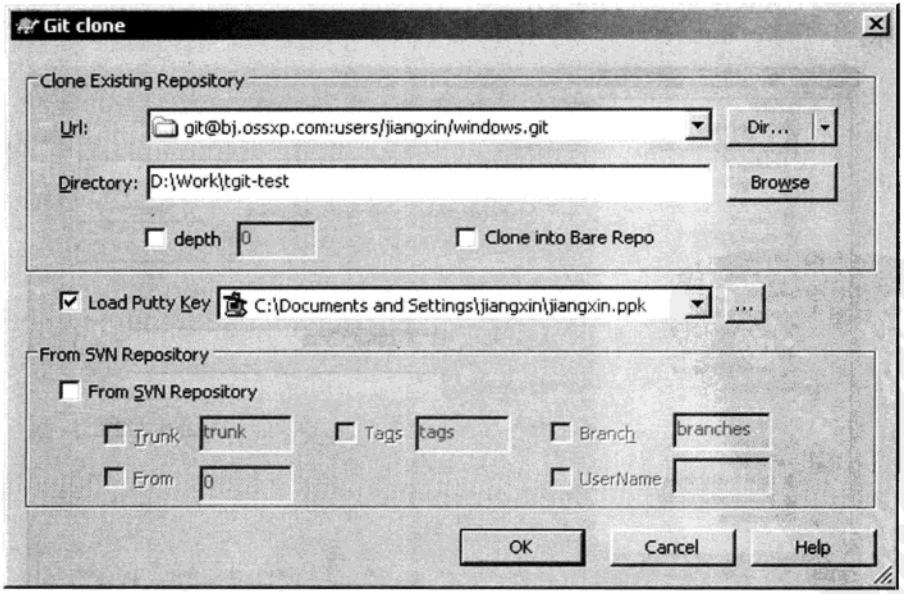



图 3-25 克隆操作选择 PuTTY 格式的私钥文件

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


如果需要更换连接服务器的 SSH 私钥，可以通过 Git 远程服务器配置界面对私钥文件进行重新设置，如图 3-26 所示。

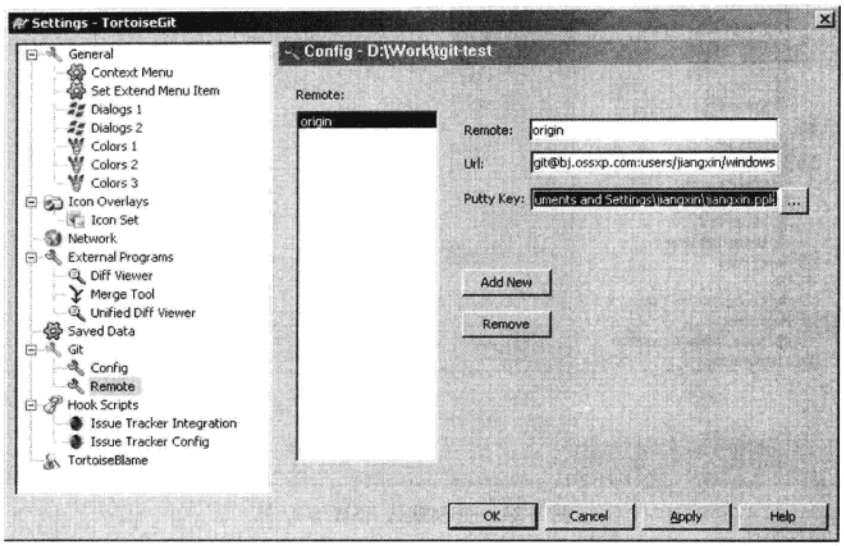



图 3-26 更换连接远程 SSH 服务器的私钥

如果系统中安装有多个 msysGit 的拷贝，可以通过 TortoiseGit 的配置界面进行选择，如图 3-27 所示。

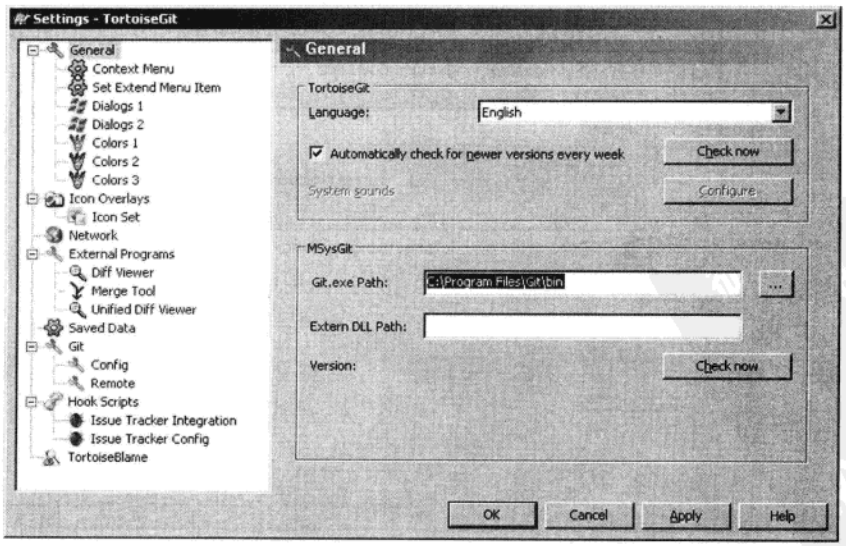



图 3-27 配置 msysGit 的可执行程序位置

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




### 3.4.7 TortoiseGit 的中文支持

虽然 TortoiseGit 在底层调用了 msysGit，但是 TortoiseGit 的中文支持与 msysGit 是有区别的。前面在介绍 msysGit 的中文支持时所进行的配制会破坏 TortoiseGit。

TortoiseGit 在提交时会将提交说明转换为 UTF-8 字符集，因此无须对 `i18n.commitEncoding` 变量进行设置。相反，如果将 `i18n.commitEncoding` 设置为 gbk 或其他字符集，则在提交对象中会包含错误的编码设置，有可能会给提交说明的显示带来麻烦。

TortoiseGit 在显示提交说明时认为所有的提交说明都是 UTF-8 编码，会转换为合适的 Windows 本地字符集显示，而无须设置 `i18n.logOutputEncoding` 变量。因为当前版本的 TortoiseGit 没有对提交对象中的 encoding 设置进行检查，所以使用 GBK 字符集的提交说明中的中文不能正常显示。

因此，如果需要同时使用 msysGit 的文字界面 Git Bash 和 TortoiseGit，而且需要在提交说明中使用中文，可以安装两套 msysGit，并确保 TortoiseGit 关联的 msysGit 没有对 `i18n.commitEncoding` 进行设置。

与 msysGit 一样，TortoiseGit 对使用中文命名的文件和目录的支持，也存在缺陷，因此应当避免在 msysGit 和 TortoiseGit 中添加用中文命名的文件和目录，如果确实需要，可以使用 Cygwin。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第 2 篇
# Git 独奏

从本篇开始，我们就真正地进入到 Git 的学习中。Git 有着陡峭的学习曲线，对有其他版本控制工具使用经验的老手也不例外，因为他们有可能会按照在其他版本控制系统中遗留的习惯去操作 Git，努力地在 Git 中寻找对应物，最终会因为 Git 的“别扭”而放弃使用，这也是我的亲身经历。

Git 的“别扭”部分源自于传统的集中式版本控制系统与以 Git 为代表的分布式版本控制系统在理念上的巨大差异，部分是因为其设计者 Linus Torvalds 对 Git 独特的创新式设计。“你应该了解真相，真相会使你自由”[①]，因此本书会将 Git 的设计原理渗透到每一个章节，让您通过不断地实践、思考、再实践来循序渐进地掌握 Git。

本篇暂时不会涉及团队如何使用 Git 的内容，而是先从个人的角度去探讨如何用好 Git。本篇是全书最重要的部分，是下一步进行团队协作必需的知识准备，也是理解全书其余各部分内容的基础。到本篇的结尾时，我们会发现通过“Git 独奏”也可以演绎出美妙的“乐曲”。



① 圣经《约翰福音 8:32》。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 4 章 Git 初始化

## 4.1 创建版本库及第一次提交

您当前使用的 Git 是 1.5.6 版或更高的版本吗？通过如下操作来查看一下您的 Git 版本：

```
$ git --version
git version 1.7.4
```

Git 是一个活跃的项目，仍在不断地进化之中，不同版本的 Git 功能不尽相同。本书对 Git 的介绍涵盖了 1.5.6 到 1.7.4 版本，这也是目前 Git 的主要版本。如果您使用的 Git 版本低于 1.5.6，那么请升级到 1.5.6 或更高的版本。本书示例使用的是 1.7 版本的 Git，我们会尽可能地指出那些与低版本不兼容的命令及参数。

在开始 Git 之旅之前，我们需要设置一下 Git 的配置变量，这是一次性的工作。即这些设置会在全局文件（用户主目录下的 .gitconfig）或系统文件（如 /etc/gitconfig）中做永久的记录。

（1）告诉 Git 当前用户的姓名和邮件地址，配置的用户名和邮件地址将在版本库提交时用到。

注意，不要照抄照搬下面的两条命令，而是用您自己的用户名和邮件地址代替这里的用户名和邮件地址，否则您的劳动成果（提交内容）可就要算到我的头上了。

```
$ git config --global user.name "Jiang Xin"
$ git config --global user.email jiangxin@ossxp.com
```

（2）设置一些 Git 别名，以便可以使用更为简洁的子命令。

例如：输入 `git ci` 即相当于 `git commit`，输入 `git st` 即相当于 `git status`。

如果拥有系统管理员权限（例如通过执行 `sudo` 命令获取管理员权限），希望注册的命令别名能够被所有用户使用，可以执行如下命令：

```
$ sudo git config --system alias.st status
$ sudo git config --system alias.ci commit
$ sudo git config --system alias.co checkout
$ sudo git config --system alias.br branch
```

也可以运行下面的命令，只在本用户的全局配置中添加 Git 命令别名：

```
$ git config --global alias.st status
$ git config --global alias.ci commit
$ git config --global alias.co checkout
$ git config --global alias.br branch
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




（3）在 Git 命令输出中开启颜色显示。

```
$ git config --global color.ui true
```

Git 的所有操作，包括创建版本库等管理操作用 git 一个命令即可完成，不像其他版本控制系统（如 Subversion），与管理相关的操作要使用另外的命令（如 svnadmin）。创建 Git 版本库，可以直接进入到工作目录中，通过执行 git init 命令完成版本库的初始化。

下面就从一个空目录开始初始化版本库，将这个版本库命名为“DEMO 版本库”，这个 DEMO 版本库将贯穿本篇始终。为了方便说明，我们使用名为 /path/to/my/workspace 的目录作为个人的工作区根目录，可以在磁盘中创建该目录并设置正确的权限。

首先建立一个新的工作目录，进入该目录后，执行 git init 创建版本库。

```
$ cd /path/to/my/workspace
$ mkdir demo
$ cd demo
$ git init
Initialized empty Git repository in /path/to/my/workspace/demo/.git/
```

实际上，如果 Git 的版本是 1.6.5 或更新的版本，可以在 git init 命令的后面直接输入目录名称，自动完成目录的创建。

```
$ cd /path/to/my/workspace
$ git init demo
Initialized empty Git repository in /path/to/my/workspace/demo/.git/
$ cd demo
```

从上面版本库初始化后的输出中可以看到，git init 命令在工作区创建了隐藏目录 .git 。

```
$ ls -aF
./  ../  .git/
```

这个隐藏的 .git 目录就是 Git 版本库（又叫仓库，repository）。

.git 版本库所在的目录为 /path/to/my/workspace/demo，它被称为工作区，目前工作区除了包含一个隐藏的 .git 版本库目录外空无一物。

下面为工作区中加点料：在工作区中创建一个文件 welcome.txt，内容就是一行“Hello.”。

```
$ echo "Hello." > welcome.txt
```

为了将这个新建立的文件添加到版本库，需要执行下面的命令：

```
$ git add welcome.txt
```

注意，到这里还没有完。Git 和大部分其他版本控制系统一样，都需要再执行一次提交操作，对于 Git 来说就是执行 git commit 命令完成提交。在提交过程中需要输入提交说明，这个要求对于 Git 来说是强制性的，不像其他很多版本控制系统（如 CVS 和

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Subversion）那样接受空白的提交说明。当 Git 提交时，如果不在命令行提供提交说明（使用 -m 参数），Git 会自动打开一个编辑器，要求您在其中输入提交说明，输入完毕后保存并退出。需要说明的是，读者要在一定程度上掌握 vim 或 emacs（Linux 下常用的两种编辑器）的编辑技巧，否则保存和退出也会成为问题。

下面进行提交。为了说明方便，使用 -m 参数直接给出了提交说明。

```
$ git ci -m "initialized."
[master (root-commit) 78cde45] initialized.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 welcome.txt
```

从上面的命令及输出中可以看出：

- 命令 git ci 实际上相当于 git commit，这是因为之前为 Git 设置了命令别名。
- 通过 -m 参数设置提交说明为："initialized."。
- 从命令输出的第一行可以看出，此次提交是提交在名为 master 的分支上，且是该分支的第一个提交（root-commit），提交 ID 为 78cde45[①]。
- 从命令输出的第二行可以看出，此次提交修改了一个文件，包含一行的插入。
- 从命令输出的第三行可以看出，此次提交创建了新文件 welcome.txt。

## 4.2 思考：为什么工作区根目录下有一个 .git 目录

Git 及其他分布式版本控制系统（如 Mercurial/Hg、Bazaar）的一个共同的显著特点是，版本库位于工作区的根目录下。对于 Git 来说，版本库位于工作区根目录下的 .git 目录中，且仅此一处，在工作区的子目录下则没有任何其他跟踪文件或目录。Git 的这种设计要比 CVS 和 Subversion 等传统的集中式版本控制工具方便多了。

传统的集中式版本控制系统的版本库和工作区是分开的，甚至是在不同的主机上，因此必须建立工作区和版本库的对应。下面来看看版本控制系统的前辈们是如何建立工作区和版本库的跟踪的，通过其各自设计的优缺点，我们会更深刻地体会到 Git 实现的必要和巧妙。

对于 CVS 而言，工作区的根目录及每一个子目录下都有一个 CVS 目录，CVS 目录中包含几个配置文件，建立了对版本库的追踪。如 CVS 目录下的 Entries 文件记录了从版本库检出到工作区的文件的名称、版本和时间戳等，通过时间戳的对比可快速扫描工作区文件的改动。这样设计的好处是，可以将工作区移动到任何其他目录中，而工作区和版本控制服务器的映射关系保持不变，这样工作区依然能够正常工作。甚至还可以将工作区的某个子目录移动到其他位置，形成新的工作区，在新的工作区下仍然可以完成版本控制相关的操作。但

① 大家实际操作中看到的 ID 肯定和这里写的不一样，具体原因会在后面的 6.1 节中予以介绍。如果碰巧您的操作也显示出了同样的 ID（78cde45），那么我建议您赶紧去买一张彩票。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


是缺点也很多，例如，如果工作区文件修改了，因为没有原始文件做比对，所以向服务器提交修改时只能对整个文件进行传输，而不能仅传输文件的改动部分，导致从客户端到服务器的网络传输效率降低。还有一个风险是信息泄漏，例如，Web 服务器的目录下如果包含了 CVS 目录，黑客就可以通过扫描 CVS/Entries 文件得到目录下的文件列表，从而获得他们想要的信息。

对于 Subversion 来说，工作区的根目录和每一个子目录下都有一个 .svn 目录。目录 .svn 中不仅包含了类似于 CVS 的跟踪目录下的配置文件，而且包含了当前工作区下每一个文件的拷贝。这些文件的原始拷贝让某些 SVN 子命令可以脱离版本库执行。而且，当由客户端向服务器提交时，可以只提交改动的部分，因为改动的文件可以与文件的原始拷贝进行差异比较。但是，这么做也有缺点，除了会像 CVS 那样因为引入 CVS 跟踪目录而有可能造成信息泄漏外，还会加倍占用工作区的空间。此外，当在工作区目录下针对文件内容进行搜索时，会因为 .svn 目录下文件的原始拷贝导致搜索结果加倍，使搜索结果混乱。

有的版本控制系统在工作区根本就没有任何跟踪文件，例如，某款商业的版本控制软件（就不点名了）的工作区就非常干净，没有任何配置文件和配置目录。但是，这样的设计更糟糕，因为它实际上是通过服务器端建立文件跟踪，在服务器端的数据库中保存了一个包含如下信息的表格：哪个客户端，在哪个本地目录检出了哪个版本的版本库文件。这样做的后果是，如果客户端将工作区移动或改名，就会导致文件的跟踪状态丢失，从而出现文件状态未知的问题。此外，客户端操作系统重装也会导致文件跟踪状态丢失。

Git 这种将版本库放在工作区根目录下的设计使得所有的版本控制操作（除了与其他远程版本库之间的互操作）都在本地即可完成，不像 Subversion 只有寥寥无几的几个命令脱离网络执行。而且，Git 没有 CVS 和 Subversion 中存在的安全泄漏问题（只要保护好 .git 目录），也不会像 Subversion 那样在搜索本地文件时出现搜索结果混乱的问题。甚至，Git 还提供了一条 git grep 命令来更好地搜索工作区的文件内容，例如，我们可以在本书的 Git 库中执行下面的命令来搜索版本库中的文件内容：

```
$ git grep "工作区文件内容搜索"
02-git-solo/010-git-init.rst:'git grep' 命令来更好地搜索工作区的文件内容。
```

Git 将版本库（.git 目录）放在工作区根目录下，那么 Git 的相关操作一定要在工作区根目录下执行吗？换句话说，当工作区中包含子目录，并在子目录中执行 Git 命令时，如何定位版本库呢？

实际上，当在 Git 工作区的某个子目录下执行操作的时候，会在工作区目录中依次向上递归查找 .git 目录，找到的 .git 目录就是工作区对应的版本库，.git 所在的目录就是工作区的根目录，文件 .git/index 记录了工作区文件的状态（实际上是暂存区的状态）。

例如，在非 Git 工作区执行 git 命令时会因为找不到 .git 目录而报错。

```
$ cd /path/to/my/workspace/
$ git status
fatal: Not a git repository (or any of the parent directories): .git
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


如果用 strace[①]命令去跟踪执行 git status 命令时的磁盘访问，会看到沿目录依次向上递归的过程。

```
$ strace -e 'trace=file' git status
...
getcwd("/path/to/my/workspace", 4096)           = 14
stat(".", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
stat(".git", 0x7fffdf1288d0)                    = -1 ENOENT (No such file or directory)
access(".git/objects", X_OK)                    = -1 ENOENT (No such file or directory)
access("./objects", X_OK)                       = -1 ENOENT (No such file or directory)
stat("..", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
chdir("..")                                     = 0
stat(".git", 0x7fffdf1288d0)                    = -1 ENOENT (No such file or directory)
access(".git/objects", X_OK)                    = -1 ENOENT (No such file or directory)
access("./objects", X_OK)                       = -1 ENOENT (No such file or directory)
stat("..", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
chdir("..")                                     = 0
stat(".git", 0x7fffdf1288d0)                    = -1 ENOENT (No such file or directory)
access(".git/objects", X_OK)                    = -1 ENOENT (No such file or directory)
access("./objects", X_OK)                       = -1 ENOENT (No such file or directory)
fatal: Not a git repository (or any of the parent directories): .git
```

当在工作区执行 Git 命令时，上面查找版本库的操作总是默默地执行，就好像什么也没有发生一样。那么有什么办法知道 Git 版本库的位置呢？如何才能知道工作区的根目录在哪里呢？可以用 Git 的一个底层命令来实现，具体操作过程如下：

（1）在工作区中建立目录 a/b/c，进入到该目录中。

```
$ cd /path/to/my/workspace/demo/
$ mkdir -p a/b/c
$ cd /path/to/my/workspace/demo/a/b/c
```

（2）显示版本库 .git 目录所在的位置。

```
$ git rev-parse --git-dir
/path/to/my/workspace/demo/.git
```

（3）显示工作区根目录。

```
$ git rev-parse --show-toplevel
/path/to/my/workspace/demo
```

（4）相对于工作区根目录的相对目录。

```
$ git rev-parse --show-prefix
a/b/c/
```

（5）显示从当前目录（cd）后退（up）到工作区的根的深度。

① Mac OS X 可以使用 dtruss 命令。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git rev-parse --show-cdup
../../../
```

传统的集中式版本控制系统的工作区和版本库都是相分离的，像 Git 这样把版本库目录放在工作区是不是太不安全了？

从存储安全的角度上来讲，将版本库放在工作区目录下有点“把鸡蛋装在一个篮子里”的味道。如果忘记了工作区中还有版本库，当直接从工作区的根执行目录删除操作时就会连版本库一并删除，这个风险的确很高。将版本库和工作区拆开似乎更加安全，但是不要忘了之前的讨论，如果将版本库和工作区拆开，就要引入其他机制以便实现版本库对工作区的追踪。

Git 克隆可以降低因为版本库和工作区混杂在一起而导致的版本库被破坏的风险。可以通过克隆操作在本机另外的磁盘 / 目录中建立 Git 克隆，并在工作区有新的提交时，手动或自动地执行向克隆版本库的推送（git push）操作。如果使用网络协议，还可以实现在其他机器上建立克隆，这样就更安全了（双机备份）。对于使用 Git 做版本控制的团队，每个人都是一个备份，因此团队开发中的 Git 版本库更安全，管理员甚至无须顾虑版本库存储的安全问题。

## 4.3 思考：git config 命令的各参数有何区别

在之前出现的 git config 命令中，有的使用了 --global 参数，有的使用了 --system 参数，这两个参数有什么区别吗？执行下面的一系列命令后，您就会明白使用不同参数的 git config 命令实际操作的文件了。

- 执行下面的命令，将打开 /path/to/my/workspace/demo/.git/config 文件进行编辑。

```
$ cd /path/to/my/workspace/demo/
$ git config -e
```

- 执行下面的命令，将打开 /home/jiangxin/.gitconfig（用户主目录下的 .gitconfig 文件）全局配置文件进行编辑。

```
$ git config -e --global
```

- 执行下面的命令，将打开 /etc/gitconfig 系统级配置文件进行编辑。如果 Git 安装在非标准位置，则这个系统级的配置文件也可能是在另外的位置。

```
$ git config -e --system
```

Git 的三个配置文件分别是版本库级别的配置文件、全局配置文件（用户主目录下）和系统级配置文件（/etc 目录下）。其中版本库级别的配置文件的优先级最高，全局配置文件次之，系统级配置文件优先级最低。这样的优先级设置可以让版本库 .git 目录下的 config 文件中的配置覆盖用户主目录下的 Git 环境配置，而用户主目录下的配置也可以覆盖系统的

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Git 配置文件。

执行前面的三个 git config 命令后会看到这三个级别的配置文件的格式和内容，原来 Git 配置文件采用的是 INI 文件格式。示例如下：

```
$ cat /path/to/my/workspace/demo/.git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
```

git config 命令可以用于读取和更改 INI 配置文件的内容。使用只带一个参数的 git config <section>.<key> 命令可用来读取 INI 配置文件中某个配置的键值，例如读取 [core] 小节的 bare 的属性值，可以用如下命令：

```
$ git config core.bare
false
```

如果想更改或设置 INI 文件中某个属性的值也非常简单，命令格式是：git config <section>.<key> <value>。可以用如下操作：

```
$ git config a.b something
$ git config x.y.z others
```

如果打开 .git/config 文件，会看到如下内容：

```
[a]
        b = something

[x "y"]
        z = others
```

对于类似于 [x "y"] 这样的配置小节会在本书第三篇介绍远程版本库的章节中经常遇到。

从上面的介绍中可以看到，使用 git config 命令可以非常方便地操作 INI 文件，实际上可以用 git config 命令操作任何其他的 INI 文件。

- 向配置文件 test.ini 中添加配置。

```
$ GIT_CONFIG=test.ini git config a.b.c.d "hello, world"
```

- 从配置文件 test.ini 中读取配置。

```
$ GIT_CONFIG=test.ini git config a.b.c.d
hello, world
```

后面介绍的 git-svn 和 Gistore 等软件就是使用该技术读写各自专有的配置文件的。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




## 4.4 思考：是谁完成的提交

在本章的一开始先为 Git 设置了全局配置变量 user.name 和 user.email，如果不设置会有什么结果呢？

执行下面的命令，删除 Git 全局配置文件中关于 user.name 和 user.email 的设置：

```
$ git config --unset --global user.name
$ git config --unset --global user.email
```

这样一来，关于用户姓名和邮件的设置都被清空了，执行下面的命令将看不到输出。

```
$ git config user.name
$ git config user.email
```

下面再尝试一次提交，看看提交的过程会有什么不同，以及提交之后显示的提交者是谁？

在下面的命令中使用了 --allow-empty 参数，这是因为如果没有对工作区的文件进行任何修改，Git 默认不会执行提交，使用 --allow-empty 参数后允许执行空白提交，操作如下：

```
$ cd /path/to/my/workspace/demo
$ git commit --allow-empty -m "who does commit?"
[master 252dc53] who does commit?
 Committer: JiangXin <jiangxin@hp.moon.ossxp.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

If the identity used for this commit is wrong, you can fix it with:

    git commit --amend --author='Your Name <you@example.com>'
```

喔，因为没有设置配置变量 user.name 和 user.email，提交后输出乱得一塌糊涂。仔细看看上面的执行 git commit 命令后的输出，原来 Git 提供了详细的帮助来告诉我们如何设置必需的配置变量，以及如何修改之前提交中出现的错误的提交者信息。

如果此时查看版本库的提交日志，会看到有两次提交。

注意，下面的输出与你动手实践时得到的输出肯定有所不同，一方面因为提交时间会不一样，另外一方面因为由 40 位十六进制数字组成的提交 ID 也不可能一样。甚至，本书中凡是您亲自完成的提交，相关的 40 位魔幻般的数字 ID 也都会不一样（原因会在后面的章节中看到）。因此，凡是涉及数字 ID 和本书中的示例不一致的时候，以你自己的数字 ID 为准，本书提供的示例仅供参考，切记。

```
$ git log --pretty=fuller
commit 252dc539b5b5f9683edd54849c8e0a246e88979c
Author:     JiangXin <jiangxin@hp.moon.ossxp.com>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
AuthorDate: Mon Nov 29 10:39:35 2010 +0800
Commit:     JiangXin <jiangxin@hp.moon.ossxp.com>
CommitDate: Mon Nov 29 10:39:35 2010 +0800

    who does commit?

commit 9e8a761ff9dd343a1380032884f488a2422c495a
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Sun Nov 28 12:48:26 2010 +0800
Commit:     Jiang Xin <jiangxin@ossxp.com>
CommitDate: Sun Nov 28 12:48:26 2010 +0800

    initialized.
```

最早的提交（提交ID为9e8a761…），提交者的信息是由之前设置的配置变量user.name和user.email给出的。而最新的提交（提交ID为252dc53…）因为删除了user.name和user.email，提交时Git对提交者的用户名和邮件地址进行了大胆的猜测，这个猜测可能是错的。

为了保证提交时提交者和作者信息的正确性，需要重新恢复user.name和user.email的设置。记住，不要照抄照搬下面的命令，请使用您自己的用户名和邮件地址。

```
$ git config --global user.name "Jiang Xin"
$ git config --global user.email jiangxin@ossxp.com
```

然后执行下面的命令，重新修改最新的提交，改正作者和提交者的错误信息。

```
$ git commit --amend --allow-empty --reset-author
```

说明：

- 参数--amend是对刚刚的提交进行修补，这样就可以改正前面的提交中错误的用户名和邮件地址，而不会产生另外的新提交。
- 参数--allow-empty使得空白提交被允许。之所以这里必须使用此参数是因为要进行的修补提交实际上是一个空白提交。
- 参数--reset-author的含义是将Author（提交者）的ID同步修改，否则只会影响提交者（Commit）的ID。使用此参数也会重置AuthorDate信息。

通过日志可以看到，最新提交的作者和提交者的信息已经改正了。

```
$ git log --pretty=fuller
commit a0c641e92b10d8bccaed1bf84ca80340fdefee6
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Mon Nov 29 11:00:06 2010 +0800
Commit:     Jiang Xin <jiangxin@ossxp.com>
CommitDate: Mon Nov 29 11:00:06 2010 +0800

    who does commit?
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
commit 9e8a761ff9dd343a1380032884f488a2422c495a
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Sun Nov 28 12:48:26 2010 +0800
Commit:     Jiang Xin <jiangxin@ossxp.com>
CommitDate: Sun Nov 28 12:48:26 2010 +0800

    initialized.
```



## 4.5 思考：随意设置提交者姓名，是否太不安全

使用过 CVS 和 Subversion 等集中式版本控制系统的用户都知道，每次提交的时候需要认证，认证成功后，登录 ID 就作为提交者 ID 出现在版本库的提交日志中。很显然，对于 CVS 和 Subversion 这样的版本控制系统而言，很难冒充他人提交。像 Git 这样的分布式版本控制系统，可以随心所欲地设定提交者，这似乎太不安全了。

Git 可以随意设置提交的用户名和邮件地址信息，这是分布式版本控制系统的特性使然，每个人都是自己版本库的主人，很难也没有必要进行身份认证，因而也就无法使用经过认证的用户名作为提交的用户名。

在进行“独奏”的时候，还要强制为自己加上一个“指纹识别”，实在是太没有必要了，但是团队合作时授权就成为必需了。通常，团队协作时会设置一个共享版本库，在团队成员向共享版本库传送（推送）新提交时，会进行用户身份认证并检查授权。一旦用户通过身份认证，一般来说不会对提交中的包含的提交者 ID 做进一步检查，但 Android 项目是个例外。

Android 项目为了更好地使用 Git 实现对代码的集中管理，开发了一套叫作 Gerrit 的审核服务器来管理 Git 提交，对提交者的邮件地址进行审核。例如下面的示例中，在向 Gerrit 服务器推送的时候，提交中的提交者邮件地址为 `jiangxin@ossxp.com`，但是在 Gerrit 中注册用户时使用的邮件地址为 `jiangxin@moon.ossxp.com`。因为两者不匹配，从而导致推送失败。

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 222 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/new/project.git
 ! [remote rejected] master -> master (you are not committer jiangxin@ossxp.com)
error: failed to push some refs to 'ssh://localhost:29418/new/project.git'
```

即使没有使用类似 Gerrit 的服务，作为提交者也不应该随意改变配置变量 `user.name` 和 `user.email` 的设置，因为当多人协同时这样做会给他人造成迷惑，也会给一些项目管理软件造成麻烦。

例如，Redmine 是一款实现需求管理和缺陷跟踪的项目管理软件，可以与 Git 版本库实现整合。Git 的提交可以直接关闭 Redmine 上的 Bug，同时 Git 的提交还可以反映出项目成员的工作进度。Redmine 中的用户（项目成员）用一个 ID 作标识，而 Git 的提交者则用

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


一个包含用户名和邮件地址的字符串，如何将 Redmine 的用户和 Git 的提交者相关联呢？Redmine 提供了一个配置界面用于设置二者之间的映射，如图 4-1 所示。

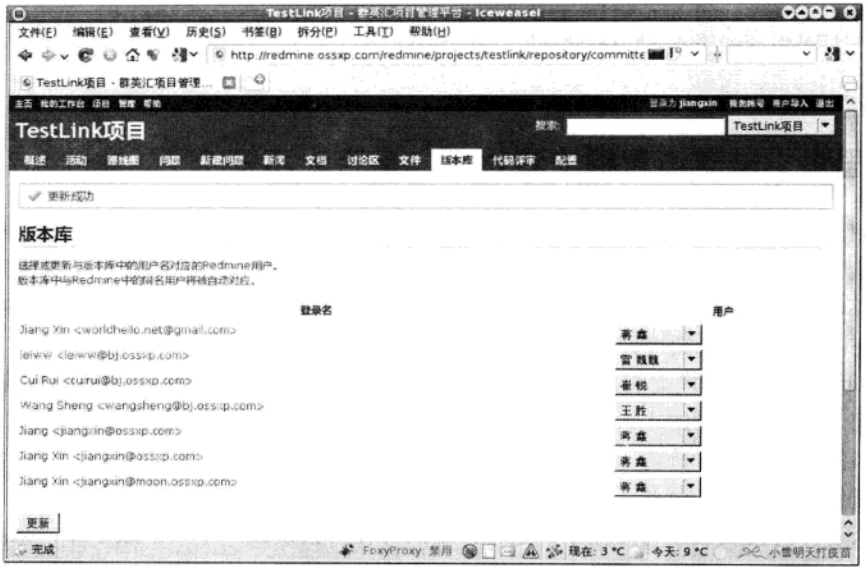



图 4-1 Redmine 中用户 ID 和 Git 提交者关联

显然，如果在 Git 提交时随意变更提交者的姓名和邮件地址，就会破坏 Redmine 软件中设置好的用户对应关系。

## 4.6 思考：命令别名是干什么的

在本章的一开始，我们通过对 alias.ci 等 Git 配置变量的设置为 Git 设置了命令别名。命令别名可以帮助用户解决从其他版本控制系统迁移到 Git 后的使用习惯问题。CVS 和 Subversion 等在提交的时候，一般习惯使用 ci（check in）子命令，在检出的时候则习惯使用 co（check out）子命令。如果 Git 不能提供对 ci 和 co 这类简洁命令的支持，对于拥有其他版本控制系统使用经验的用户来说，Git 的用户体检就会打折扣。幸好聪明的 Git 提供了别名机制，可以满足用户特殊的使用习惯。

本章前面列出的四条别名设置指令，创建的是最常用的几个 Git 别名。实际上别名还可以包含命令参数，例如下面的别名设置指令：

```
$ git config --global alias.ci "commit -s"
```

如上设置后，当使用 git ci 命令提交时，会自动带上 -s 参数，这样会在提交说明中自动添加上包含提交者姓名和邮件地址的签名标识，类似于 Signed-off-by: User

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Name <email@address> 。这对于一些项目（Git、Linux kernel、Android 等）来说是必要甚至是必需的。

不过，本书会尽量避免使用别名命令，以免由于您因为尚未设置别名而造成困扰。

## 4.7 备份本章的工作成果

执行下面的命令，算是对本章工作成果的备份：

```
$ cd /path/to/my/workspace
$ git clone demo demo-step-1
Cloning into demo-step-1...
done.
```





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第5章 Git暂存区

上一章主要学习了三个命令：git init、git add 和 git commit，这三个命令可以说是版本库创建的三部曲。同时还通过对几个问题的思考了解了 Git 版本库在工作区中的布局，Git 三个等级的配置文件及 Git 的别名命令等内容。

在上一章的实践中，DEMO 版本库经历了两次提交，可以用 git log 查看提交日志（附加的 --stat 参数可以看到每次提交的文件变更统计）。

```
$ cd /path/to/my/workspace/demo
$ git log --stat
commit a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Mon Nov 29 11:00:06 2010 +0800

    who does commit?

commit 9e8a761ff9dd343a1380032884f488a2422c495a
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Sun Nov 28 12:48:26 2010 +0800

    initialized.

 welcome.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

可以看到第一次（最早的）提交对文件 welcome.txt 有一行变更，而第二次（最新的）提交因为是使用 --allow-empty 参数进行的一次空提交，所以提交说明中看不到任何对实质内容的修改。

下面我们将仍在这个工作区继续新的实践和学习，以掌握 Git 的一个最重要概念：暂存区。

## 5.1 修改不能直接提交吗

首先更改 welcome.txt 文件，在这个文件后面追加一行。可以使用下面的命令实现内容的追加：

```
$ echo "Nice to meet you." >> welcome.txt
```

这时可以通过执行 git diff 命令看到修改后的文件与版本库中的文件的差异。（实际上这句话有问题，与本地比较的不是版本库中的文件，而是一个中间状态的文件。）

```
$ git diff
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
diff --git a/welcome.txt b/welcome.txt
index 18832d3..fd3c069 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
 Hello.
+Nice to meet you.
```

对差异输出是不是很熟悉？在之前介绍版本库的“黑暗的史前时代”时，曾经展示了 diff 命令的输出，两者的格式是一样的。

既然文件修改了，那么就提交吧。提交能够成功吗？

```
$ git commit -m "Append a nice line."
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

提交成功了吗？好像没有！

提交没有成功的证据：

（1）先来看看提交日志，如果提交成功，应该有新的提交记录出现。

下面使用了精简输出来显示日志，以便更简洁和清晰地看到提交的历史。从其中能够看出版本库中只有两个提交，都是在上一章的实践中完成的。也就是说，刚才针对修改文件的提交没有成功！

```
$ git log --pretty=oneline
a0c641e92b10d8bcca1ed1bf84ca80340fdefee6 who does commit?
9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

（2）执行 git diff 可以看到与之前相同的差异输出，这也说明了之前的提交没有成功。

（3）执行 git status 查看文件状态，可以看到文件处于修改状态，而且 git status 命令的输出和 git commit 提交失败的输出信息完全一样！

（4）对于习惯了像 CVS 和 Subversion 那样精简的状态输出的用户，可以在执行 git status 时附加上 -s 参数，显示精简格式的状态输出。

```
$ git status -s
 M welcome.txt
```

提交为什么会失败呢？再回过头来仔细看看刚才 git commit 命令提交失败后的输出：

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
#    (use "git checkout -- <file>..." to discard changes in working directory)
#
#        modified:    welcome.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

把它翻译成中文则是：

```
# 位于您当前工作的分支 master 上
# 下列修改还没有加入到提交任务（提交暂存区，stage）中，不会被提交：
#    （使用 "git add <file>..." 命令后，改动就会加入到提交任务中，
#      要在下一次提交操作时才被提交）
#    （使用 "git checkout -- <file>..." 命令，工作区中当前您不打算
#      提交的修改会被彻底清除！！！）
#
#        已修改：    welcome.txt
#
警告：提交任务是空的噢，您不要再搅扰我啦 （除非使用
      "git add" 和/或 "git commit -a" 命令）
```

也就是说，需要对修改的 welcome.txt 文件执行 git add 命令，将修改的文件添加到“提交任务”中，然后才能提交！

这个行为真的很奇怪，对于其他版本控制系统来说执行 add 操作是向版本库中添加新文件用的，修改的文件（已被版本控制跟踪的文件）在下次提交时会直接被提交。Git 的这个古怪的行为会在下面的介绍中得到解释，大家会逐渐习惯并喜欢 Git 的这个设计。

好了，现在就将修改的文件“添加”到提交任务中吧：

```
$ git add welcome.txt
```

现在再执行一些 Git 命令，看看当执行完“添加”操作后，Git 库发生了什么变化：

- 执行 git diff 没有输出，难道是被提交了？可是只是执行了“添加”到提交任务的操作，相当于一个“登记”的命令，并没有执行提交哇？

```
$ git diff
```

- 这时如果与 HEAD（当前版本库的头指针）或 master 分支（当前工作分支）进行比较，就会发现有差异。这个差异才是正常的，因为尚未真正提交嘛。

```
$ git diff HEAD
diff --git a/welcome.txt b/welcome.txt
index 18832d3..fd3c069 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
 Hello.
+Nice to meet you.
```

- 执行 git status 命令，状态输出和之前的不一样了。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   welcome.txt
#
```

❑ 再对新的 Git 状态输出进行翻译：

```
$ git status
# 位于分支 master 上
# 下列修改将被提交：
#   （如果你后悔了，可以使用 "git reset HEAD <file>..." 命令
#    将下列改动撤出提交任务（提交暂存区，stage），否则执行提交命令
#    可真的要提交喽）
#
#       已修改：   welcome.txt
#
```

不得不说，Git 太人性化了，它把各种情况下可能使用到的命令都告诉给用户了，虽然这显得有点啰嗦。如果不要这么啰嗦，可以像下面这样用简洁的方式显示状态：

```
$ git status -s
M  welcome.txt
```

上面的精简状态输出与执行 git add 之前的精简状态输出相比，有细微的差别，发现了吗？

❑ 虽然都是 M（Modified）标识，但是位置不一样。在执行 git add 命令之前，这个 M 位于第二列（第一列是一个空格），在执行完 git add 之后，字符 M 位于第一列（第二列是空白）。

❑ 位于第一列的字符 M 的含义是：版本库中的文件与处于中间状态——提交任务（提交暂存区，stage）中的文件相比有改动。

❑ 位于第二列的字符 M 的含义是：工作区当前的文件与处于中间状态——提交任务（提交暂存区，stage）中的文件相比有改动。

是不是还有一些不明白？为什么 Git 的状态输出中提示了那么多让人不解的命令？为什么存在一个提交任务的概念而又总是把它叫作暂存区（stage）？不要紧，马上就会专题讲述“暂存区”的概念。当了解了 Git 版本库的设计原理之后，理解相关 Git 命令就易如反掌了。

这时如果直接提交（git commit），加入提交任务的 welcome.txt 文件的更改就会被提交入库了。但是先不忙着执行提交，再执行一些操作，看看是否会被彻底地搞糊涂。

（1）继续修改一下 welcome.txt 文件（在文件后面再追加一行）。

```
$ echo "Bye-Bye." >> welcome.txt
```

（2）然后执行 git status，查看一下状态：

```
$ git status
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   welcome.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
```

状态输出居然是之前出现的两种不同状态输出的杂合体。

（3）如果显示精简的状态输出，也会看到前面两种精简输出的杂合体。

```
$ git status -s
MM welcome.txt
```

上面的更为复杂的 Git 状态输出可以这么理解：不但版本库中最新提交的文件与处于中间状态——提交任务（提交暂存区，stage）中的文件相比有改动，而且工作区当前的文件与处于中间状态——提交任务（提交暂存区，stage）中的文件相比也有改动。

即现在 welcome.txt 有三个不同的版本，一个在工作区，一个在等待提交的暂存区，还有一个是版本库中最新版本的 welcome.txt。通过不同的参数调用 git diff 命令可以看到不同状态下 welcome.txt 文件的差异。

（1）不带任何选项和参数调用 git diff 显示工作区的最新改动，即工作区与提交任务（提交暂存区，stage）中相比的差异。

```
$ git diff
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello.
 Nice to meet you.
+Bye-Bye.
```

（2）将工作区和 HEAD（当前工作分支）相比，会看到更多的差异。

```
$ git diff HEAD
diff --git a/welcome.txt b/welcome.txt
index 18832d3..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,3 @@
 Hello.
+Nice to meet you.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
+Bye-Bye.
```

（3）通过参数 --cached 或 --staged 调用 git diff 命令，看到的是提交暂存区（提交任务，stage）和版本库中文件的差异。

```
$ git diff --cached
diff --git a/welcome.txt b/welcome.txt
index 18832d3..fd3c069 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1 +1,2 @@
 Hello.
+Nice to meet you.
```

好了，现在是时候提交了。执行 git commit 命令进行提交：

```
$ git commit -m "which version checked in?"
[master e695606] which version checked in?
 1 files changed, 1 insertions(+), 0 deletions(-)
```

这次提交终于成功了。如何证明提交成功了呢？

（1）通过查看提交日志，看到了新的提交：

```
$ git log --pretty=oneline
e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
a0c641e92b10d8bcca1ed1bf84ca80340fdefee6 who does commit?
9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

（2）查看精简的状态输出。

状态输出中，文件名的前面少了一个字母 M，即只剩下第二列的字母 M。那么第一列的 M 哪里去了？被提交了呗。即提交任务（提交暂存区，stage）中的内容被提交到版本库中。所以，第一列会因为提交暂存区（提交任务，stage）与版本库中的状态一致而显示一个空白。

```
$ git status -s
 M welcome.txt
```

提交的 welcome.txt 是哪个版本呢？可以通过执行 git diff 或 git diff HEAD 命令查看差异。虽然命令 git diff 和 git diff HEAD 的比较过程并不相同（可以通过 strace 命令跟踪命令执行过程中的文件访问），但是会看到如下面所示的相同的差异输出结果。

```
$ git diff
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello.
 Nice to meet you.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
+Bye-Bye.
```

## 5.2 理解 Git 暂存区（stage）

将上面的实践从头至尾操作一遍，不知道您的感想如何：

- ——“被眼花缭乱的 Git 魔法彻底搞糊涂了？”
- ——“Git 为什么这么折磨人，修改的文件直接提交不就完了吗？”
- ——“看不出 Git 这么做有什么好处？”

上面的实践过程有意无意地透漏了“暂存区”的概念。为了避免用户被新概念吓坏，在暂存区出现的地方又同时使用了“提交任务”这一更易理解的概念，但是暂存区（称为 stage 或 index）才是其真正的名称。我认为 Git 暂存区的设计是 Git 最成功的设计之一，但也是最难理解的。

在版本库 .git 目录下有一个 index 文件，下面针对这个文件做一个有趣的试验。要说明的是：这个试验是用 1.7.3 版本的 Git 进行的，低版本的 Git 因为没有针对 git status 命令进行优化设计，需要运行 git diff 命令才能看到 index 文件的日期戳变化，具体操作步骤如下。

（1）首先执行 git checkout 命令（后面会介绍此命令），撤销工作区中 welcome.txt 文件尚未提交的修改。

```
$ git checkout -- welcome.txt
$ git status -s     # 执行 git diff ，如果 git 版本号小于 1.7.3
```

（2）通过状态输出可以看到工作区已经没有改动了。查看一下 .git/index 文件，注意该文件的时间戳为：19:37:44。

```
$ ls --full-time .git/index
-rw-r--r-- 1 jiangxin jiangxin 112 2010-11-29 19:37:44.625246224 +0800 .git/index
```

（3）再次执行 git status 命令，然后显示 .git/index 文件的时间戳为：19:37:44，与上面的一样。

```
$ git status -s     # 执行 git diff ，如果 git 版本号小于 1.7.3
$ ls --full-time .git/index
-rw-r--r-- 1 jiangxin jiangxin 112 2010-11-29 19:37:44.625246224 +0800 .git/index
```

（4）现在更改一下 welcome.txt 的时间戳，但是不改变它的内容。然后再执行 git status 命令，查看 .git/index 文件的时间戳为：19:42:06。

```
$ touch welcome.txt
$ git status -s     # 执行 git diff ，如果 git 版本号小于 1.7.3
$ ls --full-time .git/index
-rw-r--r-- 1 jiangxin jiangxin 112 2010-11-29 19:42:06.980243216 +0800 .git/index
```

看到了吗，时间戳改变了！

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




这个试验说明当执行 git status 命令（或者 git diff 命令）扫描工作区改动的时候，先依据 .git/index 文件中记录的（用于跟踪工作区文件的）时间戳、长度等信息判断工作区文件是否改变，如果工作区文件的时间戳改变了，说明文件的内容可能被改变了，需要打开文件，读取文件内容，与更改前的原始文件相比较，判断文件内容是否被更改。如果文件内容没有改变，则将该文件新的时间戳记录到 .git/index 文件中。因为如果要判断文件是否更改，使用时间戳、文件长度等信息进行比较要比通过文件内容比较要快得多，所以 Git 这样的实现方式可以让工作区状态扫描更快速地执行，这也是 Git 高效的原因之一。

文件 .git/index 实际上就是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名和文件的状态信息（时间戳和文件长度等）。文件的内容并没有存储在其中，而是保存在 Git 对象库 .git/objects 目录中，文件索引建立了文件和对象库中对象实体之间的对应。图 5-1 展示了工作区、版本库中的暂存区和版本库之间的关系。

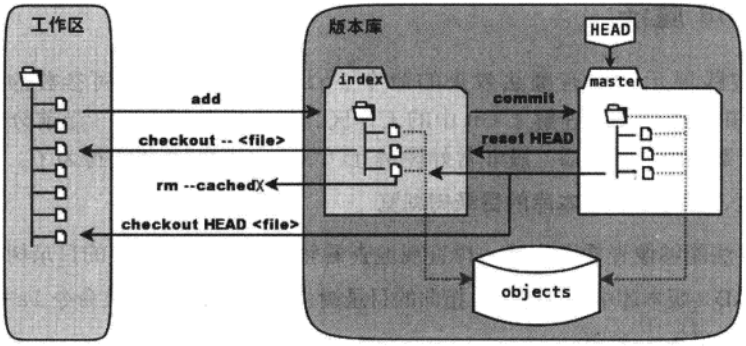



图 5-1 工作区、版本库、暂存区原理图

从图 5-1 中可以看到部分 Git 命令是如何影响工作区和暂存区的。这些命令的面纱将在接下来的几个章节中彻底揭开，下面就对这些命令进行简要说明：

- 图中左侧为工作区，右侧为版本库。在版本库中标记为 index 的区域是暂存区，标记为 master 的是 master 分支所代表的目录树。
- 图中可以看出，此时 HEAD 实际是指向 master 分支的一个“游标”，所以图示的命令中出现 HEAD 的地方可以用 master 来替换。
- 图中的 objects 标识的区域为 Git 的对象库，实际位于 .git/objects 目录下，这一点会在后面的章节中重点介绍。
- 当对工作区修改（或新增）的文件执行 git add 命令时，暂存区的目录树将被更新，同时工作区修改（或新增）的文件内容会被写入到对象库中的一个新的对象中，而该对象的 ID 被记录在暂存区的文件索引中。
- 当执行提交操作（git commit）时，暂存区的目录树会写到版本库（对象库）中，master 分支会做相应的更新，即 master 最新指向的目录树就是提交时原暂存区的目录树。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 当执行 git reset HEAD 命令时，暂存区的目录树会被重写，会被 master 分支指向的目录树所替换，但是工作区不受影响。
- 当执行 git rm --cached <file> 命令时，会直接从暂存区删除文件，工作区则不做出改变。
- 当执行 git checkout . 或 git checkout -- <file> 命令时，会用暂存区全部的文件或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。
- 当执行 git checkout HEAD . 或 git checkout HEAD <file> 命令时，会用 HEAD 指向的 master 分支中的全部或部分文件替换暂存区和工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

## 5.3 Git Diff 魔法

本章的实践展示了具有魔法效果的命令：git diff。在不同参数的作用下，git diff 的输出并不相同。在理解了 Git 中的工作区、暂存区和版本库（当前分支）的最新版本分别是三个不同的目录树后，就非常好理解 git diff 的魔法般的行为了。

### 1. 工作区、暂存区和版本库的目录树浏览

有什么办法能够像查看工作区一样直观地查看暂存区及 HEAD 中的目录树吗？

对于 HEAD（版本库中当前提交）指向的目录树，可以使用 Git 底层命令 ls-tree 来查看。

```
$ git ls-tree -l HEAD
100644 blob fd3c069c1de4f4bc9b15940f490aeb48852f3c42      25    welcome.txt
```

其中：

- 使用 -l 参数可以显示文件的大小。上面的 welcome.txt 的大小为 25 字节。
- 输出的 welcome.txt 文件条目从左至右，第一个字段是文件的属性 (rw-r--r--)，第二个字段说明是 Git 对象库中的一个 blob 对象（文件），第三个字段则是该文件在对象库中对应的 ID——一个 40 位的 SHA1 哈希值格式的 ID（这个会在后面介绍），第四个字段是文件大小，第五个字段是文件名。

在浏览暂存区中的目录树之前，首先清除工作区当前的改动。通过 git clean -fd 命令清除当前工作区中没有加入版本库的文件和目录（非跟踪文件和目录），然后执行 git checkout . 命令，用暂存区内容刷新工作区。

```
$ cd /path/to/my/workspace/demo
$ git clean -fd
$ git checkout .
```

然后开始在工作区中做出一些修改（修改 welcome.txt，再增加一个子目录和文件），

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


并添加到暂存区，最后再对工作区做出修改。

```
$ echo "Bye-Bye." >> welcome.txt
$ mkdir -p a/b/c
$ echo "Hello." > a/b/c/hello.txt
$ git add .
$ echo "Bye-Bye." >> a/b/c/hello.txt
$ git status -s
AM a/b/c/hello.txt
M  welcome.txt
```

上面的命令运行完毕后，通过精简的状态输出可以看出，工作区、暂存区和版本库当前分支的最新版本（HEAD）各不相同。先来看看工作区中文件的大小：

```
$ find . -path ./.git -prune -o -type f -printf "%-20p\t%s\n"
./welcome.txt            34
./a/b/c/hello.txt        16
```

要显示暂存区的目录树，可以使用 `git ls-files` 命令。

```
$ git ls-files -s
100644 18832d35117ef2f013c4009f5b2128dfaeff354f 0       a/b/c/hello.txt
100644 51dbfd25a804c30e9d8dc441740452534de8264b 0       welcome.txt
```

注意，这个输出与之前使用 `git ls-tree` 命令的输出不一样，其中第三个字段不是文件大小而是暂存区编号。如果想针对暂存区的目录树使用 `git ls-tree` 命令，需要先将暂存区的目录树写入 Git 对象库（用 `git write-tree` 命令），然后针对该目录树执行 `git ls-tree` 命令。

```
$ git write-tree
9431f4a3f3e1504e03659406faa9529f83cd56f8
$ git ls-tree -l 9431f4a
040000 tree 53583ee687fbb2e913d18d508aefd512465b2092       -    a
100644 blob 51dbfd25a804c30e9d8dc441740452534de8264b      34    welcome.txt
```

从上面的命令中可以看出：

- 到处都是 40 位的 SHA1 哈希值格式的 ID，可以用于指代文件内容（blob）、目录树（tree）和提交。但什么是 SHA1 哈希值 ID，作用是什么，这些疑问暂时搁置，下一章再解答。
- 命令 `git write-tree` 的输出就是写入 Git 对象库中的 Tree ID，这个 ID 将作为下一条命令的输入。
- 在 `git ls-tree` 命令中，没有把 40 位的 ID 写全，而是使用了前几位，实际上只要不与其他对象的 ID 冲突，就可以随心所欲地使用缩写 ID。
- 可以看到 `git ls-tree` 的输出显示的第一条是一个 tree 对象，即刚才创建的一级目录 a。

如果想要递归显示目录内容，则使用 `-r` 参数调用。使用参数 `-t` 可以把递归过程中遇到的每棵树都显示出来，而不只是显示最终的文件。下面执行递归操作显示目录树的内容：

```
$ git write-tree | xargs git ls-tree -l -r -t
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
040000 tree 53583ee687fbb2e913d18d508aefd512465b2092       -    a
040000 tree 514d729095b7bc203cf336723af710d41b84867b       -    a/b
040000 tree deaec688e84302d4a0b98a1b78a434be1b22ca02       -    a/b/c
100644 blob 18832d35117ef2f013c4009f5b2128dfaeff354f       7    a/b/c/hello.txt
100644 blob 51dbfd25a804c30e9d8dc441740452534de8264b      34    welcome.txt
```

好了，现在工作区、暂存区和 HEAD 三个目录树的内容各不相同。表 5-1 总结了不同文件在三个目录树中的文件大小。

表 5-1 文件不同版本的大小

| 文件名 | 工作区 | 暂存区 | HEAD |
|---|---|---|---|
| welcome.txt | 34 字节 | 34 字节 | 25 字节 |
| a/b/c/hello.txt | 16 字节 | 7 字节 | — |

### 2. Git diff 魔法

通过使用不同的参数调用 `git diff` 命令，可以对工作区、暂存区和 HEAD 中的内容进行两两比较。图 5-2 展示了不同的 `git diff` 命令的作用范围。

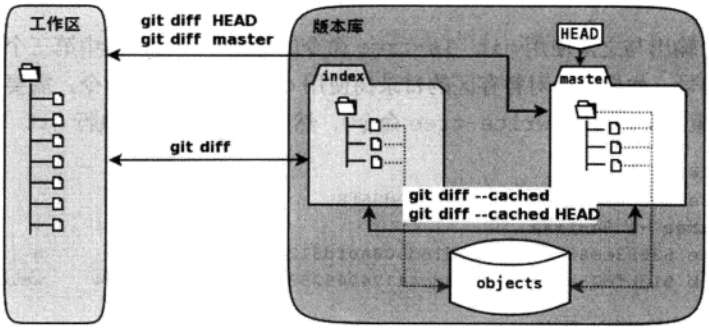



图 5-2 git diff 命令与版本库关系图

通过图 5-2 就不难理解下面代码中 `git diff` 命令的不同输出结果了。

（1）工作区和暂存区比较。

```
$ git diff
diff --git a/a/b/c/hello.txt b/a/b/c/hello.txt
index 18832d3..e8577ea 100644
--- a/a/b/c/hello.txt
+++ b/a/b/c/hello.txt
@@ -1 +1,2 @@
 Hello.
+Bye-Bye.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（2）暂存区和 HEAD 比较。

```
$ git diff --cached
diff --git a/a/b/c/hello.txt b/a/b/c/hello.txt
new file mode 100644
index 0000000..18832d3
--- /dev/null
+++ b/a/b/c/hello.txt
@@ -0,0 +1 @@
+Hello.
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello.
 Nice to meet you.
+Bye-Bye.
```

（3）工作区和 HEAD 比较。

```
$ git diff HEAD
diff --git a/a/b/c/hello.txt b/a/b/c/hello.txt
new file mode 100644
index 0000000..e8577ea
--- /dev/null
+++ b/a/b/c/hello.txt
@@ -0,0 +1,2 @@
+Hello.
+Bye-Bye.
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello.
 Nice to meet you.
+Bye-Bye.
```

## 5.4 不要使用 git commit -a

实际上，Git 的提交命令（git commit）可以带上 -a 参数，对本地所有变更的文件执行提交操作，包括对本地修改的文件和删除的文件，但不包括未被版本库跟踪的文件。

这个命令的确可以简化一些操作，减少用 git add 命令标识变更文件的步骤，但是如果习惯了使用这个“偷懒”的提交命令，就会丢掉 Git 暂存区带给用户的最大好处：对提交内容进行控制的能力。

有的用户甚至通过别名设置功能将 ci 设置为 git commit -a，这是更不可取的行为，

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


应严格禁止。本书很少会使用 git commit -a 命令。

## 5.5 搁置问题，暂存状态

查看一下当前工作区的状态：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a/b/c/hello.txt
#       modified:   welcome.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   a/b/c/hello.txt
#
```

在状态输出中，Git 体贴地告诉了用户如何将加入暂存区的文件从暂存区撤出以便让暂存区和 HEAD 一致（这样提交就不会发生）。还告诉用户，对于暂存区更新后在工作区所做的再一次修改有两个选择：或者再次添加到暂存区，或者取消工作区新做出的改动。但是现在理解涉及的命令还有些难度，一个是 git reset，一个是 git checkout。需要先理解什么是 HEAD，什么是 master 分支，以及 Git 对象存储的实现机制等问题，这样才可以更好地操作暂存区。

为此，我做出一个非常艰难的决定：就是保存当前的工作进度，在研究了 HEAD 和 master 分支的机制之后，继续对暂存区的探索。命令 git stash 就是用于保存当前工作进度的。

```
$ git stash
Saved working directory and index state WIP on master: e695606 which version
checked in?
HEAD is now at e695606 which version checked in?
```

运行完 git stash 之后，再查看工作区状态，会看见工作区尚未提交的改动（包括暂存区的改动）全都不见了。

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

“I'll be back。”（我会再回来的。）——施瓦辛格，《终结者》，1984。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 6 章 Git 对象

我们在上一章学习了 Git 的一个最重要的概念：暂存区。暂存区是一个介于工作区和版本库的中间状态，当执行提交时，实际上是将暂存区的内容提交到版本库中，而且 Git 的很多命令都会涉及暂存区的概念，例如 `git diff` 命令。

上一章也留下了很多疑惑，例如什么是 HEAD？什么是 master？为什么它们二者（在上一章）可以相互替换使用？为什么 Git 中的很多对象（如提交、树、文件内容等）都用 40 位的 SHA1 哈希值来表示？本章将会揭开这些奥秘，并且还会画出一个更为精确的版本库结构图。

## 6.1 Git 对象库探秘

前面刻意回避了对提交 ID 的说明，现在是时候来揭开由 40 位十六进制数字组成的“魔幻数字”的奥秘了。

通过查看日志的详尽输出，我们会惊讶地看到非常多的“魔幻数字”，这些“魔幻数字”实际上是 SHA1 哈希值。

```
$ git log -1 --pretty=raw
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800

    which version checked in?
```

一个提交中居然包含了三个 SHA1 哈希值表示的对象 ID：

- commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86：这是本次提交的唯一标识。
- tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9：这是本次提交所对应的目录树。
- parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6：这是本地提交的父提交（上一次提交）。

研究 Git 对象 ID 的一个重量级武器就是 `git cat-file` 命令。用下面的命令可以查看一下这三个 ID 的类型。

```
$ git cat-file -t e695606
commit
$ git cat-file -t f58d
tree
$ git cat-file -t a0c6
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
commit
```

在引用对象ID的时候，没有必要把整个的40位ID写全，只要从头开始的几位不冲突即可。下面再用 `git cat-file` 命令查看一下这几个对象的内容。

- commit 对象 e695606fc5e31b2ff9038a48a3d363f4c21a3d86

```
$ git cat-file -p e695606
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800

which version checked in?
```

- tree 对象 f58da9a820e3fd9d84ab2ca2f1b467ac265038f9

```
$ git cat-file -p f58da9a
100644 blob fd3c069c1de4f4bc9b15940f490aeb48852f3c42    welcome.txt
```

- commit 对象 a0c641e92b10d8bcca1ed1bf84ca80340fdefee6

```
$ git cat-file -p a0c641e
tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
parent 9e8a761ff9dd343a1380032884f488a2422c495a
author Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800

who does commit?
```

在上面的目录树（tree）对象中看到了一个新类型的对象：blob对象，这个对象保存着文件 `welcome.txt` 的内容，我们用 `git cat-file` 研究一下。

- 该对象的类型为blob。

```
$ git cat-file -t fd3c069c1de4f4bc9b15940f490aeb48852f3c42
blob
```

- 该对象的内容就是 `welcome.txt` 文件的内容。

```
$ git cat-file -p fd3c069c1de4f4bc9b15940f490aeb48852f3c42
Hello.
Nice to meet you.
```

这些对象保存在哪里？当然是Git库中的 `objects` 目录下了（ID的前2位作为目录名，后38位作为文件名）。用下面的命令可以看到这些对象在对象库中的实际位置。

```
$ for id in e695606 f58da9a a0c641e fd3c069; do \
  ls .git/objects/${id:0:2}/${id:2}*; done
.git/objects/e6/95606fc5e31b2ff9038a48a3d363f4c21a3d86
.git/objects/f5/8da9a820e3fd9d84ab2ca2f1b467ac265038f9
.git/objects/a0/c641e92b10d8bcca1ed1bf84ca80340fdefee6
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





```
.git/objects/fd/3c069c1de4f4bc9b15940f490aeb48852f3c42
```

图 6-1 更加清楚地显示了 Git 对象库中各个对象之间的关系。

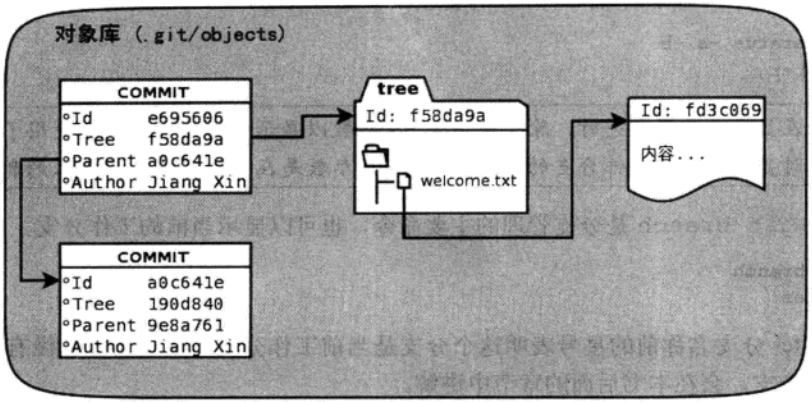



图 6-1 Git 版本库对象关系图

通过提交（Commit）对象之间的相互关联，可以很容易地识别出一条跟踪链，这条跟踪链可以在运行 `git log` 命令时通过 `--graph` 参数看到。下面的命令还使用了 `--pretty=raw` 参数，以便显示每个提交对象的 parent 属性。

```
$ git log --pretty=raw --graph e695606
* commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
| tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
| parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
| author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
| committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
|
|     which version checked in?
|
* commit a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
| tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
| parent 9e8a761ff9dd343a1380032884f488a2422c495a
| author Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800
| committer Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800
|
|     who does commit?
|
* commit 9e8a761ff9dd343a1380032884f488a2422c495a
  tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
  author Jiang Xin <jiangxin@ossxp.com> 1290919706 +0800
  committer Jiang Xin <jiangxin@ossxp.com> 1290919706 +0800

      initialized.
```

最后一个提交没有 parent 属性，所以跟踪链到此终结，这实际上就是提交的起点。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


现在来看看 HEAD 和 master 的奥秘吧。

因为在上一章的最后执行了 git stash 命令来将工作区和暂存区的改动全部封存起来，所以执行下面的命令会看到工作区和暂存区中没有改动。

```
$ git status -s -b
## master
```

**说明** 上面在显示工作区状态时，除了使用了 -s 参数以显示精简输出外，还使用了 -b 参数，以便能够同时显示出当前工作分支的名称，这个 -b 参数是在 Git 1.7.2 以后加入的新参数。

下面的 git branch 是分支管理的主要命令，也可以显示当前的工作分支。

```
$ git branch
* master
```

在 master 分支名称前的星号表明这个分支是当前工作分支。至于为什么没有其他分支，以及什么叫分支，会在本书后面的章节中讲解。

现在连续执行下面的三个命令会看到相同的输出：

```
$ git log -1 HEAD
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Mon Nov 29 17:23:01 2010 +0800

    which version checked in?
$ git log -1 master
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Mon Nov 29 17:23:01 2010 +0800

    which version checked in?
$ git log -1 refs/heads/master
commit e695606fc5e31b2ff9038a48a3d363f4c21a3d86
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Mon Nov 29 17:23:01 2010 +0800

    which version checked in?
```

也就是说，在当前版本库中，HEAD、master 和 refs/heads/master 具有相同的指向。现在到版本库（.git 目录）中一探它们的究竟。

```
$ find .git -name HEAD -o -name master
.git/HEAD
.git/logs/HEAD
.git/logs/refs/heads/master
.git/refs/heads/master
```

找到了 4 个文件，其中在 .git/logs 目录下的文件稍后再予以讨论，现在把目光锁定在 .git/HEAD 和 .git/refs/heads/master 上。

显示一下 .git/HEAD 的内容：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
$ cat .git/HEAD
ref: refs/heads/master
```

把 HEAD 的内容翻译过来就是："指向一个引用：refs/heads/master"。这个引用在哪里？当然是文件 .git/refs/heads/master 了。

看看文件 .git/refs/heads/master 的内容。

```
$ cat .git/refs/heads/master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

显示的 e695606... 所指为何物？可用 git cat-file 命令查看。

（1）显示 SHA1 哈希值指代的数据类型：

```
$ git cat-file -t e695606
commit
```

（2）显示该提交的内容：

```
$ git cat-file -p e695606fc5e31b2ff9038a48a3d363f4c21a3d86
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800

which version checked in?
```

原来分支 master 指向的是一个提交 ID（最新提交）。这样的分支实现是多么的巧妙啊：既然可以从任何提交开始建立一条历史跟踪链，用一个文件指向这个链条的最新提交，那么这个文件就可以用于追踪整个提交历史了。这个文件就是 .git/refs/heads/master 文件。

下面看一个更接近于真实的版本库结构图，如图 6-2 所示。

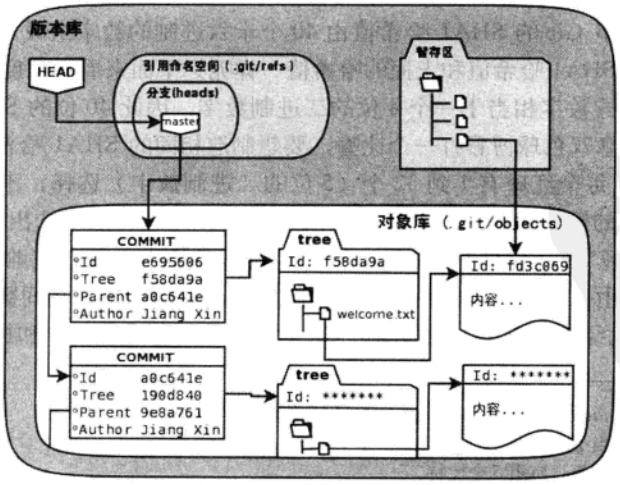



图 6-2 Git 版本库结构图

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


目录 .git/refs 是保存引用的命名空间，其中 .git/refs/heads 目录下的引用又称为分支。对于分支，既可以使用正规的长格式的表示法，如 refs/heads/master，也可以去掉前面的两级目录用 master 来表示。Git 有一个底层命令 git rev-parse 可以用于显示引用对应的提交 ID。

```
$ git rev-parse master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
$ git rev-parse refs/heads/master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
$ git rev-parse HEAD
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

可以看出它们都指向同一个对象。为什么这个对象是 40 位，而不是更少或更多？这些 ID 是如何生成的呢？

## 6.2 思考：SHA1 哈希值到底是什么，是如何生成的

哈希（hash）[①]是一种数据摘要算法（或称散列算法），是信息安全领域中重要的理论基石。该算法将任意长度的输入经过散列运算转换为固定长度的输出。固定长度的输出可以称为对应输入内容的数字摘要或哈希值。例如 SHA1 摘要算法可以处理从零到两百多万 TB[②]的输入数据，输出为固定的 160 比特的数字摘要。即使两个不同内容的输入数据量非常大、差异非常小，两者的哈希值也会显著不同。比较著名的摘要算法有：MD5 和 SHA1。Linux 下 sha1sum 命令可以用于生成摘要。

```
$ printf Git |sha1sum
5819778898df55e3a762f0c5728b457970d72cae  -
```

可以看出字符串 Git 的 SHA1 哈希值由 40 个十六进制的数字组成。那么能不能找出另外一个字符串使其 SHA1 哈希值和上面的哈希值一样呢？下面来看看难度有多大。

每个十六进制的数字相当于一个 4 位的二进制数字，因此 40 位的 SHA1 哈希值的输出实际为 160 比特。拿双色球博彩打一个比喻，要想制造相同的 SHA1 哈希值就相当于要选出 32 个[③]“红色球”，每个红球有 1 到 32 个（5 位的二进制数字）选择，不但红球之间可以重复，而且选出红球的顺序必须一致。相比“双色球博彩”总共只须选出 6 颗红球（1 到 33）外加 1 个篮球（1 到 16），SHA1“中奖”的难度差不多相当于要连续购买七期[④]“双色球”并且每一期都必需中一等奖。当然由于算法上的问题，制造冲突（相同数字摘要）的几率没有那么小[⑤]，但是已经足够小了，能够满足 Git 对不同的对象进行区分和标识了。即使有一天

① *http://en.wikipedia.org/wiki/Cryptographic_hash_function#cite_ref-13*
② $2^{64}$-1 比特，相当于 209 万 TB。1TB 相当于 1024GB。
③ 160 比特分成 32 组，每组 5 个比特。
④ 160÷24 ≈ 6.6
⑤ 相当于连续购买两期双色球彩票且都中一等奖。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


像发现了类似MD5摘要算法的冲突那样，发现了SHA1算法存在人为制造冲突的可能，那么Git可以使用更为安全的SHA-256或SHA-512的摘要算法。

可是Git中的各种对象：提交（commit）、文件内容（blob）、目录树（tree）等[①]，其对应的SHA1哈希值是如何生成的呢？下面就来展示一下。

先看一看提交的SHA1哈希值生成方法。

（1）看看HEAD对应的提交的内容，使用git cat-file命令。

```
$ git cat-file commit HEAD
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800

which version checked in?
```

（2）提交信息中总共包含234个字符。

```
$ git cat-file commit HEAD | wc -c
234
```

（3）在提交信息的前面加上内容commit 234<null>（<null>为空字符），然后执行SHA1哈希算法。

```
$ ( printf "commit 234\000"; git cat-file commit HEAD ) | sha1sum
e695606fc5e31b2ff9038a48a3d363f4c21a3d86  -
```

（4）上面命令得到的哈希值和用git rev-parse看到的是一样的。

```
$ git rev-parse HEAD
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

下面来看一看文件内容的SHA1哈希值生成方法。

（1）看看版本库中welcome.txt的内容，使用git cat-file命令。

```
$ git cat-file blob HEAD:welcome.txt
Hello.
Nice to meet you.
```

（2）文件总共包含25字节的内容。

```
$ git cat-file blob HEAD:welcome.txt | wc -c
25
```

（3）在文件内容的前面加上blob 25<null>的内容，然后执行SHA1哈希算法。

```
$ ( printf "blob 25\000"; git cat-file blob HEAD:welcome.txt ) | sha1sum
fd3c069c1de4f4bc9b15940f490aeb48852f3c42  -
```

① 还有tag对象，参见第3篇第17.2.2节。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（4）上面的命令得到的哈希值和用 git rev-parse 看到的是一样的。

```
$ git rev-parse HEAD:welcome.txt
fd3c069c1de4f4bc9b15940f490aeb48852f3c42
```

最后再来看看树的 SHA1 哈希值的形成方法。

（1）HEAD 对应的树的内容共包含 39 个字节。

```
$ git cat-file tree HEAD^{tree} | wc -c
39
```

（2）在树的内容的前面加上 tree 39<null> 的内容，然后执行 SHA1 哈希算法。

```
$ ( printf "tree 39\000"; git cat-file tree HEAD^{tree} ) | shalsum
f58da9a820e3fd9d84ab2ca2f1b467ac265038f9  -
```

（3）上面的命令得到的哈希值和用 git rev-parse 看到的是一样的。

```
$ git rev-parse HEAD^{tree}
f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
```

后面在学习里程碑（Tag）的时候会看到，Tag 对象（轻量级 Tag 除外）也是采用类似的方法在对象库中存储的。

## 6.3 思考：为什么不用顺序的数字来表示提交

到目前为止所进行的提交都是顺序提交，这可能会让您产生这么一个想法，为什么 Git 的提交不依据提交顺序对提交进行编号呢？可以把第一次提交定义为提交 1，依次递增。尤其是对于拥有像 Subversion 等集中式版本控制系统使用经验的用户而言，更会有这样的体会和想法。

集中式版本控制系统因为只有一个集中式的版本库，所以可以很容易地实现依次递增的全局唯一的提交号，像 Subversion 就是如此。Git 作为分布式版本控制系统，开发可以是非线性的，每个人都可以通过克隆版本库的方式工作在不同的本地版本库当中，在本地做的提交可以通过版本库之间的交互（推送和拉回操作）而互相分发，如果提交采用本地唯一的数字编号，在提交分发的时候会不可避免地造成冲突。这就要求提交的编号不能仅仅是本地局部有效，而是要“全球唯一”。Git 的提交通过 SHA1 哈希值作为提交 ID，的确做到了“全球唯一”。

Mercurial（Hg）是另外一个著名的分布式版本控制系统，它的提交 ID 非常有趣：同时使用了顺序的数字编号和“全球唯一”的 SHA1 哈希值。但实际上顺序的数字编号只是本地有效，对于克隆版本库来说没有意义，只有 SHA1 哈希值才是通用的编号。下面来看一个 Hg 的示例：

```
$ hg log --limit 2
修改集：      3009:2f1a3a7e8eb0
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
标签：          tip
用户：          Daniel Neuhuser <dasdasich@gmail.com>
日期：          Wed Dec 01 23:13:31 2010 +0100
摘要：          "Fixed" the CombinedHTMLDiff test

修改集：        3008:2fd3302ca7e5
用户：          Daniel Neuhuser <dasdasich@gmail.com>
日期：          Wed Dec 01 22:54:54 2010 +0100
摘要：          #559 Add 'html_permalink_text' confval
```

Hg 的设计使得在本地使用版本库更为方便，但是要在 Git 中做类似实现却很难，这是因为 Git 相比 Hg 拥有真正的分支管理功能。在 Git 中会存在当前分支中看不到的其他分支的提交，如何管理提交编号十分复杂。

幸好 Git 提供了很多方法可以方便地访问 Git 库中的对象：

- 采用部分的 SHA1 哈希值。不必把 40 位的哈希值写全，只采用开头的部分（4 位以上），只要不与现有的其他哈希值冲突即可。
- 使用 master 代表分支 master 中最新的提交，也可以使用全称 refs/heads/master 或 heads/master。
- 使用 HEAD 代表版本库中最近的一次提交。
- 符号 ^ 可以用于指代父提交。例如：
  - HEAD^ 代表版本库中的上一次提交，即最近一次提交的父提交。
  - HEAD^^ 则代表 HEAD^ 的父提交。
- 对于一个提交有多个父提交，可以在符号 ^ 后面用数字表示是第几个父提交。例如：
  - a573106^2 的含义是提交 a573106 的多个父提交中的第二个父提交。
  - HEAD^1 相当于 HEAD^，含义是 HEAD 的多个父提交中的第一个父提交。
  - HEAD^^2 的含义是 HEAD^（HEAD 父提交）的多个父提交中的第二个父提交。
- 符号 ~<n> 也可以用于指代祖先提交。例如：
  a573106~5 即相当于 a573106^^^^^ 。
- 提交所对应的树对象，可以用类似如下的语法访问：
  a573106^{tree}
- 某一次提交对应的文件对象，可以用如下的语法访问：
  a573106:path/to/file
- 暂存区中的文件对象，可以用如下的语法访问：
  :path/to/file

您可以使用 `git rev-parse` 命令在本地版本库中练习一下：

```
$ git rev-parse HEAD
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
$ git cat-file -p e695
tree f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
parent a0c641e92b10d8bcca1ed1bf84ca80340fdefee6
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
author Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291022581 +0800

which version checked in?
$ git cat-file -p e695^
tree 190d840dd3d8fa319bdec6b8112b0957be7ee769
parent 9e8a761ff9dd343a1380032884f488a2422c495a
author Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1290999606 +0800

who does commit?
$ git rev-parse e695^{tree}
f58da9a820e3fd9d84ab2ca2f1b467ac265038f9
$ git rev-parse e695^^{tree}
190d840dd3d8fa319bdec6b8112b0957be7ee769
```

在本篇的第 11.4.1 小节中，还会讲解更多访问 Git 对象的技巧，例如使用 tag 和日期来访问版本库对象。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第7章 Git重置

上一章讲解了版本库中对象的存储方式，以及分支master的实现。即master分支在版本库的引用目录（.git/refs）中体现为一个引用文件.git/refs/heads/master，其内容就是分支中最新提交的提交ID。

```
$ cat .git/refs/heads/master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

上一章还通过对提交本身数据结构的分析看到，提交可以通过对父提交的关联实现对提交历史的追溯。注意，下面的git log命令中使用了--oneline参数，类似于--pretty=oneline，但是可以显示更短小的提交ID。参数--oneline在Git 1.6.3及以后的版本中才有，老版本的Git可以使用参数--pretty=oneline --abbrev-commit替代。

```
$ git log --graph --oneline
* e695606 which version checked in?
* a0c641e who does commit?
* 9e8a761 initialized.
```

那么，是不是有新的提交发生的时候，master分支对应的引用文件中的内容就会改变呢？master分支对应的引用文件中的内容可以人为地改变吗？本章就来探讨如何用git reset命令改变分支引用文件的内容，即实现分支的重置。

## 7.1 分支游标master探秘

先来看看当有新的提交发生的时候，文件.git/refs/heads/master的内容如何改变。首先在工作区创建一个新文件，姑且叫作new-commit.txt，然后提交到版本库中。

```
$ touch new-commit.txt
$ git add new-commit.txt
$ git commit -m "does master follow this new commit?"
[master 4902dc3] does master follow this new commit?
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 new-commit.txt
```

此时工作目录下会有两个文件，其中文件new-commit.txt是新增的。

```
$ ls
new-commit.txt  welcome.txt
```

来看看master分支指向的提交ID是否改变了。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 可以看出在版本库引用空间（.git/refs/目录）下的master文件内容的确改变了，指向了新的提交。

```
$ cat .git/refs/heads/master
4902dc375672fbf52a226e0354100b75d4fe31e3
```

- 再用git log查看一下提交日志，可以看到刚刚完成的提交。

```
$ git log --graph --oneline
* 4902dc3 does master follow this new commit?
* e695606 which version checked in?
* a0c641e who does commit?
* 9e8a761 initialized.
```

引用refs/heads/master就好像是一个游标，在有新的提交发生的时候指向了新的提交。可是如果只可上、不可下，就不能称为“游标”。Git提供了git reset命令，可以将“游标”指向任意一个存在的提交ID。下面的示例就尝试人为地更改游标。（注意下面的命令中使用了--hard参数，会破坏工作区未提交的改动，慎用。）

```
$ git reset --hard HEAD^
HEAD is now at e695606 which version checked in?
```

还记得上一章介绍的HEAD^代表了HEAD的父提交吗？这条命令就相当于将master重置到上一个老的提交上。我们来看一下master文件的内容是否更改了。

```
$ cat .git/refs/heads/master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

果然，master分支的引用文件的指向更改为前一次提交的ID了，而且通过下面的命令可以看出新添加的文件new-commit.txt也丢失了。

```
$ ls
welcome.txt
```

重置命令不仅可以重置到前一次提交，而且还可以直接使用提交ID重置到任何一次提交。

（1）通过git log查询到最早的提交ID。

```
$ git log --graph --oneline
* e695606 which version checked in?
* a0c641e who does commit?
* 9e8a761 initialized.
```

（2）然后重置到最早的一次提交。

```
$ git reset --hard 9e8a761
HEAD is now at 9e8a761 initialized.
```

（3）重置后会发现welcome.txt也回退到原始版本库，曾经的修改都丢失了。

```
$ cat welcome.txt
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Hello.
```

使用重置命令很危险，会彻底地丢弃历史。那么，还能够通过浏览提交历史的办法找到丢弃的提交 ID，再使用重置命令恢复历史吗？不可能！因为重置让提交历史也改变了，提交日志如下：

```
$ git log
commit 9e8a761ff9dd343a1380032884f488a2422c495a
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Sun Nov 28 12:48:26 2010 +0800

    initialized.
```

## 7.2 用 reflog 挽救错误的重置

如果没有记下重置前 master 分支指向的提交 ID，想要重置回原来的提交似乎是一件麻烦的事情（去对象库中一个一个地找）。幸好 Git 提供了一个挽救机制，通过 .git/logs 目录下日志文件记录了分支的变更。默认非裸版本库（带有工作区）都提供分支日志功能，这是因为带有工作区的版本库都有如下设置：

```
$ git config core.logallrefupdates
true
```

查看一下 master 分支的日志文件 .git/logs/refs/heads/master 中的内容。下面的命令显示了该文件的最后几行。为了排版的需要，还将输出中的 40 位的 SHA1 提交 ID 缩短。

```
$ tail -5 .git/logs/refs/heads/master
dca47ab a0c641e Jiang Xin <...> 1290999606 +0800commit (amend): who does commit?
a0c641e e695606 Jiang Xin <...> 1291022581 +0800commit: which version checked in?
e695606 4902dc3 Jiang Xin <...> 1291435985 +0800commit: does master follow ...
4902dc3 e695606 Jiang Xin <...> 1291436302 +0800HEAD^: updating HEAD
e695606 9e8a761 Jiang Xin <...> 1291436382 +08009e8a761: updating HEAD
```

可以看出这个文件记录了 master 分支指向的变迁，最新的改变追加到文件的末尾，因此最后出现。最后一行可以看出因为执行了 git reset --hard 命令，指向的提交 ID 由 e695606 改变为 9e8a761。

Git 提供了一个 git reflog 命令，对这个文件进行操作。使用 show 子命令可以显示此文件的内容。

```
$ git reflog show master | head -5
9e8a761 master@{0}: 9e8a761: updating HEAD
e695606 master@{1}: HEAD^: updating HEAD
4902dc3 master@{2}: commit: does master follow this new commit?
e695606 master@{3}: commit: which version checked in?
a0c641e master@{4}: commit (amend): who does commit?
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



查看 git reflog 的输出和直接查看日志文件最大的不同在于显示顺序的不同，即最新改变放在了最前面显示，而且只显示每次改变的最终的 SHA1 哈希值。还有个重要的区别在于 git reflog 命令的输出中还提供了一个方便易记的表达式：<refname>@{<n>}。这个表达式的含义是引用 <refname> 之前第 <n> 次改变时的 SHA1 哈希值。

那么将引用 master 切换到两次变更之前的值，可以使用下面的命令。

- 重置 master 为两次改变之前的值。

```
$ git reset --hard master@{2}
HEAD is now at 4902dc3 does master follow this new commit?
```

- 重置后工作区中的文件 new-commit.txt 又回来了。

```
$ ls
new-commit.txt  welcome.txt
```

- 提交历史也回来了。

```
$ git log --oneline
4902dc3 does master follow this new commit?
e695606 which version checked in?
a0c641e who does commit?
9e8a761 initialized.
```

此时如果再用 git reflog 查看，会看到恢复 master 的操作也记录在日志中了。

```
$ git reflog show master | head -5
4902dc3 master@{0}: master@{2}: updating HEAD
9e8a761 master@{1}: 9e8a761: updating HEAD
e695606 master@{2}: HEAD^: updating HEAD
4902dc3 master@{3}: commit: does master follow this new commit?
e695606 master@{4}: commit: which version checked in?
```

## 7.3 深入了解 git reset 命令

重置命令（git reset）是 Git 最常用的命令之一，也是最危险最容易误用的命令。来看看 git reset 命令的用法。

```
用法一：git reset [-q] [<commit>] [--] <paths>...
用法二：git reset [--soft | --mixed | --hard | --merge | --keep] [-q] [<commit>]
```

上面列出了两个用法，其中 <commit> 都是可选项，可以使用引用或提交 ID，如果省略 <commit> 则相当于使用了 HEAD 的指向作为提交 ID。

上面列出的两种用法的区别在于，第一种用法在命令中包含路径 <paths>。为了避免路径和引用（或者提交 ID）同名而发生冲突，可以在 <paths> 前用两个连续的短线（减号）作为分隔。

第一种用法（包含了路径 <paths> 的用法）不会重置引用，更不会改变工作区，而

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


是用指定提交状态（<commit>）下的文件（<paths>）替换掉暂存区中的文件。例如命令 git reset HEAD <paths> 相当于取消之前执行的 git add <paths> 命令时改变的暂存区。

第二种用法（不使用路径 <paths> 的用法）则会重置引用 。根据不同的选项，可以对暂存区或工作区进行重置。参照下面的版本库模型图（图 7-1），来看一看不同的参数对第二种重置语法的影响。

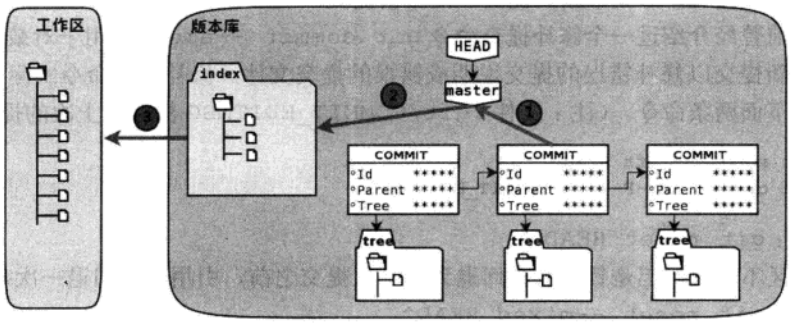



图 7-1 重置命令与版本库关系图

命令格式：git reset [--soft | --mixed | --hard ] [<commit>]

- 使用参数 --hard，如：git reset --hard <commit>。
  会执行上图中的全部动作 ①、②、③，即：
  ① 替换引用的指向。引用指向新的提交 ID。
  ② 替换暂存区。替换后，暂存区的内容和引用指向的目录树一致。
  ③ 替换工作区。替换后，工作区的内容变得和暂存区一致，也和 HEAD 所指向的目录树内容相同。
- 使用参数 --soft，如：git reset --soft <commit> 。
  会执行上图中的操作①。即只更改引用的指向，不改变暂存区和工作区。
- 使用参数 --mixed 或不使用参数（默认为 --mixed），如：git reset <commit>。
  会执行上图中的操作①和操作②。即更改引用的指向及重置暂存区，但是不改变工作区。

下面通过一些示例，看一下重置命令的不同用法。

- 命令：git reset
  仅用 HEAD 指向的目录树重置暂存区，工作区不会受到影响，相当于将之前用 git add 命令更新到暂存区的内容撤出暂存区。引用也未改变，因为引用重置到 HEAD 相当于没有重置。
- 命令：git reset HEAD
  同上。
- 命令：git reset -- filename

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


仅将文件filename的改动撤出暂存区，暂存区中其他文件不改变。相当于对命令git add filename的反向操作。

- 命令：git reset HEAD filename

  同上。

- 命令：git reset --soft HEAD^

  工作区和暂存区不改变，但是引用向前回退一次。当对最新提交的提交说明或提交的更改不满意时，撤销最新的提交以便重新提交。

  在之前曾经介绍过一个修补提交命令git commit --amend，用于对最新的提交进行重新提交以修补错误的提交说明或错误的提交文件。修补提交命令实际上相当于执行了下面两条命令。（注：文件.git/COMMIT_EDITMSG保存了上次的提交日志。）

```
$ git reset --soft HEAD^
$ git commit -e -F .git/COMMIT_EDITMSG
```

- 命令：git reset HEAD^

  工作区不改变，但是暂存区会回退到上一次提交之前，引用也会回退一次。

- 命令：git reset --mixed HEAD^

  同上。

- 命令：git reset --hard HEAD^

  彻底撤销最近的提交。引用回退到前一次，而且工作区和暂存区都会回退到上一次提交的状态。自上一次以来的提交全部丢失。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 8 章 Git 检出

在上一章我们学习了重置命令（git reset）。重置命令的一个用途就是修改引用（如 master）的游标指向。实际上在执行重置命令的时候没有使用任何参数对所要重置的分支名（如 master）进行设置，这是因为重置命令实际上所针对的是头指针 HEAD。之所以重置命令没有改变头指针 HEAD 的内容，是因为 HEAD 指向了一个引用 refs/heads/master，所以重置命令体现为分支“游标”的变更，HEAD 本身一直指向的是 refs/heads/master，并没有在重置时改变。

如果 HEAD 的内容不能改变而一直都指向 master 分支，那么 Git 如此精妙的分支设计岂不是浪费？ 如果 HEAD 要改变该如何改变呢？本章将学习检出命令（git checkout），该命令的实质就是修改 HEAD 本身的指向，该命令不会影响分支“游标”（如 master）。

## 8.1 HEAD 的重置即检出

HEAD 可以理解为“头指针”，是当前工作区的“基础版本”，当执行提交时，HEAD 指向的提交将作为新提交的父提交。看看当前 HEAD 的指向。

```
$ cat .git/HEAD
ref: refs/heads/master
```

可以看出 HEAD 指向了分支 master。此时执行 git branch 会看到当前处于 master 分支。

```
$ git branch -v
* master 4902dc3 does master follow this new commit?
```

现在使用 git checkout 命令检出该 ID 的父提交，看看会怎样。

```
$ git checkout 4902dc3^
Note: checking out '4902dc3^'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at e695606... which version checked in?
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


出现了大段的输出！翻译一下，Git 肯定又在提醒我们了。

```
$ git checkout 4902dc3^
注意：正检出 '4902dc3^'.

您现在处于‘分离头指针’状态。您可以检查、测试和提交，而不影响任何分支。
通过执行另外的一个 checkout 检出指令会丢弃在此状态下的修改和提交。

如果想保留在此状态下的修改和提交，使用 -b 参数调用 checkout 检出指令以
创建新的跟踪分支。如：

  git checkout -b new_branch_name

头指针现在指向 e695606... 提交说明为：which version checked in?
```

什么叫作“分离头指针”状态？查看一下此时 HEAD 的内容就明白了。

```
$ cat .git/HEAD
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
```

原来“分离头指针”状态指的就是 HEAD 头指针指向了一个具体的提交 ID，而不是一个引用（分支）。

查看最新提交的 reflog 也可以看到当针对提交执行 `git checkout` 命令时，HEAD 头指针被更改了：由指向 master 分支变成了指向一个提交 ID。

```
$ git reflog -1
e695606 HEAD@{0}: checkout: moving from master to 4902dc3^
```

注意上面的 reflog 是 HEAD 头指针的变迁记录，而非 master 分支。

查看一下 HEAD 和 master 对应的提交 ID，会发现现在它们指向的不一样。

```
$ git rev-parse HEAD master
e695606fc5e31b2ff9038a48a3d363f4c21a3d86
4902dc375672fbf52a226e0354100b75d4fe31e3
```

前一个是 HEAD 头指针的指向，后一个是 master 分支的指向。而且还可以看到执行 `git checkout` 命令与执行 `git reset` 命令不同，分支（master）的指向并没有改变，仍旧指向原有的提交 ID。

现在版本库的 HEAD 是指向 e695606 提交的。再做一次提交，HEAD 会如何变化呢？具体操作过程如下。

（1）先做一次修改：创建一个新文件 `detached-commit.txt`，添加到暂存区中。

```
$ touch detached-commit.txt
$ git add detached-commit.txt
```

（2）看一下状态，会发现其中有“当前不处于任何分支”的字样，显然这是因为 HEAD 处于“分离头指针”模式。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
$ git status
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   detached-commit.txt
#
```

（3）执行提交。在提交输出中也会出现 [detached HEAD ...] 的标识，这也是对用户的警示。

```
$ git commit -m "commit in detached HEAD mode."
[detached HEAD acc2f69] commit in detached HEAD mode.
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 detached-commit.txt
```

（4）此时头指针指向了新的提交。

```
$ cat .git/HEAD
acc2f69cf6f0ae346732382c819080df75bb2191
```

（5）再查看一下日志会发现新的提交是建立在之前的提交基础上的。

```
$ git log --graph --pretty=oneline
* acc2f69cf6f0ae346732382c819080df75bb2191 commit in detached HEAD mode.
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
* a0c641e92b10d8bcca1ed1bf84ca80340fdefee6 who does commit?
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

记下新的提交 ID（acc2f69），然后以 master 分支名作为参数执行 `git checkout` 命令，会切换到 master 分支上。

- 切换到 master 分支上，再没有之前大段的文字警告。

```
$ git checkout master
Previous HEAD position was acc2f69... commit in detached HEAD mode.
Switched to branch 'master'
```

- 因为 HEAD 头指针重新指向了分支，而不是处于“断头模式”（分离头指针模式）。

```
$ cat .git/HEAD
ref: refs/heads/master
```

- 切换之后，之前本地建立的新文件 `detached-commit.txt` 不见了。

```
$ ls
new-commit.txt  welcome.txt
```

- 切换之后，刚才的提交日志也不见了。

```
$ git log --graph --pretty=oneline
* 4902dc375672fbf52a226e0354100b75d4fe31e3 does master follow this new commit?
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
* a0c641e92b10d8bccaled1bf84ca80340fdefee6 who does commit?
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 * 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

刚才的提交还存在于版本库的对象库中吗？看看刚才记下的提交 ID。

```
$ git show acc2f69
commit acc2f69cf6f0ae346732382c819080df75bb2191
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Sun Dec 5 15:43:24 2010 +0800

    commit in detached HEAD mode.

diff --git a/detached-commit.txt b/detached-commit.txt
new file mode 100644
index 0000000..e69de29
```

可以看出这个提交现在仍在版本库中。由于这个提交没有被任何分支跟踪到，因此并不能保证这个提交会永久存在。实际上当 reflog 中含有该提交的日志过期后，这个提交随时都会从版本库中彻底清除。

## 8.2 挽救分离头指针

在“分离头指针”模式下进行的测试提交除了使用提交 ID（acc2f69）访问之外，不能通过 master 分支或其他引用访问到。如果这个提交是 master 分支所需要的，那么该如何处理呢？如果使用上一章介绍的 `git reset` 命令，的确可以将 master 分支重置到该测试提交“acc2f69”，但是如果那样就会丢掉 master 分支原先的提交“4902dc3”。使用合并操作（`git merge`）可以实现两者的兼顾。

下面的操作会将提交“acc2f69”合并到 master 分支中来，具体操作过程如下。

（1）确认当前处于 master 分支。

```
$ git branch -v
* master 4902dc3 does master follow this new commit?
```

（2）执行合并操作，将 acc2f69 提交合并到当前分支。

```
$ git merge acc2f69
Merge made by recursive.
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 detached-commit.txt
```

（3）工作区中多了一个 `detached-commit.txt` 文件。

```
$ ls
detached-commit.txt  new-commit.txt  welcome.txt
```

（4）查看日志，会看到不一样的分支图。即在 e695606 提交开始出现了开发分支，而分支在最新的 2b31c19 提交发生了合并。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git log --graph --pretty=oneline
*   2b31c199d5b81099d2ecd91619027ab63e8974ef Merge commit 'acc2f69'
|\
| * acc2f69cf6f0ae346732382c819080df75bb2191 commit in detached HEAD mode.
* | 4902dc375672fbf52a226e0354100b75d4fe31e3 does master follow this new commit?
|/
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
* a0c641e92b10d8bcca1ed1bf84ca80340fdefee6 who does commit?
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
```

（5）仔细看看最新提交，会看到这个提交有两个父提交。这就是合并的奥秘。

```
$ git cat-file -p HEAD
tree ab676f92936000457b01507e04f4058e855d4df0
parent 4902dc375672fbf52a226e0354100b75d4fe31e3
parent acc2f69cf6f0ae346732382c819080df75bb2191
author Jiang Xin <jiangxin@ossxp.com> 1291535485 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291535485 +0800

Merge commit 'acc2f69'
```

## 8.3 深入了解 git checkout 命令

检出命令（git checkout）是 Git 最常用的命令之一，同时也是一个很危险的命令，因为这条命令会重写工作区。检出命令的用法如下：

```
用法一：git checkout [-q] [<commit>] [--] <paths>...
用法二：git checkout [<branch>]
用法三：git checkout [-m] [[-b|--orphan] <new_branch>] [<start_point>]
```

上面列出的第一种用法和第二种用法的区别在于，第一种用法在命令中包含路径 <paths>。为了避免路径和引用（或者提交 ID）同名而发生冲突，可以在 <paths> 前用两个连续的短线（减号）作为分隔。

第一种用法的 <commit> 是可选项，如果省略则相当于从暂存区（index）进行检出。这和上一章的重置命令大不相同：重置的默认值是 HEAD，而检出的默认值是暂存区。因此重置一般用于重置暂存区（除非使用 --hard 参数，否则不重置工作区），而检出命令主要是覆盖工作区（如果 <commit> 不省略，也会替换暂存区中相应的文件）。

第一种用法（包含了路径 <paths> 的用法）不会改变 HEAD 头指针，主要是用于指定版本的文件覆盖工作区中对应的文件。如果省略 <commit>，则会用暂存区的文件覆盖工作区的文件，否则用指定提交中的文件覆盖暂存区和工作区中对应的文件。

第二种用法（不使用路径 <paths> 的用法）则会改变 HEAD 头指针。之所以后面的参数写作 <branch>，是因为只有 HEAD 切换到一个分支才可以对提交进行跟踪，否则仍然会进入“分离头指针”的状态。在“分离头指针”状态下的提交不能被引用关联到，从而可能丢失。所以用法二最主要的作用就是切换到分支。如果省略 <branch> 则相当于对工作区进行状态检查。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


第三种用法主要是创建和切换到新的分支（<new_branch>），新的分支从 <start_point> 指定的提交开始创建。新分支和我们熟悉的 master 分支没有什么实质的不同，都是在 refs/heads 命名空间下的引用。关于分支和 git checkout 命令的这个用法会在后面的章节具体介绍。

如图 8-1 所示的版本库模型图描述了 git checkout 实际完成的操作。

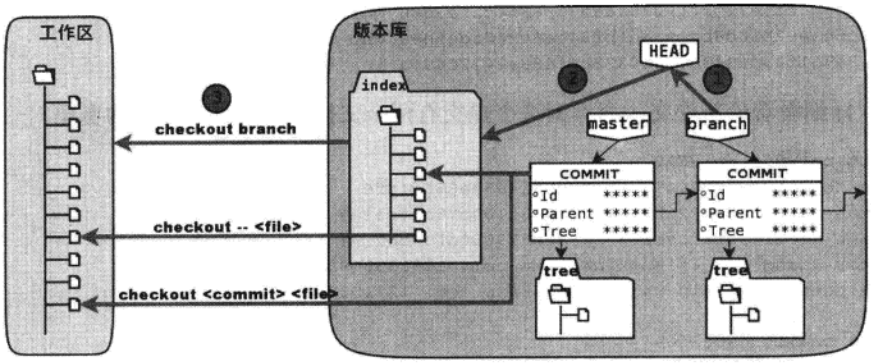



图 8-1 检出命令与版本库关系图

下面通过一些示例来具体看一下检出命令的不同用法。

- 命令：git checkout branch
  检出 branch 分支。要完成如图 8-1 中的三个步骤，更新 HEAD 以指向 branch 分支，以及用 branch 指向的树更新暂存区和工作区。
- 命令：git checkout
  汇总显示工作区、暂存区与 HEAD 的差异。
- 命令：git checkout HEAD
  同上。
- 命令：git checkout -- filename
  用暂存区中 filename 文件来覆盖工作区中的 filename 文件。相当于取消自上次执行 git add filename 以来（如果执行过）的本地修改。
  这个命令很危险，因为对于本地的修改会悄无声息地覆盖，毫不留情。
- 命令：git checkout branch -- filename
  维持 HEAD 的指向不变。用 branch 所指向的提交中的 filename 替换暂存区和工作区中相应的文件。注意会将暂存区和工作区中的 filename 文件直接覆盖。
- 命令：git checkout -- . 或写作 git checkout .
  注意 git checkout 命令后的参数为一个点（“.”）。这条命令最危险！会取消所有本地的修改（相对于暂存区）。相当于用暂存区的所有文件直接覆盖本地文件，不给用户任何确认的机会！

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第9章 恢复进度

在之前“第5章 Git暂存区”一章的结尾，曾经以终结者（The Terminator）的口吻说过：“我会再回来的”，会继续对暂存区的探索。经过了前面三章对Git对象、重置命令和检出命令的探索，现在已经拥有了足够多的武器，是时候“回归”了。

本章“回归”之后，有了前面几章的基础，再看Git状态输出中关于`git reset`或`git checkout`的指示，大家就会觉得很亲切、易如反掌了。本章还会重点介绍“回归”使用的`git stash`命令。

## 9.1 继续暂存区未完成的实践

经过了前面的实践，现在DEMO版本库应该处于master分支上，看看是不是这样。

```
$ cd /path/to/my/workspace/demo
$ git status -sb        # Git 1.7.2 及以上版本才支持 -b 参数哦
## master
$ git log --graph --pretty=oneline --stat
*   2b31c199d5b81099d2ecd91619027ab63e8974ef Merge commit 'acc2f69'
|\
| * acc2f69cf6f0ae346732382c819080df75bb2191 commit in detached HEAD mode.
| |  0 files changed, 0 insertions(+), 0 deletions(-)
* | 4902dc375672fbf52a226e0354100b75d4fe31e3 does master follow this new commit?
|/
|    0 files changed, 0 insertions(+), 0 deletions(-)
* e695606fc5e31b2ff9038a48a3d363f4c21a3d86 which version checked in?
|  welcome.txt |    1 +
|  1 files changed, 1 insertions(+), 0 deletions(-)
* a0c641e92b10d8bcca1ed1bf84ca80340fdefee6 who does commit?
* 9e8a761ff9dd343a1380032884f488a2422c495a initialized.
   welcome.txt |    1 +
   1 files changed, 1 insertions(+), 0 deletions(-)
```

还记得在之前“第5章 Git暂存区”一章的结尾是如何保存进度的吗？翻回去看一下，用的是`git stash`命令。这个命令用于保存当前进度，也是恢复进度要用的命令。

查看保存的进度用命令`git stash list`。

```
$ git stash list
stash@{0}: WIP on master: e695606 which version checked in?
```

现在就来恢复进度。使用`git stash pop`从最近保存的进度进行恢复。

```
$ git stash pop
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a/b/c/hello.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
Dropped refs/stash@{0} (c1bd56e2565abd64a0d63450fe42aba23b673cf3)
```

先不要管 git stash pop 命令的输出，后面会专题介绍 git stash 命令。通过查看工作区的状态，可以发现进度已经找回了（状态与进度保存前稍有不同）。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   a/b/c/hello.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
```

此时再看 Git 的状态输出，是否别有一番感觉呢？有了前面三章的基础，现在可以游刃有余地应对各种情况了。

（1）以当前暂存区的状态进行提交，即只提交 a/b/c/hello.txt，不提交 welcome.txt。

执行提交：

```
$ git commit -m "add new file: a/b/c/hello.txt, but leave welcome.txt alone."
[master 6610d05] add new file: a/b/c/hello.txt, but leave welcome.txt alone.
 1 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 a/b/c/hello.txt
```

查看提交后的状态：

```
$ git status -s
 M welcome.txt
```

（2）反悔了，回到之前的状态。

用重置命令放弃最新的提交：

```
$ git reset --soft HEAD^
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


查看最新的提交日志，可以看到前面的提交被抛弃了。

```
$ git log -1 --pretty=oneline
2b31c199d5b81099d2ecd91619027ab63e8974ef Merge commit 'acc2f69'
```

工作区和暂存区的状态也都维持在原来的状态。

```
$ git status -s
A  a/b/c/hello.txt
 M welcome.txt
```

（3）想将 welcome.txt 提交。

```
$ git add welcome.txt
$ git status -s
A  a/b/c/hello.txt
M  welcome.txt
```

（4）想将 a/b/c/hello.txt 撤出暂存区。

也是用重置命令。

```
$ git reset HEAD a/b/c
$ git status -s
M  welcome.txt
?? a/
```

（5）想将剩下的文件（welcome.txt）从暂存区撤出，就是说不想提交任何东西了。

还是使用重置命令，甚至可以不使用任何参数。

```
$ git reset
Unstaged changes after reset:
M       welcome.txt
```

（6）想将本地工作区所有的修改清除。即清除 welcome.txt 的改动，删除添加的目录 a 及下面的子目录和文件。

清除 welcome.txt 的改动用检出命令。

实际对于此例执行 git checkout. 也可以。

```
$ git checkout -- welcome.txt
```

工作区显示还有一个多余的目录 a。

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       a/
```

删除本地多余的目录和文件，可以使用 git clean 命令。先来测试运行以便看看哪些文件和目录会被删除，以免造成误删。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git clean -nd
Would remove a/
```

真正开始强制删除多余的目录和文件。

```
$ git clean -fd
Removing a/
```

整个世界清净了。

```
$ git status -s
```

## 9.2 使用 git stash

命令 git stash 可以用于保存和恢复工作进度，掌握这个命令对于日常的工作会有很大的帮助。关于这个命令的最主要的用法实际上通过前面的演示已经了解了。

- 命令：git stash
  保存当前的工作进度。会分别对暂存区和工作区的状态进行保存。
- 命令：git stash list
  显示进度列表。此命令显然暗示了 git stash 可以多次保存工作进度，并且在恢复的时候进行选择。
- 命令：git stash pop [--index] [<stash>]
  如果不使用任何参数，会恢复最新保存的工作进度，并将恢复的工作进度从存储的工作进度列表中清除。
  如果提供 <stash> 参数（来自于 git stash list 显示的列表），则从该 <stash> 中恢复。恢复完毕也将从进度列表中删除 <stash>。
  选项 --index 除了恢复工作区的文件外，还尝试恢复暂存区。这也就是为什么在本章一开始恢复进度的时候显示的状态和保存进度前的略有不同。
- 命令：git stash [save [--patch] [-k|--[no-]keep-index] [-q|--quiet] [<message>]]
  这条命令实际上是第一条 git stash 命令的完整版。即如果需要在保存工作进度的时候使用指定的说明，必须使用如下格式：

  ```
  git stash save "message..."
  ```

  - 使用参数 --patch 会显示工作区和 HEAD 的差异，通过对差异文件的编辑决定在进度中最终要保存的工作区的内容，通过编辑差异文件可以在进度中排除无关内容。
  - 使用 -k 或 --keep-index 参数，在保存进度后不会将暂存区重置。默认会将暂存区和工作区强制重置。
- 命令：git stash apply [--index] [<stash>]

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


除了不删除恢复的进度之外，其余和 git stash pop 命令一样。

- 命令：git stash drop [<stash>]
  删除一个存储的进度。默认删除最新的进度。
- 命令：git stash clear
  删除所有存储的进度。
- 命令：git stash branch <branchname> <stash>
  基于进度创建分支。对了，还没有讲到分支呢。;-)

## 9.3 探秘 git stash

了解一下 git stash 的机理会有几个好处：当保存了多个进度的时候知道从哪个进度恢复；综合运用前面介绍的 Git 知识点；了解 Git 的源码，Git 将不再神秘。

在执行 git stash 命令时，Git 实际调用了一个脚本文件实现相关的功能，这个脚本的文件名就是 git-stash。看看 git-stash 安装在哪里了。

```
$ git --exec-path
/usr/lib/git-core
```

如果查看一下这个目录，您会震惊的。

```
$ ls /usr/lib/git-core/
git                     git-help                git-reflog
git-add                 git-http-backend        git-relink
git-add--interactive    git-http-fetch          git-remote
git-am                  git-http-push           git-remote-ftp
git-annotate            git-imap-send           git-remote-ftps
git-apply               git-index-pack          git-remote-http
.................
... 省略 40 余行 ...
.................
```

实际上在 1.5.4 之前的版本中，Git 会将这样的一百多个以 git-<cmd> 格式命名的程序安装到可执行路径中，而这样做的唯一好处就是不用借助任何扩展机制就可以实现命令行补齐：即键入 git- 后，连续两次键入 <Tab> 键，就可以把这一百多个命令显示出来。这种方式随着 Git 子命令的增加显得越来越混乱，因此从 1.5.4 版本开始，不再提供 git-<cmd> 格式的命令，而是用唯一的 git 命令。而之前的名为 git-<cmd> 的子命令则保存在非可执行目录下，由 Git 负责加载。

在后面的章节中偶尔会看到形如 git-<cmd> 字样的名称，以及同时存在的 git <cmd> 命令。可以这样理解：git-<cmd> 作为软件本身的名称，而其命令行为 git <cmd>。

最初很多 Git 命令都是用 Shell 或 Perl 脚本语言开发的，在 Git 的发展中一些对运行效率要求高的命令用 C 语言改写。而 git-stash（至少在 Git 1.7.4 版本）还是使用 Shell 脚本开发的，研究它比研究用 C 写的命令要简单得多。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ file /usr/lib/git-core/git-stash
/usr/lib/git-core/git-stash: POSIX shell script text executable
```

解析 git-stash 脚本会比较枯燥，还是通过运行一些示例来理解更好一些。

当前的进度保存列表是空的。

```
$ git stash list
```

下面在工作区中做一些改动。

```
$ echo Bye-Bye. >> welcome.txt
$ echo hello. > hack-1.txt
$ git add hack-1.txt
$ git status -s
A  hack-1.txt
 M welcome.txt
```

可见暂存区中已经添加了新增的 hack-1.txt，修改过的 welcome.txt 并未添加到暂存区。执行 git stash 保存一下工作进度。

```
$ git stash save "hack-1: hacked welcome.txt, newfile hack-1.txt"
Saved working directory and index state On master: hack-1: hacked welcome.txt, newfile hack-1.txt
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

工作区恢复了修改前的原貌（实际上用了 git reset --hard HEAD 命令），文件 welcome.txt 的修改不见了，文件 hack-1.txt 整个都不见了。

```
$ git status -s
$ ls
detached-commit.txt  new-commit.txt  welcome.txt
```

再做一个修改，并尝试保存进度。

```
$ echo fix. > hack-2.txt
$ git stash
No local changes to save
```

进度保存失败！可见本地没有被版本控制系统跟踪的文件并不能保存进度。因此本地新文件需要先执行添加操作，然后再执行 git stash 命令。

```
$ git add hack-2.txt
$ git stash
Saved working directory and index state WIP on master: 2b31c19 Merge commit
'acc2f69'
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

不用看就知道工作区再次恢复原状。如果这时执行 git stash list 会看到有两次进度保存。

```
$ git stash list
stash@{0}: WIP on master: 2b31c19 Merge commit 'acc2f69'
stash@{1}: On master: hack-1: hacked welcome.txt, newfile hack-1.txt
```

从上面的输出中可以得出两个结论：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




- 在用 git stash 命令保存进度时，如果提供说明则更容易通过进度列表找到保存的进度。
- 每个进度的标识都是 stash@{<n>} 格式，像极了前面介绍的 reflog 的格式。

实际上，git stash 就是用前面介绍的引用和引用变更日志（reflog）来实现的。

```
$ ls -l .git/refs/stash .git/logs/refs/stash
-rw-r--r-- 1 jiangxin jiangxin 364 Dec  6 16:11 .git/logs/refs/stash
-rw-r--r-- 1 jiangxin jiangxin  41 Dec  6 16:11 .git/refs/stash
```

那么在“第 7 章 Git 重置”一章中学习的 reflog 可以派上用场了。

```
$ git reflog show refs/stash
e5c0cdc refs/stash@{0}: WIP on master: 2b31c19 Merge commit 'acc2f69'
6cec9db refs/stash@{1}: On master: hack-1: hacked welcome.txt, newfile hack-1.txt
```

对照 git reflog 的结果和前面 git stash list 的结果，可以肯定用 git stash 保存进度，实际上会将进度保存在引用 refs/stash 所指向的提交中。多次的进度保存，实际上相当于引用 refs/stash 一次又一次的变化，而 refs/stash 引用的变化由 reflog（即 .git/logs/refs/stash）所记录下来。这个实现是多么简单而巧妙啊。

一个新的疑问又出现了，如何在引用 refs/stash 中同时保存暂存区的进度和工作区中的进度呢？查看一下引用 refs/stash 的提交历史能够看出端倪。

```
$ git log --graph --pretty=raw  refs/stash -2
*   commit e5c0cdc2dedc3e50e6b72a683d928e19a1d9de48
|\  tree 780c22449b7ff67e2820e09a6332c360ddc80578
| | parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
| | parent c5edbdcc90addb06577ff60f644acd1542369194
| | author Jiang Xin <jiangxin@ossxp.com> 1291623066 +0800
| | committer Jiang Xin <jiangxin@ossxp.com> 1291623066 +0800
| |
| |     WIP on master: 2b31c19 Merge commit 'acc2f69'
| |
| * commit c5edbdcc90addb06577ff60f644acd1542369194
|/  tree 780c22449b7ff67e2820e09a6332c360ddc80578
|   parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
|   author Jiang Xin <jiangxin@ossxp.com> 1291623066 +0800
|   committer Jiang Xin <jiangxin@ossxp.com> 1291623066 +0800
|
|       index on master: 2b31c19 Merge commit 'acc2f69'
```

从提交历史中可以看到进度保存的最新提交是一个合并提交。最新的提交说明中有 WIP 字样（是 Work In Progess 的简称），代表了工作区进度。而最新提交的第二个父提交（提交历史中显示为第二个提交）有 index on master 字样，说明这个提交代表着暂存区的进度。

但是提交历史中的两个提交都指向了同一个树——tree 780c224...，这是因为最后一次做进度保存时工作区相对暂存区没有改变，这使得工作区和暂存区在引用 refs/stash 中的存储变得有些扑朔迷离。别忘了第一次进度保存工作区、暂存区和版本库都是不同的，可以用于验证关于 refs/stash 实现机制的判断。

第一次进度保存可以使用 reflog 中的语法，即用 refs/stash@{1} 来访问，也可以用简称

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


stash@{1}。下面就来研究一下第一次的进度保存。

```
$ git log --graph --pretty=raw  stash@{1} -3
*   commit 6cec9db44af38d01abe7b5025a5190c56fd0cf49
|\  tree 7250f186c6aa3e2d1456d7fa915e529601f21d71
| | parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
| | parent 4560d76c19112868a6a5692bf9379de09c0452b7
| | author Jiang Xin <jiangxin@ossxp.com> 1291622767 +0800
| | committer Jiang Xin <jiangxin@ossxp.com> 1291622767 +0800
| |
| |     On master: hack-1: hacked welcome.txt, newfile hack-1.txt
| |
| * commit 4560d76c19112868a6a5692bf9379de09c0452b7
|/  tree 5d4dd328187e119448c9171f99cf2e507e91a6c6
|   parent 2b31c199d5b81099d2ecd91619027ab63e8974ef
|   author Jiang Xin <jiangxin@ossxp.com> 1291622767 +0800
|   committer Jiang Xin <jiangxin@ossxp.com> 1291622767 +0800
|
|       index on master: 2b31c19 Merge commit 'acc2f69'
|
*   commit 2b31c199d5b81099d2ecd91619027ab63e8974ef
|\  tree ab676f92936000457b01507e04f4058e855d4df0
| | parent 4902dc375672fbf52a226e0354100b75d4fe31e3
| | parent acc2f69cf6f0ae346732382c819080df75bb2191
| | author Jiang Xin <jiangxin@ossxp.com> 1291535485 +0800
| | committer Jiang Xin <jiangxin@ossxp.com> 1291535485 +0800
| |
| |     Merge commit 'acc2f69'
```

果然上面显示的三个提交对应的三棵树各不相同。查看一下差异。用“原基线”代表进度保存时版本库的状态，即提交 2b31c199；用“原暂存区”代表进度保存时暂存区的状态，即提交 4560d76；用“原工作区”代表进度保存时工作区的状态，即提交 6cec9db。

❑ 原基线和原暂存区的差异比较。

```
$ git diff stash@{1}^2^ stash@{1}^2
diff --git a/hack-1.txt b/hack-1.txt
new file mode 100644
index 0000000..25735f5
--- /dev/null
+++ b/hack-1.txt
@@ -0,0 +1 @@
+hello.
```

❑ 原暂存区和原工作区的差异比较。

```
$ git diff stash@{1}^2 stash@{1}
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 Nice to meet you.
+Bye-Bye.
```

❑ 原基线和原工作区的差异比较。

```
$ git diff stash@{1}^1 stash@{1}
diff --git a/hack-1.txt b/hack-1.txt
new file mode 100644
index 0000000..25735f5
--- /dev/null
+++ b/hack-1.txt
@@ -0,0 +1 @@
+hello.
diff --git a/welcome.txt b/welcome.txt
index fd3c069..51dbfd2 100644
--- a/welcome.txt
+++ b/welcome.txt
@@ -1,2 +1,3 @@
 Hello.
 Nice to meet you.
+Bye-Bye.
```

用 stash@{1} 来恢复进度。

```
$ git stash apply stash@{1}
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hack-1.txt
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
```

显示进度列表，然后删除进度列表。

```
$ git stash list
stash@{0}: WIP on master: 2b31c19 Merge commit 'acc2f69'
stash@{1}: On master: hack-1: hacked welcome.txt, newfile hack-1.txt
$ git stash clear
```

删除进度列表之后，会发现 stash 相关的引用和 reflog 都不见了。

```
$ ls -l .git/refs/stash .git/logs/refs/stash
ls: cannot access .git/refs/stash: No such file or directory
ls: cannot access .git/logs/refs/stash: No such file or directory
```

通过上面的这些分析，有一定 Shell 编程基础的读者就可以尝试研究 git-stash 的代码了，在研究过程中你可能会有新的发现。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 10 章 Git 基本操作

之前的实践选取的示例都非常简单，基本上都是增加和修改文本文件，而现实情况要复杂得多，需要应对各种情况：文件删除、文件复制、文件移动、目录的组织、二进制文件、误删文件的恢复，等等。

本章要用一个更为真实的例子：通过对 Hello World 程序源代码的版本控制，来介绍工作区中其他的一些常用操作。首先我们会删除之前历次实践在版本库中留下的“垃圾”数据，然后再在其中创建一些真实的代码，并对其进行版本控制。

## 10.1 先来合个影

马上就要和之前实践遗留的数据告别了，告别之前是不是要留个影呢？在 Git 里，“留影”用的命令叫作 tag，更加专业的术语叫作“里程碑”（打 tag，或打标签）。“留影”操作如下：

```
$ cd /path/to/my/workspace/demo
$ git tag -m "Say bye-bye to all previous practice." old_practice
```

本章还不打算详细介绍里程碑的奥秘，只要知道里程碑无非也是一个引用，通过记录提交 ID（或者创建 Tag 对象）来为当前版本库的状态进行“留影”。

```
$ ls .git/refs/tags/old_practice
.git/refs/tags/old_practice

$ git rev-parse refs/tags/old_practice
41bd4e2cce0f8baa9bb4cdda62927b408c846cd6
```

留过影之后，可以执行 `git describe` 命令将最新提交显示为一个易记的名称。显示的时候会选取离该提交最近的里程碑作为“基础版本号”，后面附加标识距离“基础版本”的数字，以及该提交的 SHA1 哈希值缩写。因为最新的提交上恰好被打了一个“里程碑”，所以直接显示“里程碑”的名称。这个技术会在后面的示例代码中多次用到。

```
$ git describe
old_practice
```

## 10.2 删除文件

看看版本库当前的状态，暂存区和工作区都包含修改。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git status -s
A  hack-1.txt
 M welcome.txt
```

在这个暂存区和工作区都包含文件修改的情况下，使用删除命令更具有挑战性。删除命令有多种使用方法，有的方法很巧妙，而有的方法则需要更多的输入。为了分别介绍不同的删除方法，还要使用上一章介绍的进度保存（git stash）命令。

- 保存进度。

```
$ git stash
Saved working directory and index state WIP on master: 2b31c19 Merge commit
'acc2f69'
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

- 再恢复进度。注意不要使用 git stash pop，而是使用 git stash apply，因为这个保存的进度要被多次用到。

```
$ git stash apply
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hack-1.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   welcome.txt
#
```

## 10.2.1 本地删除不是真的删除

当前工作区的文件有：

```
$ ls
detached-commit.txt
hack-1.txt
new-commit.txt
welcome.txt
```

直接在工作区删除这些文件，会如何呢？

```
$ rm *.txt
```

通过下面的命令，可以看到在暂存区（版本库）中的文件仍然存在，并未删除。

```
$ git ls-files
detached-commit.txt
hack-1.txt
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
new-commit.txt
welcome.txt
```

从文件的状态来看，文件只是在本地进行了删除，尚未添加到暂存区（提交任务）中。也就是说：**直接在工作区删除，对暂存区和版本库没有任何影响。**

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hack-1.txt
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    detached-commit.txt
#       deleted:    hack-1.txt
#       deleted:    new-commit.txt
#       deleted:    welcome.txt
#
```

从Git状态输出中可以看出，本地删除如果要反映在暂存区中应该用`git rm`命令，对不想删除的文件执行`git checkout -- <file>`，可以让文件在工作区重现。

## 10.2.2 执行 git rm 命令删除文件

好吧，按照上面状态输出的内容，将所有的文本文件删除。执行下面的命令：

```
$ git rm detached-commit.txt hack-1.txt new-commit.txt welcome.txt
rm 'detached-commit.txt'
rm 'hack-1.txt'
rm 'new-commit.txt'
rm 'welcome.txt'
```

再看一看状态：

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    detached-commit.txt
#       deleted:    new-commit.txt
#       deleted:    welcome.txt
#
```

删除动作加入了暂存区。这时执行提交动作，就从真正意义上执行了文件删除。

```
$ git commit -m "delete trash files. (using: git rm)"
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
[master 483493a] delete trash files. (using: git rm)
 1 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 detached-commit.txt
 delete mode 100644 new-commit.txt
 delete mode 100644 welcome.txt
```

不过不要担心，文件只是在版本库的最新提交中被删除了，在历史提交中尚在。可以通过下面的命令查看历史版本的文件列表。

```
$ git ls-files --with-tree=HEAD^
detached-commit.txt
new-commit.txt
welcome.txt
```

也可以查看在历史版本中尚在的删除文件的内容。

```
$ git cat-file -p HEAD^:welcome.txt
Hello.
Nice to meet you.
```

### 10.2.3 命令 git add -u 快速标记删除

在前面执行 git rm 命令时，写下了所有要删除的文件名，好长的命令啊！能不能简化些？实际上 git add 就可以，即使用 -u 参数调用 git add 命令，含义是将本地有改动（包括修改和删除）的文件标记到暂存区。为了重现刚才的场景，先使用重置命令抛弃最新的提交，再使用进度恢复到之前的状态，具体操作过程如下。

（1）丢弃之前测试删除的试验性提交。

```
$ git reset --hard HEAD^
HEAD is now at 2b31c19 Merge commit 'acc2f69'
```

（2）恢复保存的进度。（参数 -q 使得命令进入安静模式。）

```
$ git stash apply -q
```

（3）然后删除本地文件，状态显示出依然只是在本地删除了文件，暂存区中的文件仍在。

```
$ rm *.txt
$ git status -s
 D detached-commit.txt
AD hack-1.txt
 D new-commit.txt
 D welcome.txt
```

（4）执行 git add -u 命令可以将（被版本库追踪的）本地文件的变更（修改、删除）全部记录到暂存区中。

```
$ git add -u
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（5）查看状态，可以看到工作区删除的文件全部被标记为下次提交时删除。

```
$ git status -s
D  detached-commit.txt
D  new-commit.txt
D  welcome.txt
```

（6）执行提交，删除文件。

```
$ git commit -m "delete trash files. (using: git add -u)"
[master 7161977] delete trash files. (using: git add -u)
 1 files changed, 0 insertions(+), 2 deletions(-)
 delete mode 100644 detached-commit.txt
 delete mode 100644 new-commit.txt
 delete mode 100644 welcome.txt
```

## 10.3 恢复删除的文件

经过了上面的文件删除，工作区已经没有文件了。为了说明文件移动，现在恢复一个删除的文件。前面已经说过执行了文件删除并提交，只是在最新的提交中删除了文件，历史提交中文件仍然保留，可以从历史提交中提取文件。执行下面的命令可以从历史（前一次提交）中恢复 welcome.txt 文件。

```
$ git cat-file -p HEAD~1:welcome.txt > welcome.txt
```

也可以使用 git show 命令取代 git cat-file -p 命令，效果相同。

```
$ git show HEAD~1:welcome.txt > welcome.txt
```

使用 git checkout 命令则最为简洁实用。

```
$ git checkout HEAD~1 -- welcome.txt
```

上面的命令中出现的 HEAD~1 即相当于 HEAD^ 都指的是 HEAD 的上一次提交。执行 git add -A 命令会将工作区中的所有改动及新增文件添加到暂存区，这也是一个常用的技巧。执行下面的命令后，将恢复过来的 welcome.txt 文件添加回暂存区。

```
$ git add -A
$ git status -s
A  welcome.txt
```

执行提交操作，文件 welcome.txt 又回来了。

```
$ git commit -m "restore file: welcome.txt"
[master 63992f0] restore file: welcome.txt
 1 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 welcome.txt
```

通过再次添加的方式恢复被删除的文件是最自然的恢复方法。其他版本控制系统如 CVS

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


也采用同样的方法恢复删除的文件，但是有的版本控制系统（如 Subversion ）如果这样操作会有严重的副作用——文件变更历史被人为地割裂，而且还会造成服务器存储空间的浪费。Git 通过添加方式反删除文件没有副作用，这是因为在 Git 的版本库中相同内容的文件保存在一个 blob 对象中，即便是内容不同的 blob 对象通过对象库打包也会进行存储优化。



## 10.4 移动文件

通过将 welcome.txt 改名为 README 文件来测试一下在 Git 中如何移动文件。Git 提供了 git mv 命令完成改名操作。

```
$ git mv welcome.txt README
```

可以从当前的状态中看到改名的操作。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    welcome.txt -> README
#
```

提交改名操作，在提交输出中可以看到改名前后两个文件的相似度（百分比）。

```
$ git commit -m "改名测试"
[master 7aa5ac1] 改名测试
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename welcome.txt => README (100%)
```

从提交日志中出现的文件相似度可以看出，Git 的改名操作得益于 Git 对文件追踪的强大支持（文件内容作为 blob 对象保存在对象库中）。改名操作相当于对旧文件执行删除，对新文件执行添加。实际上完全可以不使用 git mv 命令，而是以 git rm 和 git add 两条命令取而代之。为了试验不使用 git mv 命令是否可行，先撤销之前进行的提交。

- 撤销之前测试文件移动的提交。

```
$ git reset --hard HEAD^
HEAD is now at 63992f0 restore file: welcome.txt
```

- 撤销之后 welcome.txt 文件又回来了。

```
$ git status -s
$ git ls-files
welcome.txt
```

新的改名操作不使用 git mv 命令，而是直接在本地改名（文件移动），将 welcome.txt 改名为 README。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ mv welcome.txt README
$ git status -s
 D welcome.txt
?? README
```

为了考验一下 Git 的内容追踪能力，再修改一下改名后的 README 文件，即在文件末尾追加一行。

```
$ echo "Bye-Bye." >> README
```

可以使用前面介绍的 git add -A 命令。相当于对修改文件执行 git add，对删除文件执行 git rm，对本地新增文件执行 git add。

```
$ git add -A
```

查看状态，也可以看到文件重命名。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    welcome.txt -> README
#
```

执行提交。

```
$ git commit -m "README is from welcome.txt."
[master c024f34] README is from welcome.txt.
 1 files changed, 1 insertions(+), 0 deletions(-)
 rename welcome.txt => README (73%)
```

这次提交中也看到了重命名操作，但是重命名相似度不是 100%，而是 73%，这是因为重命名后的文件又追加了一行。

## 10.5 一个显示版本号的 Hello World

在本章的一开始为纪念前面的实践留了一个影，叫作 old_practice 。现在再次执行 git describe 看一下现在的版本号。

```
$ git describe
old_practice-3-gc024f34
```

也就是说：当前工作区的版本是“留影”后的第三个版本，提交 ID 是 c024f34。

下面的命令可以在提交日志中显示提交对应的里程碑（Tag）。其中参数 --decorate 可以在提交 ID 的旁边显示该提交关联的引用（里程碑或分支）。

```
$ git log --oneline --decorate -4
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
c024f34 (HEAD, master) README is from welcome.txt.
63992f0 restore file: welcome.txt
7161977 delete trash files. (using: git add -u)
2b31c19 (tag: old_practice) Merge commit 'acc2f69'
```

命令 git describe 的输出可以作为软件版本号，这个功能非常有用。因为这样可以很容易地实现将发布的软件包版本和版本库中的代码对应在一起，当发现软件包包含 Bug 时，可以最快、最准确地对应到代码上。

下面的 Hello World 程序就实现了这个功能。创建目录 src，并在 src 目录下创建以下三个文件：

（1）文件：src/main.c

没错，下面几行代码就是这个程序的主代码，和输出相关的代码就两行，一行显示“Hello, world.”，另外一行显示软件版本。在显示软件版本时用到了宏 _VERSION，这个宏的来源参考下一个文件。

```
#include "version.h"
#include <stdio.h>

int
main()
{
    printf( "Hello, world.\n" );
    printf( "version: %s.\n", _VERSION );
    return 0;
}
```

（2）文件：src/version.h.in

没错，这个文件名的后缀是 .h.in。这个文件其实是用于生成文件 version.h 的模板文件。在由此模板文件生成 version.h 的过程中，宏 _VERSION 的值“<version>”会动态替换。

```
#ifndef HELLO_WORLD_VERSION_H
#define HELLO_WORLD_VERSION_H

#define _VERSION "<version>"

#endif
```

（3）文件：src/Makefile

这个文件看起来很复杂，而且要注意所有缩进都是使用一个 <Tab> 键完成的缩进，千万不要错误地写成空格，因为这是 Makefile 的格式要求。这个文件除了定义如何由代码生成可执行文件 hello 之外，还定义了如何将模板文件 version.h.in 转换为 version.h。在转换过程中用 git describe 命令的输出替换模板文件中的 <version> 字符串。

```
OBJECTS = main.o
TARGET = hello
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
all: $(TARGET)

$(TARGET): $(OBJECTS)
        $(CC) -o $@ $^

main.o: main.c version.h

version.h: new_header

new_header:
        @sed -e "s/<version>/$$(git describe)/g"  \
                < version.h.in > version.h.tmp
        @if diff -q version.h.tmp version.h >/dev/null 2>&1; \
        then \
                rm version.h.tmp; \
        else \
                echo "version.h.in => version.h" ; \
                mv version.h.tmp version.h; \
        fi

clean:
        rm -f $(TARGET) $(OBJECTS) version.h

.PHONY: all clean
```

上述三个文件创建完毕之后，进入到 src 目录，试着运行一下。先执行 make 编译，再运行编译后的程序 hello。

```
$ cd src
$ make
version.h.in => version.h
cc    -c -o main.o main.c
cc -o hello main.o
$ ./hello
Hello, world.
version: old_practice-3-gc024f34.
```

## 10.6 使用 git add -i 选择性添加

刚刚创建的 Hello World 程序还没有添加到版本库中，在 src 目录下有下列文件：

```
$ cd /path/to/my/workspace/demo
$ ls src
hello  main.c  main.o  Makefile  version.h  version.h.in
```

这些文件中 hello、main.o 和 version.h 都是在编译时生成的文件，不应该加入到版本库中。那么选择性添加文件除了针对文件逐一使用 git add 命令外，还有其他办法吗？通过使用 -i 参数调用 git add 就是一个办法，它提供了一个交互式的界面。

执行 git add -i 命令，进入一个交互式界面，首先显示的是工作区状态。显然因为

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




版本库进行了清理，所以显得很“干净”。

```
$ git add -i
           staged     unstaged path


*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

交互式界面显示了命令列表，可以使用数字或加亮显示的命令首字母，选择相应的功能。对于此例需要将新文件加入到版本库中，所以选择“4”。

```
What now> 4
  1: src/Makefile
  2: src/hello
  3: src/main.c
  4: src/main.o
  5: src/version.h
  6: src/version.h.in
Add untracked>>
```

当选择了“4”之后，就进入了“Add untracked”的界面，显示了本地新增（尚不在版本库中）的文件列表，而且提示符也变了，由“What now>”变为“Add untracked>>”。依次输入 1、3、6 将源代码添加到版本库中。

- 输入“1”：

```
Add untracked>> 1
* 1: src/Makefile
  2: src/hello
  3: src/main.c
  4: src/main.o
  5: src/version.h
  6: src/version.h.in
```

- 输入“3”：

```
Add untracked>> 3
* 1: src/Makefile
  2: src/hello
* 3: src/main.c
  4: src/main.o
  5: src/version.h
  6: src/version.h.in
```

- 输入“6”：

```
Add untracked>> 6
* 1: src/Makefile
  2: src/hello
* 3: src/main.c
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



```
    4: src/main.o
    5: src/version.h
  * 6: src/version.h.in
  Add untracked>>
```

每次输入文件序号，对应的文件前面都添加一个星号，代表将此文件添加到暂存区。在提示符“Add untracked>>”处按回车键，完成文件添加，返回主界面。

```
Add untracked>>
added 3 paths

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

此时输入“1”查看状态，可以看到三个文件添加到暂存区中。

```
What now> 1
           staged     unstaged path
  1:       +20/-0      nothing src/Makefile
  2:       +10/-0      nothing src/main.c
  3:        +6/-0      nothing src/version.h.in

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
```

输入“7”退出交互界面。

查看文件状态，可以发现三个文件被添加到暂存区中。

```
$ git status -s
A  src/Makefile
A  src/main.c
A  src/version.h.in
?? src/hello
?? src/main.o
?? src/version.h
```

完成提交。

```
$ git commit -m "Hello world initialized."
[master d71ce92] Hello world initialized.
 3 files changed, 36 insertions(+), 0 deletions(-)
 create mode 100644 src/Makefile
 create mode 100644 src/main.c
 create mode 100644 src/version.h.in
```

## 10.7 Hello World 引发的新问题

到 src 目录中，对 Hello World 执行编译。

```
$ cd /path/to/my/workspace/demo/src
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
$ make clean && make
rm -f hello main.o version.h
version.h.in => version.h
cc     -c -o main.o main.c
cc -o hello main.o
```

运行编译后的程序，是不是对版本输出不满意呢？

```
$ ./hello
Hello, world.
version: old_practice-4-gd71ce92.
```

之所以显示长长的版本号，是因为使用了在本章最开始留的“影”。现在为 Hello world 留下一个新的“影”（一个新的里程碑）吧。

```
$ git tag -m "Set tag hello_1.0." hello_1.0
```

然后清除上次编译结果后，重新编译和运行，可以看到新的输出。

```
$ make clean && make
rm -f hello main.o version.h
version.h.in => version.h
cc     -c -o main.o main.c
cc -o hello main.o
$ ./hello
Hello, world.
version: hello_1.0.
```

还不错，显示了新的版本号。此时在工作区查看状态，会发现工作区“不干净”。

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       hello
#       main.o
#       version.h
```

编译的目标文件和从模板生成的头文件出现在了 Git 的状态输出中，这些文件会对以后的工作造成干扰。当写了新的源代码文件需要添加到版本库中时，因为这些干扰文件的存在，不得不逐一挑选要添加的文件。更为严重的是，如果不小心执行 `git add .` 或 `git add -A` 命令会将编译的目标文件及其他临时文件加入版本库中，浪费存储空间不说，甚至还会造成冲突。

Git 提供了文件忽略功能，可以用来解决这个问题。

## 10.8 文件忽略

Git 提供了文件忽略功能。当对工作区某个目录或某些文件设置了忽略后，再执行 `git`

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


status 查看状态时，被忽略的文件即使存在也不会显示为未跟踪状态，甚至根本感觉不到这些文件的存在。现在就针对 Hello world 程序目录试验一下。

```
$ cd /path/to/my/workspace/demo/src
$ git status -s
?? hello
?? main.o
?? version.h
```

可以看到 src 目录下编译的目标文件等显示为未跟踪，每一行开头的两个问号好像在向我们请求："快把我们添加到版本库里吧"。

执行下面的命令可以在这个目录下创建一个名为 .gitignore 的文件（注意文件的前面有个点），把这些要忽略的文件写在其中，文件名可以使用通配符。注意：第 2 行到第 5 行开头的右尖括号是 cat 命令的提示符，不是用户的输入。

```
$ cat > .gitignore << EOF
> hello
> *.o
> *.h
> EOF
```

看看写好的 .gitignore 文件。每个要忽略的文件显示在一行。

```
$ cat .gitignore
hello
*.o
*.h
```

再来看看当前工作区的状态。

```
$ git status -s
?? .gitignore
```

把 .gitignore 文件添加到版本库中吧。（如果不希望添加到库里，也不希望 .gitignore 文件带来干扰，可以在忽略文件中忽略自己。）

```
$ git add .gitignore
$ git commit -m "ignore object files."
[master b3af728] ignore object files.
 1 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 src/.gitignore
```

### 1. 文件 .gitignore 可以放在任何目录中

文件 .gitignore 的作用范围是其所处的目录及其子目录，因此如果把刚刚创建的 .gitignore 移动到上一层目录（仍位于工作区内）也应该有效，移动操作如下：

```
$ git mv .gitignore ..
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
#
#       renamed:    .gitignore -> ../.gitignore
#
```

果然移动 .gitignore 文件到上层目录，Hello world 程序目录下的目标文件依然被忽略着。

执行提交：

```
$ git commit -m "move .gitignore outside also works."
[master 3488f2c] move .gitignore outside also works.
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename src/.gitignore => .gitignore (100%)
```

### 2. 忽略文件有错误，后果很严重

实际上，上面写的忽略文件不是非常好，为了忽略 version.h，结果使用了通配符 *.h 会把源码目录下的有用的头文件也给忽略掉，导致应该添加到版本库的文件忘记添加。

在当前目录下创建一个新的头文件 hello.h。

```
$ echo "/* test */" > hello.h
```

在工作区状态显示中看不到 hello.h 文件。

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

只有使用了 --ignored 参数，才会在状态显示中看到被忽略的文件。

```
$ git status --ignored -s
!! hello
!! hello.h
!! main.o
!! version.h
```

要添加 hello.h 文件，使用 git add -A 和 git add . 都失效。无法用这两个命令将 hello.h 添加到暂存区中。

```
$ git add -A
$ git add .
$ git status -s
```

只有在添加操作的命令行中明确地写入文件名，并且提供 -f 参数才能真正添加。

```
$ git add -f hello.h
$ git commit -m "add hello.h"
[master 48456ab] add hello.h
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 src/hello.h
```

### 3. 忽略只对未跟踪文件有效，对于已加入版本库的文件无效

文件 hello.h 添加到版本库后，就不再受到 .gitignore 设置的文件忽略所影响了，

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


对 hello.h 的修改都会立刻被跟踪到。这是因为 Git 的文件忽略只是对未入库的文件起作用。

```
$ echo "/* end */" >> hello.h
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello.h
#
no changes added to commit (use "git add" and/or "git commit -a")
```

偷懒式提交。（使用了 -a 参数提交，不用预先执行 git add 命令。）

```
$ git commit -a -m "偷懒了，直接用 -a 参数直接提交。"
[master 613486c] 偷懒了，直接用 -a 参数直接提交。
 1 files changed, 1 insertions(+), 0 deletions(-)
```

#### 4. 本地独享式忽略文件

文件 .gitignore 设置的文件忽略是共享式的。之所以称其为“共享式”，是因为 .gitignore 被添加到版本库后成为了版本库的一部分，当版本库共享给他人（克隆），或者把版本库推送（PUSH）到集中式的服务器（或他人的版本库）时，这个忽略文件就会出现在他人的工作区中，文件忽略在他人的工作区中同样生效。

与“共享式”忽略对应的是“独享式”忽略。独享式忽略就是不会因为版本库共享，或者版本库之间的推送传递给他人的文件忽略。独享式忽略有两种方式：

- 一种是针对具体版本库的“独享式”忽略。即在版本库 .git 目录下的一个文件 .git/info/exclude 来设置文件忽略。
- 另外一种是全局的“独享式”忽略。即通过 Git 的配置变量 core.excludesfile 指定的一个忽略文件，其设置的忽略对所有本地版本库均有效。

至于哪些情况需要通过向版本库中提交 .gitignore 文件设置共享式的文件忽略，哪些情况通过 .git/info/exclude 设置只对本地有效的独享式文件忽略，这取决于要设置的文件忽略是否具有普遍意义。如果文件忽略对于所有使用此版本库工作的人都有益，就通过在版本库相应的目录下创建一个 .gitignore 文件建立忽略；否则，如果是需要忽略工作区中创建的一个试验目录或试验性的文件，则使用本地忽略。

例如，我的本地就设置着一个全局的独享的文件忽略列表（这个文件名可以随意设置）：

```
$ git config --global core.excludesfile /home/jiangxin/.gitignore
$ git config core.excludesfile
/home/jiangxin/.gitignore

$ cat /home/jiangxin/.gitignore
*~          # vim 临时文件
*.pyc       # python 的编译文件
.*.mmx      # 不是正则表达式哦，因为 FreeMind-MMX 的辅助文件以点开头
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




### 5. Git 忽略语法

关于 Git 的忽略文件的语法规则再多说几句：

- 忽略文件中的空行或以井号（#）开始的行会被忽略。
- 可以使用通配符，参见 Linux 手册：glob(7)。例如：星号（*）代表任意多字符，问号（?）代表一个字符，方括号（[abc]）代表可选字符范围等。
- 如果名称的最前面是一个路径分隔符（/），表明要忽略的文件在此目录下，而非子目录的文件。
- 如果名称的最后面是一个路径分隔符（/），表明要忽略的是整个目录，同名文件不忽略，否则同名的文件和目录都忽略。
- 通过在名称的最前面添加一个感叹号（!），代表不忽略。

下面的文件忽略示例，包含了上述要点：

```
# 这是注释行 -- 被忽略
*.a       # 忽略所有以 .a 为扩展名的文件。
!lib.a    # 但是 lib.a 文件或目录不要忽略，即使前面设置了对 *.a 的忽略。
/TODO     # 只忽略此目录下的 TODO 文件，子目录的 TODO 文件不忽略。
build/    # 忽略所有 build/ 目录下的文件。
doc/*.txt # 忽略文件如 doc/notes.txt，但是文件如 doc/server/arch.txt 不被忽略。
```

## 10.9 文件归档

如果使用压缩工具（tar、7zip、winzip、rar 等）将工作区文件归档，一不小心会把版本库（.git 目录）包含其中，甚至将工作区中的忽略文件、临时文件也包含其中。Git 提供了一个归档命令：`git archive`，可以对任意提交对应的目录树建立归档。示例如下：

- 基于最新提交建立归档文件 `latest.zip`。

```
$ git archive -o latest.zip HEAD
```

- 只将目录 `src` 和 `doc` 建立到归档 `partial.tar` 中。

```
$ git archive -o partial.tar  HEAD src doc
```

- 基于里程碑 v1.0 建立归档，并且为归档中的文件添加目录前缀 1.0。

```
$ git archive --format=tar --prefix=1.0/ v1.0 | gzip > foo-1.0.tar.gz
```

在建立归档时，如果使用树对象 ID 进行归档，则使用当前时间作为归档中文件的修改时间，而如果使用提交 ID 或里程碑等，则使用提交建立的时间作为归档中文件的修改时间。

如果使用 tar 格式建立归档，并且使用提交 ID 或里程碑 ID，还会把提交 ID 记录在归档文件的文件头中。记录在文件头中的提交 ID 可以通过 `git tar-commit-id` 命令获取。

如果希望在建立归档时忽略某些文件或目录，可以通过为相应文件或目录建立 `export-ignore` 属性加以实现。具体参见本书第 8 篇第 41 章“41.1 属性”一节。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第11章 历史穿梭

经过了之前众多的实践，版本库中已经积累了很多次提交了，从下面的命令中可以看出有 14 次提交。

```
$ git rev-list HEAD | wc -l
14
```

有很多工具可以研究和分析 Git 的历史提交，在前面的实践中，我们已经多次用到相关的 Git 命令查看历史提交、查看文件的历史版本、进行差异比较等。本章除了对之前用到的相关 Git 命令作一下总结外，还要再介绍几款图形化的客户端。

## 11.1 图形工具：gitk

gitk 是最早实现的一个图形化的 Git 版本库浏览器软件，基于 Tcl/Tk 实现，因此 gitk 非常简洁，本身就是由一个 1 万多行的 tcl 脚本写成的。gitk 的代码已经和 Git 的代码放在了同一个版本库中，gitk 随 Git 一同发布，不用特别地安装即可运行。gitk 可以显示提交的分支图，可以显示提交、文件、版本间的差异等。

在版本库中调用 gitk，就会浏览该版本库，显示其提交的分支图。gitk 可以像命令行工具一样使用不同的参数进行调用。

- 显示所有的分支。

```
$ gitk --all
```

- 显示 2 周以来的所有提交。

```
$ gitk --since="2 weeks ago"
```

- 显示某个里程碑（v2.6.12）以来，针对某些目录和文件（include/scsi 目录和 drivers/scsi 目录）的提交。

```
$ gitk v2.6.12.. include/scsi drivers/scsi
```

图 11-1 就是在 DEMO 版本库中运行 gitk --all 的显示。

从图 11-1 中可见不同颜色和形状区分的引用：

- 绿色的 master 分支。
- 黄色的 hello_1.0 和 old_practice 里程碑。
- 灰色的 stash。

gitk 使用 Tcl/Tk 开发，在显示上没有系统中原生图形应用那么漂亮的界面，甚至可以用

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


丑陋来形容。gitk 只能用于版本库浏览，如果要使用 Tcl/Tk 图形界面进行提交则需要使用另外的命令： `git gui`（或 `git citool`）命令，不过下面将要介绍的 gitg 和 qgit 在易用性上比 gitk 及 git gui 进步了不少。

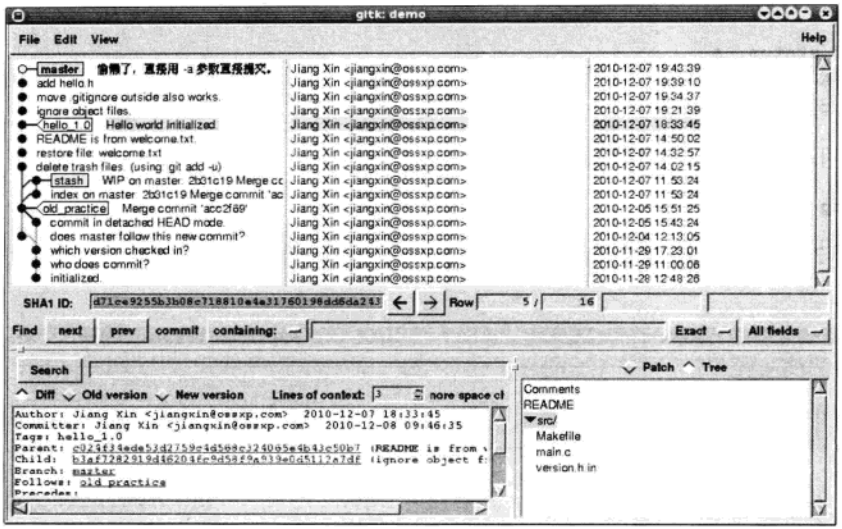



图 11-1 gitk 查看 DEMO 版本库

## 11.2 图形工具：gitg

gitg 是使用 GTK+ 图形库实现的一个 Git 版本库浏览器软件。Linux 下最著名的 Gnome 桌面环境使用的就是 GTK+，因此在 Linux 下 gitg 有着非常漂亮的原生的图形界面。gitg 不但能够实现 gitk 的全部功能，即浏览提交历史和文件，还能帮助执行提交。

在 Linux 上安装 gitg 很简单，例如在 Debian 或 Ubuntu 上，直接运行下面的命令就可以进行安装。

```
$ sudo aptitude install gitg
```

安装完毕后就可以在可执行路径中找到 gitg。

```
$ which gitg
/usr/bin/gitg
```

为了演示 gitg 具备提交功能，先在工作区做出一些修改。

（1）删除没有用到的 `hello.h` 文件。

```
$ cd /path/to/my/workspace/demo
$ rm src/hello.h
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（2）在 README 文件后面追加一行。

```
$ echo "Wait..." >> README
```

（3）查看当前工作区的状态。

```
$ git status -s
 M README
 D src/hello.h
```

现在可以在工作区下执行 gitg 命令。

```
$ gitg &
```

图 11-2 就是 gitg 的默认界面，显示了提交分支图，以及选中提交的提交信息和变更文件列表等。

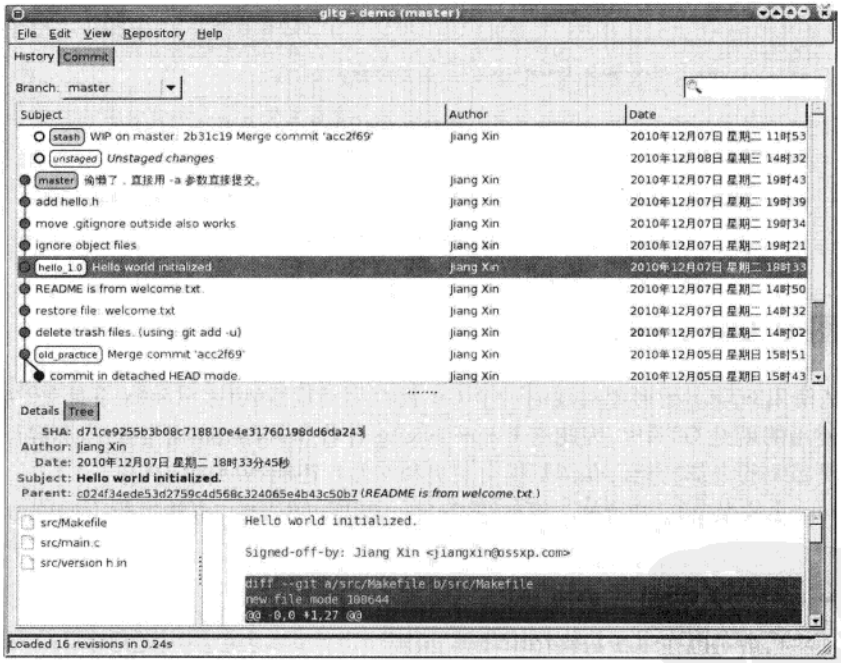



图 11-2　gitg 查看 DEMO 版本库

从图 11-2 中可以看见用不同颜色的标签显示的状态标识（包括引用）：

- 橙色的 master 分支。
- 黄色的 hello_1.0 和 old_practice 里程碑。
- 粉色的 stash 标签。
- 白色的显示工作区非暂存状态的标签。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


点击 gitg 下方窗口的标签“tree”，会显示此提交的目录树，如图 11-3 所示。

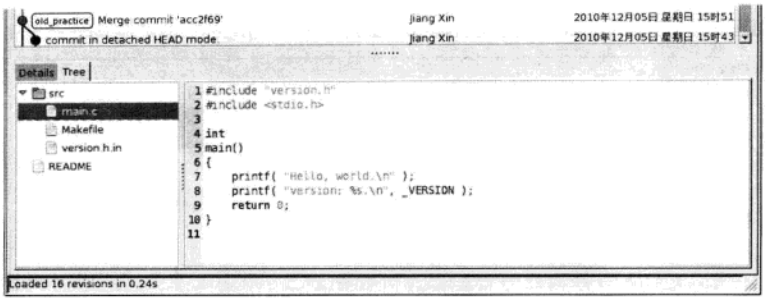



图 11-3 gitg 查看目录树

提交功能是 gitg 的一大特色。点击 gitg 顶部窗口的 commit 标签，显示如图 11-4 的界面。

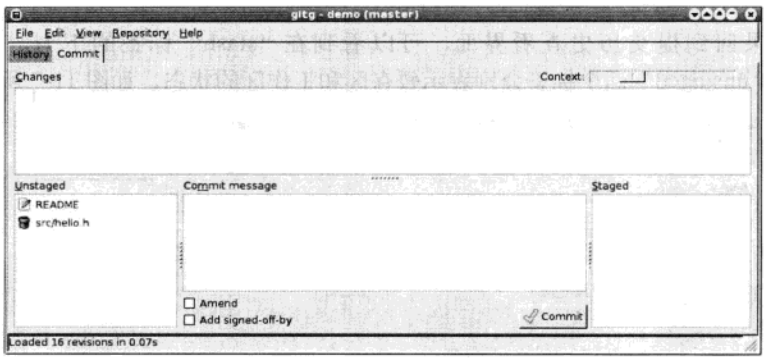



图 11-4 gitg 的提交界面

图 11-4 中，左下方窗口显示的是未更新到暂存区的本地改动。鼠标右击，在弹出菜单中选择“Stage”，如图 11-5 所示。

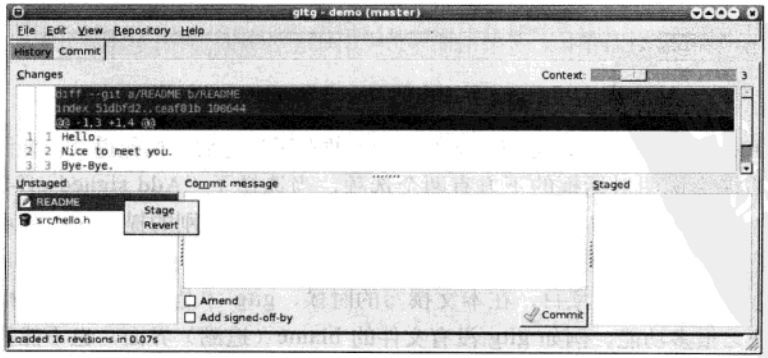



图 11-5 gitg 提交界面中的弹出菜单

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


当把文件 README 添加到暂存区后，可以看到 README 文件出现在右下方的窗口中，如图 11-6 所示。

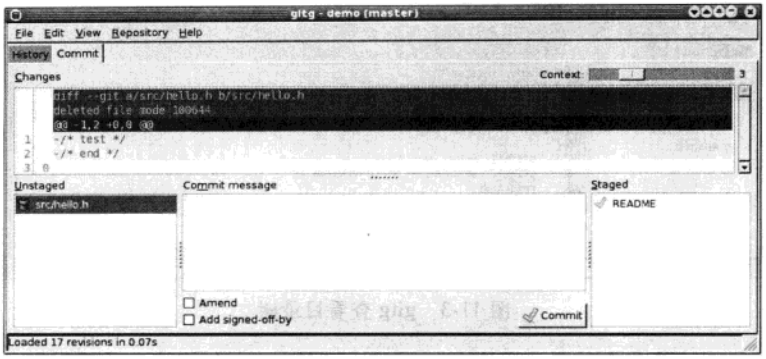



图 11-6 gitg 提交文件加入暂存区

此时如果回到提交历史查看界面，可以看到在“stash”标签的下方，同时出现了“staged”和“unstaged”两个标签分别表示暂存区和工作区的状态，如图 11-7 所示。

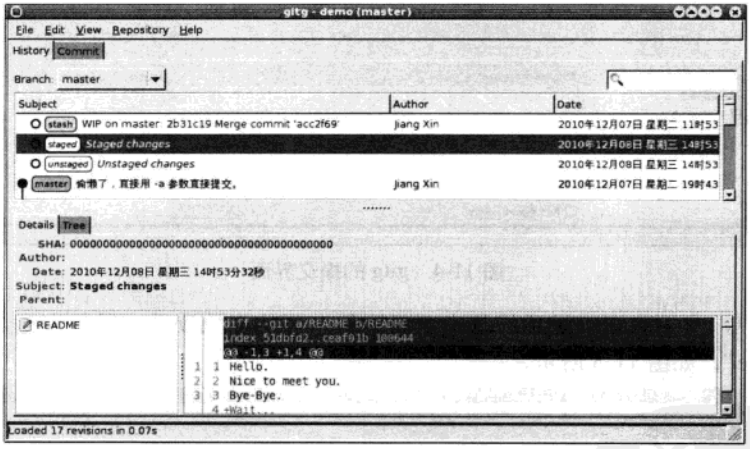



图 11-7 gitg 界面中的 staged 和 unstaged 标签

当通过 gitg 的界面选择好要提交的文件（加入暂存区）之后，执行提交，如图 11-8 所示。

图 11-8 的提交说明对话框的下方有两个选项，当选择了“Add signed-off-by”选项后，在提交日志中会自动增加相应的说明文字。图 11-9 可以看到刚刚的提交已经显示在提交历史的最顶端，在提交说明中出现了 Signed-off-by 文字说明。

gitg 还是一个比较新的项目，在本文撰写的时候，gitg 才是 0.0.6 版本，相比下面要介绍的 qgit 还缺乏很多功能。例如 gitg 没有文件的 blame（追溯）界面，也不能直接将文件检出，但是 gitg 整体的界面风格，以及易用的提交界面给人的印象非常深刻。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



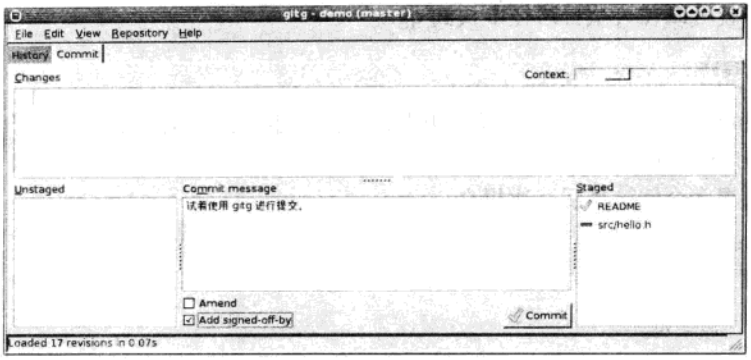




图 11-8 gitg 执行提交

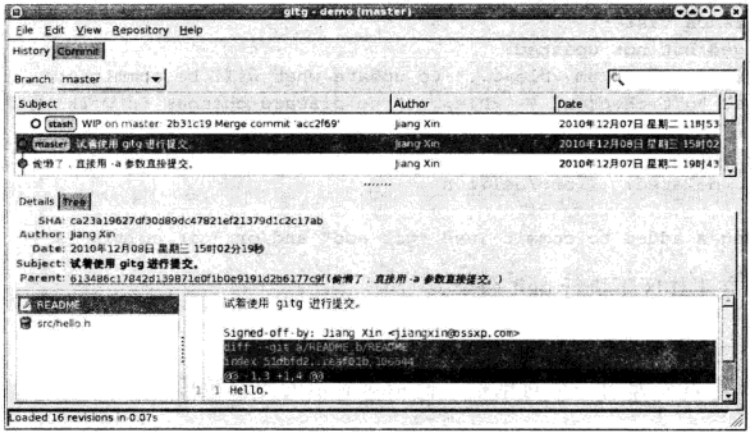



图 11-9 gitg 显示的最新提交

## 11.3 图形工具：qgit

前面介绍的 gitg 是基于 GTK+ 这一 Linux 标准的图形库，那么也许有读者已经猜到 qgit 是使用 Linux 另外一个著名的图形库 QT 实现的。QT 的知名度不亚于 GTK+，是著名的 KDE 桌面环境用到的图形库，也是蓄势待发准备和 Android 一较高低的 MeeGo 的 UI 核心。qgit 目前的版本是 2.3，相比前面介绍的 gitg，其经历的开发周期要长了不少，因此也提供了更多的功能。

在 Linux 上安装 qgit 很简单，例如在 Debian 或 Ubuntu 上，直接运行下面的命令就可以进行安装。

```
$ sudo aptitude install qgit
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


安装完毕就可以在可执行路径中找到 qgit。

```
$ which qgit
/usr/bin/qgit
```

qgit 和 gitg 一样不但能够浏览提交历史和文件，还能帮助执行提交。为了测试提交，将回滚在上一节所做的提交。

❑ 使用重置命令回滚最后一次提交。

```
$ git reset HEAD^
Unstaged changes after reset:
M       README
M       src/hello.h
```

❑ 查看当前工作区的状态。

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README
#       deleted:    src/hello.h
#
no changes added to commit (use "git add" and/or "git commit -a")
```

现在可以在工作区下执行 qgit 命令。

```
$ qgit &
```

启动 qgit，首先弹出一个对话框，提示对显示的提交范围和分支范围进行选择，如图 11-10 所示。

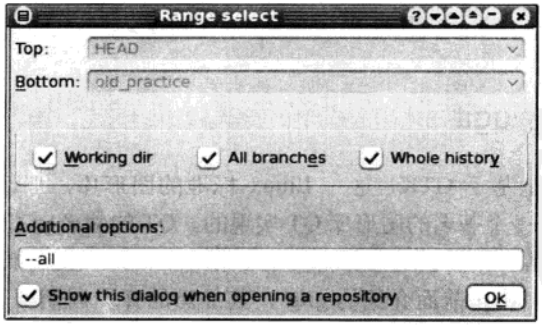



图 11-10 qgit 启动对话框

对所有的选择打钩，显示下面的 qgit 的默认界面。其中包括了提交分支图，以及选中提交的提交信息和变更文件列表等，如图 11-11 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





图 11-11 qgit 主界面

从图 11-11 中可以看见用不同颜色的标签显示的状态标识（包括引用）：

- 绿色的 master 分支。
- 黄色的 hello_1.0 和 old_practice 里程碑。
- 灰色的 stash 标签，显示了创建时候的位置，并将其包含的针对暂存区状态的提交也显示了出来。
- 最顶端显示一行绿色背景的文字：工作区有改动。

qgit 的右键菜单非常丰富，图 11-11 显示了鼠标右击提交时显示的弹出菜单，可以创建、切换标签或分支，可以将提交导出为补丁文件。

点击 qgit 右下方的变更文件列表窗口，可以选择将文件检出或直接查看，如图 11-12 所示。

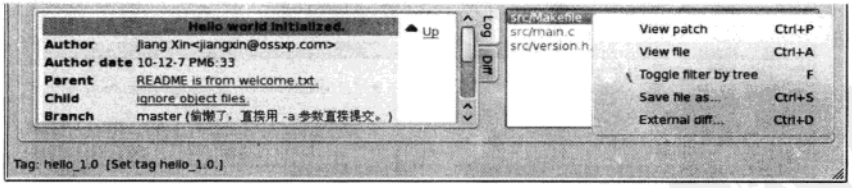



图 11-12 qgit 检出和查看文件界面

要想显示目录树，键入大写字母 T，或者单击工具条上的图标，就会在左侧显示目录树窗口，如图 11-13 所示。

图 11-13 中也显示了目录树中弹出的右键菜单。当选择查看一个文件时，会显示此文件的追溯，即显示每一行是在哪个版本由谁修改的。追溯窗口见图 11-14 右下方的窗口。

qgit 也可以执行提交。选中 qgit 顶部窗口最上一行“Working dir changes”，鼠标右击，显示的弹出菜单包含了“Commit...”选项，如图 11-15 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





图 11-13 qgit 目录树视图

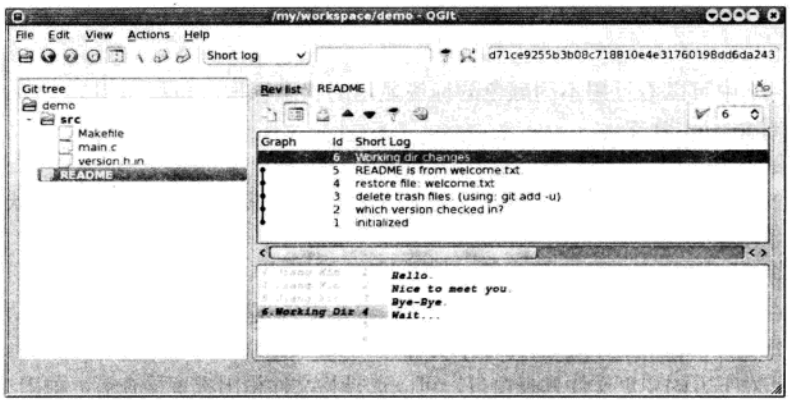



图 11-14 qgit 中的追溯界面

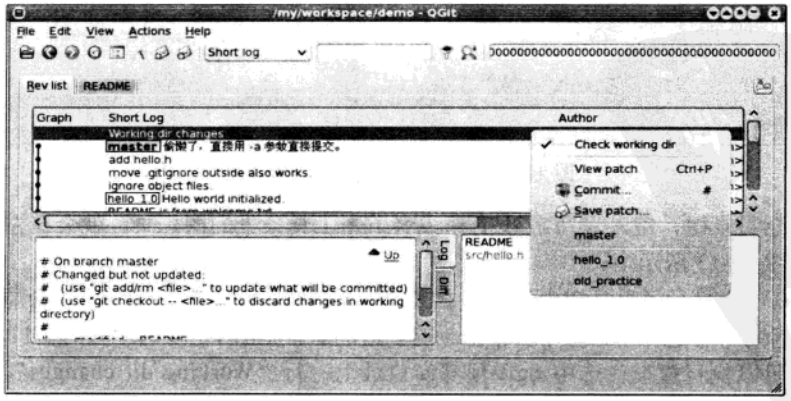



图 11-15 从 qgit 弹出菜单选择提交

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


点击弹出菜单中的“Commit...”，显示如图 11-16 所示的对话框。

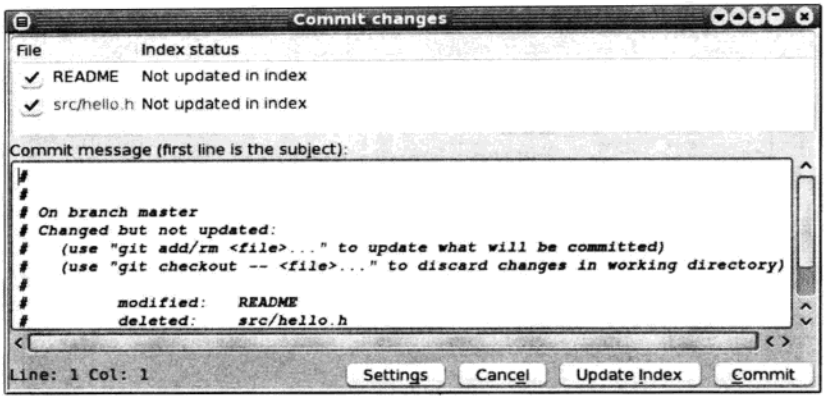



图 11-16 qgit 提交界面

如图 11-16 所示，自动选中了所有的文件。上方窗口的选中文件目前的状态是“Not updated in index”，也就是说尚未添加到暂存区。

使用 qgit 做提交，只要选择好要提交的文件列表，即使未添加到暂存区，也可以直接提交。在下方的提交窗口中写入提交日志，点击“Commit”按钮开始提交，如图 11-17 所示。

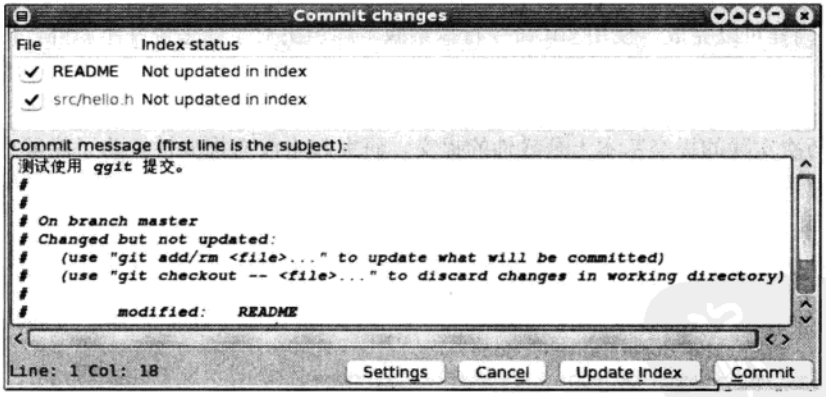



图 11-17 qgit 中编辑提交说明

提交完毕返回 qgit 主界面，在显示的提交列表的最上方，原来显示的“Working dir changes”已经更新为“Nothing to commit”，并且可以看到刚刚的提交已经显示在提交历史的最顶端，如图 11-18 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

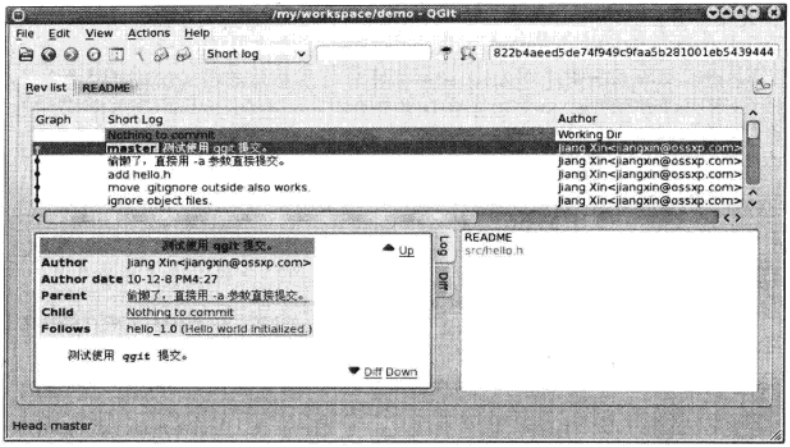




图 11-18 qgit 中显示的最新提交

## 11.4 命令行工具

上面介绍的几款图形界面的 Git 版本库浏览器最大的特色就是更好看的提交关系图，还能非常方便地浏览历史提交的目录树，并从历史提交的目录树中提取文件等。这些操作对于 Git 命令行同样可以完成。使用 Git 命令行探索版本库历史对于您来说并不新鲜，因为在前几章的实践中已经用到了相关命令，展示了对历史记录的操作。本节将对这些命令的部分要点进行强调和补充。

前面历次实践的提交基本上是线性的提交，研究起来没有挑战性。为了能够更加接近于实际又不失简洁，我构造了一个版本库放在了 Github 上。可以通过如下操作在本地克隆这个示例版本库。

```
$ cd /path/to/my/workspace/
$ git clone git://github.com/ossxp-com/gitdemo-commit-tree.git
Cloning into gitdemo-commit-tree...
remote: Counting objects: 63, done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 63 (delta 8), reused 0 (delta 0)
Receiving objects: 100% (63/63), 65.95 KiB, done.
Resolving deltas: 100% (8/8), done.
$ cd gitdemo-commit-tree
```

运行 gitg 命令，显示其提交关系图，如图 11-19 所示。

是不是有点“乱花渐欲迷人眼”的感觉。如果把提交用里程碑标识的圆圈来代表，稍加排列就会看到下面的更为直白的提交关系图，如图 11-20 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

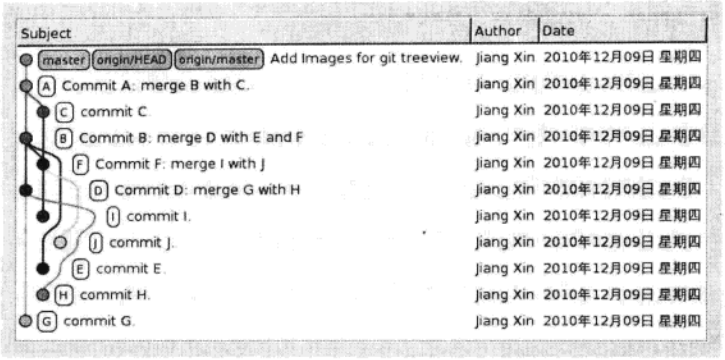




图 11-19 演示版本库的提交分支图

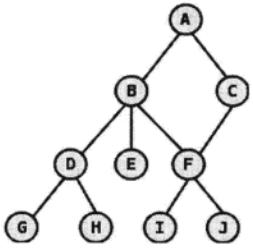



图 11-20 简化的提交分支图

Git 的大部分命令可以使用提交版本作为参数（如：`git diff <commit-id>`），有的命令则使用一个版本范围作为参数（如：`git log <rev1>..<rev2>`）。Git 的提交有着各式各样的表示法，提交范围也是一样，下面就通过两个命令 `git rev-parse` 和 `git rev-list` 分别研究一下 Git 的版本表示法和版本范围表示法。

### 11.4.1 版本表示法：git rev-parse

命令 `git rev-parse` 是 Git 的一个底层命令，其功能非常丰富（或者说杂乱），很多 Git 脚本或工具都会用到这条命令。

此命令的部分应用在“第 4 章 Git 初始化”一章中就已经看到。例如可以显示 Git 版本库的位置（`--git-dir`），当前工作区目录的深度（`--show-cdup`），甚至可以被 Git 无关的应用用于解析命令行参数（`--parseopt`）。

此命令可以显示当前版本库中的引用。

- 显示分支。

```
$ git rev-parse --symbolic --branches
master
```

- 显示里程碑。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git rev-parse --symbolic --tags
A
B
C
D
E
F
G
H
I
J
```

❑ 显示定义的所有引用。

其中 refs/remotes/ 目录下的引用称为远程分支（或远程引用），在后面的章节会予以介绍。

```
$ git rev-parse --symbolic --glob=refs/*
refs/heads/master
refs/remotes/origin/HEAD
refs/remotes/origin/master
refs/tags/A
refs/tags/B
refs/tags/C
refs/tags/D
refs/tags/E
refs/tags/F
refs/tags/G
refs/tags/H
refs/tags/I
refs/tags/J
```

命令 git rev-parse 的另外一个重要功能就是将一个 Git 对象表达式表示为对应的 SHA1 哈希值。针对本节开始克隆的版本库 gitdemo-commit-tree，做如下操作。

（1）显示 HEAD 对应的 SHA1 哈希值。

```
$ git rev-parse  HEAD
6652a0dce6a5067732c00ef0a220810a7230655e
```

（2）命令 git describe 的输出也可以解析为正确的 SHA1 哈希值。

```
$ git describe
A-1-g6652a0d
$ git rev-parse A-1-g6652a0d
6652a0dce6a5067732c00ef0a220810a7230655e
```

（3）可以同时显示多个表达式的 SHA1 哈希值。

下面的操作可以看出 master 和 refs/heads/master 都可以用于指代 master 分支。

```
$ git rev-parse  master  refs/heads/master
6652a0dce6a5067732c00ef0a220810a7230655e
6652a0dce6a5067732c00ef0a220810a7230655e
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




（4）可以用哈希值的前几位指代整个哈希值。

```
$ git rev-parse  6652  6652a0d
6652a0dce6a5067732c00ef0a220810a7230655e
6652a0dce6a5067732c00ef0a220810a7230655e
```

（5）里程碑的两种表示法均指向相同的对象。

```
$ git rev-parse  A  refs/tags/A
c9b03a208288aebdbfe8d84aeb984952a16da3f2
c9b03a208288aebdbfe8d84aeb984952a16da3f2
```

（6）里程碑 A 实际指向的是一个 Tag 对象而非提交，用下面的三个表示法可以显示提交本身的哈希值而非里程碑对象的哈希值。

还有，下面的语法也可以直接作用于轻量级里程碑（直接指向提交的里程碑），或者作用于提交本身。

```
$ git rev-parse  A^{}  A^0  A^{commit}
81993234fc12a325d303eccea20f6fd629412712
81993234fc12a325d303eccea20f6fd629412712
81993234fc12a325d303eccea20f6fd629412712
```

（7）A 的第一个父提交就是 B 所指向的提交。

回忆之前的介绍，“^”操作符代表父提交。当一个提交有多个父提交时，可以通过在符号“^”后面跟上一个数字表示第几个父提交。“A^”就相当于“A^1”。而 B^0 代表了 B 所指向的一个 Commit 对象（因为 B 是 Tag 对象）。

```
$ git rev-parse  A^  A^1  B^0
776c5c9da9dcbb7e463c061d965ea47e73853b6e
776c5c9da9dcbb7e463c061d965ea47e73853b6e
776c5c9da9dcbb7e463c061d965ea47e73853b6e
```

（8）更为复杂的表示法。

连续的“^”符号依次沿着父提交进行定位至某一祖先提交。“^”后面的数字代表该提交的第几个父提交。

```
$ git rev-parse  A^^3^2  F^2  J^{}
3252fcce40949a4a622a1ac012cb120d6b340ac8
3252fcce40949a4a622a1ac012cb120d6b340ac8
3252fcce40949a4a622a1ac012cb120d6b340ac8
```

（9）记号 ~<n> 就相当于连续 <n> 个符号“^”。

```
$ git rev-parse  A~3  A^^^  G^0
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
```

（10）显示里程碑 A 对应的目录树。下面两种写法都可以。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
`$ git rev-parse  A^{tree}  A:
95ab9e7db14ca113d5548dc20a4872950e8e08c0
95ab9e7db14ca113d5548dc20a4872950e8e08c0
```

（11）显示树里面的文件，下面两种表示法均可。

```
$ git rev-parse  A^{tree}:src/Makefile  A:src/Makefile
96554c5d4590dbde28183e9a6a3199d526eeb925
96554c5d4590dbde28183e9a6a3199d526eeb925
```

（12）暂存区里的文件和 HEAD 中的文件相同。

```
$ git rev-parse  :gitg.png  HEAD:gitg.png
fc58966ccc1e5af24c2c9746196550241bc01c50
fc58966ccc1e5af24c2c9746196550241bc01c50
```

（13）还可以通过在提交日志中查找字串的方式显示提交。

```
$ git rev-parse :/"Commit A"
81993234fc12a325d303eccea20f6fd629412712
```

（14）再有就是 reflog 相关的语法，参见“第 7 章 Git 重置”一章中关于 reflog 的介绍。

```
$ git rev-parse HEAD@{0} master@{0}
6652a0dce6a5067732c00ef0a220810a7230655e
6652a0dce6a5067732c00ef0a220810a7230655e
```

### 11.4.2 版本范围表示法：git rev-list

有的 Git 命令使用一个版本范围作为参数，命令 git rev-list 可以帮助研究 Git 的各种版本范围的语法。

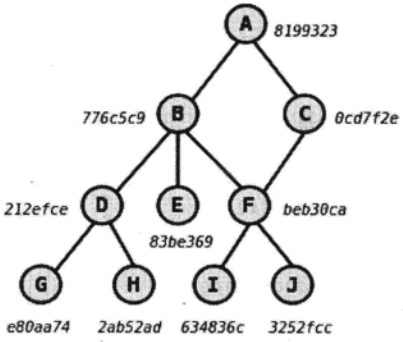



图 11-21 标记了提交 ID 的分支图

- 如图 11-21 所示，一个提交 ID 实际上就可以代表一个版本列表，含义是该版本开始的所有历史提交。

```
$ git rev-list --oneline  A
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
8199323 Commit A: merge B with C.
0cd7f2e commit C.
776c5c9 Commit B: merge D with E and F
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
634836c commit I.
3252fcc commit J.
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 两个或多个版本，相当于每个版本单独使用时指代的列表的并集。

```
$ git rev-list --oneline  D  F
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
634836c commit I.
3252fcc commit J.
2ab52ad commit H.
e80aa74 commit G.
```

- 在一个版本前面加上符号（^）含义是取反，即排除这个版本及其历史版本。

```
$ git rev-list --oneline  ^G D
212efce Commit D: merge G with H
2ab52ad commit H.
```

- 和上面等价的“点点”表示法。使用两个点连接两个版本，如 G..D，就相当于 ^G D。

```
$ git rev-list --oneline  G..D
212efce Commit D: merge G with H
2ab52ad commit H.
```

- 版本取反，参数的顺序不重要，但是“点点”表示法前后的版本顺序很重要。
  - 语法：^B C

```
$ git rev-list --oneline  ^B C
0cd7f2e commit C.
```

  - 语法：C ^B

```
$ git rev-list --oneline  C ^B
0cd7f2e commit C.
```

  - 语法：B..C 相当于 ^B C

```
$ git rev-list --oneline  B..C
0cd7f2e commit C.
```

  - 语法：C..B 相当于 ^C B

```
$ git rev-list --oneline  C..B
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
776c5c9 Commit B: merge D with E and F
212efce Commit D: merge G with H
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 三点表示法的含义是两个版本共同能够访问到的除外。如B...C将B和C共同能够访问到的F、I、J排除在外。

```
$ git rev-list --oneline  B...C
0cd7f2e commit C.
776c5c9 Commit B: merge D with E and F
212efce Commit D: merge G with H
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 三点表示法，两个版本的前后顺序没有关系。实际上r1...r2相当于r1 r2 --not $(git merge-base --all r1 r2)，所以和顺序无关。

```
$ git rev-list --oneline  C...B
0cd7f2e commit C.
776c5c9 Commit B: merge D with E and F
212efce Commit D: merge G with H
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 某提交的历史提交，自身除外，用语法r1^@表示。

```
$ git rev-list --oneline  B^@
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
634836c commit I.
3252fcc commit J.
83be369 commit E.
2ab52ad commit H.
e80aa74 commit G.
```

- 提交本身不包括其历史提交，用语法r1^!表示。

```
$ git rev-list --oneline  B^!
776c5c9 Commit B: merge D with E and F

$ git rev-list --oneline  F^! D
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
2ab52ad commit H.
```

### 11.4.3 浏览日志：git log

命令`git log`是老朋友了，在前面的章节中曾经大量地出现，用于显示提交历史。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




### 1. 显示的日志范围

命令 git log 的后面可以接表示版本范围的参数，当不使用任何表示版本范围的参数时，相当于使用了默认的参数 HEAD，即显示当前 HEAD 能够访问到的所有历史提交。下面的示例使用了前面介绍的版本范围表示法：

```
$ git log --oneline F^! D
beb30ca Commit F: merge I with J
212efce Commit D: merge G with H
2ab52ad commit H.
e80aa74 commit G.
```

### 2. 分支图显示

通过 --graph 参数调用 git log 可以显示字符界面的提交关系图，而且不同的分支还可以用不同的颜色来表示。如果希望每次查看日志的时候都看到提交关系图，可以设置一个别名，用别名来调用。

```
$ git config --global alias.glog "log --graph"
```

定义别名之后，每次希望自动显示提交关系图，就可以使用别名命令：

```
$ git glog --oneline
* 6652a0d Add Images for git treeview.
*   8199323 Commit A: merge B with C.
|\
| * 0cd7f2e commit C.
| |
|  \
*-. \   776c5c9 Commit B: merge D with E and F
|\ \ \
| | |/
| | *   beb30ca Commit F: merge I with J
| | |\
| | | * 3252fcc commit J.
| | * 634836c commit I.
| * 83be369 commit E.
*   212efce Commit D: merge G with H
|\
| * 2ab52ad commit H.
* e80aa74 commit G.
```

### 3. 显示最近的几条日志

可以使用参数 -<n>（<n> 为数字），显示最近的 <n> 条日志。例如下面的命令显示最近的 3 条日志。

```
$ git log -3 --pretty=oneline
6652a0dce6a5067732c00ef0a220810a7230655e Add Images for git treeview.
81993234fc12a325d303eccea20f6fd629412712 Commit A: merge B with C.
0cd7f2ea245d90d414e502467ac749f36aa32cc4 commit C.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


#### 4. 显示每次提交的具体改动

使用参数 -p 可以在显示日志的时候同时显示改动。

```
$ git log -p -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 9 16:07:11 2010 +0800

    Add Images for git treeview.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>

diff --git a/gitg.png b/gitg.png
new file mode 100644
index 0000000..fc58966
Binary files /dev/null and b/gitg.png differ
diff --git a/treeview.png b/treeview.png
new file mode 100644
index 0000000..a756d12
Binary files /dev/null and b/treeview.png differ
```

因为是二进制文件改动，默认不显示改动的内容。实际上 Git 的差异文件提供对二进制文件的支持，本书第 7 篇第 38 章将予以专题介绍。

#### 5. 显示每次提交的变更概要

使用 -p 参数会让日志输出显得非常冗余，当不需要知道具体的改动而只想知道改动在哪些文件上时，可以使用 --stat 参数。输出的变更概要像极了 GNU 的 diffstat 命令的输出。

```
$ git log --stat --oneline  I..C
0cd7f2e commit C.
 README     |    1 +
 doc/C.txt  |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
beb30ca Commit F: merge I with J
3252fcc commit J.
 README            |    7 +++++++
 doc/J.txt         |    1 +
 src/.gitignore    |    3 +++
 src/Makefile      |   27 +++++++++++++++++++++++++++
 src/main.c        |   10 ++++++++++
 src/version.h.in  |    6 ++++++
 6 files changed, 54 insertions(+), 0 deletions(-)
```

#### 6. 定制输出

Git 的日志输出命令提供了很多输出模板选择，可以根据需要选择冗余显示或精简显示。参数 --pretty=raw 显示 commit 的原始数据，可以显示提交对应的树 ID。

```
$ git log --pretty=raw -1
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
commit 6652a0dce6a5067732c00ef0a220810a7230655e
tree e33be9e8e7ca5f887c7d5601054f2f510e6744b8
parent 81993234fc12a325d303eccea20f6fd629412712
author Jiang Xin <jiangxin@ossxp.com> 1291882031 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291882892 +0800

    Add Images for git treeview.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
```

参数 --pretty=fuller 会同时显示作者和提交者，两者可以不同。

```
$ git log --pretty=fuller -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Thu Dec 9 16:07:11 2010 +0800
Commit:     Jiang Xin <jiangxin@ossxp.com>
CommitDate: Thu Dec 9 16:21:32 2010 +0800

    Add Images for git treeview.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
```

参数 --pretty=oneline 显然会提供最精简的日志输出。也可以使用 --oneline 参数，效果近似。

```
$ git log --pretty=oneline -1
6652a0dce6a5067732c00ef0a220810a7230655e Add Images for git treeview.
```

如果只想查看和分析某一个提交，也可以使用 git show 或 git cat-file 命令。使用 git show 显示里程碑 D 及其提交：

```
$ git show D --stat
tag D
Tagger: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 9 14:24:52 2010 +0800

create node D

commit 212efce1548795a1edb08e3708a50989fcd73cce
Merge: e80aa74 2ab52ad
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 9 14:06:34 2010 +0800

    Commit D: merge G with H

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>

 README     |    2 ++
 doc/D.txt  |    1 +
 doc/H.txt  |    1 +
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 3 files changed, 4 insertions(+), 0 deletions(-)
```

使用 git cat-file 显示里程碑 D 及其提交。参数 -p 的含义是美观的输出（pretty）。

```
$ git cat-file -p D^0
tree 1c22e90c6bf150ee1cde6cefb476abbb921f491f
parent e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
parent 2ab52ad2a30570109e71b56fa1780f0442059b3c
author Jiang Xin <jiangxin@ossxp.com> 1291874794 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291875877 +0800

Commit D: merge G with H

Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
```

### 11.4.4 差异比较：git diff

Git 差异比较功能在前面的实践中也反复地接触过了，尤其是在介绍暂存区的相关章节重点介绍了 git diff 命令如何对工作区、暂存区、版本库进行比较：

- 比较里程碑 B 和里程碑 A，用命令：git diff B A
- 比较工作区和里程碑 A，用命令：git diff A
- 比较暂存区和里程碑 A，用命令：git diff --cached A
- 比较工作区和暂存区，用命令：git diff
- 比较暂存区和 HEAD，用命令：git diff --cached
- 比较工作区和 HEAD，用命令：git diff HEAD

#### 1. 文件不同版本的差异比较

差异比较还可以使用路径参数，只显示不同版本间该路径下文件的差异。语法格式如下：

```
$ git diff <commit1> <commit2> -- <paths>
```

#### 2. 非 Git 目录 / 文件的差异比较

命令 git diff 还可以在 Git 版本库之外执行，对非 Git 目录进行比较，就像 GNU 的 diff 命令一样。之所以提供这个功能是因为 Git 差异比较命令更为强大，提供了对二进制文件差异等的扩展支持。语法格式如下：

```
$ git diff <path1> <path2>
```

#### 3. 扩展的差异语法

Git 扩展了 GNU 的差异比较语法，提供了对重命名、二进制文件、文件权限变更的支持。本书第 7 篇第 38 章将专题介绍。

#### 4. 逐词比较，而非默认的逐行比较

Git 的差异比较默认是逐行比较，分别显示改动前的行和改动后的行，到底改动在哪里还需要仔细辨别。Git 还提供一种逐词比较的输出，有的人会更喜欢。使用 --word-diff

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




参数可以显示逐词比较。

```
$ git diff --word-diff
diff --git a/src/book/02-use-git/080-git-history-travel.rst b/src/book/02-use-
git/080-git-history-travel.rst
index f740203..2dd3e6f 100644
--- a/src/book/02-use-git/080-git-history-travel.rst
+++ b/src/book/02-use-git/080-git-history-travel.rst
@@ -681,7 +681,7 @@ Git 的大部分命令可以使用提交版本作为参数（如：git diff），

::

   [-18:23:48 jiangxin@hp:~/gitwork/gitbook/src/book$-]{+$+} git log --stat
--oneline  I..C
  0cd7f2e commit C.
   README     |    1 +
   doc/C.txt  |    1 +
```

上面的逐词差异显示是有颜色的：删除内容 [-...-] 用红色表示，添加的内容 {+...+} 用绿色表示。

### 11.4.5 文件追溯：git blame

在软件开发过程中当发现 Bug 并定位到具体的代码时，Git 的文件追溯命令可以指出是谁在什么时候，以及什么版本引入的此 Bug。

当针对文件执行 `git blame` 命令时，就会逐行显示文件，在每一行的行首显示此行最早是在什么版本引入的，由谁引入的。

```
$ cd /path/to/my/workspace/gitdemo-commit-tree
$ git blame README
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800  1) DEMO program for git-scm-book.
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800  2)
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800  3) Changes
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800  4) =======
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800  5)
81993234 (Jiang Xin 2010-12-09 14:30:15 +0800  6) * create node A.
0cd7f2ea (Jiang Xin 2010-12-09 14:29:09 +0800  7) * create node C.
776c5c9d (Jiang Xin 2010-12-09 14:27:31 +0800  8) * create node B.
beb30ca7 (Jiang Xin 2010-12-09 14:11:01 +0800  9) * create node F.
^3252fcc (Jiang Xin 2010-12-09 14:00:33 +0800 10) * create node J.
^634836c (Jiang Xin 2010-12-09 14:00:33 +0800 11) * create node I.
^83be369 (Jiang Xin 2010-12-09 14:00:33 +0800 12) * create node E.
212efce1 (Jiang Xin 2010-12-09 14:06:34 +0800 13) * create node D.
^2ab52ad (Jiang Xin 2010-12-09 14:00:33 +0800 14) * create node H.
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 15) * create node G.
^e80aa74 (Jiang Xin 2010-12-09 14:00:33 +0800 16) * initialized.
```

只想查看某几行，使用 `-L n,m` 参数，命令如下：

```
$ git blame -L 6,+5 README
81993234 (Jiang Xin 2010-12-09 14:30:15 +0800  6) * create node A.
0cd7f2ea (Jiang Xin 2010-12-09 14:29:09 +0800  7) * create node C.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
776c5c9d (Jiang Xin 2010-12-09 14:27:31 +0800  8) * create node B.
beb30ca7 (Jiang Xin 2010-12-09 14:11:01 +0800  9) * create node F.
^3252fcc (Jiang Xin 2010-12-09 14:00:33 +0800 10) * create node J.
```

### 11.4.6 二分查找：git bisect

前面的文件追溯是建立在问题（Bug）已经定位（到代码上）的基础之上，然后才能通过错误的行（代码）找到人（提交者），打板子（教育或惩罚）。那么如何定位问题呢？Git 的二分查找命令可以提供帮助。

二分查找并不神秘，也不是万灵药，是建立在测试的基础之上的。实际上很多做过软件测试的人都可能使用过："最新的版本出现 Bug 了，但是在给某客户的版本却没有这个问题，所以问题肯定出在两者之间的某次代码提交上"。

#### 1. 使用二分查找

Git 提供的 `git bisect` 命令是基于版本库的、自动化的问题查找和定位工具，取代传统软件测试中针对软件发布版本的、无法定位到代码的、粗放式的测试方法。

执行二分查找，在发现问题后，首先要找到一个正确的版本，如果所发现的问题从软件最早的版本就是错的，那么就没有必要执行二分查找了，还是老老实实的 Debug 吧。但是如果能够找到一个正确的版本，即在这个正确的版本上问题没有发生，那么就可以开始使用 `git bisect` 命令在版本库中进行二分查找了：

（1）工作区切换到已知的"好版本"和"坏版本"的中间的一个版本。

（2）执行测试，如果问题重现，则将版本库当前版本库标记为"坏版本"，如果问题没有重现，则将当前版本标记为"好版本"。

（3）重复 1-2，直至最终找到第一个导致问题出现的版本。

图 11-22 是示例版本库标记了提交 ID 后的示意图，在这个示例版本库中试验二分查找流程：首先标记最新提交（HEAD）是"坏的"，G 提交是好的，然后通过查找最终定位到坏提交（B）。

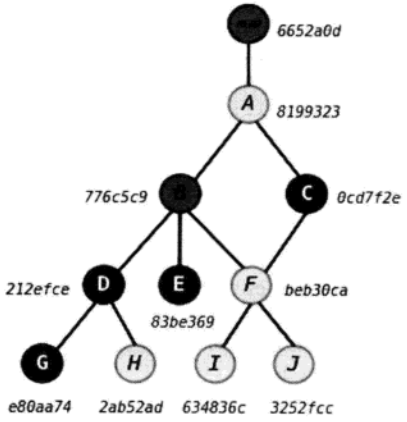



图 11-22 二分法寻找坏提交

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


在下面的试验中定义坏提交的依据很简单，如果在 doc 目录中包含文件 B.txt，则此版本是“坏”的。（这个示例太简陋，不要见笑，聪明的读者可以直接通过 doc/B.txt 文件就可追溯到 B 提交。）

下面开始通过手动测试（查找 doc/B.txt 存在与否），借助 Git 二分查找定位“问题”版本，具体操作步骤如下。

（1）首先确认工作在 master 分支。

```
$ cd /path/to/my/workspace/gitdemo-commit-tree/
$ git checkout master
Already on 'master'
```

（2）开始二分查找。

```
$ git bisect start
```

（3）当前版本已经是“坏提交”，因为存在文件 doc/B.txt。而 G 版本是“好提交”，因为不存在文件 doc/B.txt。

```
$ git cat-file -t master:doc/B.txt
blob
$ git cat-file -t G:doc/B.txt
fatal: Not a valid object name G:doc/B.txt
```

（4）将当前版本（HEAD）标记为“坏提交”，将 G 版本标记为“好提交”。

```
$ git bisect bad
$ git bisect good G
Bisecting: 5 revisions left to test after this (roughly 2 steps)
[0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
```

（5）自动定位到 C 提交。没有文件 doc/B.txt，也是一个好提交。

```
$ git describe
C
$ ls doc/B.txt
ls: 无法访问 doc/B.txt: 没有那个文件或目录
```

（6）标记当前版本（C 提交）为“好提交”。

```
$ git bisect good
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[212efce1548795a1edb08e3708a50989fcd73cce] Commit D: merge G with H
```

（7）现在定位到 D 版本，这也是一个“好提交”。

```
$ git describe
D
$ ls doc/B.txt
ls: 无法访问 doc/B.txt: 没有那个文件或目录
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（8）标记当前版本（D提交）为“好提交”。

```
$ git bisect good
Bisecting: 1 revision left to test after this (roughly 1 step)
[776c5c9da9dcbb7e463c061d965ea47e73853b6e] Commit B: merge D with E and F
```

（9）现在定位到B版本，这是一个“坏提交”。

```
$ git bisect bad
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[83be36956c007d7bfffe13805dd2081839fd3603] commit E.
```

（10）现在定位到E版本，这是一个“好提交”。当标记E为好提交之后，输出显示已经成功定位到引入坏提交的最接近的版本。

```
$ git bisect good
776c5c9da9dcbb7e463c061d965ea47e73853b6e is the first bad commit
```

（11）最终定位的坏提交用引用refs/bisect/bad标识。可以用如下方法切换到该版本。

```
$ git checkout bisect/bad
Previous HEAD position was 83be369... commit E.
HEAD is now at 776c5c9... Commit B: merge D with E and F
```

（12）当对“Bug”定位和修复后，撤销二分查找在版本库中遗留的临时文件和引用。
撤销二分查找后，版本库切换回执行二分查找之前所在的分支。

```
$ git bisect reset
Previous HEAD position was 776c5c9... Commit B: merge D with E and F
Switched to branch 'master'
```

### 2. 把“好提交”标记成了“坏提交”该怎么办?

在执行二分查找的过程中，一不小心就有可能犯错，将“好提交”标记为“坏提交”，或者相反。这将导致前面的查找过程也前功尽弃。为此，Git的二分查找提供了一个恢复查找进度的办法，具体操作过程如下。

（1）例如对提交E，本来是一个“好版本”却被错误的标记为“坏版本”。

```
$ git bisect bad
83be36956c007d7bfffe13805dd2081839fd3603 is the first bad commit
```

（2）用git bisect log命令查看二分查找的日志记录。
把二分查找的日志保存在一个文件中。

```
$ git bisect log > logfile
```

（3）编辑这个文件，删除记录了错误动作的行。
以井号（#）开始的行是注释。

```
$ cat logfile
# bad: [6652a0dce6a5067732c00ef0a220810a7230655e] Add Images for git treeview.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
# good: [e80aa7481beda65ae00e35afc4bc4b171f9b0ebf] commit G.
git bisect start 'master' 'G'
# good: [0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
git bisect good 0cd7f2ea245d90d414e502467ac749f36aa32cc4
# good: [212efce1548795a1edb08e3708a50989fcd73cce] Commit D: merge G with H
git bisect good 212efce1548795a1edb08e3708a50989fcd73cce
# bad: [776c5c9da9dcbb7e463c061d965ea47e73853b6e] Commit B: merge D with E and F
git bisect bad 776c5c9da9dcbb7e463c061d965ea47e73853b6e
```

（4）结束正在进行的出错的二分查找。

```
$ git bisect reset
Previous HEAD position was 83be369... commit E.
Switched to branch 'master'
```

（5）通过日志文件恢复进度，重启二分查找。

```
$ git bisect replay logfile
We are not bisecting.
Bisecting: 5 revisions left to test after this (roughly 2 steps)
[0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[83be36956c007d7bfffe13805dd2081839fd3603] commit E.
```

（6）再一次回到了提交 E，这一次不要标记错了哦。

```
$ git describe
E
$ git bisect good
776c5c9da9dcbb7e463c061d965ea47e73853b6e is the first bad commit
```

### 3. 二分查找使用自动化测试

Git 的二分查找命令支持 run 子命令，可以运行一个自动化测试脚本，实现自动的二分查找。

- ❑ 如果脚本的退出码是 0，正在测试的版本是一个“好版本”。
- ❑ 如果脚本的退出码是 125，正在测试的版本被跳过。
- ❑ 如果脚本的退出码是 1 到 127（125 除外），正在测试的版本是一个“坏版本”。

为本例写一个自动化测试太简单了，无非就是判断文件是否存在，存在则返回错误码 1，不存在则返回错误码 0。测试脚本 good-or-bad.sh 如下：

```
#!/bin/sh

[ -f doc/B.txt ] && exit 1
exit 0
```

用此脚本实现自动化二分查找的过程非常简单，具体操作步骤如下。

（1）从已知的坏版本 master 和好版本 G 开始新一轮的二分查找。

```
$ git bisect start master G
Bisecting: 5 revisions left to test after this (roughly 2 steps)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
[0cd7f2ea245d90d414e502467ac749f36aa32cc4] commit C.
```

（2）自动化测试，使用脚本 good-or-bad.sh。

```
$ git bisect run sh good-or-bad.sh
running sh good-or-bad.sh
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[212efce1548795a1edb08e3708a50989fcd73cce] Commit D: merge G with H
running sh good-or-bad.sh
Bisecting: 1 revision left to test after this (roughly 1 step)
[776c5c9da9dcbb7e463c061d965ea47e73853b6e] Commit B: merge D with E and F
running sh good-or-bad.sh
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[83be36956c007d7bfffe13805dd2081839fd3603] commit E.
running sh good-or-bad.sh
776c5c9da9dcbb7e463c061d965ea47e73853b6e is the first bad commit
bisect run success
```

（3）定位到的“坏版本”是 B。

```
$ git describe refs/bisect/bad
B
```

### 11.4.7 获取历史版本

提取历史提交中的文件无非就是表 11-1 中的操作，在之前的实践中已多次用到，这里不再赘述。

表 11-1 获取文件历史版本命令一览表

| 动作 | 命令格式 | 示例 |
| --- | --- | --- |
| 查看历史提交的目录树 | git ls-tree <tree-ish> <paths> | ○ git ls-tree 776c5c9 README<br>○ git ls-tree -r refs/tags/D doc |
| 整个工作区切换到历史版本 | git checkout <commit> | ○ git checkout HEAD^^ |
| 检出某文件的历史版本 | git checkout <commit> -- <paths> | ○ git checkout refs/tags/D — README<br>○ git checkout 776c5c9 — doc |
| 检出某文件的历史版本到其他文件名 | git show <commit>:<file> > new_name | ○ git show 887113d:README > README.OLD |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第12章 改变历史

我是电影《回到未来》的粉丝，偶尔会做梦，梦见穿梭到未来拿回一本2000-2050体育年鉴。操作Git也可以体验到穿梭时空的感觉，因为Git像极了一个时光机器，不但允许你在历史中穿梭，而且还能够改变历史。

本章的最开始将介绍两种最简单和最常用的历史变更操作——“悔棋”操作，就是对刚刚进行的一次或几次提交进行修补或撤销。对于跳跃式的历史记录的变更，即仅对过去某一个或某几个提交做出改变，会在“回到未来”小节详细介绍。在“丢弃历史”小节会介绍一种版本库瘦身的方法，这可能会在某些特定的场合用到。

作为分布式版本控制系统，一旦版本库被多人共享，改变历史就可能是无法完成的任务。在本章的最后，介绍还原操作以实现在不改变历史提交的情况下还原错误的改动。

## 12.1 悔棋

在日常的Git操作中，会经常出现这样的状况，输入`git commit`命令刚刚敲下回车键就后悔了：可能是提交说明中出现了错别字，或者有文件忘记提交，或者有的修改不应该提交，诸如此类。

像Subversion那样的集中式版本控制系统是“落子无悔”的系统，只能叹一口气责怪自己太不小心了。然后根据实际情况弥补：马上做一次新提交改正前面的错误；或者只能将错就错，错误的提交说明就让它一直错下去吧。因为大部分Subversion管理员不敢或不会放开修改提交说明的功能，从而导致无法对提交说明进行修改。

Git提供了“悔棋”的操作，甚至因为“单步悔棋”是如此经常的发生，乃至于Git提供了一个简洁的操作——修补式提交，命令是：`git commit --amend`。

看看当前版本库最新的两次提交：

```
$ cd /path/to/my/workspace/demo
$ git log --stat -2
commit 822b4aeed5de74f949c9faa5b281001eb5439444
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Wed Dec 8 16:27:41 2010 +0800

    测试使用 qgit 提交。

 README      |    1 +
 src/hello.h |    2 --
 2 files changed, 1 insertions(+), 2 deletions(-)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
commit 613486c17842d139871e0f1b0e9191d2b6177c9f
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Dec 7 19:43:39 2010 +0800

    偷懒了，直接用 -a 参数直接提交。

 src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

最新一次的提交的确是在上一章使用 qgit 进行的提交，但这和提交内容无关，因此需要改掉这个提交的提交说明。使用下面的命令即可做到。

```
$ git commit --amend -m "Remove hello.h, which is useless."
[master 7857772] Remove hello.h, which is useless.
 2 files changed, 1 insertions(+), 2 deletions(-)
 delete mode 100644 src/hello.h
```

上面的命令使用了 -m 参数是为了演示的方便，实际上完全可以直接输入 `git commit --amend`，在弹出的提交说明编辑界面修改提交说明，然后保存退出完成修补提交。

下面再看看最近两次的提交说明，可以看到最新的提交说明更改了（包括提交的 SHA1 哈希值），而它的父提交（即前一次提交）没有改变。

```
$ git log --stat -2
commit 78577724305e3e20aa9f2757ac5531d037d612a6
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Wed Dec 8 16:27:41 2010 +0800

    Remove hello.h, which is useless.

 README      |    1 +
 src/hello.h |    2 --
 2 files changed, 1 insertions(+), 2 deletions(-)

commit 613486c17842d139871e0f1b0e9191d2b6177c9f
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Dec 7 19:43:39 2010 +0800

    偷懒了，直接用 -a 参数直接提交。

 src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

如果最后一步操作不想删除文件 src/hello.h，而只是想修改 README，则可以按照下面的方法进行修补操作，具体操作过程如下。

（1）还原删除的 src/hello.h 文件。

```
$ git checkout HEAD^ -- src/hello.h
```

（2）此时查看状态，会看到 src/hello.h 被重新添加回暂存区。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   src/hello.h
#
```

（3）执行修补提交，不过提交说明是不是也要更改呢，因为毕竟这次提交不会删除文件了。

```
$ git commit --amend -m "commit with --amend test."
[master 2b45206] commit with --amend test.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

（4）再次查看最近的两次提交，会发现最新的提交不再删除文件 src/hello.h 了。

```
$ git log --stat -2
commit 2b452066ef6e92bceb999cf94fcce24afb652259
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Wed Dec 8 16:27:41 2010 +0800

    commit with --amend test.

 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

commit 613486c17842d139871e0f1b0e9191d2b6177c9f
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Dec 7 19:43:39 2010 +0800

    偷懒了，直接用 -a 参数直接提交。

 src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

## 12.2 多步悔棋

Git 能够提供悔棋的奥秘在于 Git 的重置命令。实际上上面介绍的单步悔棋也可以用重置命令来实现，只不过 Git 提供了一个更好用更简洁的修补提交命令而已。多步悔棋顾名思义就是可以取消最新连续的多次提交。多次悔棋并非是所有分布式版本控制系统都具有的功能，像 Mercurial/Hg 只能对最新提交悔棋一次（除非使用 MQ 插件）。Git 因为有了强大的重置命令，可以任意悔棋多次。

多步悔棋会在什么场合用到呢？软件开发中针对某个特性功能的开发就是一例。某个开发工程师领受某个特性开发的任务，于是在本地版本库进行了一系列开发、测试、修补、再测试的流程，最终特性功能开发完毕后可能在版本库中留下了多次提交。在将本地版本库改动推送（PUSH）到团队协同工作的核心版本库时，这个开发人员就想用多步悔棋的操作，

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


将多个试验性的提交合并为一个完整的提交。

以 DEMO 版本库为例，看看版本库最近的三次提交。

```
$ git log --stat --pretty=oneline -3
2b452066ef6e92bceb999cf94fcce24afb652259 commit with --amend test.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
613486c17842d139871e0f1b0e9191d2b6177c9f 偷懒了，直接用 -a 参数直接提交。
 src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
48456abfaeab706a44880eabcd63ea14317c0be9 add hello.h
 src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

想要将最近的两个提交压缩为一个，并把提交说明改为“modify hello.h”，可以使用如下方法进行操作。

（1）使用 --soft 参数调用重置命令，回到最近两次提交之前。

```
$ git reset --soft HEAD^^
```

（2）查看版本状态和最新日志。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#       modified:   src/hello.h
#
$ git log -1
commit 48456abfaeab706a44880eabcd63ea14317c0be9
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Dec 7 19:39:10 2010 +0800

    add hello.h
```

（3）执行提交操作，即完成最新两个提交压缩为一个提交的操作。

```
$ git commit -m "modify hello.h"
[master b6f0b0a] modify hello.h
 2 files changed, 2 insertions(+), 0 deletions(-)
```

（4）看看提交日志，“多步悔棋”操作成功。

```
$ git log --stat --pretty=oneline -2
b6f0b0a5237bc85de1863dbd1c05820f8736c76f modify hello.h
 README      |    1 +
 src/hello.h |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
48456abfaeab706a44880eabcd63ea14317c0be9 add hello.h
 src/hello.h |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

## 12.3 回到未来

电影《回到未来》（Back to Future）第二集，老毕福偷走时光车，到过去（1955 年）给了小毕福一本书，导致未来大变。

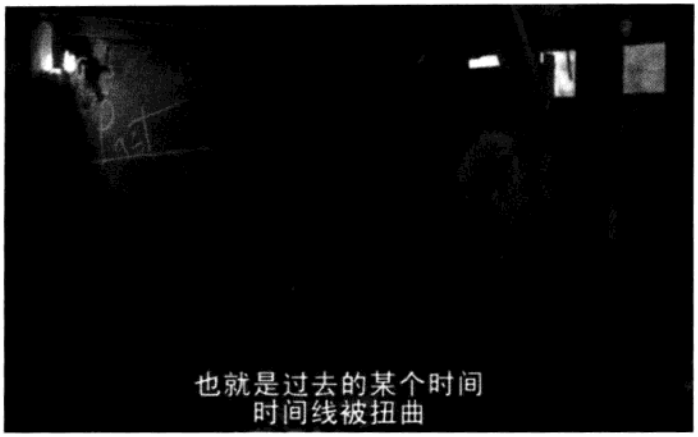



图 12-1 布朗博士正在解释为何产生两个平行的未来

Git 这一台“时光机”也有这样的能力，或者说也具有这样的行为。更改历史提交（SHA1 哈希值变更）后，即使后续提交的内容和属性都一致，但是因为后续提交中有一个属性是父提交的 SHA1 哈希值，所以一个历史提交的改变会引起连锁变化，导致所有的后续提交都发生变化，形成两条平行的时间线：一个是变更前的提交时间线，另外一条是更改历史后新的提交时间线。

把此次实践比喻成一次电影（《回到未来》）拍摄的话，舞台依然是之前的 DEMO 版本库，而剧本是这样的。

- 角色：最近的六次提交。分别依据提交顺序，编号为 A、B、C、D、E、F。

```
$ git log --oneline -6
b6f0b0a modify hello.h                                  # F
48456ab add hello.h                                     # E
3488f2c move .gitignore outside also works.             # D
b3af728 ignore object files.                            # C
d71ce92 Hello world initialized.                        # B
c024f34 README is from welcome.txt.                     # A
```

- 坏蛋：提交 D。

即不再需要对 .gitignore 文件进行移动的提交，或者这个提交将和前一次提交

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（C）压缩为一个。

- 前奏：故事人物依次出场，坏蛋 D 在图中被特殊标记，如图 12-2 所示。




图 12-2 提交示意图

- 第一幕：抛弃提交 D，将正确的提交 E 和 F 重新“嫁接”到提交 C 上，最终坏蛋被消灭，如图 12-3 所示。

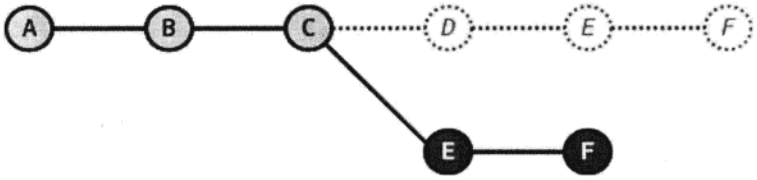



图 12-3 抛弃 D 提交示意图

- 第二幕：坏蛋 D 被 C 感化，融合为“CD”复合体，E 和 F 重新“嫁接”到“CD”复合体上，最终大团圆结局，如图 12-4 所示。

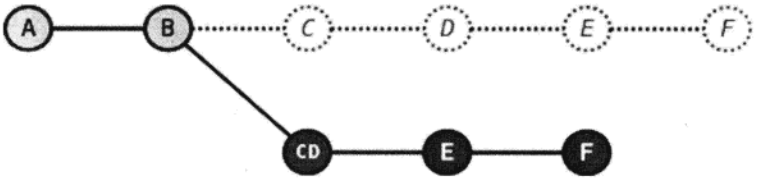



图 12-4 C、D 提交合并示意图

- 道具：分别使用三辆不同的时光车来完成“回到未来”。
  它们分别是：“核能跑车”——拣选操作、“清洁能源飞车”——变基操作、“蒸汽为动力的飞行火车”——交互式变基操作。

### 12.3.1 时间旅行一

《回到未来》第一集，布朗博士设计的第一款时间旅行车是一辆跑车，使用核燃料：钚。与之对应，此次实践使用的工具也没有太出乎想象之外，用一条新的指令——拣选指令（`git cherry-pick`）实现提交在新的分支上“重放”。

拣选指令——`git cherry-pick`，其含义是从众多的提交中挑选出一个提交应用在当前的工作分支中。该命令需要提供一个提交 ID 作为参数，操作过程相当于将该提交导出为补丁文件，然后在当前 HEAD 上重放，形成无论内容还是提交说明都一致的提交。

首先对版本库要“参演”的角色进行标记，使用尚未正式介绍的命令 `git tag`（无非

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


就是在特定命名空间建立“固定”的引用，用于对提交进行标识）。

```
$ git tag F
$ git tag E HEAD^
$ git tag D HEAD^^
$ git tag C HEAD^^^
$ git tag B HEAD~4
$ git tag A HEAD~5
```

通过日志，可以看到被标记的 6 个提交。

```
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

**1. 现在演出第一幕：干掉坏蛋 D**

（1）执行 `git checkout` 命令，暂时将 HEAD 头指针切换到 C。

切换过程显示处于非跟踪状态的警告，没有关系，因为剧情需要。

```
$ git checkout C
Note: checking out 'C'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at b3af728... ignore object files.
```

（2）执行拣选操作将 E 提交在当前 HEAD 上重放。

因为 E 和 master^ 显然指向同一角色，因此可以用下面的语法。

```
$ git cherry-pick master^
[detached HEAD fa0b076] add hello.h
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 src/hello.h
```

（3）执行拣选操作将 F 提交在当前 HEAD 上重放。

F 和 master 也具有相同指向。

```
$ git cherry-pick master
[detached HEAD f677821] modify hello.h
 2 files changed, 2 insertions(+), 0 deletions(-)
```

（4）通过日志可以看到坏蛋 D 已经不在了。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git log --oneline --decorate -6
f677821 (HEAD) modify hello.h
fa0b076 add hello.h
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

（5）通过日志还可以看出来，最新两次提交的原始创作日期（AuthorDate）和提交日期（CommitDate）不同。AuthorDate 是拣选提交的原始更改时间，而 CommitDate 是拣选操作时的时间，因此拣选后的新提交的 SHA1 哈希值也不同于所拣选的原提交的 SHA1 哈希值。

```
$ git log --pretty=fuller --decorate -2
commit f677821dfc15acc22ca41b48b8ebaab5ac2d2fea (HEAD)
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Sun Dec 12 12:11:00 2010 +0800
Commit:     Jiang Xin <jiangxin@ossxp.com>
CommitDate: Sun Dec 12 16:20:14 2010 +0800

    modify hello.h

commit fa0b076de600a53e8703545c299090153c6328a8
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Tue Dec 7 19:39:10 2010 +0800
Commit:     Jiang Xin <jiangxin@ossxp.com>
CommitDate: Sun Dec 12 16:18:34 2010 +0800

    add hello.h
```

（6）最重要的一步操作，就是要将 master 分支重置到新的提交 ID（f677821）上。

下面的切换操作使用了 reflog 的语法，即 HEAD@{1} 相当于切换回 master 分支前的 HEAD 指向，即 f677821。

```
$ git checkout master
Previous HEAD position was f677821... modify hello.h
Switched to branch 'master'
$ git reset --hard HEAD@{1}
HEAD is now at f677821 modify hello.h
```

（7）使用 qgit 查看版本库提交历史，如图 12-5 所示。

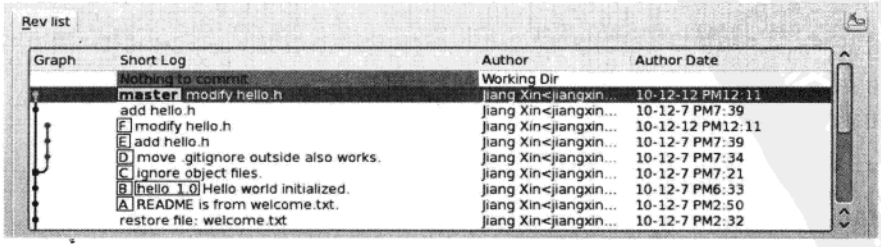



图 12-5 qgit 显示的提交分支图

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




### 2. 幕布拉上，后台重新布景

为了第二幕能够顺利演出，需要将 master 分支重新置回到提交 F 上。执行下面的操作完成“重新布景”。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

布景完毕，大幕即将再次拉开。

### 3. 现在演出第二幕：坏蛋 D 被感化，融入社会

（1）执行 git checkout 命令，暂时将 HEAD 头指针切换到坏蛋 D。

切换过程显示处于非跟踪状态的警告，没有关系，因为剧情需要。

```
$ git checkout D
Note: checking out 'D'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 3488f2c... move .gitignore outside also works.
```

（2）悔棋两次，以便将 C 和 D 融合。

```
$ git reset --soft HEAD^^
```

（3）执行提交，提交说明重用 C 提交的提交说明。

```
$ git commit -C C
[detached HEAD 53e621c] ignore object files.
 1 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
```

（4）执行拣选操作将 E 提交在当前 HEAD 上重放。

```
$ git cherry-pick E
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
[detached HEAD 1f99f82] add hello.h
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 src/hello.h
```

（5）执行拣选操作将 F 提交在当前 HEAD 上重放。

```
$ git cherry-pick F
[detached HEAD 2f13d3a] modify hello.h
 2 files changed, 2 insertions(+), 0 deletions(-)
```

（6）通过日志可以看到提交 C 和 D 被融合，所以在日志中看不到 C 的标签。

```
$ git log --oneline --decorate -6
2f13d3a (HEAD) modify hello.h
1f99f82 add hello.h
53e621c ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

（7）最重要的一步操作，就是要将 master 分支指向新的提交 ID（2f13d3a）上。

下面的切换操作使用了 reflog 的语法，即 HEAD@{1} 相当于切换回 master 分支前的 HEAD 指向，即 2f13d3a。

```
$ git checkout master
Previous HEAD position was 2f13d3a... modify hello.h
Switched to branch 'master'
$ git reset --hard HEAD@{1}
HEAD is now at 2f13d3a modify hello.h
```

（8）使用 gitk 查看版本库的提交历史，如图 12-6 所示。

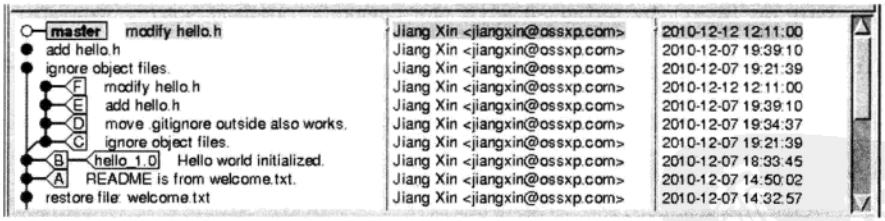



图 12-6 gitk 显示的提交分支图

### 4. 别忘了后台的重新布景

为了接下来的时间旅行二能够顺利开始，需要重新布景，将 master 分支重新置回到提交 F 上。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
用法1: git rebase  --onto <newbase>  <since>  <till>
用法2: git rebase  --onto <newbase>  <since>  [HEAD]
用法3: git rebase [--onto <since>]   <since>  <till>
用法4: git rebase [--onto <since>]   <since>  [HEAD]
```

## 12.3.2 时间旅行二

《回到未来》第二集，布朗博士改进的时间旅行车使用了未来科技，是陆天两用的飞车，而且燃料不再依赖核物质，而是使用无处不在的生活垃圾。而此次实践使用的工具也进行了升级，采用强大的 git rebase 命令。

命令 git rebase 是对提交执行变基操作，即可以实现将指定范围的提交“嫁接”到另外一个提交之上。其常用的命令行格式有：

```
用法1: git rebase --onto  <newbase>      <since>        <till>
用法2: git rebase --onto  <newbase>      <since>
用法3: git rebase                        <since>        <till>
用法4: git rebase                        <since>
用法5: git rebase -i ...
用法6: git rebase --continue
用法7: git rebase --skip
用法8: git rebase --abort
```

不要被上面的语法吓到，用法 5 会在下节（时间旅行三）中予以介绍，后三种用法则是变基运行过程被中断时可采用的命令——继续变基或终止等。

- 用法 6 是在变基遇到冲突而暂停的情况下，先完成冲突解决（添加到暂存区，不提交），然后在恢复变基操作的时候使用该命令。
- 用法 7 是在变基遇到冲突而暂停的情况下，跳过当前提交的时候使用。
- 用法 8 是在变基遇到冲突而暂停的情况下，终止变基操作，回到之前的分支时候使用。

而前四个用法如果把省略的参数补上（方括号内是省略掉的参数），看起来和用法 1 就都一致了。

下面就以归一化的 git rebase 命令格式来介绍其用法。

```
命令格式：git rebase  --onto  <newbase>  <since>  <till>
```

变基操作的过程：

（1）首先会执行 git checkout 切换到 <till>。

因为会切换到 <till>，因此如果 <till> 指向的不是一个分支（如 master），则变基操作是在 detached HEAD（分离头指针）状态进行的，当变基结束后，还要像在“时间旅行一”中那样，对 master 分支执行重置以实现变基结果在分支中生效。

（2）将 <since>..<till> 所标识的提交范围写到一个临时文件中。

还记得前面介绍的版本范围语法吗，<since>..<till> 是指包括 <till> 的所有历史提交排除 <since> 及 <since> 的历史提交后形成的版本范围。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（3）将当前分支强制重置（git reset --hard）到 <newbase>。

相当于执行：git reset --hard <newbase>。

（4）从保存在临时文件中的提交列表中，将提交逐一按顺序重新提交到重置之后的分支上。

（5）如果遇到提交已经在分支中包含，则跳过该提交。

（6）如果在提交过程遇到冲突，则变基过程暂停。用户解决冲突后，执行 git rebase --continue 继续变基操作。或者执行 git rebase --skip 跳过此提交。或者执行 git rebase --abort 就此终止变基操作切换到变基前的分支上。

很显然为了执行将 E 和 F 提交跳过提交 D，“嫁接”到提交 C 上。可以如此执行变基命令：

```
$ git rebase --onto C E^ F
```

因为 E^ 等价于 D，并且 F 和当前 HEAD 的指向相同，因此可以这样操作：

```
$ git rebase --onto C D
```

有了对变基命令的理解，就可以开始新的“回到未来”之旅了。

确认舞台已经布置完毕。

```
$ git status -s -b
## master
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

### 1. 现在演出第一幕：干掉坏蛋 D

（1）执行变基操作。

因为下面的变基操命令行使用了参数 F。F 是一个里程碑指向一个提交，而非 master，会导致后面变基完成后还需要对 master 分支执行重置。在第二幕中使用分支 master 作为参数，会发现省事不少。

```
$ git rebase --onto C E^ F
First, rewinding head to replay your work on top of it...
Applying: add hello.h
Applying: modify hello.h
```

（2）最后一步必需的操作，就是要将 master 分支指向变基后的提交上。

下面的切换操作使用了 reflog 的语法，即 HEAD@{1} 相当于切换回 master 分支前的 HEAD 指向，即 3360440。

```
$ git checkout master
Previous HEAD position was 3360440... modify hello.h
Switched to branch 'master'
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





```
$ git reset --hard HEAD@{1}
HEAD is now at 3360440 modify hello.h
```

（3）经过检查，操作完毕，收工。

```
$ git log --oneline --decorate -6
3360440 (HEAD, master) modify hello.h
1ef3803 add hello.h
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

### 2. 幕布拉上，后台重新布景

为了第二幕能够顺利演出，需要将 master 分支重新置回到提交 F 上。执行下面的操作完成“重新布景”。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

布景完毕，大幕即将再次拉开。

### 3. 现在演出第二幕：坏蛋 D 被感化，融入社会

（1）执行 git checkout 命令，暂时将 HEAD 头指针切换到坏蛋 D。

切换过程显示处于非跟踪状态的警告，没有关系，因为剧情需要。

```
$ git checkout D
Note: checking out 'D'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 3488f2c... move .gitignore outside also works.
```

（2）悔棋两次，以便将 C 和 D 融合。

```
$ git reset --soft HEAD^^
```

（3）执行提交，提交说明重用 C 提交的提交说明。

```
$ git commit -C C
[detached HEAD 2d020b6] ignore object files.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 1 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 .gitignore
```

（4）记住这个提交ID：2d020b6。

用里程碑是记忆提交ID的最好方法：

```
$ git tag newbase
$ git rev-parse newbase
2d020b62034b7a433f80396118bc3f66a60f296f
```

（5）执行变基操作，将E和F提交“嫁接”到newbase上。

下面的变基操命令行没有像之前的操作那样使用参数F，而是使用分支master。所以接下来的变基操作会直接修改master分支，而无须再对master进行重置操作。

```
$ git rebase --onto newbase E^ master
First, rewinding head to replay your work on top of it...
Applying: add hello.h
Applying: modify hello.h
```

（6）看看提交日志，看到提交C和提交D都不见了，代之以融合后的提交newbase。

还可以看到最新的提交除了和HEAD的指向一致，也和master分支的指向一致。

```
$ git log --oneline --decorate -6
2495dc1 (HEAD, master) modify hello.h
6349328 add hello.h
2d020b6 (tag: newbase) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

（7）当前的确已经在master分支上了，操作全部完成。

```
$ git branch
* master
```

（8）清理一下，然后收工。

前面的操作中为了方便创建了标识提交的新里程碑newbase，这个里程碑现在没有什么用处了，删除吧。

```
$ git tag -d newbase
Deleted tag 'newbase' (was 2d020b6)
```

#### 4. 别忘了后台的重新布景

为了接下来的时间旅行三能够顺利开始，需要重新布景，将master分支重新置回到提交F上。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




### 12.3.3 时间旅行三

《回到未来》第三集，成了铁匠的布朗博士手工打造了可以时光旅行的飞行火车，使用蒸汽作为动力。这款时间旅行火车更大、更安全、更舒适，适合一家四口外加宠物的时空旅行。与之对应本次实践也将采用“手工打造”：交互式变基。

交互式变基就是在上一节介绍的变基命令的基础上，添加了 -i 参数，在变基的时候进入一个交互界面。使用了交互界面的变基操作，不是自动化变基转换为手动确认那么没有技术含量，而是充满了魔法。

执行交互式变基操作，会将 <since>..<till> 的提交悉数罗列在一个文件中，然后自动打开一个编辑器来编辑这个文件。可以通过修改文件的内容设定变基操作，实现删除提交、将多个提交压缩为一个提交、更改提交的顺序，以及更改历史提交的提交说明等。

例如，下面的界面就是针对当前 DEMO 版本库执行的交互式变基时编辑器打开的文件：

```
pick b3af728 ignore object files.
pick 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h

# Rebase d71ce92..b6f0b0a onto d71ce92
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x <cmd>, exec <cmd> = Run a shell command <cmd>, and stop if it fails
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

从该文件中可以看出：

- 开头的四行由上到下依次对应于提交 C、D、E、F。
- 前四行默认的动作都是 pick，即应用此提交。
- 参考文件中的注释，可以通过修改动作名称，在变基的时候执行特定操作。
- 动作 reword，或者简写为 r。在变基时会应用此提交，但是在提交的时候允许用户修改提交说明。这个功能在 Git 1.6.6 之后开始提供，对于修改历史提交的提交说明非常方便。对于老版本的 Git 没有 reword 动作，可以使用 edit 动作。
- 动作 edit，或者简写为 e。也会在变基时应用此提交，但是会在应用后暂停变基，提示用户使用 `git commit --amend` 执行提交，以便对提交进行修补。当用户执行 `git commit --amend` 完成提交后，还需要执行 `git rebase --continue` 继续变基操作。用户在变基暂停状态下可以执行多次提交，从而实现把一个提交分解为

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


多个提交。edit 动作非常强大，对于老版本的 Git 没有 reword 动作，可以使用 edit 动作实现相同的效果。

- 动作 squash，或者简写为 s。该提交会与前面的提交压缩为一个。
- 动作 fixup，或者简写为 f。类似动作 squash，但是此提交的提交说明被丢弃。这个功能在 Git 1.7.0 之后开始提供，老版本的 Git 还是使用 squash 动作吧。
- 可以通过修改变基任务文件中各个提交的先后顺序，进而改变最终变基后提交的先后顺序。
- 可以修改变基任务文件，删除包含相应提交的行，这样该提交就不会被应用，进而在变基后的提交中被删除。

有了对交互式变基命令的理解，就可以开始新的“回到未来”之旅了。

确认舞台已经布置完毕。

```
$ git status -s -b
## master
$ git log --oneline --decorate -6
b6f0b0a (HEAD, tag: F, master) modify hello.h
48456ab (tag: E) add hello.h
3488f2c (tag: D) move .gitignore outside also works.
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
```

**1. 现在演出第一幕：干掉坏蛋 D**

（1）执行交互式变基操作。

```
$ git rebase -i D^
```

（2）自动用编辑器修改文件。文件内容如下：

```
pick 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h

# Rebase b3af728..b6f0b0a onto b3af728
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x <cmd>, exec <cmd> = Run a shell command <cmd>, and stop if it fails
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

（3）将第一行删除，使得上面的文件看起来像是这样（省略井号开始的注释）：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




```
pick 48456ab add hello.h
pick b6f0b0a modify hello.h
```

（4）保存退出。

变基自动开始，即刻完成。

显示下面的内容：

```
Successfully rebased and updated refs/heads/master.
```

（5）看看日志。当前分支 master 已经完成变基，消灭了“坏蛋 D”。

```
$ git log --oneline --decorate -6
78e5133 (HEAD, master) modify hello.h
11eea7e add hello.h
b3af728 (tag: C) ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

### 2. 幕布拉上，后台重新布景

为了第二幕能够顺利演出，需要将 master 分支重新置回到提交 F 上。执行下面的操作完成“重新布景”。

```
$ git checkout master
Already on 'master'
$ git reset --hard F
HEAD is now at b6f0b0a modify hello.h
```

布景完毕，大幕即将再次拉开。

### 3. 现在演出第二幕：坏蛋 D 被感化，融入社会

（1）同样执行交互式变基操作，不过因为要将 C 和 D 压缩为一个，因此变基从 C 的父提交开始。

```
$ git rebase -i C^
```

（2）自动用编辑器修改文件。文件内容如下（忽略井号开始的注释）：

```
pick b3af728 ignore object files.
pick 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h
```

（3）修改第二行（提交 D），将动作由 pick 修改为 squash 。

修改后的内容如下：

```
pick b3af728 ignore object files.
squash 3488f2c move .gitignore outside also works.
pick 48456ab add hello.h
pick b6f0b0a modify hello.h
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（4）保存退出。自动开始变基操作，在执行到 squash 命令设定的提交时，进入提交前的日志编辑状态。

显示的待编辑日志如下。很明显 C 和 D 的提交说明显示在了一起。

```
# This is a combination of 2 commits.
# The first commit's message is:

ignore object files.

# This is the 2nd commit message:

move .gitignore outside also works.
```

（5）保存退出，即完成 squash 动作标识的提交合并及后续变基操作。

看看提交日志，看到提交 C 和提交 D 都不见了，代之以一个融合后的提交出现。

```
$ git log --oneline --decorate -6
c0c2a1a (HEAD, master) modify hello.h
c1e8b66 add hello.h
db512c0 ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
```

（6）可以看到融合 C 和 D 的提交日志实际上是两者日志的融合。在前面单行显示的日志中看不出来。

```
$ git cat-file -p HEAD^^
tree 00239a5d0daf9824a23cbf104d30af66af984e27
parent d71ce9255b3b08c718810e4e31760198dd6da243
author Jiang Xin <jiangxin@ossxp.com> 1291720899 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1292153393 +0800

ignore object files.

move .gitignore outside also works.
```

时光旅行结束了，多么神奇的 Git 啊。

## 12.4 丢弃历史

历史有的时候会成为负担。例如一个人使用的版本库有一天需要作为公共版本库多人共享，最早的历史可能不希望或者没有必要继续保持存在，需要一个抛弃部分早期历史提交的精简的版本库以用于和他人共享。再比如用 Git 做文件备份，不希望备份的版本过多而导致不必要的磁盘空间占用，同样会有精简版本的需要：只保留最近的 100 次提交，抛弃之前的历史提交。那么应该如何操作呢？

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


使用交互式变基当然可以完成这样的任务，但是如果历史版本库有成百上千个，把成百上千个版本的变基动作中有 pick 的修改为 fixup 可真的很费事，实际上 Git 有更简便的方法。

现在 DEMO 版本库有如下的提交记录：

```
$ git log --oneline --decorate
c0c2a1a (HEAD, master) modify hello.h
c1e8b66 add hello.h
db512c0 ignore object files.
d71ce92 (tag: hello_1.0, tag: B) Hello world initialized.
c024f34 (tag: A) README is from welcome.txt.
63992f0 restore file: welcome.txt
7161977 delete trash files. (using: git add -u)
2b31c19 (tag: old_practice) Merge commit 'acc2f69'
acc2f69 commit in detached HEAD mode.
4902dc3 does master follow this new commit?
e695606 which version checked in?
a0c641e who does commit?
9e8a761 initialized.
```

如果希望把里程碑 A（c024f34）之前的历史提交全部清除，可以这样操作：基于里程碑 A 对应的提交构造一个根提交（即没有父提交的提交），然后再将 master 分支在里程碑 A 之后的提交变基到新的根提交上，实现对历史提交的清除。

由里程碑 A 对应的提交构造出一个根提交至少有两种方法。第一种方法是使用 `git commit-tree` 命令，可以进行如下操作。

（1）查看里程碑 A 指向的目录树。

用 A^{tree} 语法访问里程碑 A 对应的目录树。

```
$ git cat-file -p A^{tree}
100644 blob 51dbfd25a804c30e9d8dc441740452534de8264b    README
```

（2）使用 `git commit-tree` 命令直接从该目录树创建提交。

```
$ echo "Commit from tree of tag A." | git commit-tree A^{tree}
8f7f94ba6a9d94ecc1c223aa4b311670599e1f86
```

（3）命令 `git commit-tree` 的输出是一个提交的 SHA1 哈希值。查看这个提交。会发现这个提交没有历史提交，是我们需要的根提交。

```
$ git log --pretty=raw 8f7f94b
commit 8f7f94ba6a9d94ecc1c223aa4b311670599e1f86
tree 3f9b9459e9c0532ff6e3c16c3098c947d55fba41
author Jiang Xin <jiangxin@ossxp.com> 1292221037 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1292221037 +0800

    Commit from tree of tag A.
```

另外一个方法是使用 `git hash-object` 命令，从里程碑 A 指向的提交建立一个根提交。可以进行如下操作。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（1）查看里程碑 A 指向的提交。

用 A^0 语法访问里程碑 A 对应的提交。

```
$ git cat-file commit A^0
tree 3f9b9459e9c0532ff6e3c16c3098c947d55fba41
parent 63992f05a72865754809cc0772a9e1fbf134e380
author Jiang Xin <jiangxin@ossxp.com> 1291704602 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291704602 +0800

README is from welcome.txt.
```

（2）将上面的输出（里程碑 A 指向的提交）过滤掉以 parent 开头的行，并将结果保存到一个文件中。

```
$ git cat-file commit A^0 | sed -e '/^parent/ d' > tmpfile
```

（3）运行 git hash-object 命令，将文件 tmpfile 作为一个 commit 对象写入对象库。

```
$ git hash-object -t commit -w -- tmpfile
4d387fd42a023aadcf0702907b848444e8c4429c
```

（4）上面执行 git hash-object 命令的输出结果就是写入 Git 对象库中的新的提交对象 ID。查看会发现该提交就是我们需要的新的根提交。

```
$ git log --pretty=raw 4d387fd
commit 4d387fd42a023aadcf0702907b848444e8c4429c
tree 3f9b9459e9c0532ff6e3c16c3098c947d55fba41
author Jiang Xin <jiangxin@ossxp.com> 1291704602 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291704602 +0800

    README is from welcome.txt.
```

无论采用哪种方法创建了新的根提交后，就可以执行变基操作，将 master 分支在里程碑 A 之后的提交变基到新的根提交上。下面的示例选择第一种方法创建的根提交 8f7f94b。

（1）执行变基，将 master 分支里程碑 A 之后的提交全部迁移到根提交 8f7f94b 上。

```
$ git rebase --onto 8f7f94b A master
First, rewinding head to replay your work on top of it...
Applying: Hello world initialized.
Applying: ignore object files.
Applying: add hello.h
Applying: modify hello.h
```

（2）查看日志看到当前 master 分支的历史已经精简了。

```
$ git log --oneline --decorate
2584639 (HEAD, master) modify hello.h
30fe8b3 add hello.h
4dd8a65 ignore object files.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
5f2cae1 Hello world initialized.
8f7f94b Commit from tree of tag A.
```

使用图形工具查看提交历史，会看到两棵树，如图 12-7 所示：最上面的一棵树是刚刚通过变基抛弃了大部分历史提交的新的 master 分支，下面的一棵树则是变基前的提交形成的。下面的一棵树之所以还能够看到，或者说还没有从版本库中彻底清除，是因为有部分提交仍带有里程碑标签。

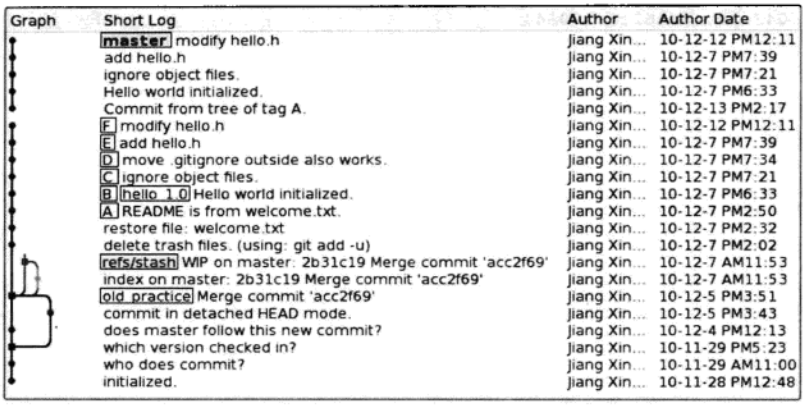



图 12-7 丢弃历史后的提交分支图

## 12.5 反转提交

前面介绍的操作都涉及对历史的修改，这对于一个人使用 Git 没有问题，但是如果多人协同就会有问题了。多人协同使用 Git，在本地版本库做的提交会通过多人之间的交互成为他人版本库的一部分，更改历史操作只能是针对自己的版本库，而无法去修改他人的版本库，正所谓“覆水难收”。在这种情况下要想修正一个错误历史提交的正确做法是反转提交，即重新做一次新的提交，相当于用错误的历史提交的反向提交，来修正错误的历史提交。

Git 反向提交命令是：`git revert`，下面在 DEMO 版本库中实践一下。注意：Subversion 的用户不要想当然地和 `svn revert` 命令对应，这两个版本控制系统中 revert 命令的功能完全不相干。

当前 DEMO 版本库最新的提交包含如下改动：

```
$ git show HEAD
commit 25846394defe16eab103b92efdaab5e46cc3dc22
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Sun Dec 12 12:11:00 2010 +0800

    modify hello.h

diff --git a/README b/README
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
index 51dbfd2..ceaf01b 100644
--- a/README
+++ b/README
@@ -1,3 +1,4 @@
 Hello.
 Nice to meet you.
 Bye-Bye.
+Wait...
diff --git a/src/hello.h b/src/hello.h
index 0043c3b..6e482c6 100644
--- a/src/hello.h
+++ b/src/hello.h
@@ -1 +1,2 @@
 /* test */
+/* end */
```

在不改变这个提交的前提下撤销对其的修改，就需要用到 git revert 反转提交。

```
$ git revert HEAD
```

运行该命令相当于将 HEAD 提交反向再提交一次，在提交说明编辑状态下暂停，显示如下（注释行被忽略）：

```
Revert "modify hello.h"

This reverts commit 25846394defe16eab103b92efdaab5e46cc3dc22.
```

可以在编辑器中修改提交说明，提交说明编辑完毕保存退出则完成反转提交。查看提交日志可以看到新的提交相当于所撤销提交的反向提交。

```
$ git log --stat -2
commit 6e6753add1601c4efa7857ab4c5b245e0e161314
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Mon Dec 13 15:19:12 2010 +0800

    Revert "modify hello.h"

    This reverts commit 25846394defe16eab103b92efdaab5e46cc3dc22.

 README      |    1 -
 src/hello.h |    1 -
 2 files changed, 0 insertions(+), 2 deletions(-)

commit 25846394defe16eab103b92efdaab5e46cc3dc22
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Sun Dec 12 12:11:00 2010 +0800

    modify hello.h

 README      |    1 +
 src/hello.h |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 13 章 Git 克隆

到现在为止，大家已经领略到 Git 的灵活性和健壮性。Git 可以通过重置随意撤销提交，可以通过变基操作更改历史，可以随意重组提交，还可以通过 reflog 的记录纠正错误的操作。但是再健壮的版本库设计，也抵挡不了存储介质的崩溃。还有一点就是不要忘了 Git 版本库是躲在工作区根目录下的 .git 目录中，如果忘了这一点直接删除工作区，就会把版本库也同时删掉，悲剧就此发生。

“不要把鸡蛋装在一个篮子里”是颠扑不破的安全法则。

在本章会学习到如何使用 git clone 命令建立版本库克隆，以及如何使用 git push 和 git pull 命令实现克隆之间的同步。

## 13.1 鸡蛋不装在一个篮子里

Git 的版本库目录和工作区在一起，因此存在一损俱损的问题，即如果删除一个项目的工作区，同时也会把这个项目的版本库删除掉。一个项目仅在一个工作区中维护太危险了，如果有两个工作区就会好很多。

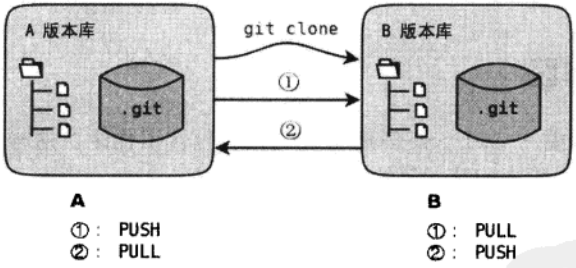



图 13-1 克隆版本库关系图

图 13-1 中一个项目使用了两个版本库进行维护，两个版本库之间通过 PULL 和 / 或 PUSH 操作实现同步：

- 版本库 A 通过克隆操作创建克隆版本库 B。
- 版本库 A 可以通过 PUSH（推送）操作，将新提交传递给版本库 B。
- 版本库 A 可以通过 PULL（拉回）操作，将版本库 B 中的新提交拉回到自身（A）。
- 版本库 B 可以通过 PULL（拉回）操作，将版本库 A 中的新提交拉回到自身（B）。
- 版本库 B 可以通过 PUSH（推送）操作，将新提交传递给版本库 A。

Git 使用 git clone 命令实现版本库克隆，主要有如下三种用法：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
用法 1: git clone <repository> <directory>
用法 2: git clone --bare    <repository> <directory.git>
用法 3: git clone --mirror <repository> <directory.git>
```

这三种用法的区别如下：

- 用法 1 将 <repository> 指向的版本库创建一个克隆到 <directory> 目录。目录 <directory> 相当于克隆版本库的工作区，文件都会检出，版本库位于工作区下的 .git 目录中。
- 用法 2 和用法 3 创建的克隆版本库都不包含工作区，直接就是版本库的内容，这样的版本库称为裸版本库。一般约定俗成裸版本库的目录名以 .git 为后缀，所以上面示例中将克隆出来的裸版本库目录名写作 <directory.git>。
- 用法 3 区别于用法 2 之处在于用法 3 克隆出来的裸版本对上游版本库进行了注册，这样可以在裸版本库中使用 git fetch 命令和上游版本库进行持续同步。
- 用法 3 只在 1.6.0 或更新版本的 Git 中才提供。

Git 的 PUSH 和 PULL 命令的用法相似，使用下面的语法：

```
git push [<remote-repos> [<refspec>]]
git pull [<remote-repos> [<refspec>]]
```

其中方括号的含义是参数可以省略，<remote-repos> 是远程版本库的地址或名称，<refspec> 是引用表达式，暂时理解为引用即可。后面的章节再具体介绍 PUSH 和 PULL 命令的细节。

下面就通过不同的 Git 命令组合，掌握版本库克隆和镜像的技巧。

## 13.2 对等工作区

不使用 --bare 或 --mirror 创建出来的克隆包含工作区，这样就会产生两个包含工作区的版本库。这两个版本库是对等的，如图 13-2 所示。

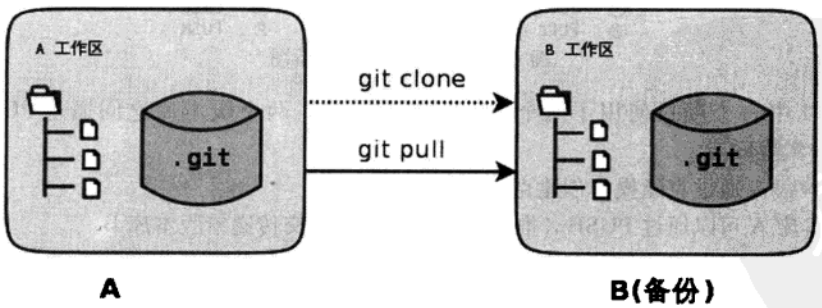



图 13-2 带工作区的对等版本库

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


这两个工作区本质上没有区别，但是往往提交是在一个版本（A）中进行的，另外一个（B）作为备份。对于这种对等工作区模式，版本库的同步只有一种可行的操作模式，就是备份库（B）执行 git pull 命令从源版本库（A）中拉回新的提交实现版本库同步。为什么不能从版本库 A 向版本库 B 执行 git push 推送操作呢？看看下面的操作。

执行克隆命令，将版本库 /path/to/my/workspace/demo 克隆到 /path/to/my/workspace/demo-backup。

```
$ git clone /path/to/my/workspace/demo /path/to/my/workspace/demo-backup
Cloning into /path/to/my/workspace/demo-backup...
done.
```

进入 demo 版本库，生成一些测试提交（使用 --allow-empty 参数可以生成空提交）。

```
$ cd /path/to/my/workspace/demo/
$ git commit --allow-empty -m "sync test 1"
[master 790e72a] sync test 1
$ git commit --allow-empty -m "sync test 2"
[master f86b7bf] sync test 2
```

能够在 demo 版本库向 demo-backup 版本库执行 PUSH 操作吗？执行一下 git push 看一看。

```
$ git push /path/to/my/workspace/demo-backup
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 274 bytes, done.
Total 2 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
remote: error: refusing to update checked out branch: refs/heads/master
remote: error: By default, updating the current branch in a non-bare repository
remote: error: is denied, because it will make the index and work tree inconsistent
remote: error: with what you pushed, and will require 'git reset --hard' to match
remote: error: the work tree to HEAD.
remote: error:
remote: error: You can set 'receive.denyCurrentBranch' configuration variable to
remote: error: 'ignore' or 'warn' in the remote repository to allow pushing into
remote: error: its current branch; however, this is not recommended unless you
remote: error: arranged to update its work tree to match what you pushed in some
remote: error: other way.
remote: error:
remote: error: To squelch this message and still keep the default behaviour, set
remote: error: 'receive.denyCurrentBranch' configuration variable to 'refuse'.
To /path/to/my/workspace/demo-backup
 ! [remote rejected] master -> master (branch is currently checked out)
error: failed to push some refs to '/path/to/my/workspace/demo-backup'
```

翻译成中文：

```
$ git push /path/to/my/workspace/demo-backup
...
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
对方说：错了：
                拒绝更新已检出的分支 refs/heads/master 。
                默认更新非裸版本库的当前分支是不被允许的，因为这将会导致
                暂存区和工作区与你推送至版本库的新提交不一致。这太古怪了。

                如果您一意孤行，也不是不允许，但是你需要为我设置如下参数：

                    receive.denyCurrentBranch = ignore|warn

                到 /path/to/my/workspace/demo-backup

 ! [对方拒绝] master -> master （分支当前已检出）
错误：部分引用的推送失败了，至 '/path/to/my/workspace/demo-backup'
```

从错误输出中可以看出，虽然可以改变 Git 的默认行为，允许向工作区推送已经检出的分支，但是这么做实在不高明。

为了实现同步，需要进入到备份版本库中，执行 `git pull` 命令。

```
$ git pull
From /path/to/my/workspace/demo
   6e6753a..f86b7bf  master     -> origin/master
Updating 6e6753a..f86b7bf
Fast-forward
```

在 demo-backup 版本库中查看提交日志，可以看到在 demo 版本库中的新提交已经被拉回到 demo-backup 版本库中。

```
$ git log --oneline -2
f86b7bf sync test 2
790e72a sync test 1
```

为什么执行 `git pull` 命令没有像执行 `git push` 命令那样提供那么多的参数呢？这是因为在执行 `git clone` 操作后，克隆出来的 demo-backup 版本库中对源版本库（上游版本库）进行了注册，所以在 demo-backup 版本库中执行拉回操作，无须设置上游版本库的地址。

在 demo-backup 版本库中可以使用下面的命令查看对上游版本库的注册信息：

```
$ cd /path/to/my/workspace/demo-backup
$ git remote -v
origin  /path/to/my/workspace/demo (fetch)
origin  /path/to/my/workspace/demo (push)
```

实际上，注册上游远程版本库的奥秘都在 Git 的配置文件中（略去无关的行）：

```
$ cat /path/to/my/workspace/demo-backup/.git/config
...
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        url = /path/to/my/workspace/demo
[branch "master"]
        remote = origin
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
merge = refs/heads/master
```

关于配置文件 [remote] 小节和 [branch] 小节的奥秘将在第 3 篇第 19 章中予以介绍。

## 13.3 克隆生成裸版本库

上一节在对等工作区模式下，工作区之间执行推送，可能会引发大段的错误输出，如果采用裸版本库则没有相应的问题。这是因为裸版本库没有工作区。没有工作区还有一个好处就是空间占用会更小。

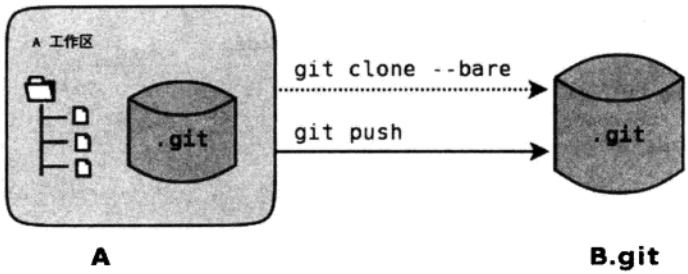



图 13-3 从版本库中克隆裸版本库

如图 13-3 所示，使用 --bare 参数将 demo 版本库克隆到 /path/to/repos/demo.git，然后就可以从 demo 版本库向克隆的裸版本库执行推送操作了。（为了说明方便，使用了 /path/to/repos/ 作为 Git 裸版本的根路径，在后面的章节中这个目录也将作为 Git 服务器端版本库的根路径。可以在磁盘中以 root 账户创建该路径并设置正确的权限。）

```
$ git clone --bare /path/to/my/workspace/demo /path/to/repos/demo.git
Cloning into bare repository /path/to/repos/demo.git...
done.
```

克隆出来的 /path/to/repos/demo.git 目录就是版本库目录，不包含工作区。

- 看看 /path/to/repos/demo.git 目录的内容。

```
$ ls -F /path/to/repos/demo.git
branches/  config  description  HEAD  hooks/  info/  objects/  packed-refs  refs/
```

- 还可以看到 demo.git 版本库中 core.bare 的配置为 true。

```
$ git --git-dir=/path/to/repos/demo.git config core.bare
true
```

进入 demo 版本库，生成一些测试提交。

```
$ cd /path/to/my/workspace/demo/
$ git commit --allow-empty -m "sync test 3"
[master d4b42b7] sync test 3
$ git commit --allow-empty -m "sync test 4"
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
[master 0285742] sync test 4
```

在 demo 版本库向 demo-backup 版本库执行 PUSH 操作，还会有错误吗？

- 不带参数执行 `git push`，因为未设定上游远程版本库，因此会报错：

```
$ git push
fatal: No destination configured to push to.
```

- 在执行 `git push` 时使用 `/path/to/repos/demo.git` 作为参数。推送成功。

```
$ git push /path/to/repos/demo.git
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 275 bytes, done.
Total 2 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
To /path/to/repos/demo.git
   f86b7bf..0285742  master -> master
```

- 看看 demo.git 版本库，是否已经完成了同步？

```
$ git --git-dir=/path/to/repos/demo.git log --oneline -2
0285742 sync test 4
d4b42b7 sync test 3
```

这个方式实现版本库本地镜像显然是更好的方法，因为可以直接在工作区修改和提交，然后执行 `git push` 命令实现推送。稍有一点遗憾的是推送命令还需要加上裸版本库的路径。这个遗憾在第 3 篇第 19 章会给出解决方案。

## 13.4 创建生成裸版本库

裸版本库不但可以通过克隆的方式创建，还可以通过 `git init` 命令以初始化的方式创建。之后的同步方式和上一节大同小异，如图 13-4 所示。

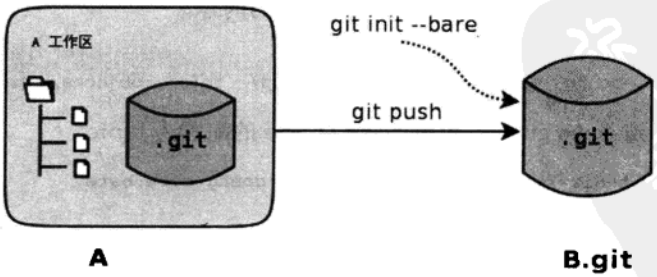



图 13-4 向裸版本库推送

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




命令 git init 在“第 4 章 Git 初始化”一章就已经用到了，是用于初始化一个版本库的。之前执行 git init 命令初始化的版本库是带工作区的，如何以裸版本库的方式初始化一个版本库呢？奥秘就在于 --bare 参数。

下面的命令会在目录 /path/to/repos/demo-init.git 中创建一个空的裸版本库。

```
$ git init --bare /path/to/repos/demo-init.git
Initialized empty Git repository in /path/to/repos/demo-init.git/
```

创建的果真是裸版本库吗？

- 看看 /path/to/repos/demo-init.git 下的内容：

```
$ ls -F /path/to/repos/demo-init.git
branches/  config  description  HEAD  hooks/  info/  objects/  refs/
```

- 看看这个版本库的配置 core.bare 的值：

```
$ git --git-dir=/path/to/repos/demo-init.git config core.bare
true
```

可是空版本库没有内容啊，那就执行 PUSH 操作为其创建内容呗。

```
$ cd /path/to/my/workspace/demo
$ git push /path/to/repos/demo-init.git
No refs in common and none specified; doing nothing.
Perhaps you should specify a branch such as 'master'.
fatal: The remote end hung up unexpectedly
error: failed to push some refs to '/path/to/repos/demo-init.git'
```

为什么出错了？翻译一下错误输出。

```
$ cd /path/to/my/workspace/demo
$ git push /path/to/repos/demo-init.git
没有指定要推送的引用，而且两个版本库也没有共同的引用。
所以什么也没有做。
可能您需要提供要推送的分支名，如 'master'。
严重错误：远程操作意外终止
错误：部分引用推送失败，至 '/path/to/repos/demo-init.git'
```

关于这个问题的详细说明要在第 3 篇第 19 章“19.4 PUSH 和 PULL 操作和远程版本库”小节中介绍，这里先说一个省略版：因为 /path/to/repos/demo-init.git 版本库刚刚初始化完成，还没有任何提交，更不要说分支了。当执行 git push 命令时，如果没有设定推送的分支，而且当前分支也没有注册到远程的某个分支，将检查远程分支是否有和本地相同的分支名（如 master），如果有，则推送，否则报错。

所以需要把 git push 命令写得再完整一些。像下面这样操作，就可以完成向空的裸版本库的推送。

```
$ git push /path/to/repos/demo-init.git master:master
Counting objects: 26, done.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Delta compression using up to 2 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (26/26), 2.49 KiB, done.
Total 26 (delta 8), reused 0 (delta 0)
Unpacking objects: 100% (26/26), done.
To /path/to/repos/demo-init.git
 * [new branch]      master -> master
```

上面的git push命令也可以简写为：git push /path/to/repos/demo-init.git master。

推送成功了吗？看看demo-init.git版本库中的提交。

```
$ git --git-dir=/path/to/repos/demo-init.git log --oneline -2
0285742 sync test 4
d4b42b7 sync test 3
```

好了继续在demo中执行几次提交。

```
$ cd /path/to/my/workspace/demo/
$ git commit --allow-empty -m "sync test 5"
[master 424aa67] sync test 5
$ git commit --allow-empty -m "sync test 6"
[master 70a5aa7] sync test 6
```

然后再向demo-init.git推送。注意这次使用的命令。

```
$ git push /path/to/repos/demo-init.git
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 273 bytes, done.
Total 2 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
To /path/to/repos/demo-init.git
   0285742..70a5aa7  master -> master
```

为什么这次使用git push命令后面没有跟上分支名呢？这是因为远程版本库（demo-init.git）中已经不再是空版本库了，有名为master的分支。

通过下面的命令可以查看远程版本库的分支。

```
$ git ls-remote /path/to/repos/demo-init.git
70a5aa7a7469076fd435a9e4f89c4657ba603ced        HEAD
70a5aa7a7469076fd435a9e4f89c4657ba603ced        refs/heads/master
```

至此相信您已经能够把鸡蛋放在不同的篮子中了，也对使用Git更加有信心了吧。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 14 章 Git 库管理

版本库管理？那不是管理员要干的事情么，怎么放在“Git 独奏”这一部分了？

没有错，这是因为对于 Git，每个用户都是自己版本库的管理员，所以在“Git 独奏”的最后一章，我们来谈一谈 Git 版本库管理的问题。如果下面的问题您没有遇到或不感兴趣，大可以放心地跳过这一章。

- 从网上克隆来的版本库，为什么对象库中找不到对象文件？而且引用目录里也看不到所有的引用文件？
- 不小心添加了一个大文件到 Git 库中，用重置命令丢弃了包含大文件的提交，可是版本库不见小，大文件仍在对象库中。
- 本地版本库的对象库里的文件越来越多，这可能导致 Git 性能的降低。

## 14.1 对象和引用哪里去了

从 Github 上克隆一个示例版本库，这个版本库在“第 11 章 历史穿梭”一章就已经克隆过一次了，现在要重新克隆一份。为了和原来的克隆相区别，我们将克隆到另外的目录。执行下面的命令。

```
$ cd /path/to/my/workspace/
$ git clone git://github.com/ossxp-com/gitdemo-commit-tree.git i-am-admin
Cloning into i-am-admin...
remote: Counting objects: 65, done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 65 (delta 8), reused 0 (delta 0)
Receiving objects: 100% (65/65), 78.14 KiB | 42 KiB/s, done.
Resolving deltas: 100% (8/8), done.
```

进入克隆的版本库，使用 `git show-ref` 命令看看所包含的引用。

```
$ cd /path/to/my/workspace/i-am-admin
$ git show-ref
6652a0dce6a5067732c00ef0a220810a7230655e refs/heads/master
6652a0dce6a5067732c00ef0a220810a7230655e refs/remotes/origin/HEAD
6652a0dce6a5067732c00ef0a220810a7230655e refs/remotes/origin/master
c9b03a208288aebdbfe8d84aeb984952a16da3f2 refs/tags/A
1a87782f8853c6e11aacba463af04b4fa8565713 refs/tags/B
9f8b51bc7dd98f7501ade526dd78c55ee4abb75f refs/tags/C
887113dc095238a0f4661400d33ea570e5edc37c refs/tags/D
6decd0ad3201ddb3f5b37c201387511059ac120c refs/tags/E
70cab20f099e0af3f870956a3fbbbda50a17864f refs/tags/F
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
96793e37c7f1c7b2ddf69b4c1e252763c11a711f refs/tags/G
476e74549047e2c5fbd616287a499cc6f07ebde0 refs/tags/H
76945a15543c49735634d58169b349301d65524d refs/tags/I
f199c10c3f1a54fa3f9542902b25b49d58efb35b refs/tags/J
```

其中以 refs/heads/ 开头的是分支；以 refs/remotes/ 开头的是远程版本库分支在本地的映射，这会在后面的章节中介绍；以 refs/tags/ 开头的是里程碑。按照之前的经验，在 .git/refs 目录下应该有这些引用所对应的文件才是。看看都在么？

```
$ find .git/refs/ -type f
.git/refs/remotes/origin/HEAD
.git/refs/heads/master
```

为什么才有两个文件？实际上当运行下面的命令后，引用目录下的文件会更少：

```
$ git pack-refs --all
$ find .git/refs/ -type f
.git/refs/remotes/origin/HEAD
```

那么本应该出现在 .git/refs/ 目录下的引用文件都到哪里去了呢？答案是这些文件被打包了，放到一个文本文件 .git/packed-refs 中了。查看一下这个文件中的内容。

```
$ head -5 .git/packed-refs
# pack-refs with: peeled
6652a0dce6a5067732c00ef0a220810a7230655e refs/heads/master
6652a0dce6a5067732c00ef0a220810a7230655e refs/remotes/origin/master
c9b03a208288aebdbfe8d84aeb984952a16da3f2 refs/tags/A
^81993234fc12a325d303eccea20f6fd629412712
```

再来看看 Git 的对象（commit、blob、tree、tag）在对象库中的存储。通过下面的命令，会发现对象库也不是原来熟悉的模样了。

```
$ find .git/objects/ -type f
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack
```

对象库中只有两个文件，本应该一个一个独立保存的对象都不见了。您应该能够猜到，所有的对象文件都被打包到这两个文件中了，其中以 .pack 结尾的文件是打包文件，以 .idx 结尾的是索引文件。打包文件和对应的索引文件只是扩展名不同，都保存于 .git/objects/pack/ 目录下。Git 对于以 SHA1 哈希值作为目录名和文件名保存的对象有一个术语，称为松散对象。松散对象打包后会提高访问效率，而且不同的对象可以通过增量存储节省磁盘空间。

通过 Git 一个底层的命令可以查看索引中包含的对象：

```
$ git show-index < .git/objects/pack/pack-*.idx | head -5
661 0cd7f2ea245d90d414e502467ac749f36aa32cc4 (0793420b)
63020 1026d9416d6fc8d34e1edfb2bc58adb8aa5a6763 (ed77ff72)
3936 15328fc6961390b4b10895f39bb042021edd07ea (13fb79ef)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
3768 1a588ca36e25f58fbeae421c36d2c39e38e991ef (86e3b0bd)
2022 1a87782f8853c6e11aacba463af04b4fa8565713 (e269ed74)
```

为什么克隆远程版本库就可以产生对象库打包及引用打包的效果呢？这是因为克隆远程版本库时，使用了“智能”的通信协议，远程 Git 服务器将对象打包后传输给本地，形成本地版本库的对象库中的一个包含所有对象的包及索引文件。无疑这样的传输方式——按需传输、打包传输——效率最高。

克隆之后的版本库在日常的提交中，产生的新对象仍旧以松散对象存在，而不是以打包的形式，日积月累会在本地版本库的对象库中形成大量的松散文件。松散对象只是进行了压缩，而没有（打包文件才有的）增量存储的功能，会浪费磁盘空间，也会降低访问效率。更为严重的是一些非正式的临时对象（暂存区操作中产生的临时对象）也以松散对象的形式保存在对象库中，造成磁盘空间的浪费。下一节就着手处理临时对象的问题。

## 14.2 暂存区操作引入的临时对象

暂存区操作有可能在对象库中产生临时对象，例如，文件反复地修改、反复地向暂存区添加，或者添加到暂存区后不提交甚至直接撤销，就会产生垃圾数据占用磁盘空间。为了说明临时对象的问题，需要准备一个大的压缩文件，10MB 即可。

在 Linux 上与内核匹配的 initrd 文件（内核启动加载的内存盘）就是一个大的压缩文件，可以用于此节的示例。将大的压缩文件放在版本库外的一个目录上，因为这个文件会多次用到。

```
$ cp /boot/initrd.img-2.6.32-5-amd64 /tmp/bigfile
$ du -sh /tmp/bigfile
11M     /tmp/bigfile
```

将这个大的压缩文件复制到工作区中，复制两份。

```
$ cd /path/to/my/workspace/i-am-admin
$ cp /tmp/bigfile bigfile
$ cp /tmp/bigfile bigfile.dup
```

然后将工作区中两个内容完全一样的大文件加入暂存区。

```
$ git add bigfile bigfile.dup
```

查看一下磁盘空间占用：

- 工作区连同版本库共占用 33MB。

```
$ du -sh .
33M     .
```

- 其中版本库只占用了 11MB。版本库空间占用是工作区的一半。
  如果再有谁说版本库空间占用一定比工作区大，可以用这个例子回击他。

```
$ du -sh .git/
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
11M     .git/
```

看看版本库中对象库内的文件，会发现多出了一个松散对象。之所以添加两个文件而只有一个松散对象，是因为 Git 对于文件的保存是将内容保存为 blob 对象中，和文件名无关，相同内容的不同文件会共享同一个 blob 对象。

```
$ find .git/objects/ -type f
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack
```

如果不想提交，想将文件撤出暂存区，则进行如下操作。

（1）查看当前暂存区的状态。

```
$ git status -s
A  bigfile
A  bigfile.dup
```

（2）将添加的文件撤出暂存区。

```
$ git reset HEAD
```

（3）通过查看状态，看到文件被撤出暂存区了。

```
$ git status -s
?? bigfile
?? bigfile.dup
```

文件撤出暂存区后，在对象库中产生的 blob 松散对象仍然存在，通过查看版本库的磁盘占用就可以看出来。

```
$ du -sh .git/
11M     .git/
```

Git 提供了 git fsck 命令，可以查看到版本库中包含的没有被任何引用关联的松散对象。

```
$ git fsck
dangling blob 2ebcd92d0dda2bad50c775dc662c6cb700477aff
```

标识为 dangling 的对象就是没有被任何引用直接或间接关联到的对象。这个对象就是前面通过暂存区操作引入的大文件的内容。如何将这个文件从版本库中彻底删除呢？ Git 提供了一个清理的命令：

```
$ git prune
```

用 git prune 清理之后，会发现：

- 用 git fsck 查看，没有未被关联到的松散对象。

```
$ git fsck
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




❑ 版本库的空间占用也小了 10MB，证明大的临时对象文件已经从版本库中删除了。

```
$ du -sh .git/
236K    .git/
```

## 14.3 重置操作引入的对象

上一节用 git prune 命令清除暂存区操作时引入的临时对象，但是如果是用重置命令抛弃的提交和文件就不会轻易地被清除。下面用同样的大文件提交到版本库中试验一下。

```
$ cd /path/to/my/workspace/i-am-admin
$ cp /tmp/bigfile bigfile
$ cp /tmp/bigfile bigfile.dup
```

将这两个大文件提交到版本库中。

❑ 添加到暂存区。

```
$ git add bigfile bigfile.dup
```

❑ 提交到版本库。

```
$ git commit -m "add bigfiles."
[master 51519c7] add bigfiles.
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 bigfile
 create mode 100644 bigfile.dup
```

❑ 查看版本库的空间占用。

```
$ du -sh .git/
11M     .git/
```

做一个重置操作，抛弃刚刚针对两个大文件做的提交。

```
$ git reset --hard HEAD^
```

重置之后，看看版本库的变化。

❑ 版本库的空间占用没有变化，还是那么“庞大”。

```
$ du -sh .git/
11M     .git/
```

❑ 查看对象库，看到三个松散对象。

```
$ find .git/objects/ -type f
.git/objects/info/packs
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff
.git/objects/d9/38dee8fde4e5053b12406c66a19183a24238e1
.git/objects/51/519c7d8d60e0f958e135e8b989a78e84122591
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack
```

- 这三个松散对象分别对应于撤销的提交、目录树，以及大文件对应的 blob 对象。

```
$ git cat-file -t 51519c7
commit
$ git cat-file -t d938dee
tree
$ git cat-file -t 2ebcd92
blob
```

向上一节一样，执行 `git prune` 命令，期待版本库空间占用会变小。可是：

- 版本库空间占用没有变化！

```
$ git prune
$ du -sh .git/
11M     .git/
```

- 执行 `git fsck` 也看不到未被关联到的对象。

```
$ git fsck
```

- 除非像下面这样执行。

```
$ git fsck --no-reflogs
dangling commit 51519c7d8d60e0f958e135e8b989a78e84122591
```

还记得前面章节中介绍的 reflog 吗？ reflog 是防止误操作的最后一道闸门。

```
$ git reflog
6652a0d HEAD@{0}: HEAD^: updating HEAD
51519c7 HEAD@{1}: commit: add bigfiles.
```

可以看到撤销的操作仍然记录在 reflog 中，正因为如此，Git 认为撤销的提交和大文件都可以被追踪到，还在使用着，所以无法用 `git prune` 命令删除。

如果确认真的要丢弃不想要的对象，需要对版本库的 reflog 做过期操作，相当于将 `.git/logs/` 下的文件清空。

- 使用下面的 reflog 过期命令做不到让刚刚撤销的提交过期，因为 reflog 的过期操作默认只会让 90 天前的数据过期。

```
$ git reflog expire --all
$ git reflog
6652a0d HEAD@{0}: HEAD^: updating HEAD
51519c7 HEAD@{1}: commit: add bigfiles.
```

- 需要为 git reflog 命令提供 `--expire=<date>` 参数，强制让 `<date>` 之前的记录全部过期。

```
$ git reflog expire --expire=now --all
$ git reflog
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


使用 now 作为时间参数，让 reflog 的全部记录都过期。没有了 reflog，即从 reflog 中看不到回滚的添加大文件的提交后，该提交对应的 commit 对象、tree 对象和 blob 对象就会成为未被关联的 dangling 对象，可以用 git prune 命令清理。下面可以看到清理后，版本库变小了。

```
$ git prune
$ du -sh .git/
244K    .git/
```

## 14.4 Git 管家：git-gc

前面两节介绍的是比较极端的情况，实际操作中会很少用到 git prune 命令来清理版本库，而是会使用一个更为常用的命令 git gc。命令 git gc 就好比 Git 版本库的管家，会对版本库进行一系列的优化动作：

（1）对分散在 .git/refs 下的文件进行打包，打包到文件 .git/packed-refs 中。

如果没有将配置 gc.packrefs 关闭，就会执行命令：git pack-refs --all --prune 实现对引用的打包。

（2）丢弃 90 天前的 reflog 记录。

会运行 reflog 过期命令：git reflog expire --all。因为采用了默认参数调用，因此只会清空 reflog 中 90 天前的记录。

（3）对松散对象进行打包。

运行 git repack 命令，凡是有引用关联的对象都被打在包里，未被关联的对象仍旧以松散对象的形式保存。

（4）清除未被关联的对象。默认只清除 2 周以前的未被关联的对象。

可以向 git gc 提供 --prune=<date> 参数，其中的时间参数传递给 git prune --expire <date>，实现对指定日期之前的未被关联的松散对象进行清理。

（5）其他清理。

如运行 git rerere gc 对合并冲突的历史记录进行过期操作。

从上面的描述中可见命令 git gc 完成了相当多的优化和清理工作，并且最大限度地照顾了安全性的需要。例如像暂存区操作引入的没有关联的临时对象会最少保留 2 个星期，而因为重置而丢弃的提交和文件则会保留最少 3 个月。

下面就把前面的例子用 git gc 再执行一遍，不过这一次添加的两个大文件要稍有不同，以便看到 git gc 打包所实现的对象增量存储的效果。

复制两个大文件到工作区。

```
$ cp /tmp/bigfile bigfile
$ cp /tmp/bigfile bigfile.dup
```

在文件 bigfile.dup 后面追加些内容，以造成 bigfile 和 bigfile.dup 内容不同。

```
$ echo "hello world" >> bigfile.dup
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


将这两个稍有不同的文件提交到版本库。

```
$ git add bigfile bigfile.dup
$ git commit -m "add bigfiles."
[master c62fa4d] add bigfiles.
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 bigfile
 create mode 100644 bigfile.dup
```

可以看到版本库中提交进来的两个不同的大文件是不同的对象。

```
$ git ls-tree HEAD | grep bigfile
100644 blob 2ebcd92d0dda2bad50c775dc662c6cb700477aff    bigfile
100644 blob 9e35f946a30c11c47ba1df351ca22866bc351e7b    bigfile.dup
```

做版本库重置，抛弃最新的提交，即抛弃添加两个大文件的提交。

```
$ git reset --hard HEAD^
HEAD is now at 6652a0d Add Images for git treeview.
```

此时的版本库有多大呢，还是像之前添加两个相同的大文件时占用11MB的空间么？

```
$ du -sh .git/
22M     .git/
```

版本库空间占用居然扩大了一倍！这显然是因为两个大文件分开存储造成的。可以用下面的命令在对象库中查看对象的大小。

```
$ find .git/objects -type f -printf "%-20p\t%s\n"
.git/objects/0c/844d2a072fd69e71638558216b69ebc57ddb64  233
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff  11184682
.git/objects/9e/35f946a30c11c47ba1df351ca22866bc351e7b  11184694
.git/objects/c6/2fa4d6cb4c082fadfa45920b5149a23fd7272e  162
.git/objects/info/packs 54
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx     2892
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack    80015
```

输出的每一行用空白分隔，前面是文件名，后面是以字节为单位的文件大小。从上面的输出中可以看出来，打包文件很小，但是有两个大的文件各自占用了11MB左右的空间。

执行 git gc 并不会删除任何对象，因为这些对象都还没有过期。但是会发现版本库的占用空间变小了。

❑ 执行 git gc 对版本库进行整理。

```
$ git gc
Counting objects: 69, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (49/49), done.
Writing objects: 100% (69/69), done.
Total 69 (delta 11), reused 63 (delta 8)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




❑ 版本库空间占用小了一半！

```
$ du -sh .git/
11M     .git/
```

❑ 原来是因为对象库重新打包，两个大文件采用了增量存储使得版本库变小。

```
$ find .git/objects -type f -printf "%-20p\t%s\n" | sort
.git/objects/info/packs 54
.git/objects/pack/pack-7cae010c1b064406cd6c16d5a6ab2f446de4076c.idx      3004
.git/objects/pack/pack-7cae010c1b064406cd6c16d5a6ab2f446de4076c.pack     11263033
```

如果想将抛弃的历史数据彻底丢弃，进行如下操作。

（1）不再保留 90 天的 reflog，而是将所有 reflog 全部即时过期。

```
$ git reflog expire --expire=now --all
```

（2）通过 git fsck 可以看到有提交成为了未被关联的提交。

```
$ git fsck
dangling commit c62fa4d6cb4c082fadfa45920b5149a23fd7272e
```

（3）这个未被关联的提交就是添加大文件的提交。

```
$ git show c62fa4d6cb4c082fadfa45920b5149a23fd7272e
commit c62fa4d6cb4c082fadfa45920b5149a23fd7272e
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 16 20:18:38 2010 +0800

    add bigfiles.

diff --git a/bigfile b/bigfile
new file mode 100644
index 0000000..2ebcd92
Binary files /dev/null and b/bigfile differ
diff --git a/bigfile.dup b/bigfile.dup
new file mode 100644
index 0000000..9e35f94
Binary files /dev/null and b/bigfile.dup differ
```

（4）不带参数调用 git gc 虽然不会清除尚未过期（未到 2 周）的大文件，但是会将未被关联的对象从打包文件中移出，成为松散文件。

```
$ git gc
Counting objects: 65, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (45/45), done.
Writing objects: 100% (65/65), done.
Total 65 (delta 8), reused 63 (delta 8)
```

（5）未被关联的对象重新成为松散文件，所以 .git 版本库的空间占用又反弹了。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ du -sh .git/
22M     .git/
$ find .git/objects -type f -printf "%-20p\t%s\n" | sort
.git/objects/0c/844d2a072fd69e71638558216b69ebc57ddb64  233
.git/objects/2e/bcd92d0dda2bad50c775dc662c6cb700477aff  11184682
.git/objects/9e/35f946a30c11c47ba1df351ca22866bc351e7b  11184694
.git/objects/c6/2fa4d6cb4c082fadfa45920b5149a23fd7272e  162
.git/objects/info/packs 54
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.idx     2892
.git/objects/pack/pack-969329578b95057b7ea1208379a22c250c3b992a.pack    80015
```

（6）实际上如果使用立即过期参数 `--prune=now` 调用 `git gc`，就不用再等2周了，直接就可以完成对未关联的对象的清理。

```
$ git gc --prune=now
Counting objects: 65, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (45/45), done.
Writing objects: 100% (65/65), done.
Total 65 (delta 8), reused 65 (delta 8)
```

（7）清理过后，版本库的空间占用降了下来。

```
$ du -sh .git/
240K    .git/
```

## 14.5 Git管家的自动执行

对于老版本库的Git，会看到帮助手册中建议用户对版本库进行周期性的整理，以便获得更好的性能，尤其是对于规模比较大的项目，但是对于整理的周期都语焉不详。

实际上对于1.6.6及以后版本的Git已经基本上不需要手动执行 `git gc` 命令了，因为部分Git命令会自动调用 `git gc --auto` 命令，在版本库确实需要整理的情况下自动开始整理操作。

目前有如下的Git命令会自动执行 `git gc --auto` 命令，实现对版本库的按需整理。

- 执行命令 `git merge` 进行合并操作后，对版本库进行按需整理。
- 执行命令 `git receive-pack`，即版本库接收其他版本库PUSH来的提交后，对版本库进行按需整理操作。

  当版本库接收到其他版本库的PUSH请求时，会调用 `git receive-pack` 命令以接收请求。在接收到推送的提交后，对版本库进行按需整理。
- 执行命令 `git rebase -i` 进行交互式变基操作后，会对版本库进行按需整理。
- 执行命令 `git am` 对mbox邮箱中通过邮件提交的补丁在版本库中进行应用的操作后，会对版本库做按需整理操作。

综上所述，对于提供共享式“写操作”的Git版本库，可以免维护。所谓的共享式写操

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


作，就是版本库作为一个裸版本库放在服务器上，团队成员可以通过 PUSH（推送）操作将提交推送到共享的裸版本中。每一次推送操作都会触发 `git gc --auto` 命令，对版本库进行按需整理。

还有，对于非独立工作的本地工作区，也可以免维护。因为和他人协同工作的本地工作区会经常执行 `git pull` 操作从他人版本库或从共享的版本库拉回新提交，执行 `git pull` 操作会触发 `git merge` 操作，因此也会对本地版本库进行按需整理。

在对版本库进行按需整理时，整理的频率是一个问题。如果整理得太勤则没有必要，还会增加系统负担；但如果疏于整理则会导致积累太多的松散文件，当真正开始版本库整理的时候会占用过多的系统资源，影响用户体验。因此实际操作中只有在特定的条件下才会触发真正的版本库整理。

主要的触发条件是：松散对象只有超过一定的数量时才会执行。在统计松散对象数量时，为了降低在 `.git/objects/` 目录下搜索松散对象对系统造成的负担，实际采取了取样搜索，即只会对对象库下的一个子目录 `.git/objects/17` 进行文件搜索。在默认的配置下，只有该目录中对象数目超过 27 个才会触发版本库的整理。至于为什么只在对象库中选择一个子目录进行松散对象的搜索，这是因为 SHA1 哈希值是完全随机的，文件在由前两位哈希值组成的目录中差不多是平均分布的。至于为什么选择 17，不知道对于作者 Junio C Hamano 有什么特殊意义，也许是向 Linus Torvalds 被评选为 20 世纪最有影响力的 100 人中排名第 17 位而致敬[①]。

可以通过设置配置变量 `gc.auto` 的值，调整 Git 管家自动运行时触发版本库整理操作的频率。默认 `gc.auto` 的值为 6700，是触发版本库整理的全部松散对象数的阈值，对于单个取样目录 `.git/objects/17` 来说，超过 (6700+255)/256 个文件时即开始版本库整理。但是注意不要将 `gc.auto` 设置为 0，否则 `git gc --auto` 命令永远不会触发版本库的整理。

① *http://en.wikipedia.org/wiki/Linus_Torvalds*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第3篇 Git和声

上一篇的各章是从个人使用的角度研究和学习 Git，通过连续的实践不但学习了 Git 的基本用法，还深入地了解了 Git 的奥秘，这些都将成为学习本篇内容的基础。本篇不再是一个人的独奏，而是多人的和声，我们将从团队使用的角度对 Git 进行研究。要知道 Git 作为版本控制系统，其主要工作就是团队协作。

团队协作和个人之间有何不同？关键就在于团队成员之间存在着数据交换：

- 数据交换需要协议，这就是第 15 章要介绍的内容。
- 数据交换可能会因为冲突造成中断，第 16 章将专题介绍如何解决冲突。
- 里程碑为数据建立标识，是数据交换的参照点，这将在第 17 章中介绍。
- 分支会为数据交换开辟不同的通道，从而减少冲突和混乱的发生，第 18 章会系统地介绍不同的分支应用模型。
- 与远程版本库进行数据交换，是 Git 协同的重要内容，这将在第 19 章中介绍。
- 本篇的最后（第 20 章）会介绍在不能直接与其他版本库交互的情况下，如何以补丁文件的方式进行数据交换。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 15 章 Git 协议与工作协同

要想团队协作使用 Git，就需要用到 Git 协议。

## 15.1 Git 支持的协议

首先来看看数据交换需要使用的协议。

Git 提供了丰富的协议支持，包括：SSH、GIT、HTTP、HTTPS、FTP、FTPS、RSYNC 及前面已经看到的本地协议等。各种不同协议的 URL 写法如表 15-1 所示。

表 15-1 Git 支持的协议

| 协议名称 | 语法格式 | 说明 |
| --- | --- | --- |
| SSH 协议（1） | ssh://[user@]example.com[:port]/path/to/repo.git/ | 可在 URL 中设置用户名和端口。默认端口 22 |
| SSH 协议（2） | [user@]example.com:path/to/repo.git/ | SCP 格式表示法，更简洁。但是非默认端口需要通过其他方式（如主机别名方式）设定 |
| GIT 协议 | git://example.com[:port]/path/to/repo.git/ | 最常用的只读协议 |
| HTTP[S] 协议 | http[s]://example.com[:port]/path/to/repo.git/ | 兼有智能协议和哑协议 |
| FTP[S] 协议 | ftp[s]://example.com[:port]/path/to/repo.git/ | 哑协议 |
| RSYNC 协议 | rsync://example.com/path/to/repo.git/ | 哑协议 |
| 本地协议（1） | file:///path/to/repo.git | |
| 本地协议（2） | /path/to/repo.git | 和 file:// 格式的本地协议类似，但有细微差别。例如克隆时不支持浅克隆，且采用直接的硬连接实现克隆 |

上面介绍的各种协议可分为两类：智能协议和哑协议。

### 1. 智能协议

在会话时使用智能协议，会在会话的两个版本库的各自一端打开相应的程序进行数据交换。使用智能协议最直观的印象就是在数据传输过程中会有清晰的进度显示，而且因为是按需传输所以传输量更小，速度更快。图 15-1 显示的就是在执行 PULL 和 PUSH 两个最常用的操作时，两个版本库各自启动辅助程序的情况。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

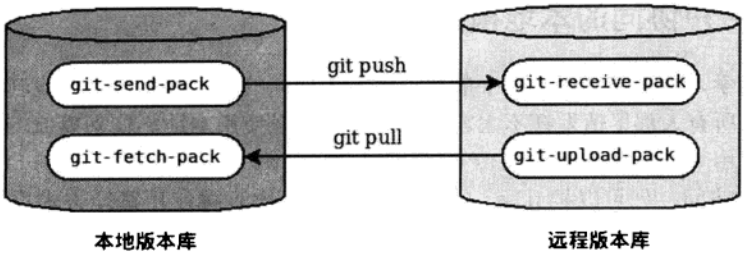




图 15-1 Git 智能协议通信示意图

上述协议中 SSH、GIT 及本地协议（file://）属于智能协议。HTTP 协议需要特殊的配置（用 git-http-backend 配置 CGI），并且客户端需要使用 Git 1.6.6 或更高的版本才能够使用智能协议。

### 2. 哑协议

和智能协议相对的是哑协议。在使用哑协议访问远程版本库的时候，远程版本库不会运行辅助程序，而是完全依靠客户端去主动“发现”。客户端需要访问文件 `.git/info/refs` 获取当前版本库的引用列表，并根据引用对应的提交 ID 直接访问对象库目录下的文件。如果对象文件被打包而不是以松散对象形式存在，则 Git 客户端还要去访问文件 `.git/objects/info/packs` 以获得打包文件列表，并据此读取完整的打包文件，从打包文件中获取对象。由此可见哑协议的效率非常之低，甚至会因为要获取一个对象而去访问整个 pack 包。

使用哑协议最直观的感受是：传输速度非常慢，而且传输进度不可见，不知道什么时候才能够完成数据传输。上述协议中，FTP 和 RSYNC 都是哑协议，没有通过 git-http-backend 或类似 CGI 程序配置的 HTTP 服务器提供的也是哑协议。因为哑协议需要索引文件 `.git/info/refs` 和 `.git/objects/info/packs` 以获取引用和包列表，因此要在版本库的钩子脚本 `post-update` 中设置运行 `git update-server-info` 以确保及时更新哑协议需要的索引文件。不过如果不使用哑协议，运行 `git update-server-info` 就没有什么必要了。

以 Git 项目本身为例，看看如何使用不同的协议地址进行版本库克隆。

- GIT 协议（智能协议）：

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

- HTTP(S) 哑协议：

```
$ git clone http://www.kernel.org/pub/scm/git/git.git
```

- HTTP(S) 智能协议[①]：

```
$ git clone https://github.com/git/git.git
```

① 使用 Git 1.6.6 或更高版本访问。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



## 15.2 多用户协同的本地模拟

在本篇的学习过程中，需要一个能够提供多人访问的版本库，显然要找到一个公共服务器，并且能让所有人都尽情发挥不太容易，但幸好可以使用本地协议来模拟。在后面的内容中，会经常使用本地协议地址 file:///path/to/repos/<project>.git 来代表对某一公共版本库的访问，您可以把 file:// 格式的 URL（比直接使用路径方式更逼真）想象为 git:// 或 http:// 格式，并且想象它是在一台远程的服务器上，而非本机。

同样地，为了模拟多人的操作，也不再使用 /path/to/my/workspace 作为工作区，而是分别使用 /path/to/user1/workspace 和 /path/to/user2/workspace 等路径来代表不同用户的工作环境。同样想象 /path/to/user1/ 和 /path/to/user2/ 是在不同的主机上，并由不同的用户进行操作。

下面就来演示一个共享版本库的搭建过程，以及两个用户 user1 和 user2 在各自的工作区中是如何工作并进行数据交换的，具体过程如下。

（1）于 /path/to/repos/shared.git 中创建一个共享的版本库。

别忘了在第 2 篇的“第 13 章 Git 克隆”一章中介绍的，以裸版本库方式创建。

```
$ git init --bare /path/to/repos/shared.git
Initialized empty Git repository in /path/to/repos/shared.git/
```

（2）用户 user1 克隆版本库。

从下面的命令输出可以看出，克隆一个刚刚初始化完成的裸版本库会显示一个警告，警告正在克隆的版本库是一个空版本库。

```
$ cd /path/to/user1/workspace
$ git clone file:///path/to/repos/shared.git project
Cloning into project...
warning: You appear to have cloned an empty repository.
```

（3）设置 user.name 和 user.email 配置变量。

要在版本库级别设置 user.name 和 user.email 配置变量（即运行 git config 命令时不使用 --global 或 --system 参数），以便和全局设置区分开，因为我们的模拟环境中所有用户都共享同一全局设置和系统设置。

```
$ cd project
$ git config user.name user1
$ git config user.email user1@sun.ossxp.com
```

（4）用户 user1 创建初始数据并提交。

```
$ echo Hello. > README
$ git add README
$ git commit -m "initial commit."
[master (root-commit) 5174bf3] initial commit.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 create mode 100644 README
```

（5）用户 user1 将本地版本库的提交推送到上游。

在下面的推送指令中使用了 origin 别名，其实际指向就是 file:///path/to/repos/shared.git。可以从 .git/config 配置文件中看到是如何实现对 origin 远程版本库注册的。关于远程版本库的内容将在第 19 章介绍。

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 210 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///path/to/repos/shared.git
 * [new branch]      master -> master
```

（6）用户 user2 克隆版本库。

用户 user2 克隆时没有显示警告，因为此时共享版本库已不再是空版本库了。

```
$ cd /path/to/user2/workspace
$ git clone file:///path/to/repos/shared.git project
Cloning into project...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

（7）同样在 user2 的本地版本库中，设置 user.name 和 user.email 配置变量，以区别全局配置设置。

```
$ cd /path/to/user2/workspace/project
$ git config user.name user2
$ git config user.email user2@moon.ossxp.com
```

（8）用户 user2 的本地版本库现在拥有和 user1 用户同样的提交。

```
$ git log
commit 5174bf33ab31a3999a6242fdcb1ec237e8f3f91a
Author: user1 <user1@sun.ossxp.com>
Date:   Sun Dec 19 15:52:29 2010 +0800

    initial commit.
```

## 15.3 强制非快进式推送

现在用户 user1 和 user2 的工作区是相同的。思考一个问题：如果两人各自在本地版本库中进行独立的提交，然后再分别向共享版本库推送，会互相覆盖么？为了回答这个问题，进行下面的实践。

首先，用户 user1 先在本地版本库中进行提交，然后将本地的提交推送到“远程”共享

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


版本库中，操作步骤如下。

（1）用户user1创建team/user1.txt文件。

假设这个项目约定：每个开发者在team目录下写一个自述文件。于是用户user1创建文件team/user1.txt。

```
$ cd /path/to/user1/workspace/project/
$ mkdir team
$ echo "I'm user1." > team/user1.txt
$ git add team
$ git commit -m "user1's profile."
[master b4f3ae0] user1's profile.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user1.txt
```

（2）用户user1将本地提交推送到服务器上。

```
$ git push
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 327 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/shared.git
   5174bf3..b4f3ae0  master -> master
```

（3）查看当前user1版本库中的日志。

```
$ git log --oneline --graph
* b4f3ae0 user1's profile.
* 5174bf3 initial commit.
```

通过上面的操作步骤，可以看到用户user1成功地更新了“远程”共享版本库。如果用户user2不知道用户user1所做的上述操作，仍在基于“远程”版本库旧数据同步而来的本地版本库中进行改动，然后用户user2向“远程”共享版本库推送，会有什么结果呢？用下面的操作验证一下。

（1）用户user2创建team/user2.txt文件。

```
$ cd /path/to/user2/workspace/project/
$ mkdir team
$ echo "I'm user1?" > team/user2.txt
$ git add team
$ git commit -m "user2's profile."
[master 8409e4c] user2's profile.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user2.txt
```

（2）用户user2将本地提交推送到服务器时出错。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git push
To file:///path/to/repos/shared.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

（3）将用户 user2 推送失败的错误日志翻译如下：

```
到版本库 file:///path/to/repos/shared.git
 ! [被拒绝]          master -> master (非快进)
错误：部分引用向 'file:///path/to/repos/shared.git' 推送失败
为防止您丢失历史数据，非快进式更新被拒绝。
在推送前请先合并远程改动，例如执行 'git pull'。
```

用户 user2 推送失败了。但这不是坏事，反倒是一件好事情，因为这避免了用户提交的相互覆盖。Git 通过检查推送操作是不是“快进式”的操作，从而保证用户的提交不会相互覆盖。一般情况下，推送只允许“快进式”推送。所谓快进式推送，就是要推送的本地版本库的提交是建立在远程版本库相应分支的现有提交基础上的，即远程版本库相应分支的最新提交是本地版本库最新提交的祖先提交。但现在用户 user2 执行的推送并非如此，是一个非快进式的推送。

- 此时用户 user2 本地版本库的最新提交及其历史提交可以用 `git rev-list` 命令显示，如下所示：

```
$ git rev-list HEAD
8409e4c72388a18ea89eecb86d68384212c5233f
5174bf33ab31a3999a6242fdcb1ec237e8f3f91a
```

- 用 `git ls-remote` 命令显示远程版本库的引用对应的 SHA1 哈希值，会发现远程版本库所包含的最新提交的 SHA1 哈希值是 b4f3ae0...，不是本地最新提交的祖先提交。

```
$ git ls-remote origin
b4f3ae0fcadce8c343f3cdc8a69c33cc98c98dfd        HEAD
b4f3ae0fcadce8c343f3cdc8a69c33cc98c98dfd        refs/heads/master
```

实际上当用户 user2 执行推送的时候，Git 就是利用类似方法判断出当前的推送不是一个快进式推送，于是产生警告并终止。

那么如何才能成功推送呢？一个不一定正确的解决方案是：强制推送。

在推送命令的后面使用 `-f` 参数可以进行强制推送，即使是非快进式的推送也会成功执行。用户 user2 执行强制推送，会强制刷新服务器中的版本。

```
$ git push -f
Counting objects: 7, done.
Delta compression using up to 2 threads.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Compressing objects: 100% (3/3), done.
Writing objects: 100% (7/7), 503 bytes, done.
Total 7 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (7/7), done.
To file:///path/to/repos/shared.git
 + b4f3ae0...8409e4c master -> master (forced update)
```

注意到了么，在强制推送的最后一行输出中显示了“强制更新（forced update）”字样。这样用户user1向版本库推送的提交由于用户user2的强制推送被覆盖了。实际上在这种情况下user1也可以强制推送，从而用自己（user1）的提交再去覆盖用户user2的提交。这样的工作模式不是协同，而是战争！

在上面用户user2使用非快进式推送强制更新版本库，实际上是危险的。滥用非快进式推送可能造成提交覆盖大战（战争是霸权的滥用）。正确地使用非快进式推送，应该是在不会造成提交覆盖“战争”的前提下，对历史提交进行修补。

下面的操作也许是一个使用非快进式推送的更好的例子。

（1）用户user2改正之前错误的录入。

细心的读者可能已经发现，用户user2在创建个人描述的文件中把自己的名字写错了。假设用户user2在刚刚完成向服务器的推送操作后也发现了这个错误，于是user2进行了下面的更改。

```
$ echo "I'm user2." > team/user2.txt
$ git diff
diff --git a/team/user2.txt b/team/user2.txt
index 27268e2..2dcb7b6 100644
--- a/team/user2.txt
+++ b/team/user2.txt
@@ -1 +1 @@
-I'm user1?
+I'm user2.
```

（2）然后用户user2将修改好的文件提交到本地版本库中。

采用直接提交还是使用修补式提交，这是一个问题。因为前次的提交已经被推送到共享版本库中，如果采用修补提交会造成前一次提交被新提交抹掉，从而在下次推送时造成非快进式推送。这时用户user2就要评估“战争”的风险：“我刚刚推送的提交，有没有可能被其他人获取了（通过git pull、git fetch或git clone操作）？”如果确认不会有他人获取，例如现在公司里只有user2自己一个人在加班，那么可以放心地进行修补操作。

```
$ git add -u
$ git commit --amend -m "user2's profile."
[master 6b1a7a0] user2's profile.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user2.txt
```

（3）采用强制推送，更新远程共享版本库中的提交。这个操作越早越好，在他人还没有来得及和服务器同步前将修补提交强制更新到服务器上。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git push -f
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 331 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/shared.git
 + 8409e4c...6b1a7a0 master -> master (forced update)
```

## 15.4 合并后推送

理性的工作协同要避免非快进式推送。一旦向服务器推送后，如果发现错误，不要使用会更改历史的操作（变基、修补提交），而是采用不会改变历史提交的反转提交等操作。

如果在向服务器推送过程中，由于他人率先推送了新的提交导致遭遇到非快进式推送的警告，应该进行如下操作才更为理性：执行 git pull 获取服务器端最新的提交并和本地提交进行合并，合并成功后再向服务器提交。

例如用户 user1 在推送时遇到了非快进式推送错误，可以通过如下操作将本地版本库的修改和远程版本库的最新提交进行合并。

（1）用户 user1 发现推送遇到了非快进式推送。

```
$ cd /path/to/user1/workspace/project/
$ git push
To file:///path/to/repos/shared.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

（2）用户 user1 运行 git pull 命令。

命令 git pull 实际包含两个动作：获取远程版本库的最新提交，以及将获取到的远程版本库提交与本地提交进行合并。

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/shared
 + b4f3ae0...6b1a7a0 master     -> origin/master  (forced update)
Merge made by recursive.
 team/user2.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 team/user2.txt
```

（3）合并之后，看看版本库的提交关系图。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


合并之后远程服务器中的最新提交 6b1a7a0 成为当前最新提交（合并提交）的父提交之一。如果再推送，则不再是非快进式的了。

```
$ git log --graph --oneline
*   bccc620 Merge branch 'master' of file:///path/to/repos/shared
|\
| * 6b1a7a0 user2's profile.
* | b4f3ae0 user1's profile.
|/
* 5174bf3 initial commit.
```

（4）执行 git push 命令，成功完成到远程版本库的推送。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 686 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To file:///path/to/repos/shared.git
   6b1a7a0..bccc620  master -> master
```

## 15.5 禁止非快进式推送

非快进式推送如果被滥用就会成为项目的灾难：

- 团队成员之间的提交战争取代了本应的相互协作。
- 造成不必要的冲突，为他人造成麻烦。
- 在提交历史中引入包含修补提交前后两个版本的怪异的合并提交。

Git 提供了至少两种方式对非快进式推送进行限制。一个是通过版本库的配置，另一个是通过版本库的钩子脚本。

将版本库的配置变量 receive.denyNonFastForwards 设置为 true 可以禁止任何用户进行非快进式推送。下面的示例中，可以看到对一个已经预先设置为禁止非快进式推送的版本库执行非快进式推送操作，将会被禁止，即使使用强制推送操作也会被禁止。

（1）更改服务器版本库 /path/to/repos/shared.git 的配置变量。

```
$ git --git-dir=/path/to/repos/shared.git config \
      receive.denyNonFastForwards true
```

（2）在用户 user1 的工作区执行重置操作，以便在后面执行推送时产生非快进式推送。

```
$ git reset --hard HEAD^1
$ git log --graph --oneline
* b4f3ae0 user1's profile.
* 5174bf3 initial commit.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（3）用户 user1 即便使用强制推送也不会成功。

在出错信息中看到服务器端拒绝执行：[remote rejected]。

```
$ git push -f
Total 0 (delta 0), reused 0 (delta 0)
remote: error: denying non-fast-forward refs/heads/master (you should pull first)
To file:///path/to/repos/shared.git
 ! [remote rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
```

另外一个方法是通过钩子脚本进行设置，可以仅对某些情况下的非快进式推送进行限制，而不是不分青红皂白地一概拒绝。例如：只对部分用户进行限制，而允许特定用户执行非快进式推送，或者允许某些分支可以进行强制提交而其他分支不可以。第 5 篇第 30 章会介绍 Gitolite 服务架设，通过授权文件（实际上通过版本库的 update 钩子脚本实现）对版本库非快进式推送做出更为精细的授权控制。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第16章 冲突解决

上一章介绍了 Git 协议，并且使用本地协议来模拟一个远程的版本库，以两个不同用户的身份检出该版本库，和该远程版本库进行交互——交换数据、协同工作。在上一章的协同中只遇到了一个小小的麻烦——非快进式推送，可以通过执行拉回操作（git pull），成功完成合并后再推送。

但是在真实的运行环境中，用户间协同并不总是会一帆风顺，只要有合并就可能会有冲突。本章就重点介绍冲突解决机制。

## 16.1 拉回操作中的合并

为了降低难度，上一章的实践中用户 user1 执行 git pull 操作解决非快进式推送问题似乎非常简单，就好像只要把共享版本库中的最新提交直接拉回到本地，然后就可以推送了，其他的好像什么都没有发生一样。真的是这样么？

（1）用户 user1 向共享版本库推送时，因为 user2 强制推送已经改变了共享版本库中的提交状态，导致 user1 推送失败，如图 16-1 所示。

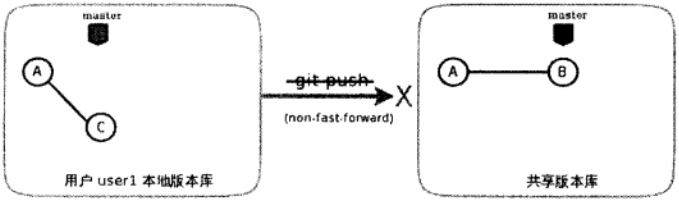



图 16-1 非快进式推送被禁止

（2）用户 user1 执行拉回操作的第一阶段，将共享版本库 master 分支的最新提交获取到本地，并更新到本地版本库特定的引用 refs/remotes/origin/master（简称为 origin/master），如图 16-2 所示。

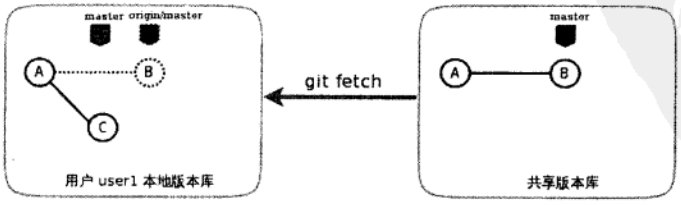



图 16-2 执行获取操作

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（3）用户 user1 执行拉回操作的第二阶段，将本地分支 master 和共享版本库本地跟踪分支 origin/master 进行合并操作，如图 16-3 所示。

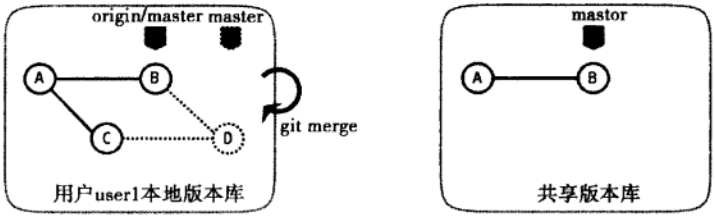



图 16-3 执行合并操作

（4）用户 user1 执行推送操作，将本地提交推送到共享版本库中，如图 16-4 所示。

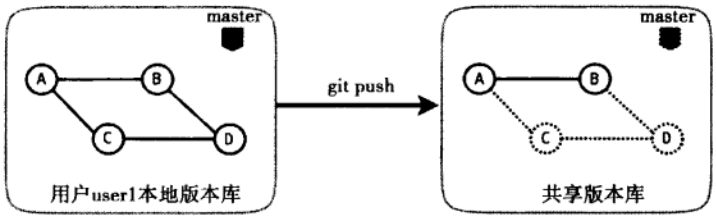



图 16-4 执行推送操作

实际上拉回操作（git pull）是由两个步骤组成的：一个是获取操作（git fetch），另一个是合并操作（git merge），即：

```
git pull = git fetch + git merge
```

图 16-2 示意的获取操作看似很简单，实际上要到第 19 章介绍远程版本库的章节才能够讲明白，现在只需要根据图示将获取操作理解为将远程的共享版本库的对象（提交、里程碑、分支等）复制到本地即可。

合并操作是本章要介绍的重点。合并操作可以由拉回操作（git pull）隐式地执行，将其他版本库的提交和本地版本库的提交进行合并。还可以针对本版本库中的其他分支（将在第 18 章中介绍）进行显示的合并操作，将其他分支的提交和当前分支的提交进行合并。

合并操作的命令行格式如下：

```
git merge [选项 ...] <commit>...
```

合并操作的大多数情况，只须提供一个 <commit>（提交 ID 或对应的引用：分支、里程碑等）作为参数。合并操作将 <commit> 对应的目录树和当前工作分支的目录树的内容进行合并，合并后的提交以当前分支的提交作为第一个父提交，以 <commit> 为第二个父提交。合并操作还支持将多个 <commit> 代表的分支和当前分支进行合并，过程类似。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


默认情况下，合并后的结果会自动提交，但是如果提供 --no-commit 选项，则合并后的结果会放入暂存区，用户可以对合并结果进行检查、更改，然后手动提交。

合并操作并非总会成功，因为合并的不同提交可能同时修改了同一文件相同区域的内容，导致冲突。冲突会造成合并操作的中断，冲突的文件被标识，用户可以对标识为冲突的文件进行冲突解决操作，然后更新暂存区，再提交，最终完成合并操作。

根据合并操作是否遇到冲突，以及不同的冲突类型，可以分为以下几种情况：成功的自动合并、逻辑冲突、真正的冲突和树冲突。下面分别予以介绍。

## 16.2 合并一：自动合并

Git 的合并操作非常智能，大多数情况下会自动完成合并。不管是修改不同的文件，还是修改相同的文件（文件的不同位置），或者文件名变更。

### 16.2.1 修改不同的文件

如果用户 user1 和 user2 各自的本地提交中修改了不同的文件，当一个用户将改动推送到服务器后，另外一个用户推送就会遇到非快进式推送错误，需要先合并再推送。因为两个用户修改了不同的文件，合并不会遇到麻烦。

在上一章的操作过程中，两个用户的本地版本库和共享版本库可能不一致，为了确保版本库状态的一致性，以便下面的实践能够正常执行，分别在两个用户的本地版本库中执行下面的操作。

```
$ git fetch
$ git reset --hard origin/master
```

下面的实践中，两个用户分别修改不同的文件，其中一个用户要尝试合并操作将本地提交和另外一个用户的提交合并，具体操作过程如下。

（1）用户 user1 修改 team/user1.txt 文件，提交并推送到共享服务器。

```
$ cd /path/to/user1/workspace/project/
$ echo "hack by user1 at `date`" >> team/user1.txt
$ git add -u
$ git commit -m "update team/user1.txt"
$ git push
```

（2）用户 user2 修改 team/user2.txt 文件，提交。

```
$ cd /path/to/user2/workspace/project/
$ echo "hack by user2 at `date`" >> team/user2.txt
$ git add -u
$ git commit -m "update team/user2.txt"
```

（3）用户 user2 在推送的时候，会遇到非快进式推送的错误而被终止。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git push
To file:///path/to/repos/shared.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/shared.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

（4）用户 user2 执行获取（git fetch）操作。获取到的提交更新到本地用于跟踪共享版本库 master 分支的本地引用 origin/master 中。

```
$ git fetch
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/shared
   bccc620..25fce74  master     -> origin/master
```

（5）用户 user2 执行合并操作，完成自动合并。

```
$ git merge origin/master
Merge made by recursive.
 team/user1.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

（6）用户 user2 推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 747 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To file:///path/to/repos/shared.git
   25fce74..0855b86  master -> master
```

（7）通过提交日志，可以看到成功合并的提交及其两个父提交的关系图。

```
$ git log -3 --graph --stat
*   commit 0855b86678d1cf86ccdd13adaaa6e735715d6a7e
|\  Merge: f53acdf 25fce74
| | Author: user2 <user2@moon.ossxp.com>
| | Date:   Sat Dec 25 23:00:55 2010 +0800
| |
| |     Merge remote branch 'origin/master'
| |
| * commit 25fce74b5e73b960c42e4a463d03d462919b674d
| | Author: user1 <user1@sun.ossxp.com>
| | Date:   Sat Dec 25 22:54:53 2010 +0800
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
| |
| |     update team/user1.txt
| |
| |  team/user1.txt |    1 +
| |  1 files changed, 1 insertions(+), 0 deletions(-)
| |
* | commit f53acdf6a76e0552b562f5aaa4d40ff19e8e2f77
|/  Author: user2 <user2@moon.ossxp.com>
|   Date:   Sat Dec 25 22:56:49 2010 +0800
|
|       update team/user2.txt
|
|    team/user2.txt |    1 +
|    1 files changed, 1 insertions(+), 0 deletions(-)
```

## 16.2.2 修改相同文件的不同区域

当用户user1和user2在本地提交中修改相同的文件，但是修改的是文件的不同位置时，则两个用户的提交仍可成功合并，具体操作过程如下。

（1）为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

（2）用户user1在自己的工作区中修改README文件，在文件的第一行插入内容，更改后的文件内容如下。

```
User1 hacked.
Hello.
```

（3）用户user1对修改进行本地提交并推送到共享版本库。

```
$ git add -u
$ git commit -m "User1 hack at the beginning."
$ git push
```

（4）用户user2在自己的工作区中修改README文件，在文件的最后插入内容，更改后的文件内容如下。

```
Hello.
User2 hacked.
```

（5）用户user2对修改进行本地提交。

```
$ git add -u
$ git commit -m "User2 hack at the end."
```

（6）用户user2执行获取（git fetch）操作。获取到的提交更新到本地用于跟踪共享版本库master分支的本地引用origin/master中。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///path/to/repos/shared
   0855b86..07e9d08  master     -> origin/master
```

（7）用户 user2 执行合并操作，完成自动合并。

```
$ git merge refs/remotes/origin/master
Auto-merging README
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

（8）用户 user2 推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 607 bytes, done.
Total 6 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (6/6), done.
To file:///path/to/repos/shared.git
   07e9d08..2a67e6f  master -> master
```

（9）如果追溯一下 README 文件每一行的来源，可以看到分别是 user1 和 user2 更改的最前和最后的一行。

```
$ git blame README
07e9d082 (user1 2010-12-25 23:12:17 +0800 1) User1 hacked.
^5174bf3 (user1 2010-12-19 15:52:29 +0800 2) Hello.
bb0c74fa (user2 2010-12-25 23:14:27 +0800 3) User2 hacked.
```

### 16.2.3 同时更改文件名和文件内容

如果一个用户将文件移动到其他目录（或修改文件名），另外一个用户针对重命名前的文件进行了修改，还能够实现自动合并么？这对于其他版本控制系统可能是一个难题，例如 Subversion 就不能很好地处理，还为此引入了一个“树冲突”的新名词。Git 对于此类冲突能够很好地处理，可以自动解决冲突实现自动合并，具体操作过程如下。

（1）为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

（2）用户 user1 在自己的工作区中将文件 README 进行重命名，本地提交并推送到共享版本库。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cd /path/to/user1/workspace/project/
$ mkdir doc
$ git mv README doc/README.txt
$ git commit -m "move document to doc/."
$ git push
```

（3）用户user2在自己的工作区中修改README文件，在文件的最后插入内容，并本地提交。

```
$ cd /path/to/user2/workspace/project/
$ echo "User2 hacked again." >> README
$ git add -u
$ git commit -m "User2 hack README again."
```

（4）用户user2执行获取（git fetch）操作。获取到的提交更新到本地跟踪共享版本库master分支的本地引用origin/master中。

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From file:///path/to/repos/shared
   0855b86..07e9d08  master     -> origin/master
```

（5）用户user2执行合并操作，完成自动合并。

```
$ git merge refs/remotes/origin/master
Merge made by recursive.
 README => doc/README.txt |    0
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename README => doc/README.txt (100%)
```

（6）用户user2推送合并后的本地版本库到共享版本库。

```
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 636 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To file:///path/to/repos/shared.git
   9c51cb9..f73db10  master -> master
```

（7）使用-m参数可以查看合并操作所做出的修改。

```
$ git log -1 -m --stat
commit f73db106c820f0c6d510f18ae8c67629af9c13b7 (from 887488eee19300c566c272ec84b
236026b0303c6)
Merge: 887488e 9c51cb9
Author: user2 <user2@moon.ossxp.com>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    Date:   Sat Dec 25 23:36:57 2010 +0800

        Merge remote branch 'refs/remotes/origin/master'

     README          |    4 ----
     doc/README.txt  |    4 ++++
     2 files changed, 4 insertions(+), 4 deletions(-)

    commit f73db106c820f0c6d510f18ae8c67629af9c13b7 (from 9c51cb91bfe12654e2de1d61d72
2161db0539644)
    Merge: 887488e 9c51cb9
    Author: user2 <user2@moon.ossxp.com>
    Date:   Sat Dec 25 23:36:57 2010 +0800

        Merge remote branch 'refs/remotes/origin/master'

     doc/README.txt |    1 +
     1 files changed, 1 insertions(+), 0 deletions(-)
```

## 16.3 合并二：逻辑冲突

自动合并如果成功地执行，则大多数情况下就意味着完事大吉，但是在某些特殊情况下，合并后的结果虽然在 Git 看来是完美的合并，实际上却存在着逻辑冲突。

一个典型的逻辑冲突是一个用户修改了一个文件的文件名，而另外的用户在其他文件中引用旧的文件名，这样的合并虽然能够成功但是包含着逻辑冲突。例如：

（1）一个 C 语言的项目中存在头文件 `hello.h`，该头文件定义了一些函数声明。

（2）用户 user1 将 `hello.h` 文件改名为 `api.h`。

（3）用户 user2 写了一个新的源码文件 `foo.c` 并在该文件中包含了 `hello.h` 文件。

（4）两个用户的提交合并后，会因为源码文件 `foo.c` 找不到所包含的文件 `hello.h` 而导致项目编译失败。

再举一个逻辑冲突的示例。假如一个用户修改了函数返回值而另外的用户使用旧的函数返回值，虽然成功合并但是存在逻辑冲突：

（1）函数 `compare(obj1, obj2)` 用于比较两个对象 obj1 和 obj2。返回 1 代表比较的两个对象相同，返回 0 代表比较的两个对象不同。

（2）用户 user1 修改了该函数的返回值，返回 0 代表两个对象相同，返回 1 代表 obj1 大于 obj2，返回 −1 则代表 obj1 小于 obj2。

（3）用户 user2 不知道 user1 对该函数的改动，仍以该函数原返回值判断两个对象的异同。

（4）两个用户的提交合并后，不会出现编译错误，但是软件中会潜藏着重大的 Bug。

上面的两个逻辑冲突的示例，尤其是最后一个非常难以捕捉。如果因此而贬低 Git 的自动合并，或者对每次自动合并的结果疑神疑鬼，进而花费大量精力去分析合并的结果，则是因噎废食、得不偿失。一个好的项目实践是每个开发人员都为自己的代码编写可运行的单元

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


测试，项目每次编译时都要执行自动化测试，捕捉潜藏的Bug。在2010年OpenParty上的一个报告中，我介绍了如何在项目中引入单元测试及自动化集成，可以参考下面的链接：

- *http://www.beijing-open-party.org/topic/9*
- *http://wenku.baidu.com/view/63bf7d160b4e767f5acfcef6.html*

## 16.4 合并三：冲突解决

如果两个用户修改了同一文件的同一区域，则在合并的时候会遇到冲突而导致合并过程中断。这是因为Git并不能越俎代庖地替用户做出决定，而是把决定权交给用户。在这种情况下，Gits标识出合并冲突，等待用户对冲突做出抉择。

下面的实践非常简单，两个用户都修改doc/README.txt文件，在第二行“Hello.”的后面加上自己的名字，具体操作过程如下。

（1）为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

（2）用户user1在自己的工作区中修改doc/README.txt文件（仅改动了第二行）。修改后内容如下：

```
User1 hacked.
Hello, user1.
User2 hacked.
User2 hacked again.
```

（3）用户user1对修改进行本地提交并推送到共享版本库。

```
$ git add -u
$ git commit -m "Say hello to user1."
$ git push
```

（4）用户user2在自己的工作区中修改doc/README.txt文件（仅改动了第二行）。修改后内容如下：

```
User1 hacked.
Hello, user2.
User2 hacked.
User2 hacked again.
```

（5）用户user2对修改进行本地提交。

```
$ git add -u
$ git commit -m "Say hello to user2."
```

（6）用户user2执行拉回操作，遇到冲突。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


git pull 操作相当于 git fetch 和 git merge 两个操作。

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/shared
   f73db10..a123390  master     -> origin/master
Auto-merging doc/README.txt
CONFLICT (content): Merge conflict in doc/README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

执行 git pull 时所做的合并操作由于遇到冲突导致中断。来看看处于合并冲突状态时工作区和暂存区的状态。

执行 git status 命令，可以从状态输出中看到文件 doc/README.txt 处于未合并（冲突）的状态，这个文件在两个不同的提交中都做了修改。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      doc/README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

那么 Git 是如何记录合并过程及冲突的呢？实际上合并过程是通过 .git 目录下的几个文件进行记录的：

- 文件 .git/MERGE_HEAD 记录所合并的提交 ID。
- 文件 .git/MERGE_MSG 记录合并失败的信息。
- 文件 .git/MERGE_MODE 标识合并状态。

版本库暂存区中则会记录冲突文件的多个不同版本。可以使用 git ls-files 命令查看：

```
$ git ls-files -s
100644 ea501534d70a13b47b3b4b85c39ab487fa6471c2 1       doc/README.txt
100644 5611db505157d312e4f6fb1db2e2c5bac2a55432 2       doc/README.txt
100644 036dbc5c11b0a0cefc8247cf0e9a3e678f8de060 3       doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

在上面的输出中，每一行分为四个字段，前两个分别是文件的属性和 SHA1 哈希值。第三个字段是暂存区编号。当合并冲突发生后，会用到 0 以上的暂存区编号。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 编号为1的暂存区用于保存冲突文件修改之前的副本，即冲突双方共同的祖先版本。可以用 :1:<filename> 访问。

```
$ git show :1:doc/README.txt
User1 hacked.
Hello.
User2 hacked.
User2 hacked again.
```

- 编号为2的暂存区用于保存当前冲突文件在当前分支中修改的副本。可以用 :2:<filename> 访问。

```
$ git show :2:doc/README.txt
User1 hacked.
Hello, user2.
User2 hacked.
User2 hacked again.
```

- 编号为3的暂存区用于保存当前冲突文件在合并版本（分支）中修改的副本。可以用 :3:<filename> 访问。

```
$ git show :3:doc/README.txt
User1 hacked.
Hello, user1.
User2 hacked.
User2 hacked again.
```

对暂存区中冲突文件的上述三个副本无须了解太多，这三个副本实际上是提供冲突解决工具，用于实现三向文件合并的。

工作区的版本则可能同时包含了成功的合并及冲突的合并，其中冲突的合并会用特殊的标记（<<<<<<< ======= >>>>>>>）进行标识。查看当前工作区中冲突的文件：

```
$ cat doc/README.txt
User1 hacked.
<<<<<<< HEAD
Hello, user2.
=======
Hello, user1.
>>>>>>> a123390b8936882bd53033a582ab540850b6b5fb
User2 hacked.
User2 hacked again.
```

特殊标识 <<<<<<<（七个小于号）和 =======（七个等号）之间的内容是当前分支所更改的内容。特殊标识 =======（七个等号）和 >>>>>>>（七个大于号）之间的内容是所合并的版本更改的内容。

冲突解决的实质就是通过编辑操作，将冲突标识符所标识的冲突内容替换为合适的内容，并去掉冲突标识符。编辑完毕后执行 git add 命令将文件添加到暂存区（标号0），然

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


后再提交就完成了冲突解决。

当工作区处于合并冲突状态时，无法再执行提交操作。此时有两个选择：放弃合并操作，或者对合并冲突进行冲突解决操作。放弃合并操作非常简单，只须执行 git reset 将暂存区重置即可。下面重点介绍如何进行冲突解决的操作。有两个方法进行冲突解决，一个是对少量冲突非常适合的手工编辑操作，另外一个是使用图形化冲突解决工具。

### 16.4.1 手工编辑完成冲突解决

先来看看不使用工具，直接手动编辑完成冲突解决。打开文件 doc/README.txt，将冲突标识符所标识的文字替换为 Hello, user1 and user2.。修改后的文件内容如下：

```
User1 hacked.
Hello, user1 and user2.
User2 hacked.
User2 hacked again.
```

然后添加到暂存区，并提交：

```
$ git add -u
$ git commit -m "Merge completed: say hello to all users."
```

查看最近三次提交的日志，会看到最新的提交就是一个合并提交：

```
$ git log --oneline --graph -3
*   bd3ad1a Merge completed: say hello to all users.
|\
| * a123390 Say hello to user1.
* | 60b10f3 Say hello to user2.
|/
```

提交完成后，会看到 .git 目录下与合并相关的文件 .git/MERGE_HEAD、.git/MERGE_MSG、.git/MERGE_MODE 文件都自动删除了。

如果查看暂存区，会发现冲突文件在暂存区中的三个副本也都清除了（实际在对编辑完成的冲突文件执行 git add 后就已经清除了）。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0       doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

### 16.4.2 图形工具完成冲突解决

上面介绍的通过手工编辑完成冲突解决并不复杂，对于简单的冲突是最快捷的解决方法。但是如果冲突的区域过多、过大，并且缺乏原始版本作为参照，冲突解决过程就会显得非常的不便，这种情况下使用图形工具就显得非常有优势。

还以上面的合并冲突为例介绍使用图形工具进行冲突解决的方法。为了制造一个冲突，

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


首先把 user2 辛辛苦苦完成的冲突解决提交回滚，再执行合并重新进入冲突状态。

（1）将冲突解决的提交丢弃，即强制重置到前一个版本。

```
$ git reset --hard HEAD^
```

（2）这时查看状态，会显示当前工作分支的最新提交和共享版本库的 master 分支的最新提交出现了偏离。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
nothing to commit (working directory clean)
```

（3）那么执行合并操作吧。冲突发生了。

```
$ git merge refs/remotes/origin/master
Auto-merging doc/README.txt
CONFLICT (content): Merge conflict in doc/README.txt
Automatic merge failed; fix conflicts and then commit the result.
```

下面就演示使用图形工具如何解决冲突。使用图形工具进行冲突解决需要事先在操作系统中安装相关的工具软件，如：kdiff3、meld、tortoisemerge、araxis 等。而启动图形工具进行冲突解决也非常简单，只须执行命令 `git mergetool` 即可。

```
$ git mergetool
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff
diffuse ecmerge p4merge araxis emerge vimdiff
Merging:
doc/README.txt

Normal merge conflict for 'doc/README.txt':
  {local}: modified
  {remote}: modified
Hit return to start merge resolution tool (kdiff3):
```

运行 `git mergetool` 命令后，会显示支持的图形工具列表，并提示用户选择可用的冲突解决工具。默认会选择系统中已经安装的工具软件，如 `kdiff3`。直接按下回车键，自动打开 `kdiff3` 进入冲突解决界面。

启动 kdiff3 后，如图 16-5 所示，上方三个窗口由左至右显示冲突文件的三个版本，分别是：

A：暂存区 1 中的版本（共同祖先版本）。

B：暂存区 2 中的版本（当前分支更改的版本）。

C：暂存区 3 中的版本（他人更改的版本）。

kdiff3 下方的窗口是合并后文件的编辑窗口。如图 16-6 所示，点击标记为“合并冲突”的一行，在弹出菜单中出现 A、B、C 三个选项，分别代表从 A、B、C 三个窗口复制相关内容到当前位置。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

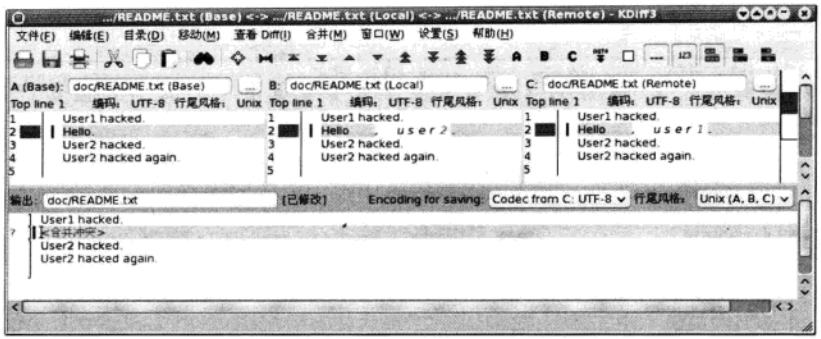




图 16-5 kdiff3 冲突解决界面

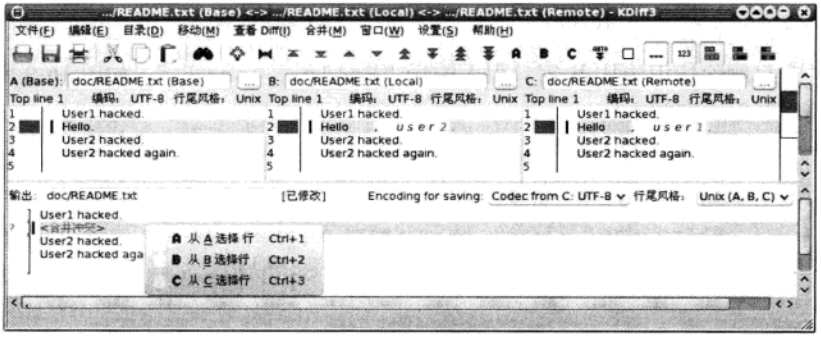



图 16-6 kdiff3 合并冲突行的弹出菜单

当通过图 16-6 显示的弹出菜单选择了 B 和 C 后，可以在图 16-7 中看到在合并窗口中出现了标识 B 和 C 的行，分别代表 user2 和 user1 对该行的修改。

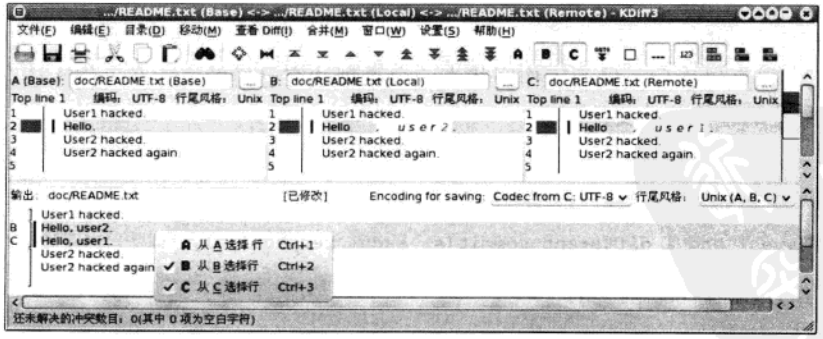



图 16-7 在 kdiff3 的冲突区域同时选取 B 和 C 的修改

在合并窗口进行编辑，将“Hello, user1.”修改为“Hello, user1 and user2.”，如图 16-8 所示。修改后，可以看到该行的标识由 C 改变为 m，含义是该行是经过手工修

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


改的行。

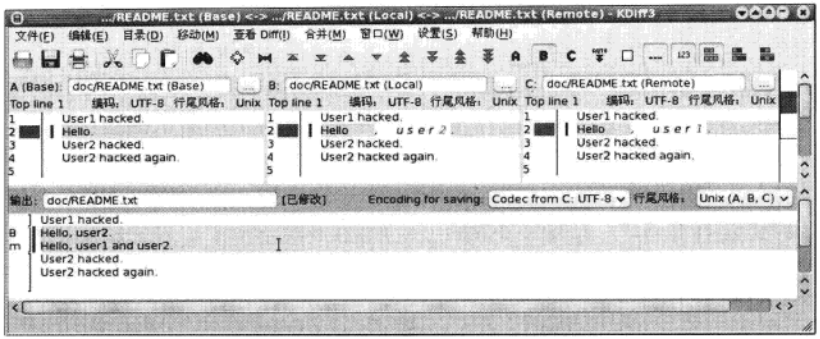



图 16-8 在 kdiff3 的冲突区域编辑内容

在合并窗口删除标识为从 B 窗口引入的行“Hello, user2.”，如图 16-9 所示。保存退出即完成图形化冲突解决。

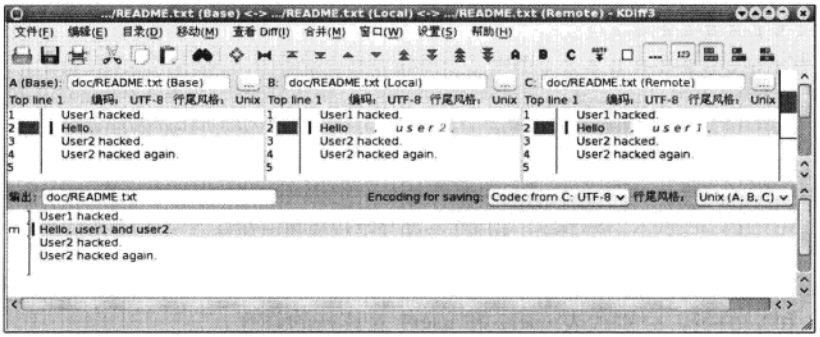



图 16-9 完成 kdiff3 冲突区域的编辑

图形工具保存退出后，显示工作区状态，会看到冲突已经解决。在工作区还会遗留一个以 .orig 结尾的合并前的文件副本。

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Changes to be committed:
#
#       modified:   doc/README.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
#       doc/README.txt.orig
```

查看暂存区会发现暂存区中的冲突文件的三个副本都已经清除。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0       doc/README.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

执行提交和推送。

```
$ git commit -m "Say hello to all users."
[master 7f7bb5e] Say hello to all users.
$ git push
Counting objects: 14, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 712 bytes, done.
Total 8 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To file:///path/to/repos/shared.git
   a123390..7f7bb5e  master -> master
```

查看最近三次的提交日志，会看到最新的提交是一个合并提交。

```
$ git log --oneline --graph -3
*   7f7bb5e Say hello to all users.
|\
| * a123390 Say hello to user1.
* | 60b10f3 Say hello to user2.
|/
```

## 16.5 合并四：树冲突

如果一个用户将某个文件改名，另外一个用户将同样的文件改为另外的名字，当这两个用户的提交进行合并操作时，Git 显然无法替用户做出裁决，于是就产生了冲突。这种因为文件名修改造成的冲突，称为树冲突。这种树冲突的解决方式比较特别，因此专题介绍。

仍旧使用前面的版本库进行此次实践。为确保两个用户的本地版本库和共享版本库状态一致，先分别对两个用户的本地版本库执行拉回操作。

```
$ git pull
```

下面就分别以两个用户的身份执行提交，将同样的一个文件改为不同的文件名，制造一个树冲突，具体操作过程如下。

（1）用户 user1 将文件 doc/README.txt 改名为 readme.txt，提交并推送到共享版本库。

```
$ cd /path/to/user1/workspace/project
$ git mv doc/README.txt readme.txt
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git commit -m "rename doc/README.txt to readme.txt"
[master 615c1ff] rename doc/README.txt to readme.txt
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename doc/README.txt => readme.txt (100%)
$ git push
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 282 bytes, done.
Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
To file:///path/to/repos/shared.git
   7f7bb5e..615c1ff  master -> master
```

（2）用户user2将文件doc/README.txt改名为README，并做本地提交。

```
$ cd /path/to/user2/workspace/project
$ git mv doc/README.txt README
$ git commit -m "rename doc/README.txt to README"
[master 20180eb] rename doc/README.txt to README
 1 files changed, 0 insertions(+), 0 deletions(-)
 rename doc/README.txt => README (100%)
```

（3）用户user2执行git pull操作，遇到合并冲突。

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
From file:///path/to/repos/shared
   7f7bb5e..615c1ff  master     -> origin/master
CONFLICT (rename/rename): Rename "doc/README.txt"->"README" in branch "HEAD"
rename "doc/README.txt"->"readme.txt" in "615c1ffaa41b2798a56854259caeeb1020c51721"
Automatic merge failed; fix conflicts and then commit the result.
```

因为两个用户同时更改了同一文件的文件名并且改成了不同的名字，于是引发冲突。此时查看状态会看到：

```
$ git status
# On branch master
# Your branch and 'refs/remotes/origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       added by us:        README
#       both deleted:       doc/README.txt
#       added by them:      readme.txt
#
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
no changes added to commit (use "git add" and/or "git commit -a")
```

此时查看一下用户 user2 本地版本库的暂存区，可以看到因为冲突在编号为 1、2、3 的暂存区出现了相同 SHA1 哈希值的对象，但是文件名各不相同。

```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 2       README
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 1       doc/README.txt
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 3       readme.txt
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

其中在暂存区 1 中是改名之前的 doc/README.txt，在暂存区 2 中是用户 user2 改名后的文件名 README，而暂存区 3 是其他用户（user1）改名后的文件 readme.txt。

此时的工作区中存在两个相同的文件 README 和 readme.txt 分别是用户 user2 和 user1 对 doc/README.txt 重命名之后的文件。

```
$ ls -l readme.txt README
-rw-r--r-- 1 jiangxin jiangxin 72 12 月 27 12:25 README
-rw-r--r-- 1 jiangxin jiangxin 72 12 月 27 16:53 readme.txt
```

### 16.5.1 手工操作解决树冲突

这时 user2 应该和 user1 商量一下到底应该将该文件改成什么名字。如果双方最终确认应该采用 user2 重命名的名称，则 user2 应该进行下面的操作完成冲突解决，具体操作过程如下。

（1）删除文件 readme.txt。

在执行 git rm 操作过程时会弹出三条警告，说共有三个文件待合并。

```
$ git rm readme.txt
README: needs merge
doc/README.txt: needs merge
readme.txt: needs merge
rm 'readme.txt'
```

（2）删除文件 doc/README.txt。

执行删除过程，弹出的警告少了一条，因为前面的删除操作已经将一个冲突文件撤出暂存区了。

```
$ git rm doc/README.txt
README: needs merge
doc/README.txt: needs merge
rm 'doc/README.txt'
```

（3）添加文件 README。

```
$ git add README
```

（4）这时查看一下暂存区，会发现所有文件都在暂存区 0 中。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git ls-files -s
100644 463dd451d94832f196096bbc0c9cf9f2d0f82527 0       README
100644 430bd4314705257a53241bc1d2cb2cc30f06f5ea 0       team/user1.txt
100644 a72ca0b4f2b9661d12d2a0c1456649fc074a38e3 0       team/user2.txt
```

（5）提交完成冲突解决。

```
$ git commit -m "fixed tree conflict."
[master e82187e] fixed tree conflict.
```

（6）查看一下最近三次提交日志，看到最新的提交是一个合并提交。

```
$ git log --oneline --graph -3 -m --stat
*   e82187e (from 615c1ff) fixed tree conflict.
|\
| |  README     |    4 ++++
| |  readme.txt |    4 ----
| |  2 files changed, 4 insertions(+), 4 deletions(-)
| * 615c1ff rename doc/README.txt to readme.txt
| |  doc/README.txt |    4 ----
| |  readme.txt     |    4 ++++
| |  2 files changed, 4 insertions(+), 4 deletions(-)
* | 20180eb rename doc/README.txt to README
|/
|    README          |    4 ++++
|    doc/README.txt  |    4 ----
|    2 files changed, 4 insertions(+), 4 deletions(-)
```

## 16.5.2 交互式解决树冲突

树冲突虽然不能像文件冲突那样使用图形工具进行冲突解决，但还是可以使用 git mergetool 命令，通过交互式问答快速解决此类冲突。

首先将 user2 的工作区重置到前一次提交，再执行 git merge 引发树冲突。

（1）重置到前一次提交。

```
$ cd /path/to/user2/workspace/project
$ git reset --hard HEAD^
HEAD is now at 20180eb rename doc/README.txt to README
$ git clean -fd
```

（2）执行 git merge 引发树冲突。

```
$ git merge refs/remotes/origin/master
CONFLICT (rename/rename): Rename "doc/README.txt"->"README" in branch "HEAD"
rename "doc/README.txt"->"readme.txt" in "refs/remotes/origin/master"
Automatic merge failed; fix conflicts and then commit the result.
$ git status -s
AU README
DD doc/README.txt
UA readme.txt
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


上面的操作所引发的树冲突，可以执行 git mergetool 命令进行交互式冲突解决，会如下逐一提示用户进行选择。

（1）执行 git mergetool 命令。忽略其中的提示和警告。

```
$ git mergetool
merge tool candidates: opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff
diffuse ecmerge p4merge araxis emerge vimdiff
Merging:
doc/README.txt
README
readme.txt

mv: 无法获取 "doc/README.txt" 的文件状态 (stat): 没有那个文件或目录
cp: 无法获取 "./doc/README.txt.BACKUP.13869.txt" 的文件状态 (stat): 没有那个文件或目录
mv: 无法将 ".merge_file_I3gfzy" 移动至 "./doc/README.txt.BASE.13869.txt": 没有那个文件或目录
```

（2）询问对文件 doc/README.txt 的处理方式。输入 d 选择将该文件删除。

```
Deleted merge conflict for 'doc/README.txt':
  {local}: deleted
  {remote}: deleted
Use (m)odified or (d)eleted file, or (a)bort? d
```

（3）询问对文件 README 的处理方式。输入 c 选择将该文件保留（创建）。

```
Deleted merge conflict for 'README':
  {local}: created
  {remote}: deleted
Use (c)reated or (d)eleted file, or (a)bort? c
```

（4）询问对文件 readme.txt 的处理方式。输入 d 选择将该文件删除。

```
Deleted merge conflict for 'readme.txt':
  {local}: deleted
  {remote}: created
Use (c)reated or (d)eleted file, or (a)bort? d
```

（5）查看当前状态，只有一些尚未清理的临时文件，而冲突已经解决。

```
$ git status -s
?? .merge_file_I3gfzy
?? README.orig
```

（6）提交完成冲突解决。

```
$ git commit -m "fixed tree conflict."
[master e070bc9] fixed tree conflict.
```

（7）向共享服务器推送。

```
$ git push
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 457 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///path/to/repos/shared.git
   615c1ff..e070bc9  master -> master
```

## 16.6 合并策略

Git 合并操作支持很多合并策略，默认会选择最适合的合并策略。例如，和一个分支进行合并时会选择 recursive 合并策略，当和两个或两个以上的其他分支进行合并时采用 octopus 合并策略。可以通过传递参数使用指定的合并策略，命令行如下：

```
git merge [-s <strategy>] [-X <strategy-option>] <commit>...
```

其中参数 -s 用于设定合并策略，参数 -X 用于为所选的合并策略提供附加的参数。

下面分别介绍不同的合并策略：

（1）resolve

该合并策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略被认为是最安全、最快的合并策略。

（2）recursive

该合并策略只能用于合并两个头（即当前分支和另外的一个分支），使用三向合并策略。这个合并策略是合并两个头指针时的默认合并策略。

当合并的头指针拥有一个以上的祖先的时候，会针对多个公共祖先创建一个合并的树，并以此作为三向合并的参照。这个合并策略被认为可以实现冲突的最小化，而且可以发现和处理由于重命名导致的合并冲突。

这个合并策略可以使用下列选项。

- ours

  在遇到冲突的时候，选择我们的版本（当前分支的版本），而忽略他人的版本。如果他人的改动和本地改动不冲突，会将他人的改动合并进来。

  不要将此模式和后面介绍的单纯的 ours 合并策略相混淆。后面介绍的 ours 合并策略直接丢弃其他分支的变更，无论冲突与否。

- theirs

  和 ours 选项相反，遇到冲突时选择他人的版本，丢弃我们的版本。

- subtree[=path]

  这个选项使用子树合并策略，比下面介绍的 subtree（子树合并）策略的定制能力更强。下面的 subtree 合并策略要对两个树的目录移动进行猜测，而 recursive 合并策略可以通过此参数直接对子树目录进行设置。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（3）octopus

可以合并两个以上的头指针，但是拒绝执行需要手动解决的复杂合并。主要的用途是将多个主题分支合并到一起。这个合并策略是对三个及三个以上的头指针进行合并时的默认合并策略。

（4）ours

可以合并任意数量的头指针，但是合并的结果总是使用当前分支的内容，丢弃其他分支的内容。

（5）subtree

这是一个经过调整的 recursive 策略。当合并树 A 和 B 时，如果 B 和 A 的一个子树相同，B 首先进行调整以匹配 A 的树的结构，以免两棵树在同一级别进行合并。同时也针对两棵树的共同祖先进行调整。

关于子树合并会在第 4 篇的第 24 章“子树合并”中详细介绍。



## 16.7 合并相关的设置

可以通过 `git config` 命令设置与合并相关的配置变量，对合并进行配置。下面是一些常用的设置。

（1）merge.conflictstyle

该配置变量定义冲突文件中冲突的标记风格，有两个可用的风格，默认的“merge”或“diff3”。

默认的“merge”风格使用标准的冲突分界符（<<<<<<< ======= >>>>>>>）对冲突内容进行标识，其中的两个文字块分别是本地的修改和他人的修改。

如果使用“diff3”风格，则会在冲突中出现三个文字块，分别是：<<<<<<< 和 ||||||| 之间的本地更改版本、||||||| 和 ======= 之间的原始（共同祖先）版本和 ======= 和 >>>>>>> 之间的他人更改的版本。例如：

```
User1 hacked.
<<<<<<< HEAD
Hello, user2.
||||||| merged common ancestors
Hello.
=======
Hello, user1.
>>>>>>> a123390b8936882bd53033a582ab540850b6b5fb
User2 hacked.
User2 hacked again.
```

（2）merge.tool

设定执行 `git mergetool` 进行冲突解决时调用的图形化工具。配置变量 `merge.tool` 可以设置为如下内置支持的工具：“kdiff3”、“tkdiff”、“meld”、“xxdiff”、“emerge”、

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



“vimdiff”、“gvimdiff”、“diffuse”、“ecmerge”、“tortoisemerge”、“p4merge”、“araxis” 和“opendiff”。

```
$ git config --global merge.tool kdiff3
```

如果将 merge.tool 设置为其他值，则使用自定义工具进行冲突解决。自定义工具需要通过 mergetool.<tool>.cmd 对自定义工具的命令行进行设置。

（3）mergetool.<tool>.path

如果 git mergetool 支持的冲突解决工具安装在特殊位置，可以使用 mergetool.<tool>.path 对工具 <tool> 的安装位置进行设置。例如：

```
$ git config --global mergetool.kdiff3.path /path/to/kdiff3
```

（4）mergetool.<tool>.cmd

如果所用的冲突解决工具不在内置的工具列表中，还可以使用 mergetool.<tool>.cmd 对自定义工具的命令行进行设置，同时要将 merge.tool 设置为 <tool>。

自定义工具的命令行可以使用 Shell 变量。例如：

```
$ git config --global merge.tool mykdiff3
$ git config --global mergetool.mykdiff3.cmd '/usr/bin/kdiff3
       -L1 "$MERGED (Base)" -L2 "$MERGED (Local)" -L3 "$MERGED (Remote)"
       --auto -o "$MERGED" "$BASE" "$LOCAL" "$REMOTE"'
```

（5）merge.log

是否在合并提交的提交说明中包含合并提交的概要信息。默认为 false。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 17 章 Git 里程碑

里程碑即 Tag，是人为对提交进行的命名。这和 Git 的提交 ID 是否太长无关，使用任何数字版本号无论长短，都没有使用一个直观的表意的字符串来得方便。例如：用里程碑名称“v2.1”对应于软件的 2.1 发布版本就比使用提交 ID 要直观得多。

对于里程碑，实际上我们并不陌生，在第 2 篇的“第 10 章 Git 基本操作”中，就介绍了使用里程碑来对工作进度“留影”纪念，并使用 `git describe` 命令显示里程碑和提交 ID 的组合来代表软件的版本号。本章将详细介绍里程碑的创建、删除和共享，还会介绍里程碑存在的三种不同形式：轻量级里程碑、带注释的里程碑和带签名的里程碑。

接下来的三章，将对一个名为“Hello World”的示例版本库进行研究，这个版本库不需要我们从头建立，可以直接从 Github[①] 上克隆。先使用下面的方法在本地创建一个镜像，用作本地用户的共享版本库。

- 进入本地版本库根目录下。

```
$ mkdir -p /path/to/repos/
$ cd /path/to/repos/
```

- 从 Github 上镜像 `hello-world.git` 版本库。

  如果 Git 是 1.6.0 或更新的版本，可以使用下面的命令建立版本库镜像。

```
$ git clone --mirror git://github.com/ossxp-com/hello-world.git
```

  否则使用下面的命令建立版本库镜像。

```
$ git clone --bare \
  git://github.com/ossxp-com/hello-world.git hello-world.git
```

完成上面的操作后，就在本地建立了一个裸版本库 `/path/to/repos/hello-world.git`。接下来用户 user1 和 user2 分别在各自的工作区克隆这个裸版本库。使用如下命令即可：

```
$ git clone file:///path/to/repos/hello-world.git \
            /path/to/user1/workspace/hello-world
$ git clone file:///path/to/repos/hello-world.git \
            /path/to/user2/workspace/hello-world
$ git --git-dir=/path/to/user1/workspace/hello-world/.git \
      config user.name user1
$ git --git-dir=/path/to/user1/workspace/hello-world/.git \
```

① *https://github.com/ossxp-com/hello-world/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        config user.email user1@sun.ossxp.com
$ git --git-dir=/path/to/user2/workspace/hello-world/.git \
        config user.name user2
$ git --git-dir=/path/to/user2/workspace/hello-world/.git \
        config user.email user2@moon.ossxp.com
```

## 17.1 显示里程碑

里程碑可以使用 git tag 命令来显示，里程碑还可以在其他命令的输出中出现，下面分别对这些命令加以介绍。

### 1. 命令 git tag

不带任何参数执行 git tag 命令，即可显示当前版本库的里程碑列表。

```
$ cd /path/to/user1/workspace/hello-world
$ git tag
jx/v1.0
jx/v1.0-i18n
jx/v1.1
jx/v1.2
jx/v1.3
jx/v2.0
jx/v2.1
jx/v2.2
jx/v2.3
```

里程碑创建的时候可能包含一个说明。在显示里程碑的时候同时显示说明，使用 -n<num> 参数，显示最多 <num> 行里程碑的说明。

```
$ git tag -n1
jx/v1.0         Version 1.0
jx/v1.0-i18n    i18n support for v1.0
jx/v1.1         Version 1.1
jx/v1.2         Version 1.2: allow spaces in username.
jx/v1.3         Version 1.3: Hello world speaks in Chinese now.
jx/v2.0         Version 2.0
jx/v2.1         Version 2.1: fixed typo.
jx/v2.2         Version 2.2: allow spaces in username.
jx/v2.3         Version 2.3: Hello world speaks in Chinese now.
```

还可以使用通配符对输出进行过滤。只显示名称和通配符相符的里程碑。

```
$ git tag -l jx/v2*
jx/v2.0
jx/v2.1
jx/v2.2
jx/v2.3
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 2. 命令 git log

在查看日志时使用参数 --decorate 可以看到提交对应的里程碑及其他引用。

```
$ git log --oneline --decorate
3e6070e (HEAD, tag: jx/v1.0, origin/master, origin/HEAD, master) Show version.
75346b3 Hello world initialized.
```

### 3. 命令 git describe

使用命令 git describe 将提交显示为一个易记的名称。这个易记的名称来自于建立在该提交上的里程碑，若该提交没有里程碑则使用该提交历史版本上的里程碑并加上可理解的寻址信息。

- 如果该提交恰好被打上一个里程碑，则显示该里程碑的名字。

```
$ git describe
jx/v1.0
$ git describe 384f1e0
jx/v2.2
```

- 若提交没有对应的里程碑，但是在其祖先版本上建有里程碑，则使用类似 <tag>-<num>-g<commit> 的格式显示。

  其中 <tag> 是最接近的祖先提交的里程碑名字，<num> 是该里程碑和提交之间的距离，<commit> 是该提交的精简提交 ID。

```
$ git describe 610e78fc95bf2324dc5595fa684e08e1089f5757
jx/v2.2-1-g610e78f
```

- 如果工作区对文件有修改，还可以通过后缀 -dirty 表示出来。

```
$ echo hacked >> README; git describe --dirty; git checkout -- README
jx/v1.0-dirty
```

- 如果提交本身没有包含里程碑，可以通过传递 --always 参数显示精简提交 ID，否则会出错。

```
$ git describe master^ --always
75346b3
```

  命令 git describe 是非常有用的命令，可以将该命令的输出用作软件的版本号。在此之前曾经演示过这个应用，马上还会看到。

### 4. 命令 git name-rev

命令 git name-rev 和 git describe 类似，会显示提交 ID 及其对应的一个引用。默认优先使用分支名，除非使用 --tags 参数。还有一个显著的不同就是，如果提交上没有相对应的引用，则会使用最新提交上的引用名称并加上向后回溯的符号 ~<num>。

- 默认优先显示分支名。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git name-rev HEAD
HEAD master
```

- 使用 --tags 优先使用里程碑。
  之所以在对应的里程碑引用名称后面加上后缀 ^0，是因为该引用指向的是一个 tag 对象而非提交。用 ^0 后缀指向对应的提交。

```
$ git name-rev HEAD --tags
HEAD tags/jx/v1.0^0
```

- 如果提交上没有对应的引用名称，则会使用新提交上的引用名称并加上后缀 ~<num>。后缀的含义是第 <num> 个祖先提交。

```
$ git name-rev --tags 610e78fc95bf2324dc5595fa684e08e1089f5757
610e78fc95bf2324dc5595fa684e08e1089f5757 tags/jx/v2.3~1
```

- 命令 git name-rev 可以对标准输入中的提交 ID 进行改写，使用管道符号对前一个命令的输出进行改写，会显示神奇的效果。

```
$ git log --pretty=oneline origin/helper/master | \
git name-rev --tags --stdin
bb4fef88fee435bfac04b8389cf193d9c04105a6 (tags/jx/v2.3^0) Translate for Chinese.
610e78fc95bf2324dc5595fa684e08e1089f5757 (tags/jx/v2.3~1) Add I18N support.
384f1e0d5106c9c6033311a608b91c69332fe0a8 (tags/jx/v2.2^0) Bugfix: allow spaces in username.
e5e62107f8f8d0a5358c3aff993cf874935bb7fb (tags/jx/v2.1^0) fixed typo: -help to --help
5d7657b2f1a8e595c01c812dd5b2f67ea133f456 (tags/jx/v2.0^0) Parse arguments using getopt_long.
3e6070eb2062746861b20e1e6235fed6f6d15609 (tags/jx/v1.0^0) Show version.
75346b3283da5d8117f3fe66815f8aaaf5387321 (tags/jx/v1.0~1) Hello world initialized.
```

## 17.2 创建里程碑

创建里程碑依然是使用 git tag 命令。创建里程碑的用法有以下几种：

```
用法1：git tag              <tagname> [<commit>]
用法2：git tag -a           <tagname> [<commit>]
用法3：git tag -m <msg>     <tagname> [<commit>]
用法4：git tag -s           <tagname> [<commit>]
用法5：git tag -u <key-id> <tagname> [<commit>]
```

其中：

- 用法 1 是创建轻量级里程碑。
- 用法 2 和用法 3 相同，都是创建带说明的里程碑。其中用法 3 直接通过 -m 参数提供里程碑创建说明。
- 用法 4 和用法 5 相同，都是创建带 GnuPG 签名的里程碑。其中用法 5 用 -u 参数选择指定的私钥进行签名。
- 创建里程碑需要输入里程碑的名字（<tagname>）和一个可选的提交 ID（<commit>）。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


如果没有提供提交ID，则基于头指针HEAD创建里程碑。

### 17.2.1 轻量级里程碑

轻量级里程碑最简单，创建时无须输入描述信息。我们来看看如何创建轻量级里程碑：

（1）先创建一个空提交。

```
$ git commit --allow-empty -m "blank commit."
[master 60a2f4f] blank commit.
```

（2）在刚刚创建的空提交上创建一个轻量级里程碑，名为mytag。

省略了<commit>参数，相当于在HEAD上即最新的空提交上创建里程碑。

```
$ git tag mytag
```

（3）查看里程碑，可以看到该里程碑已经创建。

```
$ git tag -l my*
mytag
```

#### 1. 轻量级里程碑的奥秘

当创建了里程碑mytag后，会在版本库的.git/refs/tags目录下创建一个新文件。

- 查看一下这个引用文件的内容，会发现是一个40位的SHA1哈希值。

```
$ cat .git/refs/tags/mytag
60a2f4f31e5dddd777c6ad37388fe6e5520734cb
```

- 用git cat-file命令检查轻量级里程碑指向的对象。轻量级里程碑实际上指向的是一个提交。

```
$ git cat-file -t mytag
commit
```

- 查看该提交的内容，发现就是刚刚进行的空提交。

```
$ git cat-file -p mytag
tree 1d902fedc4eb732f17e50f111dcecb638f10313e
parent 3e6070eb2062746861b20e1e6235fed6f6d15609
author user1 <user1@sun.ossxp.com> 1293790794 +0800
committer user1 <user1@sun.ossxp.com> 1293790794 +0800

blank commit.
```

#### 2. 轻量级里程碑的缺点

轻量级里程碑的创建过程没有记录，因此无法知道是谁创建的里程碑，何时创建的里程碑。在团队协同开发时，尽量不要采用此种偷懒的方式创建里程碑，而是采用后两种方式。

还有git describe命令默认不使用轻量级里程碑生成版本描述字符串。

- 执行git describe命令，发现生成的版本描述字符串，使用的是前一个版本上的

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


里程碑名称。

```
$ git describe
jx/v1.0-1-g60a2f4f
```

- 使用 --tags 参数，也可以将轻量级里程碑用作版本描述符。

```
$ git describe --tags
mytag
```

### 17.2.2 带说明的里程碑

带说明的里程碑，就是使用参数 -a 或 -m <msg> 调用 git tag 命令，在创建里程碑的时候提供一个关于该里程碑的说明。下面来看看如何创建带说明的里程碑：

（1）还是先创建一个空提交。

```
$ git commit --allow-empty -m "blank commit for annotated tag test."
[master 8a9f3d1] blank commit for annotated tag test.
```

（2）在刚刚创建的空提交上创建一个带说明的里程碑，名为 mytag2。

下面的命令使用了 -m <msg> 参数，在命令行给出了新建里程碑的说明。

```
$ git tag -m "My first annotated tag." mytag2
```

（3）查看里程碑，可以看到该里程碑已经创建。

```
$ git tag -l my* -n1
mytag           blank commit.
mytag2          My first annotated tag.
```

下面来看看带说明里程碑的奥秘。当创建了带说明的里程碑 mytag2 后，会在版本库的 .git/refs/tags 目录下创建一个新的引用文件。

- 查看一下这个引用文件的内容：

```
$ cat .git/refs/tags/mytag2
149b6344e80fc190bda5621cd71df391d3dd465e
```

- 用 git cat-file 命令检查该里程碑（带说明的里程碑）指向的对象，会发现指向的不再是一个提交，而是一个 tag 对象。

```
$ git cat-file -t mytag2
tag
```

- 查看该提交的内容，会发现 mytag2 对象的内容不是之前我们熟悉的提交对象的内容，而是包含了创建里程碑时的说明，以及对应的提交 ID 等信息。

```
$ git cat-file -p mytag2
object 8a9f3d16ce2b4d39b5d694de10311207f289153f
type commit
tag mytag2
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
tagger user1 <user1@sun.ossxp.com> Sun Jan 2 14:10:07 2011 +0800

My first annotated tag.
```

由此可见使用带说明的里程碑，会在版本库中建立一个新的对象（tag 对象），这个对象会记录创建里程碑的用户（tagger），创建里程碑的时间，以及为什么要创建里程碑。这就避免了轻量级里程碑因为匿名创建而无法追踪的缺点。

带说明的里程碑是一个 tag 对象，在版本库中以一个对象的方式存在，并用一个 40 位的 SHA1 哈希值来表示。这个哈希值的生成方法和前面介绍的 commit 对象、tree 对象、blob 对象一样。至此，Git 对象库的四类对象我们就都已经研究到了。

```
$ git cat-file tag mytag2 | wc -c
148
$ (printf "tag 148\000"; git cat-file tag mytag2) | sha1sum
149b6344e80fc190bda5621cd71df391d3dd465e  -
```

虽然 mytag2 本身是一个 tag 对象，但在很多 Git 命令中，可以直接将其视为一个提交。下面的 `git log` 命令，显示 mytag2 指向的提交日志。

```
$ git log -1 --pretty=oneline mytag2
8a9f3d16ce2b4d39b5d694de10311207f289153f blank commit for annotated tag test.
```

有时，需要得到里程碑指向的提交对象的 SHA1 哈希值。

- 直接用 `git rev-parse` 命令查看 mytag2 得到的是 tag 对象的 ID，并非提交对象的 ID。

```
$ git rev-parse mytag2
149b6344e80fc190bda5621cd71df391d3dd465e
```

- 使用下面几种不同的表示法，则可以获得 mytag2 对象所指向的提交对象的 ID。

```
$ git rev-parse mytag2^{commit}
8a9f3d16ce2b4d39b5d694de10311207f289153f
$ git rev-parse mytag2^{}
8a9f3d16ce2b4d39b5d694de10311207f289153f
$ git rev-parse mytag2^0
8a9f3d16ce2b4d39b5d694de10311207f289153f
$ git rev-parse mytag2~0
8a9f3d16ce2b4d39b5d694de10311207f289153f
```

### 17.2.3 带签名的里程碑

带签名的里程碑和上面介绍的带说明的里程碑本质上是一样的，都是在创建里程碑的时候在 Git 对象库中生成一个 tag 对象，只不过带签名的里程碑多做了一个工作：为里程碑对象添加 GnuPG 签名。

创建带签名的里程碑也非常简单，使用参数 `-s` 或 `-u <key-id>` 即可。还可以使用 `-m <msg>` 参数直接在命令行中提供里程碑的描述。创建带签名的里程碑的一个前提是需要安装

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


GnuPG，并且建立相应的公钥/私钥对。

GnuPG 可以在各个平台上安装。

❑ 在 Linux 如 Debian/Ubuntu 上安装，执行：

```
$ sudo aptitude install gnupg
```

❑ 在 Mac OS X 上可以通过 Homebrew 安装：

```
$ brew install gnupg
```

❑ 在 Windows 上可以通过 cygwin 安装 gnupg。

为了演示创建带签名的里程碑，还是事先创建一个空提交：

```
$ git commit --allow-empty -m "blank commit for GnuPG-signed tag test."
[master ebcf6d6] blank commit for GnuPG-signed tag test.
```

直接在刚刚创建的空提交上创建一个带签名的里程碑 mytag3 很可能会失败：

```
$ git tag -s -m "My first GPG-signed tag." mytag3
gpg: "user1 <user1@sun.ossxp.com>" 已跳过：私钥不可用
gpg: signing failed: 私钥不可用
error: gpg failed to sign the tag
error: unable to sign the tag
```

之所以签名失败，是因为找不到签名可用的公钥/私钥对。使用下面的命令可以查看当前可用的 GnuPG 公钥。

```
$ gpg --list-keys
/home/jiangxin/.gnupg/pubring.gpg
---------------------------------
pub   1024D/FBC49D01 2006-12-21 [有效至：2016-12-18]
uid                  Jiang Xin <worldhello.net@gmail.com>
uid                  Jiang Xin <jiangxin@ossxp.com>
sub   2048g/448713EB 2006-12-21 [有效至：2016-12-18]
```

可以看到 GnuPG 的公钥链（pubring）中只包含了 Jiang Xin 用户的公钥，尚没有 uesr1 用户的公钥。

实际上在创建带签名的里程碑时，并非一定要使用签名者本人的公钥/私钥对进行签名，使用 -u <key-id> 参数调用就可以用指定的公钥/私钥对进行签名，对于此例可以使用 FBC49D01 作为 <key-id>。但如果没有可用的公钥/私钥对，或者希望使用提交者本人的公钥/私钥对进行签名，就需要为提交者：user1 <user1@sun.ossxp.com> 创建对应的公钥/私钥对。

使用命令 gpg --gen-key 来创建公钥/私钥对。

```
$ gpg --gen-key
```

按照提示一步一步操作即可。需要注意的有：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 在创建公钥 / 私钥对时，会提示输入用户名，输入 User1，提示输入邮件地址，输入 user1@sun.ossxp.com，其他可以采用默认值。
- 在提示输入密码时，为了简单起见可以直接按下回车，即使用空口令。
- 在生成公钥 / 私钥对过程中，会提示用户做一些随机操作以便产生更好的随机数，这时不停的晃动鼠标就可以了。

创建完毕，再查看一下公钥链。

```
$ gpg --list-keys
/home/jiangxin/.gnupg/pubring.gpg
---------------------------------
pub   1024D/FBC49D01 2006-12-21 [有效至：2016-12-18]
uid                  Jiang Xin <worldhello.net@gmail.com>
uid                  Jiang Xin <jiangxin@ossxp.com>
sub   2048g/448713EB 2006-12-21 [有效至：2016-12-18]

pub   2048R/37379C67 2011-01-02
uid                  User1 <user1@sun.ossxp.com>
sub   2048R/2FCFB3E2 2011-01-02
```



很显然用户 user1 的公钥 / 私钥对已经建立。现在就可以直接使用 -s 参数来创建带签名的里程碑了。

```
$ git tag -s -m "My first GPG-signed tag." mytag3
```

查看里程碑，可以看到该里程碑已经创建。

```
$ git tag -l my* -n1
mytag           blank commit.
mytag2          My first annotated tag.
mytag3          My first GPG-signed tag.
```

和带说明的里程碑一样，在 Git 对象库中也建立了一个 tag 对象。查看该 tag 对象可以看到其中包含了 GnuPG 签名。

```
$ git cat-file tag mytag3
object ebcf6d6b06545331df156687ca2940800a3c599d
type commit
tag mytag3
tagger user1 <user1@sun.ossxp.com> 1293960936 +0800

My first GPG-signed tag.
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

iQEcBAABAgAGBQJNIEboAAoJEO9W1fg3N5xn42gH/jFDEKobqlupNKFvmkI1t9d6
lApDFUdcFMPWvxo/eq8VjcQyRcb1X1bGJj+pxXk455fDL1NWonaJa6HE6RLu868x
CQIWqWelkCelfm05GE9FnPd2SmJsiDkTPZzINya1Hyl F5ZbrExH506JyCFk//FC2
8zRApSbrsj3yAWMStW0fGqHKLuYq+sdepzGnnFnhhzkJhusMHUkTIfpLwaprhMsm
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
1IIxKNm9i0Zf/tzq4a/R0N8NiFHl/9M95iV200I9PuuRWedV0tEPS6Onax2yT3JE
I/w9gtIBOeb5uAz2Xrt5AUwt9JJTk5mmv2HBqWCq5wefxs/ub26iPmef35PwAgA=
=jdrN
-----END PGP SIGNATURE-----
```

要验证签名的有效性，如果直接使用 gpg 命令会比较麻烦，因为需要将这个文件拆分为两个，一个是不包含签名的里程碑内容，另外一个是签名本身。还好可以使用命令 `git tag -v` 来验证里程碑签名的有效性。

```
$ git tag -v mytag3
object ebcf6d6b06545331df156687ca2940800a3c599d
type commit
tag mytag3
tagger user1 <user1@sun.ossxp.com> 1293960936 +0800

My first GPG-signed tag.
gpg: 于 2011 年 01 月 02 日 星期日 17 时 35 分 36 秒 CST 创建的签名，使用 RSA，钥匙号 37379C67
```

## 17.3 删除里程碑

如果里程碑建立在了错误的提交上，或者对里程碑的命名不满意，可以删除里程碑。删除里程碑使用命令 `git tag -d`，下面用命令删除里程碑 mytag。

```
$ git tag -d mytag
Deleted tag 'mytag' (was 60a2f4f)
```

里程碑没有类似 reflog 的变更记录机制，一旦删除不易恢复，慎用。在删除里程碑 mytag 的命令输出中，会显示该里程碑所对应的提交 ID，一旦发现删除错误，赶紧补救还来得及。下面的命令实现对里程碑 mytag 的重建。

```
$ git tag mytag 60a2f4f
```

Git 没有提供对里程碑重命名的命令，如果对里程碑名字不满意的话，可以删除旧的里程碑，然后重新用新的名称创建里程碑。

为什么没有提供重命名里程碑的命令呢？按理说只要将 `.git/refs/tags/` 下的引用文件改名就可以了。这是因为里程碑的名字不但反映在 `.git/refs/tags` 引用目录下的文件名，而且对于带说明或签名的里程碑，里程碑的名字还反映在 tag 对象的内容中。尤其是带签名的里程碑，如果修改里程碑的名字，不但里程碑对象内容势必要变化，而且里程碑也要重新进行签名，这显然难以自动实现。

在第 6 篇第 35 章的“35.4 Git 版本库整理”一节中会介绍使用 `git filter-branch` 命令实现对里程碑自动重命名的方法，但是那个方法也不能毫发无损地实现对签名里程碑的重命名，被重命名的签名里程碑中的签名会被去除，从而成为带说明的里程碑。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 17.4 不要随意更改里程碑

里程碑建立后，如果需要修改，可以使用同样的里程碑名称重新建立，不过需要加上 -f 或 --force 参数强制覆盖已有的里程碑。

更改里程碑要慎重，一个原因是里程碑从概念上讲是对历史提交的一个标记，不应该随意变动。另外一个原因是里程碑一旦被他人同步，如果修改里程碑，已经同步该里程碑的用户并不会自动更新，这就导致一个相同名称的里程碑在不同用户的版本库中的指向不同。下面就看看如何与他人共享里程碑。

## 17.5 共享里程碑

现在看看用户 user1 的工作区状态。可以看出现在的工作区相比上游有三个新的提交。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 3 commits.
#
nothing to commit (working directory clean)
```

那么如果执行 git push 命令向上游推送，会将本地创建的三个里程碑推送到上游吗？通过下面的操作来试一试。

- 向上游推送。

```
$ git push
Counting objects: 3, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 512 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To file:///path/to/repos/hello-world.git
   3e6070e..ebcf6d6  master -> master
```

- 通过执行 git ls-remote 可以查看上游版本库的引用，会发现本地建立的三个里程碑，并没有推送到上游。

```
$ git ls-remote origin my*
```

创建的里程碑，默认只在本地版本库中可见，不会因为对分支执行推送而将里程碑也推送到远程版本库。这样的设计显然更为合理，否则的话，每个用户本地创建的里程碑都自动向上游推送，那么上游的里程碑将有多么杂乱，而且不同用户创建的相同名称的里程碑会互相覆盖。

### 1. 显式推送以共享里程碑

如果用户确实需要将某些本地建立的里程碑推送到远程版本库，需要在 git push 命令

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


中明确地表示出来。下面在用户user1的工作区执行命令，将mytag里程碑共享到上游版本库。

```
$ git push origin mytag
Total 0 (delta 0), reused 0 (delta 0)
To file:///path/to/repos/hello-world.git
 * [new tag]         mytag -> mytag
```

如果需要将本地建立的所有里程碑全部推送到远程版本库，可以使用通配符。

```
$ git push origin refs/tags/*
Counting objects: 2, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 687 bytes, done.
Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
To file:///path/to/repos/hello-world.git
 * [new tag]         mytag2 -> mytag2
 * [new tag]         mytag3 -> mytag3
```

再用命令git ls-remote查看上游版本库的引用，会发现本地建立的三个里程碑，已经能够在上游中看到了。

```
$ git ls-remote origin my*
60a2f4f31e5dddd777c6ad37388fe6e5520734cb        refs/tags/mytag
149b6344e80fc190bda5621cd71df391d3dd465e        refs/tags/mytag2
8a9f3d16ce2b4d39b5d694de10311207f289153f        refs/tags/mytag2^{}
5dc2fc52f2dcb84987f511481cc6b71ec1b381f7        refs/tags/mytag3
ebcf6d6b06545331df156687ca2940800a3c599d        refs/tags/mytag3^{}
```

**2. 用户从版本库执行拉回操作，会自动获取里程碑么？**

用户user2的工作区中如果执行git fetch或git pull操作，能自动将用户user1推送到共享版本库中的里程碑获取到本地版本库么？下面实践一下。

（1）进入user2的工作区。

```
$ cd /path/to/user2/workspace/hello-world/
```

（2）执行git pull命令，从上游版本库获取提交。

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From file:///path/to/repos/hello-world
   3e6070e..ebcf6d6  master     -> origin/master
 * [new tag]         mytag3     -> mytag3
From file:///path/to/repos/hello-world
 * [new tag]         mytag      -> mytag
 * [new tag]         mytag2     -> mytag2
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Updating 3e6070e..ebcf6d6
Fast-forward
```

（3）可见执行 git pull 操作，能够在获取远程共享版本库的提交的同时，获取新的里程碑。下面的命令可以看到本地版本库中的里程碑。

```
$ git tag -n1 -l my*
mytag           blank commit.
mytag2          My first annotated tag.
mytag3          My first GPG-signed tag.
```

### 3. 里程碑变更能够自动同步吗？

里程碑可以被强制更新。当里程碑被改变后，已经获取到里程碑的版本库再次使用获取或拉回操作，能够自动更新里程碑吗？答案是不能。可以看看下面的操作。

（1）用户 user2 强制更新里程碑 mytag2。

```
$ git tag -f -m "user2 update this annotated tag." mytag2 HEAD^
Updated tag 'mytag2' (was 149b634)
```

（2）里程碑 mytag2 已经是不同的对象了。

```
$ git rev-parse mytag2
0e6c780ff0fe06635394db9dac6fb494833df8df
$ git cat-file -p mytag2
object 8a9f3d16ce2b4d39b5d694de10311207f289153f
type commit
tag mytag2
tagger user2 <user2@moon.ossxp.com> Mon Jan 3 01:14:18 2011 +0800

user2 update this annotated tag.
```

（3）为了更改远程共享服务器中的里程碑，同样需要显式推送。即在推送时写上要推送的里程碑名称。

```
$ git push origin mytag2
Counting objects: 1, done.
Writing objects: 100% (1/1), 171 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To file:///path/to/repos/hello-world.git
   149b634..0e6c780  mytag2 -> mytag2
```

（4）切换到另外一个用户 user1 的工作区。

```
$ cd /path/to/user1/workspace/hello-world/
```

（5）用户 user1 执行拉回操作，没有获取到新的里程碑。

```
$ git pull
Already up-to-date.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（6）用户user1必须显式地执行拉回操作。即要在git pull的参数中使用引用表达式。

所谓引用表达式就是用冒号分隔的引用名称或通配符。用在这里代表用远程共享版本库的引用refs/tag/mytag2覆盖本地版本库的同名引用。

```
$ git pull origin refs/tags/mytag2:refs/tags/mytag2
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From file:///path/to/repos/hello-world
 - [tag update]      mytag2     -> mytag2
Already up-to-date.
```

关于里程碑的共享和同步操作，看似很烦琐，但用心体会就会感觉到Git关于里程碑共享的设计是非常合理和人性化的：

- 里程碑共享，必须显式的推送。即在推送命令的参数中，标明要推送哪个里程碑。显式推送是防止用户随意推送里程碑导致共享版本库中里程碑泛滥的方法。当然还可以参考第5篇“第30章 Gitolite服务架设”的相关章节为共享版本库添加授权，只允许部分用户向服务器推送里程碑。
- 执行获取或拉回操作，自动从远程版本库获取新里程碑，并在本地版本库中创建。获取或拉回操作，只会将获取的远程分支所包含的新里程碑同步到本地，而不会将远程版本库的其他分支中的里程碑获取到本地。这既方便了里程碑的取得，又防止本地里程碑因同步远程版本库而泛滥。
- 如果本地已有同名的里程碑，默认不会从上游同步里程碑，即使两者里程碑的指向是不同的。理解这一点非常重要。这也就要求里程碑一旦共享，就不要再修改。

## 17.6 删除远程版本库的里程碑

假如向远程版本库推送里程碑后，忽然发现里程碑创建在了错误的提交上，为了防止其他人获取到错误的里程碑，应该尽快将里程碑删除。

删除本地里程碑非常简单，使用git tag -d <tagname>就可以了，但是如何撤销已经推送到远程版本库的里程碑呢？需要登录到服务器上吗？或者需要麻烦管理员吗？不必！可以直接在本地版本库执行命令删除远程版本库中的里程碑。

使用git push命令可以删除远程版本库中的里程碑。用法如下：

```
命令：git push <remote_url>  :<tagname>
```

该命令的最后一个参数实际上是一个引用表达式，引用表达式一般的格式为<ref>:<ref>。该推送命令使用的引用表达式冒号前的引用被省略，其含义是将一个空值推送到远程版本库对应的引用中，亦即删除远程版本库中相关的引用。这个命令不但可以用于删除里程碑，在下一章还可以用它删除远程版本库中的分支。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


下面演示在用户 user1 的工作区执行下面的命令删除远程共享版本库中的里程碑 mytag2。

（1）切换到用户 user1 工作区。

```
$ cd /path/to/user1/workspace/hello-world
```

（2）执行推送操作删除远程共享版本库中的里程碑。

```
$ git push origin :mytag2
To file:///path/to/repos/hello-world.git
 - [deleted]         mytag2
```

（3）查看远程共享库中的里程碑，发现 mytag2 的确已经被删除。

```
$ git ls-remote origin my*
60a2f4f31e5dddd777c6ad37388fe6e5520734cb        refs/tags/mytag
5dc2fc52f2dcb84987f511481cc6b71ec1b381f7        refs/tags/mytag3
ebcf6d6b06545331df156687ca2940800a3c599d        refs/tags/mytag3^{}
```

## 17.7 里程碑命名规范

在正式项目的版本库管理中，要为里程碑创建订立一些规则，诸如：

- 对创建里程碑进行权限控制，参考后面 Git 服务器架设的相关章节。
- 不要使用轻量级里程碑（只用于本地临时性里程碑），而是要使用带说明的里程碑，甚至要求必须使用带签名的里程碑。
- 如果使用带签名的里程碑，可以考虑设置专用账户，使用专用的私钥创建签名。
- 里程碑的命名要使用统一的风格，并很容易和最终产品显示的版本号相对应。

Git 的里程碑命名还有一些特殊的约定需要遵守。实际上，下面的这些约定对于下一章要介绍的分支及任何其他引用均适用：

- 不要以符号“-”开头。以免在命令行中被当成命令的选项。
- 可以包含路径分隔符“/”，但是路径分隔符不能位于最后。
  使用路径分隔符创建 tag 实际上会在引用目录下创建子目录。例如名为 demo/v1.2.1 的里程碑，就会创建目录 .git/refs/tags/demo 并在该目录下创建引用文件 v1.2.1。
- 不能出现两个连续的点“..”。因为两个连续的点被用于表示版本范围，当然更不能使用三个连续的点。
- 如果在里程碑命名中使用了路径分隔符“/”，就不能在任何一个分隔路径中以点“.”开头。这是因为里程碑在用简写格式表达时，可能造成以一个点“.”开头。这样的引用名称在用作版本范围的最后一个版本时，本来两点操作符变成了三点操作符，从而造成歧义。
- 不能在里程碑名称的最后出现点“.”。否则作为第一个参数出现在表示版本范围的表达式中时，本来版本范围表达式可能用的是两点操作符，结果被误作三点操作符。
- 不能使用特殊字符，如：空格、波浪线“~”、脱字符“^”、冒号“:”、问号“?”、星

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


号“*”、方括号“[”，以及字符 \177（删除字符）或小于 \040（32）的 Ascii 码都不能使用。

这是因为波浪线“~”和 脱字符“^”都用于表示一个提交的祖先提交。冒号被用作引用表达式来分隔两个不同的引用，或者用于分隔引用代表的树对象和该目录树中的文件。问号、星号和方括号在引用表达式中都被用作通配符。

- 不能以“.lock”为结尾。因为以“.lock”结尾的文件是里程碑操作过程中的临时文件。
- 不能包含“@{”字串。否则易和 reflog 的“@{<num>}”语法相混淆。
- 不能包含反斜线“\”。因为反斜线用于命令行或 shell 脚本会造成意外。

Git 还专门为检查引用名称是否符合规范提供了一个命令：git check-ref-format。若该命令返回值为 0，则引用名称符合规范，若返回值为 1，则不符合规范。

```
$ git check-ref-format refs/tags/.name || echo "返回 $?，不合法的引用"
返回 1，不合法的引用
```

### 1. Linux 中的里程碑

Linux 内核项目无疑是使用 Git 版本库时间最久远，也是最重量级的项目。研究 Linux 内核项目本身的里程碑命名和管理，无疑会为自己的项目提供借鉴。

（1）首先看看 Linux 中的里程碑命名。可以看到里程碑都是以字母 v 开头。

```
$ git ls-remote --tags \
  git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-2.6-stable.git \
  v2.6.36*
25427f38d3b791d986812cb81c68df38e8249ef8        refs/tags/v2.6.36
f6f94e2ab1b33f0082ac22d71f66385a60d8157f        refs/tags/v2.6.36^{}
8ed88d401f908a594cd74a4f2513b0fabd32b699        refs/tags/v2.6.36-rc1
da5cabf80e2433131bf0ed8993abc0f7ea618c73        refs/tags/v2.6.36-rc1^{}
...
7619e63f48822b2c68d0e108677340573873fb93        refs/tags/v2.6.36-rc8
cd07202cc8262e1669edff0d97715f3dd9260917        refs/tags/v2.6.36-rc8^{}
9d389cb6dcae347cfcdadf2a1ec5e66fc7a667ea        refs/tags/v2.6.36.1
bf6ef02e53e18dd14798537e530e00b80435ee86        refs/tags/v2.6.36.1^{}
ee7b38c91f3d718ea4035a331c24a56553e90960        refs/tags/v2.6.36.2
a1346c99fc89f2b3d35c7d7e2e4aef8ea4124342        refs/tags/v2.6.36.2^{}
```

（2）以 -rc<num> 为后缀的是先于正式版发布的预发布版本。

可以看出这个里程碑是一个带签名的里程碑。关于此里程碑的说明也是再简练不过了。

```
$ git show v2.6.36-rc1
tag v2.6.36-rc1
Tagger: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sun Aug 15 17:42:10 2010 -0700

Linux 2.6.36-rc1
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
iEYEABECAAYFAkxoiWgACgkQF3YsRnbiHLtYKQCfQSIVcj2hvLj6IWgP9xK2FE7T
bPoAniJ1CjbwLxQBudRi71FvubqPLuVC
=iuls
-----END PGP SIGNATURE-----

commit da5cabf80e2433131bf0ed8993abc0f7ea618c73
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sun Aug 15 17:41:37 2010 -0700

    Linux 2.6.36-rc1

diff --git a/Makefile b/Makefile
index 788111d..f3bdff8 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
-SUBLEVEL = 35
-EXTRAVERSION =
+SUBLEVEL = 36
+EXTRAVERSION = -rc1
 NAME = Sheep on Meth

 # *DOCUMENTATION*
```

（3）正式发布版去掉了预发布版的后缀。

```
$ git show v2.6.36
tag v2.6.36
Tagger: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Wed Oct 20 13:31:18 2010 -0700

Linux 2.6.36

The latest and greatest, and totally bug-free.  At least until 2.6.37
comes along and shoves it under a speeding train like some kind of a
bully.
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

iEYEABECAAYFAky/UcwACgkQF3YsRnbiHLvg/ACffKjAb1fD6fpqcHbSijHHpbP3
4SkAnR4xOy7iKhmfS50ZrVsOkFFTuBHG
=JD3z
-----END PGP SIGNATURE-----

commit f6f94e2ab1b33f0082ac22d71f66385a60d8157f
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Wed Oct 20 13:30:22 2010 -0700
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



```
    Linux 2.6.36

diff --git a/Makefile b/Makefile
index 7583116..860c26a 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 36
-EXTRAVERSION = -rc8
+EXTRAVERSION =
 NAME = Flesh-Eating Bats with Fangs

 # *DOCUMENTATION*
```

（4）正式发布后的升级/修正版本是通过最后一位数字的变动来体现的。

```
$ git show v2.6.36.1
tag v2.6.36.1
Tagger: Greg Kroah-Hartman <gregkh@suse.de>
Date:   Mon Nov 22 11:04:17 2010 -0800

This is the 2.6.36.1 stable release
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2.0.15 (GNU/Linux)

iEYEABECAAYFAkzqvrIACgkQMUfUDdst+ym9VQCgmE1LK2eC/LE9HkscsxL1X62P
8F0AnRI28EHENLXC+FBPt+AFWoT9f1N8
=BX5O
-----END PGP SIGNATURE-----

commit bf6ef02e53e18dd14798537e530e00b80435ee86
Author: Greg Kroah-Hartman <gregkh@suse.de>
Date:   Mon Nov 22 11:03:49 2010 -0800

    Linux 2.6.36.1

diff --git a/Makefile b/Makefile
index 860c26a..dafd22a 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
 SUBLEVEL = 36
-EXTRAVERSION =
+EXTRAVERSION = .1
 NAME = Flesh-Eating Bats with Fangs
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
# *DOCUMENTATION*
```

### 2. Android 项目

看看其他项目的里程碑命名，会发现不同项目关于里程碑的命名各不相同。但是对于同一个项目要在里程碑命名上遵照同一标准，并能够和软件版本号正确地对应。

Android 项目是一个非常有特色的使用 Git 版本库的项目，在后面会用两章介绍 Android 项目为 Git 带来的两个新工具。看看 Android 项目的里程碑编号对自己版本库的管理有无启发。

（1）看看 Android 项目中的里程碑命名，会发现其里程碑的命名格式为 android-< 大版本号 >_r< 小版本号 > 。

```
$ git ls-remote --tags \
  git://android.git.kernel.org/platform/manifest.git \
  android-2.2*
6a03ae8f564130cbb4a11acfc49bd705df7c8df6        refs/tags/android-2.2.1_r1
599e242dea48f84e2f26054b0d1721e489043440        refs/tags/android-2.2.1_r1^{}
656ba6fdbd243153af6ec31017de38641060bf1e        refs/tags/android-2.2_r1
27cd0e346d1f3420c5747e01d2cb35e9ffd025ea        refs/tags/android-2.2_r1^{}
f6b7c499be268f1613d8cd70f2a05c12e01bcb93        refs/tags/android-2.2_r1.1
bd3e9923773006a0a5f782e1f21413034096c4b1        refs/tags/android-2.2_r1.1^{}
03618e01ec9bdd06fd8fe9afdbdcbaf4b84092c5        refs/tags/android-2.2_r1.2
ba7111e1d6fd26ab150bafa029fd5eab8196dad1        refs/tags/android-2.2_r1.2^{}
e03485e978ce1662a1285837f37ed39eadaedb1d        refs/tags/android-2.2_r1.3
7386d2d07956be6e4f49a7e83eafb12215e835d7        refs/tags/android-2.2_r1.3^{}
```

（2）里程碑的创建过程中使用了专用账号和 GnuPG 签名。

```
$ git show android-2.2_r1
tag android-2.2_r1
Tagger: The Android Open Source Project <initial-contribution@android.com>
Date:   Tue Jun 29 11:28:52 2010 -0700

Android 2.2 release 1
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBMKjtm6K0/gZqxDngRAlBUAJ9QwgFbUL592FgRZLTLLbzhKsSQ8ACffQu5
Mjxg5X9oc+7N1DfdU+pmOcI=
=0NG0
-----END PGP SIGNATURE-----

commit 27cd0e346d1f3420c5747e01d2cb35e9ffd025ea
Author: The Android Open Source Project <initial-contribution@android.com>
Date:   Tue Jun 29 11:27:23 2010 -0700

    Manifest for android-2.2_r1

diff --git a/default.xml b/default.xml
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
index 4f21453..aaa26e3 100644
--- a/default.xml
+++ b/default.xml
@@ -3,7 +3,7 @@
   <remote  name="korg"
            fetch="git://android.git.kernel.org/"
            review="review.source.android.com" />
-  <default revision="froyo"
+  <default revision="refs/tags/android-2.2_r1"
            remote="korg" />
...
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 18 章 Git 分支

分支是我们的老朋友了，第 6 章、第 7 章和第 8 章就已经从实现原理上介绍了分支。您想必已经知道了分支 master 的存在方式无非就是在目录 .git/refs/heads 下的文件（或称引用）而已。也看到了分支 master 的指向如何随着提交而变化，如何通过 git reset 命令而重置，以及如何使用 git checkout 命令而检出。

之前的章节都只用到了一个分支：master 分支，而在本章会接触到多个分支。会从应用的角度上介绍分支的几种不同类型：发布分支、特性分支和卖主分支。在本章可以学习到如何对多分支进行操作，如何创建分支，如何切换到其他分支，以及分支之间的合并、变基等。

## 18.1 代码管理之殇

分支是代码管理的利器。如果没有有效的分支管理，代码管理就适应不了复杂的开发过程和项目的需要。在实际的项目实践中，单一分支的单线开发模式远远不够，因为：

- 成功的软件项目大多要经过多个开发周期，发布多个软件版本。每个已经发布的版本都可能发现 Bug，这就需要对历史版本进行更改。
- 有前瞻性的项目管理，新版本的开发往往是和当前版本同步进行的。如果两个版本的开发都混杂在 master 分支中，肯定会是一场灾难。
- 如果产品要针对不同的客户定制，肯定是希望客户越多越好。如果所有的客户定制都混杂在一个分支中，必定会带来混乱。如果使用多个分支管理不同的定制，但若是管理不善，分支之间定制功能的迁移就会成为头痛的问题。
- 即便是所有成员都在为同一个项目的同一个版本进行工作，每个人领受任务却不尽相同，有的任务开发周期会很长，有的任务需要对软件架构进行较大的修改，如果所有人都工作在同一分支中，就会因为过多过频的冲突导致效率低下。
- 敏捷开发（不管是极限编程 XP 还是 Scrum 或其他）是最有效的项目管理模式，其最有效的一个实践就是快速迭代、每晚编译。如果不能将项目的各个功能模块的开发通过分支进行隔离，在软件集成上就会遭遇困难。

### 18.1.1 发布分支

在 2006 年我接触到一个项目团队，使用 Subversion 做版本控制。最为困扰项目经理的是刚刚修正产品的一个 Bug，马上又会接二连三地发现新的 Bug。在访谈开发人员，询问开发人员是如何修正 Bug 的时候，开发人员的回答让我大吃一惊：“当发现产品出现 Bug 的时

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


候，我要中断当前的工作，把我正在开发的新功能的代码注释掉，然后再去修改 Bug，修改好就生成一个 war 包（Java 开发网站项目）给运维部门，扔到网站上去。”

于是我就画了下面的一个图（图 18-1），大致描述了这个团队进行 Bug 修正的过程，从中可以很容易地看出问题的端倪。这个图对于 Git 甚至其他版本库控制系统同样适用。

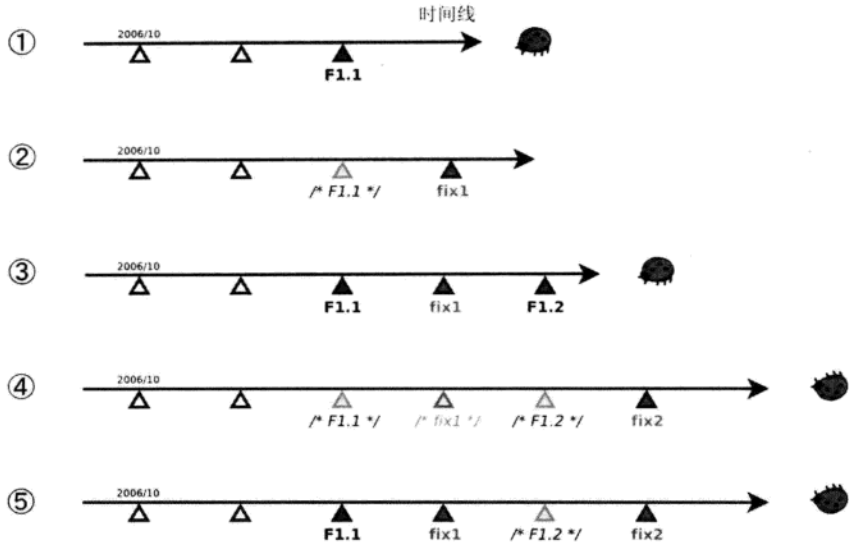



图 18-1 为什么 Bug 没完没了

说明：

- 图 18-1 中的图示①，开发者针对功能 1 做了一个提交，编号“F1.1”。这时客户报告已发布的产品出现了 Bug。
- 于是开发者匆忙地干了起来，图示②显示了该开发者修正 Bug 的过程：将新提交的针对功能 1 的代码“F1.1”注释掉，然后提交一个修正 Bug 的提交（编号：fix1）。
- 开发者编译出已发布软件的修订版交给客户，接着开始功能 1 的开发。图示③显示了开发者针对功能 1 做出了一个新的提交“F1.2”。
- 客户再次在已发布版本中发现一个 Bug。开发者再次开始 Bug 修正工作。
- 图示④和图示⑤显示了此工作模式下非常容易在修复一个 Bug 的时候引入新的 Bug。
- 图示④的问题在于开发者注释功能 1 的代码时，不小心将“fix1”的代码也注释掉了，导致曾经修复的 Bug 在已发布软件的修订版中重现。
- 图示⑤的问题在于开发者没有将功能 1 的代码剔出干净，导致在已发布产品的修订版本中引入了不完整和不需要的功能代码。用户可能看到一个新的但是不能使用的菜单项，甚至更糟。

使用版本控制系统的分支功能，可以避免对已发布的软件版本进行 Bug 修正时引入新功

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


能的代码，或者因误删其他 Bug 修正代码导致已修复问题重现。在这种情况下创建的分支有一个专有的名称：Bugfix 分支或发布分支（Release Branch）。之所以称为发布分支，是因为在软件新版本发布后经常使用此技术进行软件维护，发布升级版本。

图 18-2 演示了如何使用发布分支应对 Bug 修正的过程。

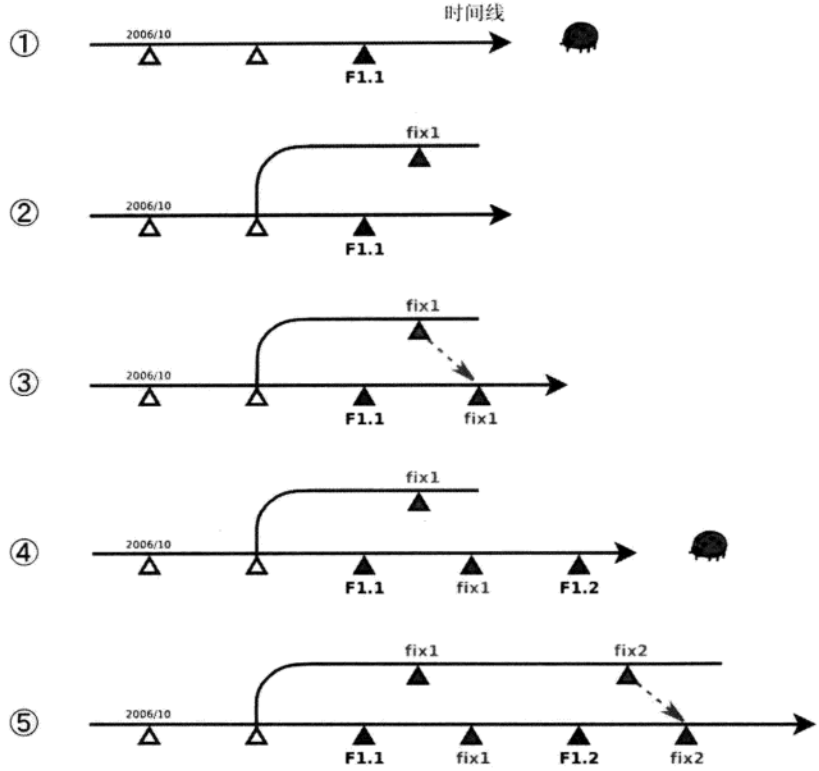



图 18-2 使用发布分支的 Bug 修正过程

说明：

- 图 18-2 中的图示②，可以看到开发者创建了一个发布分支（Bugfix 分支），在分支中提交修正代码“fix1”。注意此分支是自上次软件发布时最后一次提交进行创建的，因此分支中没有包含开发者为新功能所做的提交“F1.1”，是一个“干净”的分支。
- 图示③可以看出从发布分支向主线做了一次合并，这是因为在主线上也同样存在该 Bug，需要在主线上也做出相应的更改。
- 图示④，开发者继续开发，针对功能 1 执行了一个新的提交，编号“F1.2”。这时，客户报告有新的 Bug。
- 继续在发布分支上进行 Bug 修正，参考图示⑤。当修正完成（提交“fix2”）时，基于发布分支创建一个新的软件版本发给客户。不要忘了向主线合并，因为同样的 Bug

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


可能在主线上也存在。

关于如何基于一个历史提交创建分支，以及如何在分支之间进行合并，在本章后面的内容中会详细介绍。

### 18.1.2 特性分支

有这么一个软件项目，项目已经延期了可是还是看不到一点要完成的样子。最终老板变得有些不耐烦了，说道："那么就砍掉一些功能吧"。项目经理听闻一阵眩晕，因为项目经理知道自己负责的这个项目采用的是单一主线开发，要将一个功能从中撤销，工作量非常大，而且还可能会牵涉到其他相关模块的变更。

图18-3就是这个项目的版本库示意图，显然这个项目的代码管理没有使用分支。

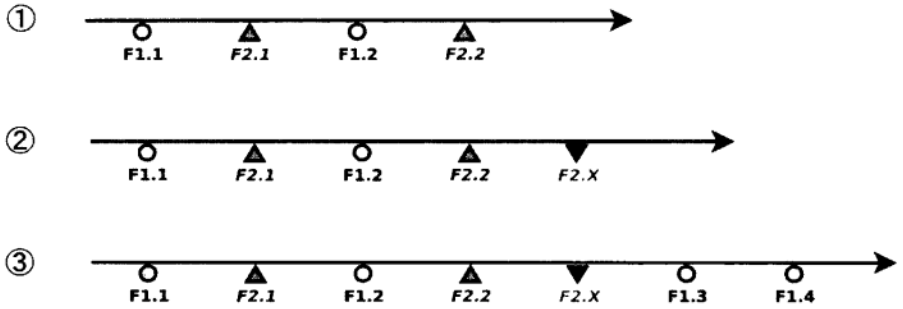



图18-3 没有使用分支导致项目拖延

说明：

- 图18-3中的图示①，用圆圈代表功能1的历次提交，用三角代替功能2的历次提交。因为所有开发者都在主线上工作，所以提交混杂在一起。
- 当老板决定功能2不在这一版本的产品中发布，延期到下一个版本时，功能2的开发者做了一个（或者若干个）反向提交，即图示②中的倒三角（代号为"F2.X"）标识的反向提交，将功能2的所有历史提交全部撤销。
- 图示③表示除了功能2外的其他开发继续进行。

那么负责开发功能2的开发者干什么呢？或者放一个长假，或者在本地开发，与版本库隔离，即不向版本库提交，直到延期的项目终于发布之后再将代码提交。这两种方法都是不可取的，尤其是后一种隔离开发最危险，如果因为病毒感染、文件误删、磁盘损坏，就会导致全部工作损失殆尽。我管理过的一个项目组就曾经遇到过这样的情况。

采用分支将某个功能或模块的开发与开发主线独立出来，是解决类似问题的办法，这种用途的分支被称为特性分支（Feature Branch）或主题分支（Topic Branch）。图18-4就展示了如何使用特性分支帮助纠正要延期的项目，协同多用户的开发。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

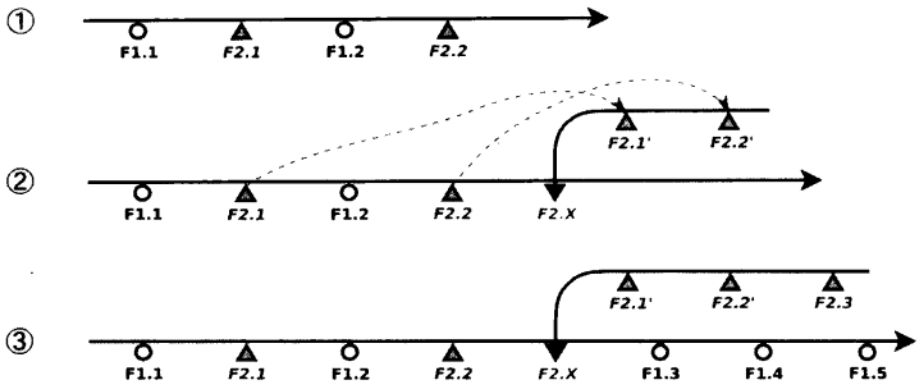




图 18-4 使用特性分支协同多功能开发



说明：

- 图 18-4 中的图示①和前面的一样，都是多个开发者的提交混杂在开发主线中。
- 图示②是当得知功能 2 不在此次产品发布中后，功能 2 的开发者所做的操作。
- 首先，功能 2 的开发者提交一个（或若干个）反向提交，将功能 2 的相关代码全部撤销。图中倒三角（代号为“F2.X”）的提交就是一个反向提交。
- 接着，功能 2 的开发者从反向提交开始创建一个特性分支。
- 最后，功能 2 的开发者将功能 2 的历史提交拣选到特性分支上。对于 Git 可以使用拣选命令 `git cherry-pick`。
- 图示③中可以看出包括功能 2 在内的所有功能和模块都继续提交，但是提交的分支各不相同。功能 2 的开发者将代码提交到特性分支上，其他开发者还提交到主线上。

那么在什么情况下使用特性分支呢？试验性、探索性的功能开发应该为其建立特性分支。功能复杂、开发周期长（有可能在本次发布中取消）的模块应该为其建立特性分支。会更改软件体系架构，破坏软件集成，或者容易导致冲突、影响他人开发进度的模块，应该为其建立特性分支。

在使用 CVS 或 Subversion 等版本控制系统建立分支时，或者因为太慢（CVS）或者因为授权原因需要找管理员进行操作，非常的不方便。Git 的分支管理就方便多了，一是开发者可以在本地版本库中随心所欲地创建分支，二是管理员可以对共享版本库进行设置允许开发者创建特定名称的分支，这样开发者的本地分支可以推送到服务器实现数据的备份。关于 Git 服务器的分支授权参照本书第 5 篇的 Gitolite 服务器架设的相关章节。

### 18.1.3 卖主分支

有的项目要引用到第三方的代码模块并且需要对其进行定制，有的项目甚至整个就是基于某个开源项目进行的定制。如何有效地管理本地定制和第三方（上游）代码的变更就成为

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


了一个难题。卖主分支（Vendor Branch）可以部分解决这个难题。

所谓卖主分支，就是在版本库中创建一个专门和上游代码进行同步的分支，一旦有上游代码发布就检入到卖主分支中。图 18-5 就是一个典型的卖主分支工作流程。

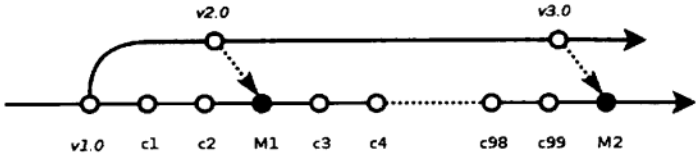



图 18-5 卖主分支工作流程

说明：

- 在主线上检入上游软件版本 1.0 的代码。在图中标记为 v1.0 的提交即是。
- 然后在主线上进行定制开发，c1、c2 分别代表历次定制提交。
- 当上游有了新版本发布后，例如 2.0 版本，就将上游新版本的源代码提交到卖主分支中。图中标记为 v2.0 的提交即是。
- 然后在主线上合并卖主分支上的新提交，合并后的提交显示为 M1。

如果定制较少，使用卖主分支可以工作得很好，但是如果定制的内容非常多，在合并的时候就会遇到非常多的冲突。定制的代码越多、混杂的越厉害，冲突解决就越困难。

本章的内容尚不能针对复杂的定制开发给出满意的版本控制解决方案，本书第 4 篇的“第 22 章 Topgit 协同模型”会介绍一个针对复杂定制开发的更好的解决方案。

## 18.2 分支命令概述

在 Git 中分支管理使用命令 git branch。该命令的主要用法如下：

```
用法 1：git branch
用法 2：git branch <branchname>
用法 3：git branch <branchname> <start-point>
用法 4：git branch -d <branchname>
用法 5：git branch -D <branchname>
用法 6：git branch -m <oldbranch> <newbranch>
用法 7：git branch -M <oldbranch> <newbranch>
```

说明：

- 用法 1 用于显示本地分支列表。当前分支在输出中会显示为特别的颜色，并用星号“*”标识出来。
- 用法 2 和用法 3 用于创建分支。用法 2 基于当前头指针（HEAD）指向的提交创建分支，新分支的分支名为 <branchname>。用法 3 基于提交 <start-point> 创建新分支，新分支的分支名为 <branchname>。
- 用法 4 和用法 5 用于删除分支。用法 4 在删除分支 <branchname> 时会检查所要删除的分支是否已经合并到其他分支中，否则拒绝删除。用法 5 会强制删除分支

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


<branchname>，即使该分支没有合并到任何一个分支中。

- 用法 6 和用法 7 用于重命名分支。如果版本库中已经存在名为 <newbranch> 的分支，用法 6 拒绝执行重命名，而用法 7 会强制执行。

下面就通过“Hello World”项目演示 Git 的分支管理。

## 18.3 “Hello World”开发计划

上一章从 Github[①] 上检出的 hello-world 包含了一个 C 语言开发的应用，现在假设项目 hello-world 做产品发布，版本号定为 1.0，则进行下面的里程碑操作。

（1）为 hello-world 创建里程碑 v1.0 。

```
$ cd /path/to/user1/workspace/hello-world/
$ git tag -m "Release 1.0" v1.0
```

（2）将新建的里程碑推送到远程共享版本库。

```
$ git push origin refs/tags/v1.0
Counting objects: 1, done.
Writing objects: 100% (1/1), 158 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To file:///path/to/repos/hello-world.git
 * [new tag]         v1.0 -> v1.0
```

到现在为止还没有运行 hello-world 程序呢，现在就在开发者 user1 的工作区中运行一下，具体操作过程如下。

（1）进入 src 目录，编译程序。

```
$ cd src
$ make
version.h.in => version.h
cc    -c -o main.o main.c
cc -o hello main.o
```

（2）使用参数 --help 运行 hello 程序，可以查看帮助信息。

说明：hello 程序的帮助输出中有一个拼写错误，本应该是 --help 的地方写成了 -help。这是有意为之。

```
$ ./hello --help
Hello world example v1.0
Copyright Jiang Xin <jiangxin AT ossxp DOT com>, 2009.

Usage:
    hello
```

① *https://github.com/ossxp-com/hello-world/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        say hello to the world.

hello <username>
        say hi to the user.

hello -h, -help
        this help screen.
```

（3）不带参数运行，向全世界问候。

说明：最后一行显示版本为“v1.0”，这显然是来自于新建立的里程碑“v1.0”。

```
$ ./hello
Hello world.
(version: v1.0)
```

（4）执行命令的时候，后面添加用户名作为参数，则向该用户问候。

说明：下面在运行 hello 的时候，显然出现了一个 Bug，即用户名中间如果出现了空格，输出的欢迎信息只包含了部分的用户名。这个 Bug 也是有意为之。

```
$ ./hello Jiang Xin
Hi, Jiang.
(version: v1.0)
```

既然 1.0 版本已经发布了，现在是时候制订下一个版本 2.0 的开发计划了。计划如下：

- 多语种支持。

  为 hello-world 添加多语种支持，使得软件运行的时候能够使用中文或其他本地化语言进行问候。

- 用 getopt 进行命令行解析。

  对命令行参数解析框架进行改造，以便实现更灵活、更易扩展的命令行处理。在 1.0 版本中，程序内部解析命令行参数使用了简单的字符串比较，非常不灵活。从源文件 src/main.c 中可以看到当前实现的简陋和局限。

```
$ git grep -n argv
main.c:20:main(int argc, char **argv)
main.c:24:      } else if ( strcmp(argv[1],"-h") == 0 ||
main.c:25:                  strcmp(argv[1],"--help") == 0 ) {
main.c:28:          printf ("Hi, %s.\n", argv[1]);
```

最终决定由开发者 user2 负责多语种支持的功能，由开发者 user1 负责用 getopt 进行命令行解析的功能。

## 18.4 基于特性分支的开发

有了前面“代码管理之殇”的铺垫，在领受任务之后，开发者 user1 和 user2 应该为自己负责的功能创建特性分支。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 18.4.1 创建分支 user1/getopt

开发者 user1 负责用 getopt 进行命令行解析的功能，因为这个功能用到 getopt 函数，于是将这个分支命名为 user1/getopt。开发者 user1 使用 git branch 命令创建该特性分支，具体操作过程如下。

（1）确保是在开发者 user1 的工作区中。

```
$ cd /path/to/user1/workspace/hello-world/
```

（2）开发者 user1 基于当前 HEAD 创建分支 user1/getopt。

```
$ git branch user1/getopt
```

（3）使用 git branch 创建分支，并不会自动切换。查看当前分支可以看到仍然工作在 master 分支（用星号“*”标识）中。

```
$ git branch
* master
  user1/getopt
```

（4）执行 git checkout 命令切换到新分支上。

```
$ git checkout user1/getopt
Switched to branch 'user1/getopt'
```

（5）再次查看分支列表，当前工作分支的标记符（星号）已经落在 user1/getopt 分支上。

```
$ git branch
  master
* user1/getopt
```

分支实际上是创建在目录 .git/refs/heads 下的引用，版本库初始时创建的 master 分支就是在该目录下。在第 2 篇“第 7 章 Git 重置”的章节中，已经介绍过 master 分支的实现，实际上这也是所有分支的实现方式。

- 查看一下 .git/refs/heads 目录下的引用。可以在该目录下看到 master 文件，和一个 user1 目录。而在 user1 目录下是文件 getopt。

```
$ ls -F .git/refs/heads/
master  user1/
$ ls -F .git/refs/heads/user1/
getopt
```

- 引用文件 .git/refs/heads/user1/getopt 记录的是一个提交 ID。

```
$ cat .git/refs/heads/user1/getopt
ebcf6d6b06545331df156687ca2940800a3c599d
```

- 因为分支 user1/getopt 是基于头指针 HEAD 创建的，因此当前该分支和 master 分支的指向是一致的。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cat .git/refs/heads/master
ebcf6d6b06545331df156687ca2940800a3c599d
```

❑ 当前的工作分支为user1/getopt，记录在头指针文件 .git/HEAD 中。

切换分支命令 git checkout 对文件 .git/HEAD 的内容进行更新。可以参照第2篇“第8章 Git检出”的相关章节。

```
$ cat .git/HEAD
ref: refs/heads/user1/getopt
```

## 18.4.2 创建分支 user2/i18n

开发者 user2 要完成多语种支持的工作任务，于是决定将分支定名为 user2/i18n。每一次创建分支通常都需要完成以下两个工作：

❑ 创建分支：执行 git branch <branchname> 命令创建新分支。

❑ 切换分支：执行 git checkout <branchname> 命令切换到新分支。

有没有简单的操作，在创建分支后立即切换到新分支上呢？有的，Git 提供了这样一个命令，能够将上述两条命令所执行的操作一次性完成。用法如下：

```
git checkout -b <new_branch> [<start_point>]
```

即检出命令 git checkout 通过参数 -b <new_branch> 实现了创建分支和切换分支两个动作的合二为一。下面开发者 user2 就使用 git checkout 命令来创建分支，具体操作过程如下。

（1）进入到开发者 user2 的工作目录，并和上游同步一次。

```
$ cd /path/to/user2/workspace/hello-world/
$ git pull
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
From file:///path/to/repos/hello-world
 * [new tag]         v1.0       -> v1.0
Already up-to-date.
```

（2）执行 git checkout -b 命令，创建并切换到新分支 user2/i18n 上。

```
$ git checkout -b user2/i18n
Switched to a new branch 'user2/i18n'
```

（3）查看本地分支列表，会看到已经创建并切换到 user2/i18n 分支上了。

```
$ git branch
  master
* user2/i18n
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 18.4.3 开发者 user1 完成功能开发

开发者 user1 开始在 user1/getopt 分支中工作，重构 hello-world 中的命令行参数解析的代码。重构时采用 getopt_long 函数。

您可以试着更改，不过在 hello-world 中已经保存了一份改好的代码，可以直接检出。

（1）确保是在 user1 的工作区中。

```
$ cd /path/to/user1/workspace/hello-world/
```

（2）执行下面的命令，用里程碑 jx/v2.0 标记的内容（已实现用 getopt 进行命令行解析的功能）替换暂存区和工作区。

下面的 git checkout 命令的最后是一个点“.”，因此检出只更改了暂存区和工作区，而没有修改头指针。

```
$ cd /path/to/user1/workspace/hello-world/
$ git checkout jx/v2.0 -- .
```

（3）查看状态，会看到分支仍保持为 user1/getopt，但文件 src/main.c 被修改了。

```
$ git status
# On branch user1/getopt
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   src/main.c
#
```

（4）比较暂存区和 HEAD 的文件差异，可以看到为实现用 getopt 进行命令行解析功能而对代码的改动。

```
$ git diff --cached
diff --git a/src/main.c b/src/main.c
index 6ee936f..fa5244a 100644
--- a/src/main.c
+++ b/src/main.c
@@ -1,4 +1,6 @@
 #include <stdio.h>
+#include <getopt.h>
+
 #include "version.h"

 int usage(int code)
@@ -19,15 +21,44 @@ int usage(int code)
 int
 main(int argc, char **argv)
 {
-    if (argc == 1) {
+    int c;
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
+    char *uname = NULL;
+
+    while (1) {
+        int option_index = 0;
+        static struct option long_options[] = {
+            {"help", 0, 0, 'h'},
+            {0, 0, 0, 0}
+        };
...
```

（5）开发者user1提交代码，完成开发任务。

```
$ git commit -m "Refactor: use getopt_long for arguments parsing."
[user1/getopt 0881ca3] Refactor: use getopt_long for arguments parsing.
 1 files changed, 36 insertions(+), 5 deletions(-)
```

（6）提交完成之后，可以看到这时user1/getopt分支和master分支的指向不同了。

```
$ git rev-parse user1/getopt master
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
ebcf6d6b06545331df156687ca2940800a3c599d
```

（7）编译运行hello-world。

注意输出中的版本号显示。

```
$ cd src
$ make clean
rm -f hello main.o version.h
$ make
version.h.in => version.h
cc    -c -o main.o main.c
cc -o hello main.o
$ ./hello
Hello world.
(version: v1.0-1-g0881ca3)
```

### 18.4.4 将user1/getopt分支合并到主线

既然开发者user1负责的功能开发完成了，那就合并到开发主线master上吧，这样测试团队（如果有的话）就可以基于开发主线master进行软件集成和测试了。具体操作过程如下。

（1）为将分支合并到主线，首先user1将工作区切换到主线，即master分支。

```
$ git checkout master
Switched to branch 'master'
```

（2）然后执行git merge命令以合并user1/getopt分支。

```
$ git merge user1/getopt
Updating ebcf6d6..0881ca3
Fast-forward
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



```
 src/main.c |   41 ++++++++++++++++++++++++++++++++++++++++-----
 1 files changed, 36 insertions(+), 5 deletions(-)
```

（3）本次合并非常顺利，实际上合并后 master 分支和 user1/getopt 指向同一个提交。这是因为合并前的 master 分支的提交就是 usr1/getopt 分支的父提交，所以此次合并相当于将分支 master 重置到 user1/getopt 分支。

```
$ git rev-parse user1/getopt master
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
```

（4）查看状态信息可以看到本地分支和远程分支的跟踪关系。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
```

（5）上面的状态输出中显示本地 master 分支比远程共享版本库的 master 分支领先，可以运行 git cherry 命令查看哪些提交领先（未被推送到上游跟踪分支中）。

```
$ git cherry
+ 0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
```

（6）执行推送操作，完成本地分支向远程分支的同步。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 689 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/hello-world.git
   ebcf6d6..0881ca3  master -> master
```

（7）删除 user1/getopt 分支。

既然特性分支 user1/getopt 已经合并到主线上了，那么该分支已经完成了历史使命，可以放心地将其删除。

```
$ git branch -d user1/getopt
Deleted branch user1/getopt (was 0881ca3).
```

开发者 user2 对多语种支持功能有些犯愁，需要多花些时间，那么就先不等他了。

## 18.5 基于发布分支的开发

用户在使用 1.0 版的 hello-word 过程中发现了两个错误，报告给项目组。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 第一个问题是：帮助信息中出现文字错误。本应该写为“--help”却写成了“-help”。
- 第二个问题是：当执行hello-world的程序，提供带空格的用户名时，问候语中显示的是不完整的用户名。

  例如执行“./hello Jiang Xin”，本应该输出“Hi, Jiang Xin.”，却只输出了“Hi, Jiang.”。

为了能够及时修正1.0版本中存在的这两个Bug，将这两个Bug的修正工作分别交给两个开发者user1和user2完成：

- 开发者user1负责修改文字错误的Bug。
- 开发者user2负责修改显示用户名不完整的bug。

现在的版本库中master分支相比1.0发布时添加了新功能代码，即开发者user1推送的用getopt进行命令行解析的相关代码。如果基于master分支对用户报告的两个Bug进行修改，就会引入尚未经过测试、可能不稳定的新功能的代码。在之前“代码管理之殇”中介绍的发布分支，恰恰适用于此场景。

### 18.5.1 创建发布分支

要想解决在1.0版本中发现的Bug，就需要基于1.0发行版的代码创建发布分支。

（1）软件hello-world的1.0发布版在版本库中有一个里程碑相对应。

```
$ cd /path/to/user1/workspace/hello-world/
$ git tag -n1 -l v*
v1.0            Release 1.0
```

（2）基于里程碑v1.0创建发布分支hello-1.x。

注：使用了git checkout命令创建分支，最后一个参数v1.0是新分支hello-1.x创建的基准点。如果没有里程碑，使用提交ID也是一样。

```
$ git checkout -b hello-1.x v1.0
Switched to a new branch 'hello-1.x'
```

（3）用git rev-parse命令可以看到hello-1.x分支对应的提交ID和里程碑v1.0指向的提交一致，但是和master不一样。

提示：因为里程碑v1.0是一个包含提交说明的里程碑，因此为了显示其对应的提交ID，使用了特别的记法“v1.0^{}”。

```
$ git rev-parse hello-1.x v1.0^{} master
ebcf6d6b06545331df156687ca2940800a3c599d
ebcf6d6b06545331df156687ca2940800a3c599d
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
```

（4）开发者user1将分支hello-1.x推送到远程共享版本库，因为开发者user2修改Bug时也要用到该分支。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git push origin hello-1.x
Total 0 (delta 0), reused 0 (delta 0)
To file:///path/to/repos/hello-world.git
 * [new branch]      hello-1.x -> hello-1.x
```

（5）开发者 user2 从远程共享版本库获取新的分支。

开发者 user2 执行 `git fetch` 命令，将远程共享版本库的新分支 `hello-1.x` 复制到本地引用 `origin/hello-1.x`[①]上。

```
$ cd /path/to/user2/workspace/hello-world/
$ git fetch
From file:///path/to/repos/hello-world
 * [new branch]      hello-1.x  -> origin/hello-1.x
```

（6）开发者 user2 切换到 hello-1.x 分支。

本地引用 `origin/hello-1.x` 称为远程分支，第 19 章将专题介绍。该远程分支不能直接检出，而是需要基于该远程分支创建本地分支。第 19 章会介绍一个更为简单的基于远程分支建立本地分支的方法，本例先用标准的方法建立分支。

```
$ git checkout -b hello-1.x origin/hello-1.x
Branch hello-1.x set up to track remote branch hello-1.x from origin.
Switched to a new branch 'hello-1.x'
```

## 18.5.2 开发者 user1 工作在发布分支

开发者 user1 修改帮助信息中的文字错误，具体操作过程如下。

（1）编辑文件 `src/main.c`，将“`-help`”字符串修改为“`--help`”。

```
$ cd /path/to/user1/workspace/hello-world/
$ vi src/main.c
...
```

（2）开发者 user1 的改动可以从下面的差异比较中看到。

```
$ git diff
diff --git a/src/main.c b/src/main.c
index 6ee936f..e76f05e 100644
--- a/src/main.c
+++ b/src/main.c
@@ -11,7 +11,7 @@ int usage(int code)
             "              say hello to the world.\n\n"
             "    hello <username>\n"
             "              say hi to the user.\n\n"
-            "    hello -h, -help\n"
+            "    hello -h, --help\n"
             "              this help screen.\n\n", _VERSION);
```

① 该引用的全称为 refs/remotes/origin/hello-1.x。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    return code;
}
```

（3）执行提交。

```
$ git add -u
$ git commit -m "Fix typo: -help to --help."
[hello-1.x b56bb51] Fix typo: -help to --help.
 1 files changed, 1 insertions(+), 1 deletions(-)
```

（4）推送到远程共享版本库。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 349 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To file:///path/to/repos/hello-world.git
   ebcf6d6..b56bb51  hello-1.x -> hello-1.x
```

### 18.5.3 开发者 user2 工作在发布分支

开发者 user2 针对问候时用户名显示不全的 Bug 进行更改，具体操作过程如下。

（1）进入开发者 user2 的工作区，并确保工作在 hello-1.x 分支中。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout hello-1.x
```

（2）编辑文件 src/main.c，修改代码中的 Bug。

```
$ vi src/main.c
```

（3）实际上在 hello-world 版本库中包含了我的一份修改，可以看看和您的更改是否一致。

下面的命令将我对此 Bug 的修改保存为一个补丁文件。

```
$ git format-patch jx/v1.1..jx/v1.2
0001-Bugfix-allow-spaces-in-username.patch
```

（4）应用我对此 Bug 的改动补丁[①]。

如果您已经自己完成了修改，可以先执行 git stash 保存自己的修改进度，然后执行下面的命令应用补丁文件。当应用完补丁后，再执行 git stash pop 将您的改动合并到工作区。如果我们的改动一致（英雄所见略同），将不会有冲突。

```
$ patch -p1 < 0001-Bugfix-allow-spaces-in-username.patch
```

① 应用由 git format-patch 生成的补丁文件，最好使用 git am 命令。这里为简单起见使用 GNU patch 命令。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
patching file src/main.c
```

（5）看看代码的改动吧。

```
$ git diff
diff --git a/src/main.c b/src/main.c
index 6ee936f..f0f404b 100644
--- a/src/main.c
+++ b/src/main.c
@@ -19,13 +19,20 @@ int usage(int code)
 int
 main(int argc, char **argv)
 {
+    char **p = NULL;
+
     if (argc == 1) {
         printf ("Hello world.\n");
     } else if ( strcmp(argv[1],"-h") == 0 ||
                 strcmp(argv[1],"--help") == 0 ) {
                 return usage(0);
     } else {
-        printf ("Hi, %s.\n", argv[1]);
+        p = &argv[1];
+        printf ("Hi,");
+        do {
+            printf (" %s", *p);
+        } while (*(++p));
+        printf (".\n");
     }

     printf( "(version: %s)\n", _VERSION );
```

（6）本地测试一下改进后的软件，看看 Bug 是否已经被改正。如果运行结果能显示出完整的用户名，则 Bug 成功修正。

```
$ cd src/
$ make
version.h.in => version.h
cc    -c -o main.o main.c
cc -o hello main.o
$ ./hello Jiang Xin
Hi, Jiang Xin.
(version: v1.0-dirty)
```

（7）提交代码。

```
$ git add -u
$ git commit -m "Bugfix: allow spaces in username."
[hello-1.x e64f3a2] Bugfix: allow spaces in username.
 1 files changed, 8 insertions(+), 1 deletions(-)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 18.5.4 开发者 user2 合并推送

开发者 user2 在本地版本库完成提交后，不要忘记向远程共享版本库进行推送。但在推送分支 hello-1.x 时开发者 user2 没有开发者 user1 那么幸运，因为此时远程共享版本库的 hello-1.x 分支已经被开发者 user1 推送过一次，因此开发者 user2 在推送过程中会遇到非快进式推送问题。

```
$ git push
To file:///path/to/repos/hello-world.git
 ! [rejected]        hello-1.x -> hello-1.x (non-fast-forward)
error: failed to push some refs to 'file:///path/to/repos/hello-world.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' section of 'git push --help' for details.
```

就像在“第 15 章 Git 协议和工作协同”一章中介绍的那样，开发者 user2 需要执行一个拉回操作，将远程共享服务器的改动获取到本地并和本地提交进行合并。

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/hello-world
   ebcf6d6..b56bb51  hello-1.x  -> origin/hello-1.x
Auto-merging src/main.c
Merge made by recursive.
 src/main.c |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

通过显示分支图的方式查看日志，可以看到在执行 git pull 操作后发生了合并。

```
$ git log --graph --oneline
*   8cffe5f Merge branch 'hello-1.x' of file:///path/to/repos/hello-world into hello-1.x
|\
| * b56bb51 Fix typo: -help to --help.
* | e64f3a2 Bugfix: allow spaces in username.
|/
* ebcf6d6 blank commit for GnuPG-signed tag test.
* 8a9f3d1 blank commit for annotated tag test.
* 60a2f4f blank commit.
* 3e6070e Show version.
* 75346b3 Hello world initialized.
```

现在开发者 user2 可以将合并后的本地版本库中的提交推送给远程共享版本库了。

```
$ git push
Counting objects: 14, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 814 bytes, done.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Total 8 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To file:///path/to/repos/hello-world.git
   b56bb51..8cffe5f  hello-1.x -> hello-1.x
```

### 18.5.5 发布分支的提交合并到主线

当开发者 user1 和 user2 都相继在 hello-1.x 分支中将相应的 Bug 修改完后，就可以从 hello-1.x 分支中编译新的软件产品交给客户使用了。接下来别忘了在主线 master 分支中也做出同样的更改，因为在 hello-1.x 分支中修改的 Bug 同样也存在于主线 master 分支中。

**1. 拣选操作**

使用 Git 提供的拣选命令，就可以直接将发布分支上进行的 Bug 修正合并到主线上。下面就以开发者 user2 的身份进行操作，具体操作过程如下。

（1）进入 user2 工作区并切换到 master 分支。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout master
```

（2）从远程共享版本库同步 master 分支。

同步后本地 master 分支包含了开发者 user1 提交的命令行参数解析重构的代码。

```
$ git pull
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From file:///path/to/repos/hello-world
   ebcf6d6..0881ca3  master     -> origin/master
Updating ebcf6d6..0881ca3
Fast-forward
 src/main.c |   41 ++++++++++++++++++++++++++++++++++++++++-----
 1 files changed, 36 insertions(+), 5 deletions(-)
```

（3）查看分支 hello-1.x 的日志，确认要拣选的提交 ID。

从下面的日志中可以看出分支 hello-1.x 的最新提交是一个合并提交，而要拣选的提交分别是其第一个父提交和第二个父提交，可以分别用“hello-1.x^1”和“hello-1.x^2”表示。

```
$ git log -3 --graph --oneline hello-1.x
*   8cffe5f Merge branch 'hello-1.x' of file:///path/to/repos/hello-world into
    hello-1.x
|\
| * b56bb51 Fix typo: -help to --help.
* | e64f3a2 Bugfix: allow spaces in username.
|/
```

（4）执行拣选操作。先将开发者 user2 提交的修正代码拣选到当前分支（即主线）。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


拣选操作遇到了冲突，见下面的命令输出。

```
$  git cherry-pick hello-1.x^1
Automatic cherry-pick failed.  After resolving the conflicts,
mark the corrected paths with 'git add <paths>' or 'git rm <paths>'
and commit the result with:

        git commit -c e64f3a216d346669b85807ffcfb23a21f9c5c187
```

（5）拣选操作发生冲突，通过查看状态可以看到是在文件 src/main.c 上发生了冲突。

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      src/main.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

### 2. 冲突发生的原因

为什么发生了冲突呢？这是因为拣选 hello-1.x 分支上的一个提交到 master 分支时，因为两个甚至多个提交在重叠的位置更改代码所致。通过下面的命令可以看到到底是哪些提交引起的冲突。

```
$ git log master...hello-1.x^1
commit e64f3a216d346669b85807ffcfb23a21f9c5c187
Author: user2 <user2@moon.ossxp.com>
Date:   Sun Jan 9 13:11:19 2011 +0800

    Bugfix: allow spaces in username.

commit 0881ca3f62ddadcddec08bd9f2f529a44d17cfbf
Author: user1 <user1@sun.ossxp.com>
Date:   Mon Jan 3 22:44:52 2011 +0800

    Refactor: use getopt_long for arguments parsing.
```

可以看出引发冲突的提交一个是当前工作分支 master 上的最新提交，即开发者 user1 的重构命令行参数解析的提交，而另外一个引发冲突的是要拣选的提交，即开发者 user2 针对用户名显示不全所做的错误修正提交。一定是因为这两个提交的更改发生了重叠导致了冲突的发生。下面就来解决冲突。

### 3. 冲突解决

冲突解决可以使用图形界面工具，不过对于本例直接编辑冲突文件，手工进行冲突解决也很方便。打开文件 src/main.c 就可以看到发生冲突的区域都用特有的标记符标识出来，参见表 18-1 中左侧一列的内容。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


表 18-1 冲突解决前后对照

| 冲突文件 src/main.c 标识出的冲突内容 | 冲突解决后的内容对照 |
| --- | --- |
| 21 int | 21 int |
| 22 main(int argc, char **argv) | 22 main(int argc, char **argv) |
| 23 { | 23 { |
| 24 <<<<<<< HEAD | |
| 25 int c; | 24 int c; |
| 26 char ***uname** = NULL; | 25 char ***p** = NULL; |
| 27 | 26 |
| 28 while (1) { | 27 while (1) { |
| 29 int option_index = 0; | 28 int option_index = 0; |
| 30 static struct option long_options[] = { | 29 static struct option long_options[] = { |
| 31 {"help", 0, 0, 'h'}, | 30 {"help", 0, 0, 'h'}, |
| 32 {0, 0, 0, 0} | 31 {0, 0, 0, 0} |
| 33 }; | 32 }; |
| 34 | 33 |
| 35 c = getopt_long(argc, argv, "h", | 34 c = getopt_long(argc, argv, "h", |
| 36 long_options, &option_index); | 35 long_options, &option_index); |
| 37 if (c == −1) | 36 if (c == -1) |
| 38 break; | 37 break; |
| 39 | 38 |
| 40 switch (c) { | 39 switch (c) { |
| 41 case 'h': | 40 case 'h': |
| 42 return usage(0); | 41 return usage(0); |
| 43 default: | 42 default: |
| 44 return usage(1); | 43 return usage(1); |
| 45 } | 44 } |
| 46 } | 45 } |
| 47 | 46 |
| 48 if (optind < argc) { | 47 if (optind < argc) { |
| 49 **uname** = **argv**[optind]; | 48 **p** = **&argv**[optind]; |
| 50 } | 49 } |
| 51 | 50 |
| 52 if (**uname** == **NULL**) { | 51 if (**p** == **NULL** \|\| ***p** == **NULL**) { |
| 53 ======= | |
| 54 char **p = NULL; | |
| 55 | |
| 56 if (argc == 1) { | |
| 57 >>>>>>> e64f3a2... Bugfix: allow spaces in username. | |
| 58 printf ("Hello world.\n"); | 52 printf ("Hello world.\n"); |
| 59 } else { | 53 } else { |
| 60 <<<<<<< HEAD | |
| 61 printf ("Hi, %s.\n", uname); | |
| 62 ======= | |
| 63 p = &argv[1]; | |
| 64 printf ("Hi,"); | 54 printf ("Hi,"); |
| 65 do { | 55 do { |
| 66 printf (" %s", *p); | 56 printf (" %s", *p); |
| 67 } while (*(++p)); | 57 } while (*(++p)); |
| 68 printf (".\n"); | 58 printf (".\n"); |
| 69 >>>>>>> e64f3a2... Bugfix: allow spaces in username. | |
| 70 } | 59 } |
| 71 | 60 |
| 72 printf( "(version: %s)\n", _VERSION ); | 61 printf( "(version: %s)\n", _VERSION ); |
| 73 return 0; | 62 return 0; |
| 74 } | 63 } |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


在文件 src/main.c 冲突内容中，第 25-52 行及第 61 行是 master 分支中由开发者 user1 重构命令行解析时提交的内容，而第 54 ～ 56 行及第 63-68 行则是分支 hello-1.x 中由开发者 user2 提交的修正用户名显示不全的 Bug 的相应代码。

表 18-1 右侧的一列则是冲突解决后的内容。为了和冲突前的内容相对照，重新进行了排版，并对差异内容进行加粗显示。您可以参照完成冲突解决。

将手动编辑完成的文件 src/main.c 添加到暂存区才真正地完成了冲突解决。

```
$ git add src/main.c
```

因为是拣选操作，提交时最好重用所拣选提交的提交说明和作者信息，而且也省下了自己写提交说明的麻烦。使用下面的命令完成提交操作。

```
$ git commit -C hello-1.x^1
[master 10765a7] Bugfix: allow spaces in username.
 1 files changed, 8 insertions(+), 4 deletions(-)
```

#### 4. 完成全部拣选操作

接下来再将开发者 user1 在分支 hello-1.x 中的提交也拣选到当前分支。所拣选的提交非常简单，不过是修改了提交说明中的文字错误而已，拣选操作也不会引发异常，直接完成。

```
$ git cherry-pick hello-1.x^2
Finished one cherry-pick.
[master d81896e] Fix typo: -help to --help.
 Author: user1 <user1@sun.ossxp.com>
 1 files changed, 1 insertions(+), 1 deletions(-)
```

现在通过日志可以看到 master 分支已经完成了对已知 Bug 的修复。

```
$ git log -3 --graph --oneline
* d81896e Fix typo: -help to --help.
* 10765a7 Bugfix: allow spaces in username.
* 0881ca3 Refactor: use getopt_long for arguments parsing.
```

查看状态可以看到当前的工作分支相对于远程服务器有两个新提交。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
nothing to commit (working directory clean)
```

执行推送命令将本地 master 分支同步到远程共享版本库。

```
$ git push
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 802 bytes, done.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Total 8 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
To file:///path/to/repos/hello-world.git
   0881ca3..d81896e  master -> master
```

## 18.6 分支变基

### 18.6.1 完成 user2/i18n 特性分支的开发

开发者 user2 针对多语种开发的工作任务还没有介绍呢，在最后就借着“实现”这个稍微复杂的功能来学习一下 Git 分支的变基操作，具体操作过程如下。

（1）进入 user2 的工作区，并切换到 user2/i18n 分支。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout user2/i18n
Switched to branch 'user2/i18n'
```

（2）使用 gettext 为软件添加多语言支持。您可以尝试实现该功能。不过在 hello-world 中已经保存了一份实现该功能的代码（见里程碑 jx/v1.0-i18n），可以直接拿过来用。

（3）里程碑 jx/v1.0-i18n 最后的两个提交实现了多语言支持功能。

```
$ git log --oneline -2 --stat jx/v1.0-i18n
ade873c Translate for Chinese.
 src/locale/zh_CN/LC_MESSAGES/helloworld.po |   30 ++++++++++++++++++++++------
 1 files changed, 23 insertions(+), 7 deletions(-)
0831248 Add I18N support.
 src/Makefile                               |   21 ++++++++++-
 src/locale/helloworld.pot                  |   46 ++++++++++++++++++++++++++++++
 src/locale/zh_CN/LC_MESSAGES/helloworld.po |   46 ++++++++++++++++++++++++++++++
 src/main.c                                 |   18 ++++++++--
 4 files changed, 125 insertions(+), 6 deletions(-)
```

（4）可以通过拣选命令将这两个提交拣选到 user2/i18n 分支中，相当于在分支 user2/i18n 中实现了多语言支持的开发。

```
$ git cherry-pick jx/v1.0-i18n~1
...
$ git cherry-pick jx/v1.0-i18n
...
```

（5）看看当前分支拣选后的日志。

```
$ git log --oneline -2
7acb3e8 Translate for Chinese.
90d873b Add I18N support.
```

（6）为了测试刚刚“开发”完成的多语言支持功能，先对源码执行编译。

```
$ cd src
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ make
version.h.in => version.h
cc    -c -o main.o main.c
msgfmt -o locale/zh_CN/LC_MESSAGES/helloworld.mo
locale/zh_CN/LC_MESSAGES/helloworld.po
cc -o hello main.o
```

（7）查看帮助信息，会发现帮助信息已经本地化。

注意：帮助信息中仍然有文字错误，--help误写为-help。

```
$ ./hello --help
Hello world 示例 v1.0-2-g7acb3e8
版权所有 蒋鑫 <jiangxin AT ossxp DOT com>, 2009

用法：
    hello
            世界你好。

    hello <username>
            向用户问您好。

    hello -h, -help
            显示本帮助页。
```

（8）不带用户名运行hello，也会输出中文。

```
$ ./hello
世界你好。
(version: v1.0-2-g7acb3e8)
```

（9）带用户名运行hello，会向用户问候。

注意：程序仍然存在只显示部分用户名的问题。

```
$ ./hello Jiang Xin
您好，Jiang.
(version: v1.0-2-g7acb3e8)
```

（10）推送分支user2/i18n到远程共享服务器。

推送该特性分支的目的并非是与他人在此分支上协同工作，主要是为了进行数据备份。

```
$ git push origin user2/i18n
Counting objects: 21, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), 2.91 KiB, done.
Total 17 (delta 6), reused 1 (delta 0)
Unpacking objects: 100% (17/17), done.
To file:///path/to/repos/hello-world.git
 * [new branch]      user2/i18n -> user2/i18n
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




### 18.6.2 分支 user2/i18n 变基

在测试刚刚完成的具有多语种支持功能的 hello-world 时，之前改正的两个 Bug 又重现了。这并不奇怪，因为分支 user2/i18n 基于 master 分支创建的时候，这两个 Bug 还没有发现呢，更不要说改正了。

在最早刚刚创建 user2/i18n 分支时，版本库的结构非常简单，如图 18-6 所示。

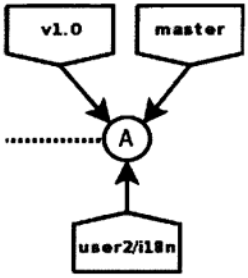



图 18-6 分支 user2/i18n 创建初始版本库的分支状态

但是当前 master 分支中不但包含了对两个 Bug 的修正，还包含了开发者 user1 调用 getopt 对命令行参数解析进行的代码重构。图 18-7 显示的是当前版本库 master 分支和 user2/i18n 分支的关系图。

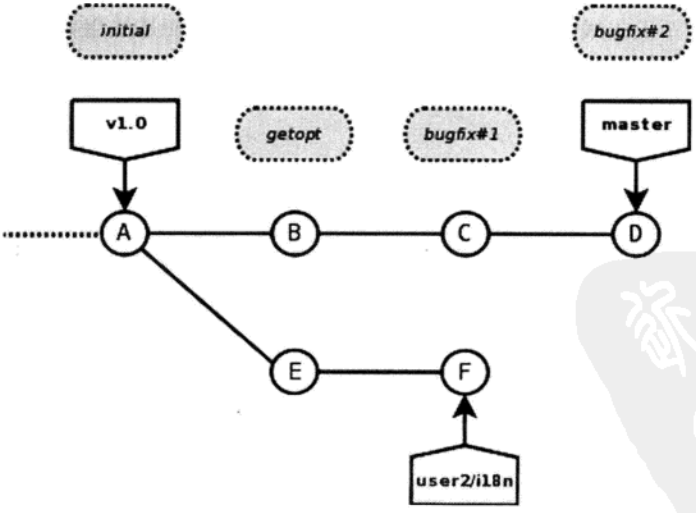



图 18-7 当前版本库分支示意图

开发者 user2 要将分支 user2/i18n 中的提交合并到主线 master 中，可以采用上一节介绍的分支合并操作。如果执行分支合并操作，版本库的状态将会如图 18-8 所示：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

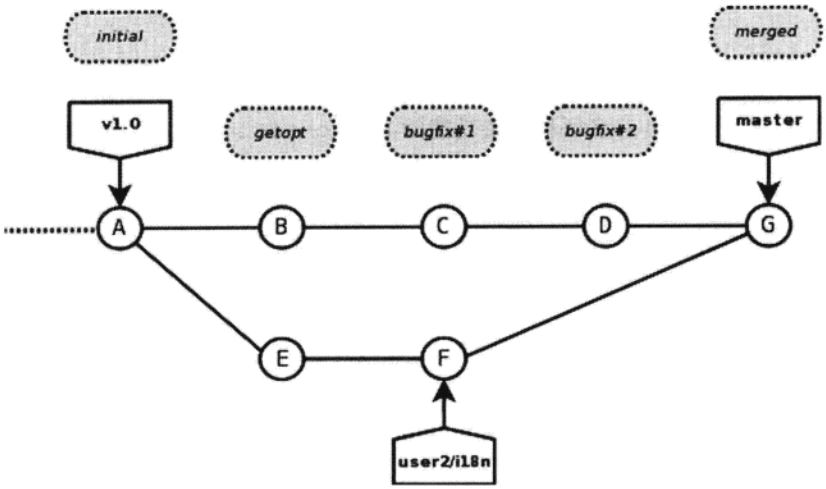




图 18-8 使用分支合并时版本库的分支状态

这样操作有利有弊。有利的一面是开发者在 user2/i18n 分支中的提交不会发生改变，这一点对于提交已经被他人共享时很重要。再有因为 user2/i18n 分支是基于 v1.0 创建的，这样可以很容易将多语言支持功能添加到 1.0 版本的 hello-world 中。不过这些对于本项目来说都不重要。至于不利的一面，就是这样的合并操作会产生三个提交（包括一个合并提交），对于要对提交进行审核的项目团队来说增加了代码审核的负担。因此很多项目在特性分支合并到开发主线的时候，都不推荐使用合并操作，而是使用变基操作。如果执行变基操作，版本库相关分支的关系图就如图 18-9 所示。

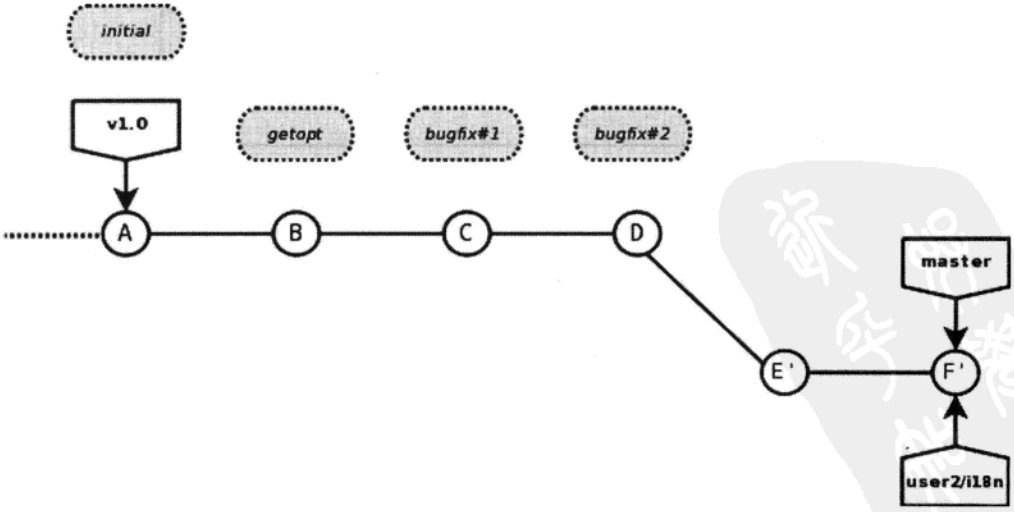



图 18-9 使用变基操作版本库的分支状态

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




很显然，采用变基操作的分支关系图要比采用合并操作的简单多了，看起来更像是集中式版本控制系统特有的顺序提交。因为减少了一个提交，也会减轻代码审核的负担。

下面开发者 user2 就通过变基操作将特性分支 user2/i18n 合并到主线，具体操作过程如下。

（1）首先确保开发者 user2 的工作区位于分支 user2/i18n 上。

```
$ cd /path/to/user2/workspace/hello-world/
$ git checkout user2/i18n
```

（2）执行变基操作。

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add I18N support.
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Failed to merge in the changes.
Patch failed at 0001 Add I18N support.

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".
```

变基遇到了冲突，看来这回的麻烦可不小。冲突是在合并 user2/i18n 分支中的提交“Add I18N support”时遇到的。首先回顾一下变基的原理，参见第 2 篇“第 12 章 改变历史”的相关章节。对于本例，在进行变基操作时会先切换到 user2/i18n 分支，并强制重置到 master 分支所指向的提交。然后再将原 user2/i18n 分支的提交一一拣选到新的 user2/i18n 分支上。运行下面的命令可以查看可能导致冲突的提交列表。

```
$ git rev-list --pretty=oneline user2/i18n^...master
d81896e60673771ef1873b27a33f52df75f70515 Fix typo: -help to --help.
10765a7ef46981a73d578466669f6e17b73ac7e3 Bugfix: allow spaces in username.
90d873bb93cd7577b7638f1f391bd2ece3141b7a Add I18N support.
0881ca3f62ddadcddec08bd9f2f529a44d17cfbf Refactor: use getopt_long for arguments
parsing
```

刚刚发生的冲突是在拣选提交“Add I18N suppport”时出现的，所以在冲突文件中标识为他人版本的是 user2 添加多语种支持功能的提交，而冲突文件中标识为自己版本的是修正两个 Bug 的提交及开发者 user1 提交的重构命令行参数解析的提交。下面的两个表格（表 18-2 和表 18-3）是文件 src/main.c 发生冲突的两个主要区域，表格的左侧一列是冲突文件中的内容，右侧一列则是冲突解决后的内容。为了方便对照进行了适当排版。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


**表 18-2 变基冲突区域一解决前后对照**

| 变基冲突区域一内容（文件 src/main.c） | 冲突解决后的内容对照 |
| --- | --- |
| 12 int usage(int code) | 12 int usage(int code) |
| 13 { | 13 { |
| 14 printf(_("Hello world example %s\n" | 14 printf(_("Hello world example %s\n" |
| 15 "Copyright Jiang Xin <jiangxin AT ossxp ...\n" | 15 "Copyright Jiang Xin <jiangxin AT ossxp ...\n" |
| 16 "\n" | 16 "\n" |
| 17 "Usage:\n" | 17 "Usage:\n" |
| 18 " hello\n" | 18 " hello\n" |
| 19 " say hello to the world.\n\n" | 19 " say hello to the world.\n\n" |
| 20 " hello <username>\n" | 20 " hello <username>\n" |
| 21 " say hi to the user.\n\n" | 21 " say hi to the user.\n\n" |
| 22 <<<<<<< HEAD | |
| 23 " hello -h, --help\n" | 22 " hello -h, --help\n" |
| 24 " this help **screen.\n\n**", _VERSION); | 23 " this help **screen.\n\n**"), _VERSION); |
| 25 \|\|\|\|\|\|\| merged common ancestors | |
| 26 " hello -h, -help\n" | |
| 27 " this help screen.\n\n", _VERSION); | |
| 28 ======= | |
| 29 " hello -h, -help\n" | |
| 30 " this help screen.\n\n"), _VERSION); | |
| 31 >>>>>>> Add I18N support. | |
| 32 return code; | 24 return code; |
| 33 } | 25 } |

**表 18-3 变基冲突区域二解决前后对照**

| 变基冲突区域二内容（文件 src/main.c） | 冲突解决后的内容对照 |
| --- | --- |
| 38 <<<<<<< HEAD | |
| 39 int c; | 30 int c; |
| 40 char **p = NULL; | 31 char **p = NULL; |
| 41 | 32 |
| | 33 **setlocale( LC_ALL, "" );** |
| | 34 **bindtextdomain("helloworld","locale");** |
| | 35 **textdomain("helloworld");** |
| | 36 |
| 42 while (1) { | 37 while (1) { |
| 43 int option_index = 0; | 38 int option_index = 0; |
| 44 static struct option long_options[] = { | 39 static struct option long_options[] = { |
| 45 {"help", 0, 0, 'h'}, | 40 {"help", 0, 0, 'h'}, |
| 46 {0, 0, 0, 0} | 41 {0, 0, 0, 0} |
| 47 }; | 42 }; |
| 48 | 43 |
| 49 c = getopt_long(argc, argv, "h", | 44 c = getopt_long(argc, argv, "h", |
| 50 long_options, &option_index); | 45 long_options, &option_index); |
| 51 if (c == -1) | 46 if (c == -1) |
| 52 break; | 47 break; |
| 53 | 48 |
| 54 switch (c) { | 49 switch (c) { |
| 55 case 'h': | 50 case 'h': |
| 56 return usage(0); | 51 return usage(0); |
| 57 default: | 52 default: |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 变基冲突区域二内容（文件 src/main.c） | 冲突解决后的内容对照 |
| --- | --- |
| 58 return usage(1);<br>59 }<br>60 }<br>61<br>62 if (optind < argc) {<br>63 p = &argv[optind];<br>64 }<br>65<br>66 if (p == NULL \|\| *p == NULL) {<br>67 printf ("**Hello world.\n**");<br>68 \|\|\|\|\|\|\| merged common ancestors<br>69 if (argc == 1) {<br>70 printf ("Hello world.\n");<br>71 } else if ( strcmp(argv[1],"-h") == 0 \|\|<br>72 strcmp(argv[1],"--help") == 0 ) {<br>73 return usage(0);<br>74 =======<br>75 setlocale( LC_ALL, "" );<br>76 bindtextdomain("helloworld","locale");<br>77 textdomain("helloworld");<br>78<br>79 if (argc == 1) {<br>80 printf ( _("Hello world.\n") );<br>81 } else if ( strcmp(argv[1],"-h") == 0 \|\|<br>82 strcmp(argv[1],"--help") == 0 ) {<br>83 return usage(0);<br>84 >>>>>>> Add I18N support.<br>85 } else {<br>86 <<<<<<< HEAD<br>87 printf ("**Hi,**");<br>88 do {<br>89 printf (" %s", *p);<br>90 } while (*(++p));<br>91 printf (".\n");<br>92 \|\|\|\|\|\|\| merged common ancestors<br>93 printf ("Hi, %s.\n", argv[1]);<br>94 =======<br>95 printf (_("Hi, %s.\n"), argv[1]);<br>96 >>>>>>> Add I18N support.<br>97 } | 53 return usage(1);<br>54 }<br>55 }<br>56<br>57 if (optind < argc) {<br>58 p = &argv[optind];<br>59 }<br>60<br>61 if (p == NULL \|\| *p == NULL) {<br>**62** printf ( **_("Hello world.\n")** );<br><br><br><

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   src/Makefile
#       new file:   src/locale/helloworld.pot
#       new file:   src/locale/zh_CN/LC_MESSAGES/helloworld.po
#       modified:   src/main.c
#
```

现在不要执行提交，而是继续变基操作。变基操作会自动完成对冲突解决的提交，并对分支中的其他提交继续执行变基，直至全部完成。

```
$ git rebase --continue
Applying: Add I18N support.
Applying: Translate for Chinese.
```

图 18-10 显示了版本库执行完变基后的状态。

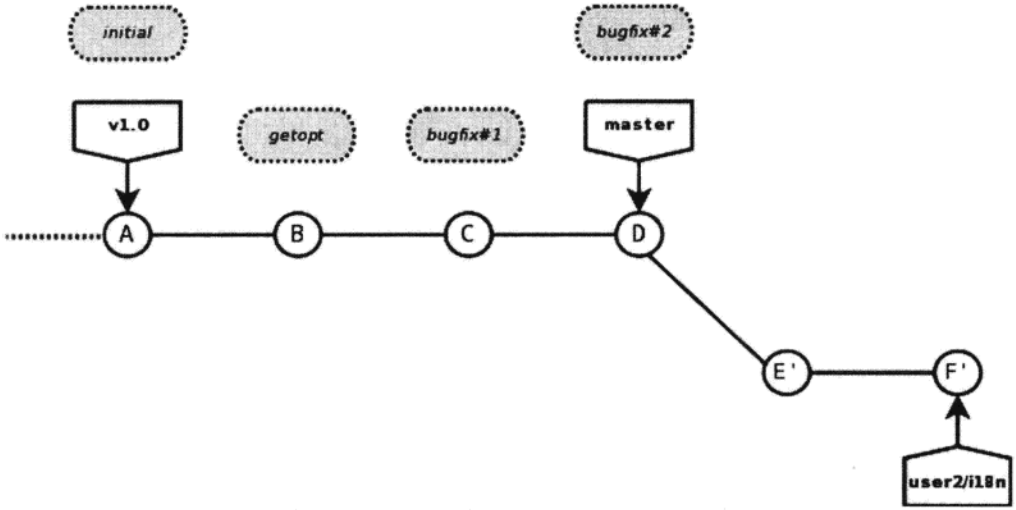



图 18-10 变基操作完成后版本库的分支状态

现在需要将user2/i18n分支的提交合并到主线master中。实际上不需要在master分支上再执行繁琐的合并操作，而是可以直接用推送操作——用本地的user2/i18n分支直接更新远程版本库的master分支。

```
$ git push origin user2/i18n:master
Counting objects: 21, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), 2.91 KiB, done.
Total 17 (delta 6), reused 1 (delta 0)
Unpacking objects: 100% (17/17), done.
To file:///path/to/repos/hello-world.git
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


仔细看看上面运行的 git push 命令，终于看到了引用表达式中冒号前后使用了不同名字的引用。含义是用本地的 user2/i18n 引用的内容（提交 ID）更新远程共享版本库的 master 引用内容（提交 ID）。

执行拉回操作，可以发现远程共享版本库的 master 分支的确被更新了。通过拉回操作本地的 master 分支也随之更新。

（1）切换到 master 分支，会从提示信息中看到本地 master 分支落后远程共享版本库 master 分支两个提交。

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

（2）执行拉回操作，将本地 master 分支同步到和远程共享版本库相同的状态。

```
$ git pull
Updating d81896e..c4acab2
Fast-forward
 src/Makefile                                |   21 ++++++++-
 src/locale/helloworld.pot                   |   46 ++++++++++++++++++++
 src/locale/zh_CN/LC_MESSAGES/helloworld.po |   62 ++++++++++++++++++++++++++++
 src/main.c                                  |   18 +++++++--
 4 files changed, 141 insertions(+), 6 deletions(-)
 create mode 100644 src/locale/helloworld.pot
 create mode 100644 src/locale/zh_CN/LC_MESSAGES/helloworld.po
```

特性分支 user2/i18n 也完成了历史使命，可以删除了。因为之前 user2/i18n 已经推送到远程共享版本库，如果想要删除分支不要忘了也将远程分支同时删除。

（1）删除本地版本库的 user2/i18n 分支。

```
$ git branch -d user2/i18n
Deleted branch user2/i18n (was c4acab2).
```

（2）删除远程共享版本库的 user2/i18n 分支。

```
$ git push origin :user2/i18n
To file:///path/to/repos/hello-world.git
 - [deleted]         user2/i18n
```

**补充** 实际上变基之后 user2/i18n 分支的本地化模板文件（helloworld.pot）和汉化文件（helloworld.po）都需要做出相应更新，否则 hello-world 的一些输出不能进行本地化。更新模板需要删除文件 helloworld.pot，再执行命令 make po。重新翻译中文本地化文件，可以使用工具 lokalize 或 kbabel。具体的操作过程就不再赘述了[①]。

① 在本篇第 20 章“20.3.1 StGit”一节会弥补这一遗憾，完善 Hello Word 的本地化。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 19 章　远程版本库

Git 作为分布式版本库控制系统，每个人都是本地版本库的主人，可以在本地的版本库中随心所欲地创建分支和里程碑。当需要多人协作时，问题就出现了：

- 如何避免因为用户把所有的本地分支都推送到共享版本库，从而造成共享版本库上分支的混乱？
- 如何避免不同用户针对不同特性开发创建了相同名字的分支而造成分支名称的冲突？
- 如何避免用户随意在共享版本库中创建里程碑而导致里程碑名称上的混乱和冲突？
- 当用户向共享版本库及其他版本库推送时，每次都需要输入长长的版本库 URL，太不方便了。
- 当用户需要经常从多个不同的他人版本库中获取提交时，有没有办法不要总是输入长长的版本库 URL ？
- 如果不带任何其他参数执行 `git fetch`、`git pull` 和 `git push` 到底是和哪个远程版本库及哪个分支进行交互？

本章介绍的 `git remote` 命令就是用于实现对远程版本库的便捷访问，建立远程分支和本地分支的对应，使得 `git fetch`、`git pull` 和 `git push` 能够更为便捷地进行操作。

## 19.1　远程分支

上一章介绍 Git 分支的时候，每一个版本库最多只和一个上游版本库（远程共享版本库）进行交互，实际上 Git 允许一个版本库和任意多的版本库进行交互。首先执行下面的命令，基于 `hello-world.git` 版本库再创建几个新的版本库。

```
$ cd /path/to/repos/
$ git clone --bare hello-world.git hello-user1.git
Cloning into bare repository hello-user1.git...
done.
$ git clone --bare hello-world.git hello-user2.git
Cloning into bare repository hello-user2.git...
done.
```

现在有了三个共享版本库：`hello-world.git`、`hello-user1.git` 和 `hello-user2.git`。现在有一个疑问，如果一个本地版本库需要和上面三个版本库进行互操作，三个共享版本库都存在一个 master 分支，会不会互相干扰、冲突或覆盖呢？

先来看看 `hello-world` 远程共享版本库中包含的分支有哪些：

```
$ git ls-remote --heads file:///path/to/repos/hello-world.git
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
8cffe5f135821e716117ee59bdd53139473bd1d8        refs/heads/hello-1.x
bb4fef88fee435bfac04b8389cf193d9c04105a6        refs/heads/helper/master
cf71ae3515e36a59c7f98b9db825fd0f2a318350        refs/heads/helper/v1.x
c4acab26ff1c1125f5e585ffa8284d27f8ceea55        refs/heads/master
```

原来远程共享版本库中有四个分支，其中 hello-1.x 分支是开发者 user1 创建的。现在重新克隆该版本库，如下：

```
$ cd /path/to/my/workspace/
$ git clone file:///path/to/repos/hello-world.git
...
$ cd /path/to/my/workspace/hello-world
```

执行 git branch 命令检查分支，会吃惊地看到只有一个分支 master。

```
$ git branch
* master
```

那么远程版本库中的其他分支哪里去了？为什么本地只有一个分支呢？执行 git show-ref 命令可以看到全部的本地引用。

```
$ git show-ref
c4acab26ff1c1125f5e585ffa8284d27f8ceea55 refs/heads/master
c4acab26ff1c1125f5e585ffa8284d27f8ceea55 refs/remotes/origin/HEAD
8cffe5f135821e716117ee59bdd53139473bd1d8 refs/remotes/origin/hello-1.x
bb4fef88fee435bfac04b8389cf193d9c04105a6 refs/remotes/origin/helper/master
cf71ae3515e36a59c7f98b9db825fd0f2a318350 refs/remotes/origin/helper/v1.x
c4acab26ff1c1125f5e585ffa8284d27f8ceea55 refs/remotes/origin/master
3171561b2c9c57024f7d748a1a5cfd755a26054a refs/tags/jx/v1.0
aaff5676a7c3ae7712af61dfb9ba05618c74bbab refs/tags/jx/v1.0-i18n
e153f83ee75d25408f7e2fd8236ab18c0abf0ec4 refs/tags/jx/v1.1
83f59c7a88c04ceb703e490a86dde9af41de8bcb refs/tags/jx/v1.2
1581768ec71166d540e662d90290cb6f82a43bb0 refs/tags/jx/v1.3
ccca267c98380ea7fffb241f103d1e6f34d8bc01 refs/tags/jx/v2.0
8a5b9934aacdebb72341dcadbb2650cf626d83da refs/tags/jx/v2.1
89b74222363e8cbdf91aab30d005e697196bd964 refs/tags/jx/v2.2
0b4ec63aea44b96d498528dcf3e72e1255d79440 refs/tags/jx/v2.3
60a2f4f31e5dddd777c6ad37388fe6e5520734cb refs/tags/mytag
5dc2fc52f2dcb84987f511481cc6b71ec1b381f7 refs/tags/mytag3
51713af444266d56821fe3302ab44352b8c3eb71 refs/tags/v1.0
```

从 git show-ref 的输出中发现了几个不寻常的引用，这些引用以 refs/remotes/origin/ 为前缀，并且名称和远程版本库的分支名一一对应。这些引用实际上就是从远程版本库的分支复制过来的，称为远程分支。

Git 的 git branch 命令也能够查看这些远程分支，不过要加上“-r”参数：

```
$ git branch -r
  origin/HEAD -> origin/master
  origin/hello-1.x
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
origin/helper/master
origin/helper/v1.x
origin/master
```

Git 这样的设计是非常巧妙的，在从远程版本库执行获取操作时，不是把远程版本库的分支原封不动地复制到本地版本库的分支中，而是复制到另外的命名空间。如在克隆一个版本库时，会将远程分支都复制到目录 .git/refs/remotes/origin/ 下。这样从不同的远程版本库执行获取操作，因为通过命名空间的相互隔离，所以就避免了在本地的相互覆盖。

那么克隆操作产生的远程分支为什么都有一个名为“origin/”的前缀呢？奥秘就在配置文件 .git/config 中。下面的几行内容出自该配置文件，为了说明方便显示了行号。

```
6 [remote "origin"]
7   fetch = +refs/heads/*:refs/remotes/origin/*
8   url = file:///path/to/repos/hello-world.git
```

这个小节可以称为 [remote] 小节，该小节以 origin 为名注册了一个远程版本库。该版本库的 URL 地址由第 8 行给出，会发现这个 URL 地址就是执行 git clone 命令时所用的地址。最具魔法的配置是第 7 行，这一行设置了执行 git fetch origin 操作时使用的默认引用表达式：

- 该引用表达式以加号（+）开头，含义是强制进行引用的替换，即使即将进行的替换是非快进式的。
- 引用表达式中使用了通配符，冒号前面的含有通配符的引用指的是远程版本库的所有分支，冒号后面的引用含义是复制到本地的远程分支目录中。

正因为有了上面的 [remote] 配置小节，当执行 git fetch origin 操作时，就相当于执行了下面的命令，将远程版本库的所有分支复制为本地的远程分支。

```
git fetch origin +refs/heads/*:refs/remotes/origin/*
```

远程分支不是真正意义上的分支，是类似于里程碑一样的引用。如果针对远程分支执行检出命令，会看到大段的错误警告。

```
$ git checkout origin/hello-1.x
Note: checking out 'origin/hello-1.x'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 8cffe5f... Merge branch 'hello-1.x' of
file:///path/to/repos/hello-world into hello-1.x
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


上面大段的错误信息实际上告诉我们一件事，远程分支类似于里程碑，如果检出就会使得头指针 HEAD 处于分离头指针状态。实际上除了以 refs/heads 为前缀的引用之外，如果检出任何其他引用，都将使工作区处于分离头指针状态。如果对远程分支进行修改就需要创建新的本地分支。

## 19.2 分支追踪

为了能够在远程分支 refs/remotes/origin/hello-1.x 上进行工作，需要基于该远程分支创建本地分支。远程分支可以使用简写 origin/hello-1.x。如果 Git 的版本是 1.6.6 或更新的版本，可以使用下面的命令同时完成分支的创建和切换。

```
$ git checkout hello-1.x
Branch hello-1.x set up to track remote branch hello-1.x from origin.
Switched to a new branch 'hello-1.x'
```

如果 Git 的版本比较老，或注册了多个远程版本库，因此存在多个名为 hello-1.x 的远程分支，就不能使用上面简洁的分支创建和切换命令，而需要使用在上一章中学习到的分支创建命令，显式地从远程分支中创建本地分支。

```
$ git checkout -b hello-1.x origin/hello-1.x
Branch hello-1.x set up to track remote branch hello-1.x from origin.
Switched to a new branch 'hello-1.x'
```

在上面基于远程分支创建本地分支的过程中，命令输出的第一行说的是建立了本地分支和远程分支的跟踪。和远程分支建立跟踪后，本地分支就具有下列特征：

- 检查工作区状态时，会显示本地分支和被跟踪远程分支提交之间的关系。
- 当执行 git pull 命令时，会和被跟踪的远程分支进行合并（或者变基），如果两者出现版本偏离的话。
- 当执行 git push 命令时，会推送到远程版本库的同名分支中。

下面就在基于远程分支创建的本地跟踪分支中进行操作，看看本地分支是如何与远程分支建立关联的，具体操作过程如下。

（1）先将本地 hello-1.x 分支向后重置两个版本。

```
$ git reset --hard HEAD^^
HEAD is now at ebcf6d6 blank commit for GnuPG-signed tag test.
```

（2）然后查看状态，显示当前分支相比跟踪分支落后了 3 个版本。

之所以落后三个版本而非两个版本是因为 hello-1.x 的最新提交是一个合并提交，包含两个父提交，因此上面的重置命令丢弃掉了三个提交。

```
$ git status
# On branch hello-1.x
# Your branch is behind 'origin/hello-1.x' by 3 commits, and can be fast-
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
forwarded.
#
nothing to commit (working directory clean)
```

(3) 执行 git pull 命令，会自动与跟踪的远程分支进行合并，相当于找回最新的3个提交。

```
$ git pull
Updating ebcf6d6..8cffe5f
Fast-forward
 src/main.c |   11 ++++++++++--
 1 files changed, 9 insertions(+), 2 deletions(-)
```

但是如果基于本地分支创建另外一个本地分支则没有分支跟踪的功能。下面就从本地的 hello-1.x 分支中创建 hello-jx 分支。

(1) 从 hello-1.x 分支中创建新的本地分支 hello-jx。

下面的创建分支操作只有一行输出，看不到分支间建立跟踪的提示。

```
$ git checkout -b hello-jx hello-1.x
Switched to a new branch 'hello-jx'
```

(2) 将 hello-jx 分支重置。

```
$ git reset --hard HEAD^^
HEAD is now at ebcf6d6 blank commit for GnuPG-signed tag test.
```

(3) 检查状态看不到分支间的跟踪信息。

```
$ git status
# On branch hello-jx
nothing to commit (working directory clean)
```

(4) 执行 git pull 命令会报错。

```
$ git pull
You asked me to pull without telling me which branch you
want to merge with, and 'branch.hello-jx.merge' in
your configuration file does not tell me, either. Please
specify which branch you want to use on the command line and
try again (e.g. 'git pull <repository> <refspec>').
See git-pull(1) for details.

If you often merge with the same branch, you may want to
use something like the following in your configuration file:

    [branch "hello-jx"]
    remote = <nickname>
    merge = <remote-ref>

    [remote "<nickname>"]
    url = <url>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    fetch = <refspec>

See git-config(1) for details.
```

将上面命令执行中的错误信息翻译过来，就是：

```
$ git pull
您让我执行拉回操作，但是没有告诉我您希望与哪个远程分支进行合并，
而且也没有通过配置 'branch.hello-jx.merge' 来告诉我。

请在命令行中提供足够的参数，如 'git pull <repository> <refspec>' 。
或者如果您经常与同一个分支进行合并，可以和该分支建立跟踪。在配置
中添加如下配置信息：

    [branch "hello-jx"]
    remote = <nickname>
    merge = <remote-ref>

    [remote "<nickname>"]
    url = <url>
    fetch = <refspec>
```

为什么用同样方法建立的分支 hello-1.x 和 hello-jx，差距咋就那么大呢？奥秘就在于从远程分支创建本地分支，自动建立了分支间的跟踪，而从一个本地分支创建另外一个本地分支则没有。看看配置文件 .git/config 中是不是专门为分支 hello-1.x 创建了相应的配置信息？

```
 9 [branch "master"]
10    remote = origin
11    merge = refs/heads/master
12 [branch "hello-1.x"]
13    remote = origin
14    merge = refs/heads/hello-1.x
```

其中第 9 ～ 11 行是针对 master 分支设置的分支间跟踪，是在版本库克隆的时候自动建立的。而第 12 ～ 14 行是前面基于远程分支创建本地分支时建立的。至于分支 hello-jx 则没有建立相关的配置。

如果希望在基于一个本地分支创建另外一个本地分支时也能够使用分支间的跟踪功能，就要在创建分支时提供 --track 参数。下面实践一下，具体操作过程如下。

（1）删除之前创建的 hello-jx 分支。

```
$ git checkout master
Switched to branch 'master'
$ git branch -d hello-jx
Deleted branch hello-jx (was ebcf6d6).
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（2）使用参数 --track 重新基于 hello-1.x 创建 hello-jx 分支。

```
$ git checkout --track -b hello-jx hello-1.x
Branch hello-jx set up to track local branch hello-1.x.
Switched to a new branch 'hello-jx'
```

（3）从 Git 库的配置文件中会看到为 hello-jx 分支设置的跟踪。

因为跟踪的是本版本库的本地分支，所以第 16 行设置的远程版本库的名字为一个点。

```
15 [branch "hello-jx"]
16    remote = .
17    merge = refs/heads/hello-1.x
```

## 19.3 远程版本库

名为 origin 的远程版本库是在版本库克隆时注册的，那么如何注册新的远程版本库呢？

### 1. 注册远程版本库

下面将版本库 file:///path/to/repos/hello-user1.git 以 new-remote 为名进行注册。

```
$ git remote add new-remote file:///path/to/repos/hello-user1.git
```

如果再打开版本库的配置文件 .git/config 会看到新的配置。

```
12 [remote "new-remote"]
13    url = file:///path/to/repos/hello-user1.git
14    fetch = +refs/heads/*:refs/remotes/new-remote/*
```

执行 git remote 命令，可以更为方便地显示已经注册的远程版本库。

```
$ git remote -v
new-remote      file:///path/to/repos/hello-user1.git (fetch)
new-remote      file:///path/to/repos/hello-user1.git (push)
origin  file:///path/to/repos/hello-world.git (fetch)
origin  file:///path/to/repos/hello-world.git (push)
```

现在执行 git fetch 并不会从新注册的 new-remote 远程版本库中获取，因为当前分支设置的默认远程版本库为 origin。要想从 new-remote 远程版本库中获取，需要为 git fetch 命令增加一个参数 new-remote。

```
$ git fetch new-remote
From file:///path/to/repos/hello-user1
 * [new branch]      hello-1.x  -> new-remote/hello-1.x
 * [new branch]      helper/master -> new-remote/helper/master
 * [new branch]      helper/v1.x -> new-remote/helper/v1.x
 * [new branch]      master     -> new-remote/master
```

从上面的命令输出中可以看出，远程版本库的分支复制到本地版本库前缀为 new-

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


remote 的远程分支中去了。用 git branch -r 命令可以看到新增了几个远程分支。

```
$ git branch -r
  new-remote/hello-1.x
  new-remote/helper/master
  new-remote/helper/v1.x
  new-remote/master
  origin/HEAD -> origin/master
  origin/hello-1.x
  origin/helper/master
  origin/helper/v1.x
  origin/master
```

### 2. 更改远程版本库的地址

如果远程版本库的 URL 地址改变，需要更换，该如何处理呢？手工修改 .git/config 文件是一种方法，用 git config 命令进行更改是第二种方法，还有一种方法是用 git remote 命令，如下：

```
$ git remote set-url new-remote file:///path/to/repos/hello-user2.git
```

可以看到注册的远程版本库的 URL 地址已经更改。

```
$ git remote -v
new-remote      file:///path/to/repos/hello-user2.git (fetch)
new-remote      file:///path/to/repos/hello-user2.git (push)
origin  file:///path/to/repos/hello-world.git (fetch)
origin  file:///path/to/repos/hello-world.git (push)
```

从上面的输出中可以发现每一个远程版本库都显示两个 URL 地址，分别是执行 git fetch 和 git push 命令时用到的 URL 地址。既然有两个地址，就意味着这两个地址可以不同，用下面的命令可以为推送操作设置单独的 URL 地址。

```
$ git remote set-url --push new-remote /path/to/repos/hello-user2.git
$ git remote -v
new-remote      file:///path/to/repos/hello-user2.git (fetch)
new-remote      /path/to/repos/hello-user2.git (push)
origin  file:///path/to/repos/hello-world.git (fetch)
origin  file:///path/to/repos/hello-world.git (push)
```

当单独为推送设置了 URL 后，配置文件 .git/config 的对应 [remote] 小节也会增加一条新的名为 pushurl 的配置。如下：

```
12 [remote "new-remote"]
13    url = file:///path/to/repos/hello-user2.git
14    fetch = +refs/heads/*:refs/remotes/new-remote/*
15    pushurl = /path/to/repos/hello-user2.git
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 3. 更改远程版本库的名称

如果对远程版本库的注册名称不满意，也可以进行修改。例如将 new-remote 名称修改为 user2，使用下面的命令：

```
$ git remote rename new-remote user2
```

完成改名后，不但远程版本库的注册名称更改过来了，就连远程分支的名称都会自动进行相应的更改。可以通过执行 git remote 和 git branch -r 命令查看。

```
$ git remote
origin
user2
$ git branch -r
  origin/HEAD -> origin/master
  origin/hello-1.x
  origin/helper/master
  origin/helper/v1.x
  origin/master
  user2/hello-1.x
  user2/helper/master
  user2/helper/v1.x
  user2/master
```

### 4. 远程版本库更新

当注册了多个远程版本库并希望获取所有远程版本库的更新时，Git 提供了一个简单的命令。

```
$ git remote update
Fetching origin
Fetching user2
```

如果某个远程版本库不想在执行 git remote update 时获得更新，可以通过参数关闭自动更新。例如下面的命令关闭远程版本库 user2 的自动更新。

```
$ git config remote.user2.skipDefaultUpdate true
$ git remote update
Fetching origin
```

### 5. 删除远程版本库

如果想要删除注册的远程版本库，用 git remote 的 rm 子命令可以实现。例如删除注册的 user2 版本库。

```
$ git remote rm user2
```

## 19.4 PUSH 和 PULL 操作与远程版本库

Git 分支一章已经介绍过对于新建立的本地分支（没有建立和远程分支的追踪），执行 git push 命令是不会被推送到远程版本库中的，这样的设置是非常安全的，避免了因为误

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


操作将本地分支创建到远程版本库中。当不带任何参数执行 git push 命令时，实际的执行过程是：

- 如果为当前分支设置了 <remote>，即由配置 branch.<branchname>.remote 给出了远程版本库代号，则不带参数执行 git push 相当于执行了 git push <remote>。
- 如果没有为当前分支设置 <remote>，则不带参数执行 git push 相当于执行了 git push origin。
- 要推送的远程版本库的 URL 地址由 remote.<remote>.pushurl 给出。如果没有配置，则使用 remote.<remote>.url 配置的 URL 地址。
- 如果为注册的远程版本库设置了 push 参数，即通过 remote.<remote>.push 配置了一个引用表达式，则使用该引用表达式执行推送。
- 否则使用“:”作为引用表达式。该表达式的含义是同名分支推送，即对所有在远程版本库中有同名分支的本地分支执行推送。

这也就是为什么在一个本地新建分支中执行 git push 推送操作不会推送也不会报错的原因，因为远程不存在同名分支，所以根本就没有对该分支执行推送。而如果本地其他分支在远程版本库有同名分支且本地包含更新的话，会对这些分支进行推送。

在 Git 分支一章中就已经知道，如果需要在远程版本库中创建分支，则执行命令：git push <remote> <new_branch>。即通过将本地分支推送到远程版本库的方式在远程版本库中创建分支。但是在接下来的使用中会遇到麻烦：不能执行 git pull 操作（不带参数）将远程版本库中其他人推送的提交获取到本地。这是因为没有建立本地分支和远程分支的追踪，即没有设置 branch.<branchname>.remote 的值和 branch.<branchname>.merge 的值。

关于不带参数执行 git pull 命令的解释如下：

- 如果为当前分支设置了 <remote>，即由配置 branch.<branchname>.remote 出了远程版本库代号，则不带参数执行 git pull 相当于执行了 git pull <remote>。
- 如果没有为当前分支设置 <remote>，则不带参数执行 git pull 相当于执行了 git pull origin。
- 要获取的远程版本库的 URL 地址由 remote.<remote>.url 给出。
- 如果为注册的远程版本库设置了 fetch 参数，即通过 remote.<remote>.fetch 配置了一个引用表达式，则使用该引用表达式执行获取操作。
- 接下来要确定合并的分支。如果设定了 branch.<branchname>.merge，则对其设定的分支执行合并，否则报错退出。

在执行 git pull 操作的时候可以通过参数 --rebase 设置使用变基而非合并操作，将本地分支的改动变基到跟踪分支上。为了避免因为忘记使用 --rebase 参数导致分支的合并，可以执行如下命令进行设置。注意将 <branchname> 替换为对应的分支名称。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git config branch.<branchname>.rebase true
```

有了这个设置之后，如果是在<branchname>工作分支中执行git pull命令，在遇到本地分支和远程分支出现偏离的情况时，会采用变基操作，而不是默认的合并操作。

如果为本地版本库设置参数branch.autosetuprebase，值为true，则在基于远程分支建立本地追踪分支时，会自动配置branch.<branchname>.rebase参数，在执行git pull命令时使用变基操作取代默认的合并操作。

## 19.5 里程碑和远程版本库

远程版本库中的里程碑同步到本地版本库，会使用同样的名称，而不会像分支那样移动到另外的命名空间（远程分支）中，这可能会给本地版本库中的里程碑带来混乱。当和多个远程版本库交互时，这个问题就更为严重。

前面的Git里程碑一章已经介绍了当执行git push命令推送时，默认不会将本地创建的里程碑带入远程版本库，这样可以避免远程版本库上里程碑的泛滥。但是执行git fetch命令从远程版本库获取分支的最新提交时，如果获取的提交上建有里程碑，这些里程碑会被获取到本地版本库。当删除注册的远程版本库时，远程分支会被删除，但是该远程版本库引入的里程碑不会被删除，日积月累本地版本库中的里程碑可能会变得愈加混乱。

可以在执行git fetch命令的时候，设置不获取里程碑只获取分支及提交。通过提供-n或--no-tags参数可以实现。示例如下：

```
$ git fetch --no-tags file:///path/to/repos/hello-world.git \
      refs/heads/*:refs/remotes/hello-world/*
```

在注册远程版本库的时候，也可以使用--no-tags参数，避免将远程版本库的里程碑引入本地版本库。例如：

```
$ git remote add --no-tags hell-world \
      file:///path/to/repos/hello-world.git
```

## 19.6 分支和里程碑的安全性

通过前面章节的探讨，会感觉到Git的使用真的是太方便、太灵活了，但是需要掌握的知识点和窍门也太多了。为了避免没有经验的用户在团队共享的Git版本库中误操作，就需要对版本库进行一些安全上的设置。本书第5篇Git服务器搭建的相关章节会具体介绍如何配置用户授权等版本库安全性设置。

实际上Git版本库本身也提供了一些安全机制避免对版本库的破坏。

- 用reflog记录对分支的操作历史。

  默认创建的带工作区的版本库都会包含core.logallrefupdates为true的配置，这样在版本库中建立的每个分支都会创建对应的reflog。但是创建的裸版本库默

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


认不包含这个设置，也就不会为每个分支设置 reflog。如果团队的规模较小，可能因为分支误操作导致数据丢失，可以考虑为裸版本库添加 core.logallrefupdates 的相关配置。

- 关闭非快进式推送。

  如果将配置 receive.denyNonFastForwards 设置为 true，则禁止一切非快进式推送。但这个配置有些矫枉过正，更好的方法是搭建基于 SSH 协议的 Git 服务器，通过钩子脚本更灵活地进行配置。例如：允许来自某些用户的强制提交，而其他用户不能执行非快进式推送。

- 关闭分支删除功能。

  如果将配置 receive.denyDeletes 设置为 true，则禁止删除分支。同样更好的方法是通过架设基于 SSH 协议的 Git 服务器，配置分支删除的用户权限。





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 20 章 补丁文件交互

之前各个章节的版本库间的交互都是通过 git push 和 / 或 git pull 命令来实现的，这是 Git 最主要的交互模式，但并不是全部。使用补丁文件是另外一种交互方式，适用于参与者众多的大型项目进行的分布式开发。例如 Git 项目本身的代码提交就主要由贡献者通过邮件传递补丁文件来实现的。作者在写书过程中发现了 Git 的两个 Bug，就是以补丁形式通过邮件贡献给 Git 项目的，下面两个链接就是相关邮件的存档。

- 关于 Git 文档错误的 Bugfix：
  *http://marc.info/?l=git&m=129248415230151*
- 关于 git-apply 的一个 Bugfix:
  *http://article.gmane.org/gmane.comp.version-control.git/162100*

这种使用补丁文件进行提交的方式可以提高项目的参与度。因为任何人都可以参与项目的开发，只要会将提交转化为补丁，会发邮件即可。

## 20.1 创建补丁

Git 提供了将提交批量转换为补丁文件的命令：git format-patch。该命令后面的参数是一个版本范围列表，会将包含在此列表中的提交一一转换为补丁文件，每个补丁文件包含一个序号并从提交说明中提取字符串作为文件名。

下面演示一下在 user1 工作区中，如何将 master 分支的最近 3 个提交转换为补丁文件。

（1）进入 user1 工作区，切换到 master 分支。

```
$ cd /path/to/user1/workspace/hello-world/
$ git checkout master
$ git pull
```

（2）执行下面的命令将最近 3 个提交转换为补丁文件。

```
$ git format-patch -s HEAD~3..HEAD
0001-Fix-typo-help-to-help.patch
0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
```

在上面的 git format-patch 命令中使用了 -s 参数，会在导出的补丁文件中添加当前用户的签名。这个签名并非 GnuPG 式的数字签名，不过是将作者姓名添加到提交说明中而已，和在本书第 2 篇开头介绍的 git commit -s 命令的效果相同。虽然签名很不起眼，但是对于以补丁的方式提交数据却非常重要，因为以补丁方式的提交可能因为合并冲突或其

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


他原因使得最终提交的作者 ID 显示为代为提交的项目管理员的 ID，在提交说明中加入原始作者的署名信息大概是作者唯一露脸的机会。如果在提交时忘了使用 -s 参数添加签名，可以在用 git format-path 命令创建补丁文件的时候补救。

看一下补丁文件的文件头，在下面代码中的第 7 行可以看到新增的签名。

```
 1 From d81896e60673771ef1873b27a33f52df75f70515 Mon Sep 17 00:00:00 2001
 2 From: user1 <user1@sun.ossxp.com>
 3 Date: Mon, 3 Jan 2011 23:48:56 +0800
 4 Subject: [PATCH 1/3] Fix typo: -help to --help.
 5
 6
 7 Signed-off-by: user1 <user1@sun.ossxp.com>
 8 ---
 9  src/main.c |    2 +-
10  1 files changed, 1 insertions(+), 1 deletions(-)
```

补丁文件有一个类似邮件一样的文件头（第 1 ～ 4 行），提交日志的第一行作为邮件标题（Subject），其余的提交说明作为邮件内容（如果有的话），文件补丁用三个横线和提交说明分开。

实际上这些补丁文件可以直接拿来作为邮件发送给项目的负责人。Git 提供了一个辅助邮件发送的命令 git send-email。下面用该命令将这三个补丁文件以邮件的形式发送出去。

```
$ git send-email *.patch
0001-Fix-typo-help-to-help.patch
0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
The following files are 8bit, but do not declare a Content-Transfer-Encoding.
    0002-Add-I18N-support.patch
    0003-Translate-for-Chinese.patch
Which 8bit encoding should I declare [UTF-8]?
Who should the emails appear to be from? [user1 <user1@sun.ossxp.com>]
Emails will be sent from: user1 <user1@sun.ossxp.com>
Who should the emails be sent to? jiangxin
Message-ID to be used as In-Reply-To for the first email?
...
Send this email? ([y]es|[n]o|[q]uit|[a]ll): a
...
```

命令 git send-email 提供交互式字符界面，输入正确的收件人地址，邮件就批量地发送出去了。

## 20.2 应用补丁

在前面通过 git send-email 命令发送邮件给 jiangxin 用户。现在使用 Linux 上的 mail 命令检查一下邮件。

```
$ mail
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Mail version 8.1.2 01/15/2001.  Type ? for help.
"/var/mail/jiangxin": 3 messages 3 unread
>N  1 user1@sun.ossxp.c  Thu Jan 13 18:02   38/1120  [PATCH 1/3] Fix typo: -help
to --help.
 N  2 user1@sun.ossxp.c  Thu Jan 13 18:02  227/6207
=?UTF-8?q?=5BPATCH=202/3=5D=20Add=20I18N=20support=2E?=
 N  3 user1@sun.ossxp.c  Thu Jan 13 18:02   95/2893
=?UTF-8?q?=5BPATCH=203/3=5D=20Translate=20for=20Chinese=2E?=
&
```

如果邮件不止这三封，需要将三个包含补丁的邮件挑选出来保存到另外的文件中。例如在 mail 命令的提示符（&）下输入命令将选定的邮件保存为单独的文件。

```
& s 1-3 user1-mail-archive
"user1-mail-archive" [New file]
& q
```

上面的操作在本地创建了一个由开发者 user1 的补丁邮件组成的归档文件 user1-mail-archive，这个文件是 mbox 格式的，可以用 mail 命令打开。

```
$ mail -f user1-mail-archive
Mail version 8.1.2 01/15/2001.  Type ? for help.
"user1-mail-archive": 3 messages
>   1 user1@sun.ossxp.c  Thu Jan 13 18:02   38/1121  [PATCH 1/3] Fix typo: -help
    to --help.
    2 user1@sun.ossxp.c  Thu Jan 13 18:02  227/6208  =?UTF-8?q?=5BPATCH=202/3=5D=
    20Add=20I18N=20support=2E?=
    3 user1@sun.ossxp.c  Thu Jan 13 18:02   95/2894  =?UTF-8?q?=5BPATCH=203/3=5D=
    20Translate=20for=20Chinese=2E?=
& q
```

使用 git am 命令可以使得保存在 mbox 中的邮件批量地应用在版本库中。am 是 apply email 的缩写。下面就演示一下如何使用 git am 命令应用补丁，具体操作过程如下。

（1）基于 HEAD~3 版本创建一个本地分支，以便在该分支下应用补丁。

```
$ git checkout -b user1 HEAD~3
Switched to a new branch 'user1'
```

（2）将 mbox 文件 user1-mail-archive 中的补丁全部应用在当前分支上。

```
$ git am user1-mail-archive
Applying: Fix typo: -help to --help.
Applying: Add I18N support.
Applying: Translate for Chinese.
```

（3）补丁成功应用上了，看看提交日志。

```
$ git log -3 --pretty=fuller
commit 2d9276af9df1a2fdb71d1e7c9ac6dff88b2920a1
Author:     Jiang Xin <jiangxin@ossxp.com>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
AuthorDate: Thu Jan 13 18:02:03 2011 +0800
Commit:     user1 <user1@sun.ossxp.com>
CommitDate: Thu Jan 13 18:21:16 2011 +0800

    Translate for Chinese.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
    Signed-off-by: user1 <user1@sun.ossxp.com>

commit 41227f492ad37cdd99444a5f5cc0c27288f2bca4
Author:     Jiang Xin <jiangxin@ossxp.com>
AuthorDate: Thu Jan 13 18:02:02 2011 +0800
Commit:     user1 <user1@sun.ossxp.com>
CommitDate: Thu Jan 13 18:21:15 2011 +0800

    Add I18N support.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
    Signed-off-by: user1 <user1@sun.ossxp.com>

commit 4a3380fb7ae90039633dec84acc2aab85398efad
Author:     user1 <user1@sun.ossxp.com>
AuthorDate: Thu Jan 13 18:02:01 2011 +0800
Commit:     user1 <user1@sun.ossxp.com>
CommitDate: Thu Jan 13 18:21:15 2011 +0800

    Fix typo: -help to --help.

    Signed-off-by: user1 <user1@sun.ossxp.com>
```

从提交信息中可以看出：

- 提交的时间信息使用了邮件发送的时间。
- 作者（Author）的信息被保留，和补丁文件中的一致。
- 提交者（Commit）全都设置为 user1，因为提交是在 user1 的工作区完成的。
- 提交说明中的签名信息被保留。实际上 git am 命令也可以提供 -s 参数，在提交说明中附加执行命令用户的签名。

对于不习惯在控制台用 mail 命令接收邮件的用户，可以通过邮件附件、U 盘或其他方式获取 git format-patch 生成的补丁文件，将补丁文件保存在本地，通过管道符调用 git am 命令应用补丁。

```
$ ls *.patch
0001-Fix-typo-help-to-help.patch  0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
$ cat *.patch | git am
Applying: Fix typo: -help to --help.
Applying: Add I18N support.
Applying: Translate for Chinese.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Git 还提供一个命令 git apply，可以应用一般格式的补丁文件，但是不能执行提交，也不能保持补丁中的作者信息。实际上 git apply 命令和 GNU patch 命令类似，细微差别将在本书第 7 篇的“第 38 章 补丁中的二进制文件”中予以介绍。

## 20.3 StGit 和 Quilt

一个复杂功能的开发一定是由多个提交来完成的，对于在以接收和应用补丁文件为开发模式的项目中，复杂的功能需要通过多个补丁文件来完成。补丁文件因为要经过审核才能被接受，因此针对一个功能的多个补丁文件一定要保证各个都是精品：补丁 1 用来完成一个功能点，补丁 2 用来完成第二个功能点，等等。一定不能出现这样的情况：补丁 3 用于修正补丁 1 的错误，补丁 10 改正了补丁 7 中的文字错误，等等。这样就带来补丁管理的难题。

实际上基于特性分支的开发又何尝不是如此？在将特性分支归并到开发主线前，要接受团队的评审，特性分支的开发者一定想将特性分支上的提交进行重整，把一些提交合并或拆分。使用变基命令可以实现提交的重整，但是操作起来会比较困难，有什么好办法呢？

### 20.3.1 StGit

StGit[①]是 Stacked Git 的简称。StGit 就是解决上面提到的两个难题的答案。实际上 StGit 在设计上参考了一个著名的补丁管理工具 Quilt，并且可以输出 Quilt 兼容的补丁列表。在本节的后半部分会介绍 Quilt。

StGit 是一个 Python 项目，安装起来还是很方便的。在 Debian/Ubuntu 下，可以直接通过包管理器安装：

```
$ sudo aptitude install stgit stgit-contrib
```

下面还是用 hello-world 版本库，进行 StGit 的实践。

（1）首先检出 hello-world 版本库。

```
$ cd /path/to/my/workspace/
$ git clone file:///path/to/repos/hello-world.git stgit-demo
$ cd stgit-demo
```

（2）在当前工作区初始化 StGit。

```
$ stg init
```

（3）现在补丁列表为空。

```
$ stg series
```

（4）将最新的三个提交转换为 StGit 补丁。

① *http://www.procode.org/stgit/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ stg uncommit -n 3
Uncommitting 3 patches ...
  Now at patch "translate-for-chinese"
done
```

（5）现在补丁列表中有三个文件了。

第一列是补丁的状态符号。加号（+）代表该补丁已经应用在版本库中，大于号（>）用于标识当前的补丁。

```
$ stg ser
+ fix-typo-help-to-help
+ add-i18n-support
> translate-for-chinese
```

（6）现在查看 master 分支的日志，发现和之前没有两样。

```
$ git log -3 --oneline
c4acab2 Translate for Chinese.
683448a Add I18N support.
d81896e Fix typo: -help to --help.
```

（7）执行 StGit 补丁出栈的命令，会将补丁撤出应用。使用 -a 参数会将所有补丁撤出应用。

```
$ stg pop
Popped translate-for-chinese
Now at patch "add-i18n-support"
$ stg pop -a
Popped add-i18n-support -- fix-typo-help-to-help
No patch applied
```

（8）再来看看版本库的日志，会发现最新的三个提交都不见了。

```
$ git log -3 --oneline
10765a7 Bugfix: allow spaces in username.
0881ca3 Refactor: use getopt_long for arguments parsing.
ebcf6d6 blank commit for GnuPG-signed tag test.
```

（9）查看补丁列表的状态，会看到每个补丁前都用减号（－）标识。

```
$ stg ser
- fix-typo-help-to-help
- add-i18n-support
- translate-for-chinese
```

（10）执行补丁入栈，即应用补丁，使用命令 stg push 或 stg goto，注意 stg push 命令和 git push 命令风马牛不相及。

```
$ stg push
Pushing patch "fix-typo-help-to-help" ... done (unmodified)
Now at patch "fix-typo-help-to-help"
$ stg goto add-i18n-support
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Pushing patch "add-i18n-support" ... done (unmodified)
Now at patch "add-i18n-support"
```

（11）现在处于应用 add-i18n-support 补丁的状态。这个补丁有些问题，本地化语言模板有错误，我们来修改一下。

```
$ cd src/
$ rm locale/helloworld.pot
$ make po
xgettext -s -k_ -o locale/helloworld.pot main.c
msgmerge locale/zh_CN/LC_MESSAGES/helloworld.po locale/helloworld.pot -o locale/
temp.po
. 完成。
mv locale/temp.po locale/zh_CN/LC_MESSAGES/helloworld.po
```

（12）现在查看工作区，发现工作区有改动。

```
$ git status -s
 M locale/helloworld.pot
 M locale/zh_CN/LC_MESSAGES/helloworld.po
```

（13）不要提交，而是执行 stg refresh 命令更新补丁。

```
$ stg refresh
Now at patch "add-i18n-support"
```

（14）这时再查看工作区，发现本地修改不见了。

```
$ git status -s
```

（15）执行 stg show 会看到当前的补丁 add-i18n-support 已经更新。

```
$ stg show
...
```

（16）将最后一个补丁应用到版本库，遇到冲突。这是因为最后一个补丁是对中文本地化文件的翻译，因为翻译前的模板文件被更改了所以造成了冲突。

```
$ stg push
Pushing patch "translate-for-chinese" ... done (conflict)
Error: 1 merge conflict(s)
       CONFLICT (content): Merge conflict in
       src/locale/zh_CN/LC_MESSAGES/helloworld.po
Now at patch "translate-for-chinese"
```

（17）这个冲突文件很好解决，直接编辑冲突文件 helloworld.po 即可。编辑好之后，注意一下第 50 行和第 62 行是否像下面写的一样。

```
50 "    hello -h, --help\n"
51 "            显示本帮助页。\n"
...
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
61 msgid "Hi,"
62 msgstr "您好，"
```

（18）执行 git add 命令完成冲突解决。

```
$ git add locale/zh_CN/LC_MESSAGES/helloworld.po
```

（19）不要提交，而是使用 stg refresh 命令更新补丁，同时更新提交。

```
$ stg refresh
Now at patch "translate-for-chinese"
$ git status -s
```

（20）看看修改后的程序，是不是都能显示中文了。

```
$ ./hello
世界你好。
(version: v1.0-5-g733c6ea)
$ ./hello Jiang Xin
您好，Jiang Xin.
(version: v1.0-5-g733c6ea)
$ ./hello -h
...
```

（21）导出补丁，使用命令 stg export。导出的是 Quilt 格式的补丁集。

```
$ cd /path/to/my/workspace/stgit-demo/
$ stg export -d patches
Checking for changes in the working directory ... done
```

（22）看看导出补丁的目标目录。

```
$ ls patches/
add-i18n-support  fix-typo-help-to-help  series  translate-for-chinese
```

（23）其中文件 series 是补丁文件的列表，列在前面的补丁先被应用。

```
$ cat patches/series
# This series applies on GIT commit d81896e60673771ef1873b27a33f52df75f70515
fix-typo-help-to-help
add-i18n-support
translate-for-chinese
```

通过上面的演示可以看出 StGit 可以非常方便地对提交进行整理，整理提交时无须使用复杂的变基命令，而是采用提交 StGit 化、修改文件、执行 stg refresh 的工作流程即可更新补丁和提交。StGit 还可以将补丁导出为补丁文件，虽然导出的补丁文件没有像 git format-patch 那样加上代表顺序的数字前缀，但是用文件 series 标注了补丁文件的先后顺序。实际上可以在执行 stg export 时添加 -n 参数为补丁文件添加数字前缀。

StGit 还有一些功能，如合并补丁 / 提交，插入新补丁 / 提交等，请参照 StGit 帮助，恕不一一举例。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 20.3.2 Quilt

Quilt[①]是一款补丁列表管理软件，用Shell语言开发，安装也很简单，在Debian/Ubuntu上直接用下面的命令即可安装：

```
$ sudo aptitude install quilt
```

Quilt约定俗成将补丁集放在项目根目录下的子目录patches中，否则需要通过环境变量QUILT_PATCHES对路径进行设置。为了减少麻烦，在上面用stg export导出补丁的时候就导出到了patches目录下。

简单说一下Quilt的使用，会发现真的和StGit很像，实际上是先有的Quilt，后有的StGit。Quilt使用过程如下。

（1）重置到三个提交前的版本，否则应用补丁的时候会失败。不要忘了删除src/locale目录。

```
$ git reset --hard HEAD~3
$ rm -rf src/locale/
```

（2）显示补丁列表。

```
$ quilt series
01-fix-typo-help-to-help
02-add-i18n-support
03-translate-for-chinese
```

（3）应用一个补丁。

```
$ quilt push
Applying patch 01-fix-typo-help-to-help
patching file src/main.c

Now at patch 01-fix-typo-help-to-help
```

（4）下一个补丁是什么？

```
$ quilt next
02-add-i18n-support
```

（5）应用全部补丁。

```
$ quilt push -a
Applying patch 02-add-i18n-support
patching file src/Makefile
patching file src/locale/helloworld.pot
patching file src/locale/zh_CN/LC_MESSAGES/helloworld.po
```

① *http://savannah.nongnu.org/projects/quilt*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
patching file src/main.c

Applying patch 03-translate-for-chinese
patching file src/locale/zh_CN/LC_MESSAGES/helloworld.po

Now at patch 03-translate-for-chinese
```

Quilt 的功能还有很多，请参照 Quilt 的在线帮助，恕不一一举例。

Git 提供了一个名为 `git quiltimport` 的命令，可以非常方便地将 Quilt 格式的补丁集转化为一个一个的 Git 提交，是前面介绍的 `git am` 命令的一个补充。例如要将位于 `patches` 目录下的 Quilt 补丁集应用到版本库中，可以执行下面的命令：

```
$ git quiltimport
```





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第 4 篇
# Git 协同模型

分布式的版本控制会不会造成开发中的无序，导致版本管理的崩溃？习惯了如 Subversion 这类集中式版本控制系统的用户一定会有这个疑问。

作为分布式版本控制系统，每一个克隆都是一个完整的版本库，可以提供一个版本控制服务器所能提供的一切服务，即每个人的电脑都是一台服务器。与之相反，像 Subversion 那样的集中式版本控制系统，只拥有唯一的版本控制服务器，所有团队成员都使用客户端与之交互，大部分操作要通过网络传输来实现。对于习惯了和唯一服务器交互的团队，转换到 Git 后，该如何协同团队的工作呢？“第 21 章　经典 Git 协同模型”会介绍集中式和金字塔式两种主要的协同工作模型。

基于某个项目进行二次开发，需要使用不同的工作模型。原始的项目称为上游项目，不能直接在上游项目中提交，可能是因为授权的原因，或者是因为目标用户的需求不同。这种基于上游项目进行二次开发，实际上是对各个独特的功能分支进行管理，同时又能对上游项目的开发进度进行兼收并蓄式的合并。“第 22 章　Topgit 协同模型”会重点介绍这一方面的内容。

多个版本库组成一个项目，在实际应用中并不罕见。一部分原因可能是版本库需要依赖第三方的版本库，这时第 23 章介绍的“子模组协同模型”就可以派上用场了。有的时候还要对第三方的版本库进行定制，“第 24 章　子树合并”提供了一个解决方案。有的时候，为了管理方便（授权或项目确实太庞杂），多个版本库共同组成一个大的项目，例如 Google Android 项目就是由近 200 个版本库组成的。“第 25 章　Android 式多版本库协同”提供了一个非常有趣的解决方案，解决了“子模组协同模型”管理上的难题。

在本篇的最后（第 26 章），会介绍 git-svn 这一工具。可能因为公司对代码严格的授权要求，而不能将公司的版本控制服务器从 Subversion 迁移到 Git（实际可以通过对 Git 版本库细粒度拆分实现授权管理），可是这并不能阻止个人使用 git-svn 作为前端工具操作 Subversion 版本库。git-svn 可以让 Git 和 Subversion 完美地协同工作。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 21 章　经典 Git 协同模型

## 21.1　集中式协同模型

可以像集中式版本控制系统那样使用 Git，在一个大家都可以访问到的服务器上架设 Git 服务器，所有人都可以从该服务器克隆代码，将本地提交推送到服务器上，如图 21-1 所示。

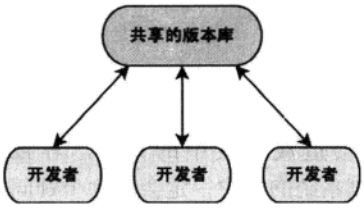



图 21-1　集中式协同模型

回忆一下在使用 Subversion 等集中式版本控制系统时，对服务器管理上的要求：

- 只允许拥有账号的用户访问版本库。
- 甚至只允许用户访问版本库中的某些路径，其他路径则不能访问。
- 特定目录只允许特定用户执行写操作。
- 服务器可以通过钩子实现特殊功能，如对 commit log 的检查、数据镜像等。

对于这些需求，Git 大部分都能支持，甚至能够做到更多：

- 可以设置谁能访问版本库，谁不能访问版本库。
- 具有更丰富的写操作授权。可以限制哪些分支不允许写，哪些路径不允许写。
- 可以设置谁能够创建新的分支。
- 可以设置谁能够创建新的版本库。
- 可以设置谁能够强制更新。
- 服务器端同样支持钩子脚本。

当然想要完整地实现上述的功能需求，需要使用特殊的服务器软件（如 Gitolite）来架设 Git 服务器，这也是通过 Git 实现传统集中式工作模型的核心。在本书第 5 篇介绍服务器部署的时候，会介绍如何用 Gitolite 架设 Git 服务器，以实现集中式协同模型对版本库授权和管理上的要求。

但是也要承认，在“读授权”上 Git 做不到很精细，这是分布式版本控制系统的机制使然。按模块分解 Git 版本库，并结合后面介绍的多版本库协同解决方案可以克服 Git 读授权

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的局限。

- Git 不支持对版本库读取的精确授权，只能是非 0 即 1 的授权。即或者能够读取一个版本库的全部，或者什么也读不到。
- 因为 Git 的提交是一个整体，提交中包含了完整目录树（tree）的哈希值，因此完整性不容破坏。
- Git 是分布式版本控制系统，如果允许不完整的克隆，那么本地就是截然不同的版本库，在向服务器推送的时候，会被拒绝，或者产生新的分支。

### 21.1.1 传统集中式协同模型

对于简单的代码修改，可以像传统集中式版本控制系统（Subversion）那样工作，参照图 21-2 所示的工作流程图。

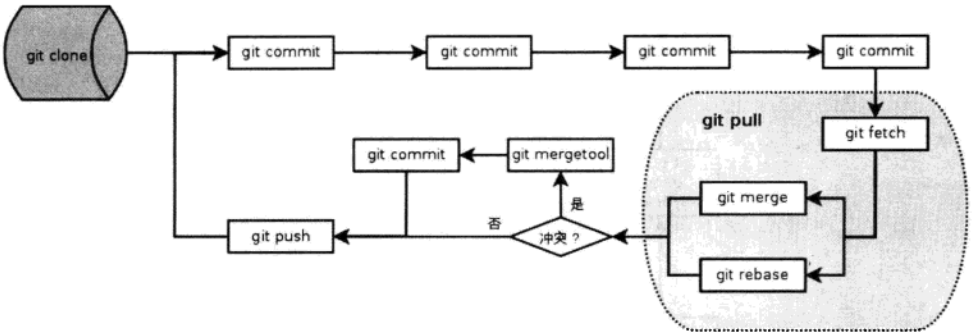



图 21-2 集中式协同模型工作流 1

但是对于复杂的修改（代码重构 / 增加复杂功能），这个工作模式就有些不合适了。

第一个问题是：很容易将不成熟的代码带入共享的版本库，破坏共享版本库相应分支的代码稳定性。例如破坏编译、破坏每日集成。这是因为开发者克隆版本库后，直接工作在默认的跟踪分支上，当不小心执行 `git push` 命令时，就会将自己的提交推送到服务器上。

为了避免上面的问题，开发者可能会延迟推送，在软件开发的整个过程中（例如一个月）只在本地提交，而不向服务器推送，这样会产生更严重的问题：数据丢失。开发者可能因为操作系统感染病毒，或者不小心删除目录，或者硬盘故障导致工作成果的彻底丢失，这对个人和团队来说都是灾难。

解决这个问题的方法也很简单，就是在本地创建本地分支（功能分支），并且同时在服务器端（共享版本库）也创建自己独享的功能分支。本地提交推送到共享版本库中自己独享的分支上。当开发完成之后，将功能分支合并到主线上，推送到共享版本库，完成开发。当然如果不再需要该特性分支时就要做些清理工作。参见图 21-3 所示的工作流程图。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

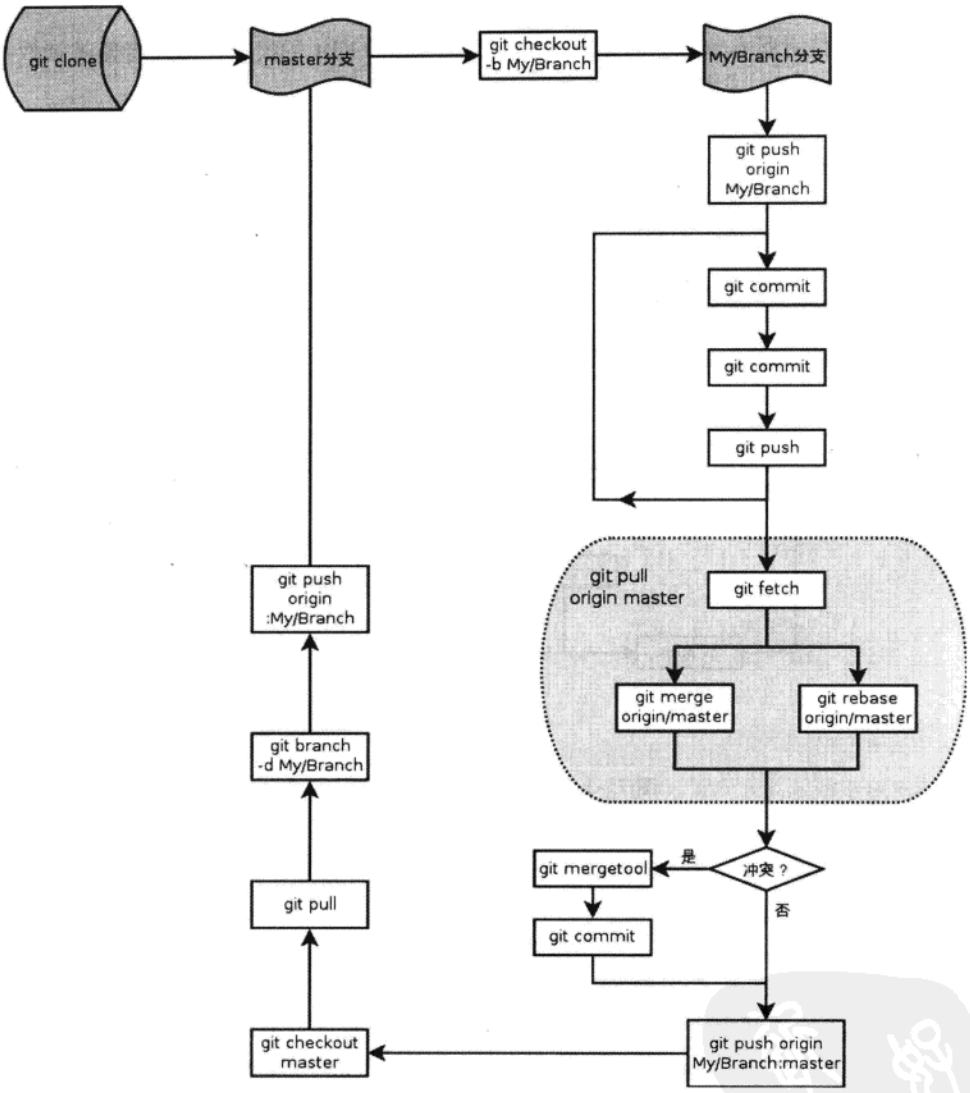




图 21-3 集中式协同模型工作流 2

## 21.1.2 Gerrit 特殊的集中式协同模型

### 1. 传统集中式协同模型的缺点

传统集中式协同模型的主要问题是在管理上：谁能够向版本库推送？可以信赖某人向版本库推送吗？

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


对于在一个相对固定的团队内部使用集中式协同模型没有问题，因为大家彼此信赖，都熟悉项目的相关领域。但是对于公开的项目（开源项目）来说，采用集中式的协同模型，必然只能有部分核心人员具有“写”权限，很多有能力的参与者会被拒之门外，这不利于项目的发展。因此集中式协同模型主要应用在公司范围内和商业软件开发中，而不会成为开源项目的首选。

**2. 强制代码审核的集中式协同模型**

Android 项目采用了独树一帜的集中式管理模型——通过 Gerrit 架设的审核服务器对提交进行强制审核。Android 是由近 200 个 Git 版本库组成的庞大的项目，为了对庞大的版本库进行管理，Android 项目开发了两个工具 repo 和 Gerrit 进行版本库的管理。其中 Gerrit 服务器为 Android 项目引入了特别的集中式协同模型。

Gerrit 服务器通过 SSH 协议管理 Git 版本库，并实现了一个 Web 界面的评审工作流。任何注册用户都可以参与到项目中来，都可以将 Git 提交推送到 Gerrit 管理下的 Git 版本库（通过 Gerrit 启动的特殊 SSH 端口）。Git 推送不能直接推送到分支，而是推送到特殊的引用 `refs/for/<branch-name>`，此提交会自动转换为形如 `refs/changes/<nn>/<review-id>/<patch-set>` 的补丁集，此补丁集在 Gerrit 的 Web 界面中显示为对应的评审任务。评审任务进入审核流程，通过相关负责人的审核后才被接受，并合并到正式的版本库中。

本书的第 5 篇第 32 章会详细介绍 Gerrit 代码审核服务器的部署和使用。

## 21.2 金字塔式协同模型

自从分布式版本库控制系统（Mercurial/Hg、Bazaar、Git 等）诞生之后，有越来越多的开源项目迁移了版本控制系统，例如从 Subversion 或 CVS 迁移到分布式版本控制系统。因为众多的开源项目逐渐认识到，集中式的版本控制管理方式阻止了更多的人参与项目的开发，对项目的发展不利。

集中式版本控制系统的最大问题是，如果没有在服务器端授权，就无法提交，也就无法保存自己的更改。开源项目虽然允许所有人访问代码库，但是不可能授权“写操作”给所有的人，否则代码质量无法控制（Gerrit 审核服务器是例外）。与此相对照的是，在使用了分布式版本控制系统之后，任何人都可以在本地克隆一个和远程版本库一模一样的版本库，本地的版本库允许任何操作，这就极大地调动了开发者投入项目研究的积极性。

分布式的开发必然带来协同的问题，如何能够让一个素不相识的开发者将他的贡献提交到项目中？如何能够最大化地发动和汇聚全球的智慧？开源社区逐渐发展出金字塔模型（如图 21-4 所示），而这也是必然之选。

金字塔模型的含义是，虽然理论上每个开发者的版本库都是平等的，但是会有一个公认的权威的版本库，这个版本库由一个或多个核心开发者负责维护（具有推送的权限）。核心的开发人员负责审核其他贡献者的提交，审核可以通过邮件传递的补丁或访问（PULL）贡献者开放的代码库进行。由此构成了以核心开发团队为顶层的、所有贡献者共同参与的开发



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


者金字塔。

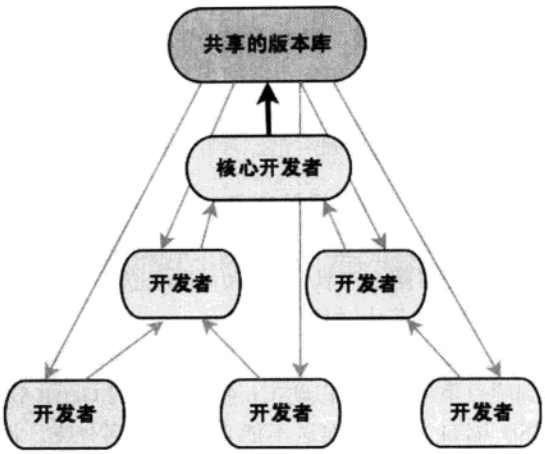



图 21-4 金字塔式协同模型

Linux 社区就是典型的金字塔结构。Linus Torvalds 的版本库是公认的官方版本库，允许核心成员的提交。其他贡献者的提交必须经过一个或多个核心成员的审核后，才能经由核心成员代为推送到官方版本库。

采用这种金字塔式协同模型不需要复杂的 Git 服务器设置，只需要项目管理者提供一个让其他人只读访问的版本库即可。当然管理者要能够通过某种方法向该版本库推送，以便其他人能够通过该版本库获得更新。

### 21.2.1 贡献者开放只读版本库

因为不能直接向项目只读共享的版本库提交，为了能让项目的管理者获取自己的提交，贡献者需要提供项目管理者访问自己版本库的方法。建立一个自己拥有的只读共享版本库是一个简单易行的方法。第 5 篇“搭建 Git 服务器”的相关章节会介绍几种快速搭建只读 Git 版本库的方法，包括：用 HTTP 智能协议搭建 Git 服务器，用 Git 协议搭建 Git 服务器。

贡献者建立自己的只读共享版本库后，需要检查和整理自己贡献的提交，检查项目如下：

- 贡献的提交要处于一个单独的特性分支中，并且要为该特性分支取一个有意义的名字。使用贡献者的名字及简单的概括性文字是非常好的特性分支名。例如对我来说可能创建名为 `jiangxin/fix-bug-xxx` 的分支。
- 贡献的提交是否基于上游对应分支的最新提交？如果不是，需要变基到上游最新提交，以免产生合并。

  项目的管理者会尽量避免不必要的合并，因此会要求贡献者的提交尽量基于项目的最新提交来进行。使用下面的方式建立跟踪远程分支的本地分支，可以很简单地实现在执行 `git pull` 操作时使用变基操作取代合并操作。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git checkout -b jiangxin/fix-bug-xxx origin/master
$ git config branch.jiangxin/fix-bug-xxx.rebase true
hack, hack, hack
$ git pull
```

然后贡献者就可以向项目管理者发送通知邮件，告诉项目管理者有新贡献的代码等待他的审核。邮件中大致包括以下内容：

- 为什么要修改项目的代码。
- 相应的修改是否经过了测试，或者提交中是否包含了单元测试。
- 自己版本库的访问地址。
- 特性分支的名称。

Git 提供了一个名为 `git request-pull` 的命令，可以非常方便地生成上述信息。

### 21.2.2 以补丁方式贡献代码

使用补丁文件方式贡献代码也是开源项目常用的协同方式。Git 项目本身就是采用该方式运作的。运作方式如下：

（1）每个用户先在本地版本库修改代码。

（2）修改完成后，通过执行 `git format-patch` 命令将提交转换为补丁。

（3）如果提交很多且比较杂乱，可以考虑使用 StGit 对提交进行重整。

（4）调用 `git send-email` 命令，或者通过图形界面的邮件客户端软件将补丁发给邮件列表及项目维护者。

（5）项目维护者认可贡献者提交的补丁后，执行 `git am` 命令应用补丁。

第 3 篇的“第 20 章 补丁文件交互”已经详细介绍了该模式的工作流程，请参考相关章节。





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 22 章 Topgit 协同模型

如果没有 Topgit[①]，就不会有此书。因为发现了 Topgit，我才下定决心在公司大范围推广 Git；因为 Topgit 才激发了我对 Git 的好奇之心。

## 22.1 作者版本控制系统的三个里程碑

### 1. Subversion 和卖主分支

从 2005 年开始我专心于开源软件的研究、定制开发和整合，在这之后的几年，一直使用 Subversion 做版本控制。对于定制开发工作，Subversion 有一种称为卖主分支（Vendor Branch）的模式。

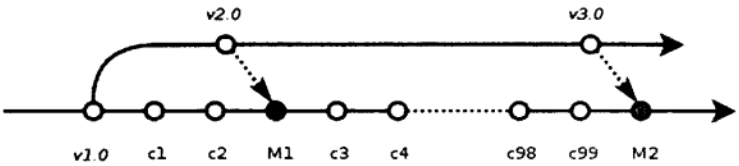



图 22-1 卖主分支工作模式图

卖主分支的工作模式如图 22-1 所示：

- 图 22-1 由左至右，提交随着时间而递增。
- 主线 trunk 用于对定制开发的过程进行跟踪。
- 主线的第一个提交 v1.0 是导入上游（该开源软件的官方版本库）发布的版本。
- 之后在 v1.0 提交之处建立分支，是为卖主分支（vendor branch）。
- 主线上依次进行了 c1、c2 两次提交，是基于 v1.0 进行的定制开发。
- 上游有了新版本，提交到卖主分支上，即 v2.0 提交。和 v1.0 相比除了大量的文件更改外，还可能有文件增加和删除。
- 然后在主线上执行从卖主分支到主线的合并，即提交 M1。因为此时主线上的改动相对较少，合并 v2.0 并不太费事。
- 主线继续开发。可能同时有针对不同需求的定制开发，在主线上会有越来越多的提交，如上图从 c3 到 c99 的近百次提交。

① *http://repo.or.cz/w/topgit.git*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 如果在卖主分支上导入上游的新版本 v3.0，合并将会非常痛苦。因为主线上针对不同需求的定制开发已经混杂在一起了！

实践证明，Subversion 的卖主分支对于大规模的定制开发非常不适合。向上游新版本的迁移会随着定制功能和提交的增多变得越来越困难。

### 2. Hg 和 MQ

在 2008 年，我们的版本库迁移到 Mercurial（水银，又称为 Hg），并工作在“Hg + MQ”模式下，我自以为找到了定制开发版本控制的终极解决方案，那时我们已被 Subversion 的卖主分支折磨得太久了。

Hg 和 Git 一样也是一种分布式版本控制系统，MQ 是 Hg 的一个扩展，可以实现提交和补丁两种模式之间的转换。Hg 版本库上的提交可以通过 `hg qimport` 命令转化为补丁列表，也可以通过 `hg qpush`、`hg qpop` 等命令在补丁列表上游移（出栈和入栈），入栈的补丁转化为 Hg 版本库的提交，补丁出栈会从 Hg 版本库移走最新的提交。

相比 Subversion 的卖主分支，使用“Hg + MQ”的好处在于：

- 针对不同需求的定制开发，其提交被限定在各自独立的补丁文件中而不会混杂在一起。
- 针对同一个需求的定制开发，无论多少次的更改都体现为补丁文件的变化，而补丁文件本身也是被版本控制的。
- 各个补丁之间是顺序依赖关系，形成一个 Quilt 格式的补丁列表。
- 向上游新版本迁移过程的工作量降低了，除了因为针对各个功能的定制开发被隔离，还有迁移过程也非常具有可操作性。

将定制开发迁移至上游新版本的过程是：先将所有补丁“出栈”，再将上游新版本提交到主线，然后依次将补丁“入栈”。如果上游新版本的代码改动较大，补丁入栈可能会遇到冲突，在冲突解决完毕后执行 `hg qref` 命令即可完成定制开发到新的上游版本的迁移。

但是如果需要在定制开发上进行多人协作，“Hg + MQ”的弊病就显现了。因为“Hg + MQ”工作模式下，定制开发的成果是一个补丁库，在补丁库上进行协作的难度非常大，当发生冲突的时候，补丁文件本身的冲突会让人眼花缭乱。这就引发了我们第三次版本控制系统大迁移。

### 3. Git 和 Topgit

2009 年，我们把目光锁定在 Topgit 上。Topgit 是 Topic Git 的简写，是用 shell 脚本语言开发的辅助工具，对 Git 功能进行了扩展，用于管理特性分支的开发。Topgit 为特性分支引入了基准分支的概念，并以此管理特性分支间的依赖，让特性分支向上游新版本的迁移变得非常简单。

Topgit 的主要特点有：

- 上游的原始代码位于开发主线（`master` 分支），而每一个定制开发都对应于一条 Git 特性分支（`refs/heads/t/feature_name`）。
- 特性分支之间的依赖关系不像“Hg + MQ”那样是逐一依赖模式，而是可以任意设定



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


分支之间的依赖：多重依赖、平行依赖等。

- 特性分支和其依赖的分支可以导出为 Quilt 格式的补丁列表。
- 因为针对某一需求的定制开发限定在特定的特性分支中，可以多人协同参与，和正常的 Git 开发别无二致。

## 22.2 Topgit 原理

图 22-2 是一个近似的 Topgit 实现图（略去了重要的 top-bases 分支）。

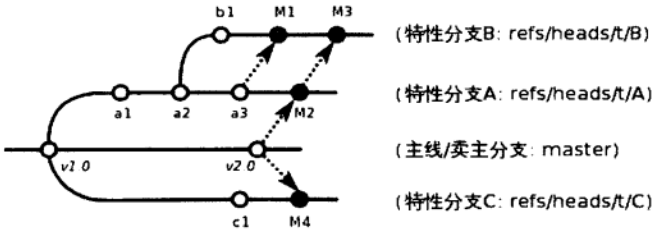



图 22-2 Topgit 特性分支关系图

在图 22-2 中，主线上的 v1.0 是上游版本的一次提交。特性分支 A 和 C 都直接依赖主线 master，而特性分支 B 则依赖特性分支 A。提交 M1 是特定分支 B 因为特性分支 A 更新而做的一次迁移。提交 M2 和 M4 则分别是特性分支 A 和 C 因为上游出现了新版本 v2.0 而做的迁移。当然特性分支 B 也要做相应的迁移，是为 M3。

上述的图示非常粗糙，因为如果按照这样的设计很难将特性分支导出为补丁文件。例如特性分支 B 导出为补丁，实际上应该是 M2 和 M3 之间的差异，而绝不是 a2 和 M3 之间的差异。Topgit 为了能够实现将分支导出为补丁，又为每个特性的开发引入了一个特殊的引用 (refs/top-bases/*)，用于追踪特性分支的“变基”，称为特性分支的基准分支。所有特性分支的基准分支也形成了复杂的分支关系图，如图 22-3 所示。

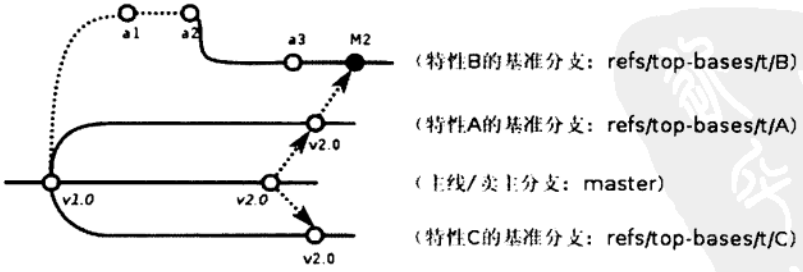



图 22-3 Topgit 特性分支的基准分支关系图

把图 22-2 和图 22-3 两张分支图重合，重合点之间的差异就可用于将特性分支导出为补

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


丁文件。

上面的特性分支 B 还只是依赖一个分支，如果出现一个分支依赖多个特性分支的话，情况就会更加复杂，也更能体现出这种设计方案的精妙。

Topgit 还在每个特性分支工作区的根目录引入两个文件，用于记录分支的依赖及关于此分支的说明：

- 文件 .topdeps 记录该分支所依赖的分支列表。
  该文件通过 tg create 命令在创建特性分支时自动创建，或者通过 tg depend add 命令来添加新依赖。
- 文件 .topmsg 记录该分支的描述信息。
  该文件通过 tg create 命令在创建特性分支时创建，可以手动编辑。

## 22.3 Topgit 的安装

Topgit 用 shell 脚本语言开发，可以安装在所有的类 Unix 环境中，例如 Linux、Mac OS X，以及 Windows 下的 Cygwin。下面的官方网站链接介绍了 Topgit 的安装和使用方法：*http://repo.or.cz/w/topgit.git?a=blob;f=README*。

### 1. Linux 下安装 Topgit

安装官方的 Topgit 版本，直接克隆官方的版本库，执行 make 即可：

```
$ git clone git://repo.or.cz/topgit.git
$ cd topgit
$ make
$ make install
```

默认会把 Topgit 的可执行文件 tg 安装在 $HOME/bin（用户主目录下的 bin 目录）下，如果没有将 ~/bin 加入环境变量 $PATH 中，就可能无法执行 tg 命令。

如果具有 root 权限，可以在编译和安装时向 make 命令传递 prefix 变量，将 Topgit 安装在系统目录中。

```
$ make prefix=/usr
$ sudo make prefix=/usr install
```

我对 Topgit 做了一些增强和改进[①]，在后面的章节将予以介绍。如果想安装改进后的版本，需要预先安装 Quilt 补丁管理工具，然后进行如下操作。

```
$ git clone git://github.com/ossxp-com/topgit.git
$ cd topgit
$ QUILT_PATCHES=debian/patches quilt push -a
$ make prefix=/usr
```

① 我对 Topgit 的改进采用了 Topgit 的开发模式，如果大家发现我的改动没有及时地跟上上游代码，用户可以自行使用 Topgit 将改动迁移到最新的上游版本。还要提醒的是，不要把我的错误算到上游开发者头上。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ sudo make prefix=/usr install
```

如果用的是 Ubuntu 或 Debian Linux 操作系统，还可以这么安装。

（1）先安装 Debian/Ubuntu 打包依赖的相关工具软件。

```
$ sudo aptitude install quilt debhelper \
                        build-essential fakeroot dpkg-dev
```

（2）再调用 dpkg-buildpackage 命令，编译出 DEB 包，再安装。

```
$ git clone git://github.com/ossxp-com/topgit.git
$ cd topgit
$ dpkg-buildpackage -b -rfakeroot
$ sudo dpkg -i ../topgit_*.deb
```

（3）安装完毕后，重新加载命令行补齐，可以更方便地使用 tg 命令。

```
$ . /etc/bash_completion
```

### 2. Mac OS X 下安装 Topgit

在 Mac OS X 下安装官方版本的 Topgit，在使用中会遇到各种各样的问题。这是因为 Mac OS X 下部分 shell 命令的行为和相应的 GNU 命令的行为不一致，例如 echo、paste 和 sed 命令等。

在 Mac OS X 下可以使用 Homebrew 安装所需的 GNU 工具。如下：

```
$ brew install gnu-sed
$ brew install quilt
```

然后别忘了安装改造后的 Topgit。

```
$ git clone git://github.com/ossxp-com/topgit.git
$ cd topgit
$ QUILT_PATCHES=debian/patches quilt push -a
$ make prefix=/usr
$ sudo make prefix=/usr install
```

### 3. Windows 下安装 Topgit

Windows 下的 msysGit 因为缺乏 Topgit 依赖的命令行工具（如 fgrep、install、make、mkfifo、mktemp、tsort 等），安装和运行 Topgit 会遇到困难。从安装好的 MSYS[①] 或 MSYS-CN[②] 中提取所需要的软件到 msysGit 环境中，可以实现 Topgit 在 msysGit 中的安装和运行。

Windows 下的 Cygwin 拥有一个完整的 POSIX 环境，当安装了所需的工具（make、quilt[③] 等）后，就可以正常地编译和使用 Topgit。

---

① *http://www.mingw.org/wiki/msys*

② *http://code.google.com/p/msys-cn/*

③ 如果要安装改进后的 Topgit。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


注意如果克隆 Topgit 版本库后工作区文件的换行符是 DOS 格式换行符（CRLF），在安装过程中会遇到麻烦。克隆改进的 Topgit 则不会出现类似问题，这是因为在工作区根目录下存在一个 .gitattributes[①] 文件，可以保证检出的工作区文件采用 Unix 格式的换行符（LF）。

在 Cygwin 下安装改进后的 Topgit 使用如下方法：

```
$ git clone git://github.com/ossxp-com/topgit.git
$ cd topgit
$ QUILT_PATCHES=debian/patches quilt push -a
$ make prefix=/usr
$ make prefix=/usr install
```

## 22.4 Topgit 的使用

通过前面的原理部分，可以发现 Topgit 为管理特性分支所引入的配置文件和基准分支都是和 Git 兼容的。

- 在 refs/top-bases/ 命名空间下的引用，用于记录特性分支的基准分支。
- 在特性分支的工作区根目录下引入两个文件 .topdeps 和 .topmsg，用于记录分支的依赖和说明。
- 引入新的钩子脚本 hooks/pre-commit，用于在提交时检查分支依赖有没有发生循环等。

Topgit 的命令行的一般格式为：

```
tg [global_option] <subcmd> [command_options...] [arguments...]
```

说明：在子命令前的全局选项，目前可用的只有 -r <remote>，用于设定远程版本库，默认为 origin。在子命令后可以跟子命令相关的参数。

下面就来介绍 Topgit 常用的子命令。

### 1. tg help 命令

tg help 命令显示帮助信息。在 tg help 后面提供子命令名称，可以获得该子命令详细的帮助信息。

### 2. tg create 命令

tg create 命令用于创建新的特性分支。用法如下：

```
tg [...] create NAME [DEPS...|-r RNAME]
```

其中：

- NAME 是新的特性分支的分支名，必须提供。一般约定俗成：NAME 以 t/ 前缀开头的表明此分支是一个 Topgit 特性分支。
- DEPS... 是可选的一个或多个依赖分支名。如果不提供依赖分支名，则使用当前分

① 参见第 8 篇第 40 章“40.3 换行符问题”。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


支作为新的特性分支的依赖分支。

- -r RNAME 选项，将远程分支作为依赖分支，不常用。

tg create 命令会创建新的特性分支 refs/heads/NAME，以及特性分支的基准分支 refs/top-bases/NAME，并且在项目根目录下创建文件 .topdeps 和 .topmsg。还会提示用户编辑 .topmsg 文件，输入详细的特性分支描述信息。

为了试验 Topgit 命令，找一个示例版本库或干脆创建一个版本库。在示例版本库的 master 分支下输入如下命令创建一个名为 t/feature1 的特性分支：

```
$ tg create t/feature1
tg: Automatically marking dependency on master
tg: Creating t/feature1 base from master...
Switched to a new branch 't/feature1'
tg: Topic branch t/feature1 set up. Please fill .topmsg now and make initial commit.
tg: To abort: git rm -f .top* && git checkout master && tg delete t/feature1
```

提示信息中以“tg:”开头的是 Topgit 产生的说明。其中提示用户编辑 .topmsg 文件，然后执行一次提交操作完成 Topgit 特性分支的创建。

如果想撤销此次操作，删除项目根目录下的 .top* 文件，切换到 master 分支，然后执行 tg delete t/feature1 命令删除 t/feature1 分支及特性分支的基准分支 refs/top-bases/t/feature1。

输入 git status 可以看到当前已经切换到 t/feature1 分支，并且 Topgit 已经创建了 .topdeps 和 .topmsg 文件，并已将这两个文件加入到暂存区。

```
$ git status
# On branch t/feature1
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .topdeps
#       new file:   .topmsg
#
$ cat .topdeps
master
```

打开 .topmsg 文件，会看到下面的内容（前面增加了行号）：

```
1   From: Jiang Xin <jiangxin@ossxp.com>
2   Subject: [PATCH] t/feature1
3
4   <patch description>
5
6   Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
```

其中第 2 行是关于该特性分支的简短描述，第 4 行是详细描述，可以写多行。编辑完成，别忘了提交，提交之后才完成 Topgit 分支的创建。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git add -u
$ git commit -m "create tg branch t/feature1"
```

如果这时想创建一个新的特性分支 t/feature2，并且也是要依赖 master，注意需要在命令行中提供 master 作为第二个参数，以设定依赖分支。因为当前所处的分支为 t/feature1，如果不提供指定的依赖分支就会自动依赖当前分支。

```
$ tg create t/feature2 master
$ git commit -m "create tg branch t/feature2"
```

下面的命令将创建 t/feature3 分支，该分支依赖 t/feature1 和 t/feature2。

```
$ tg create t/feature3 t/feature1 t/feature2
$ git commit -m "create tg branch t/feature3"
```

### 3. tg info 命令

tg info 命令用于显示当前分支或指定的 Topgit 分支的信息。用法如下：

```
tg [...] info [NAME]
```

其中 NAME 是可选的 Topgit 分支名。例如执行下面的命令会显示分支 t/feature3 的信息：

```
$ tg info
Topic Branch: t/feature3 (1/1 commit)
Subject: [PATCH] t/feature3
Base: 0fa79a5
Depends: t/feature1
         t/feature2
Up-to-date.
```

切换到 t/feature1 分支，做一些修改并提交：

```
$ git checkout t/feature1
$ echo Hi > hacks.txt
$ git add hacks.txt
$ git commit -m "hacks in t/feature1."
```

然后再来看 t/feature3 的状态：

```
$ tg info t/feature3
Topic Branch: t/feature3 (1/1 commit)
Subject: [PATCH] t/feature3
Base: 0fa79a5
Depends: t/feature1
         t/feature2
Needs update from:
        t/feature1 (1/1 commit)
```

状态信息显示 t/feature3 不再是最新的状态（Up-to-date），因为依赖的 t/feature1 分支包含新的提交，而需要从 t/feature1 获取更新。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


#### 4. tg update 命令

tg update 命令用于更新分支，即从依赖的分支获取最新的提交合并到当前分支。同时在 refs/top-bases/ 命名空间下的特性分支的基准分支也会更新。

```
tg [...] update [NAME]
```

其中 NAME 是可选的 Topgit 分支名。下面就对需要更新的 t/feature3 分支执行 tg update 命令。

```
$ git checkout t/feature3
$ tg update
tg: Updating base with t/feature1 changes...
Merge made by recursive.
 hacks.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 feature1
tg: Updating t/feature3 against new base...
Merge made by recursive.
 hacks.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 feature1
```

从上面的输出信息可以看出执行了两次分支合并操作，一次是针对 refs/top-bases/t/feature3 引用指向的特性分支的基准分支，另外一次针对的是 refs/heads/t/feature3 特性分支。

执行 tg update 命令因为要涉及分支的合并，因此并非每次都会成功。例如在 t/feature3 和 t/feature1 中同时对同一个文件（如 hacks.txt）进行修改。然后在 t/feature3 中再执行 tg update 可能就会报错，进入冲突解决状态。

```
$ tg update t/feature3
tg: Updating base with t/feature1 changes...
Merge made by recursive.
 hacks.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
tg: Updating t/feature3 against new base...
Auto-merging hacks.txt
CONFLICT (content): Merge conflict in hacks.txt
Automatic merge failed; fix conflicts and then commit the result.
tg: Please commit merge resolution. No need to do anything else
tg: You can abort this operation using `git reset --hard` now
tg: and retry this merge later using `tg update`.
```

可以看出第一次对 refs/top-bases/t/feature3 引用指向的特性分支的基准分支合并成功，但对 t/feature3 特性分支进行的合并出错了。

执行 tg info 命令查看一下当前分支 t/feature3 的状态。

```
$ tg info
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Topic Branch: t/feature3 (3/2 commits)
Subject: [PATCH] t/feature3
Base: 37dcb62
* Base is newer than head! Please run 'tg update'.
Depends: t/feature1
         t/feature2
Up-to-date.
```

从上面 tg info 命令的输出可以看出当前分支的状态是 Up-to-date，但是输出中包含一个提示：分支的基（Base）要比头（Head）新，请执行 tg update 命令。

这时如果执行 tg summary 命令的话，可以看到 t/feature3 处于 B（Break）状态。

```
$ tg summary
        t/feature1                      [PATCH] t/feature1
 0      t/feature2                      [PATCH] t/feature2
>     B t/feature3                      [PATCH] t/feature3
```

执行 git status 命令，可以看出因为两个分支同时修改了文件 hacks.txt 而导致冲突。

```
$ git status
# On branch t/feature3
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      hacks.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

可以编辑 hacks.txt 文件，或者调用冲突解决工具解决冲突，之后再提交，这才真正完成了此次 tg update。

```
$ git mergetool
$ git commit -m "resolved conflict with t/feature1."
$ tg info
Topic Branch: t/feature3 (4/2 commits)
Subject: [PATCH] t/feature3
Base: 37dcb62
Depends: t/feature1
         t/feature2
Up-to-date.
```

### 5. tg summary 命令

tg summary 命令用于显示 Topgit 管理的特性分支的列表及各个分支的状态。用法如下：

```
tg [...] summary [-t | --sort | --deps | --graphviz]
```

不带任何参数执行 tg summary 是最常用的 Topgit 命令。在介绍不带参数的 tg summary 命令之前，先看看该命令的其他用法。

使用 -t 参数显示特性分支列表。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ tg summary -t
t/feature1
t/feature2
t/feature3
```

使用 --deps 参数除了显示 Topgit 特性分支外，还会显示特性分支的依赖分支。

```
$ tg summary  --deps
t/feature1 master
t/feature2 master
t/feature3 t/feature1
t/feature3 t/feature2
```

使用 --sort 参数按照分支依赖的顺序显示分支列表，除了显示 Topgit 分支外，还会显示依赖的非 Topgit 分支。

```
$ tg summary  --sort
t/feature3
t/feature2
t/feature1
master
```

使用 --graphviz 会输出 GraphViz 格式文件，可以用于显示特性分支之间的关系。

```
$ tg summary --graphviz | dot -T png -o topgit.png
```

生成的特性分支关系图如图 22-4 所示。

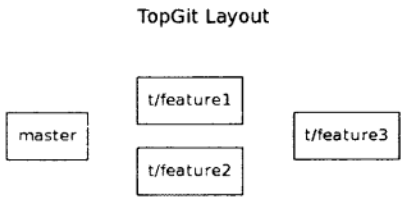



图 22-4 Topgit 特性分支依赖关系图

不带任何参数执行 tg summary 显示分支列表及状态。这是最常用的 Topgit 命令之一。

```
$ tg summary
        t/feature1                      [PATCH] t/feature1
 0      t/feature2                      [PATCH] t/feature2
>       t/feature3                      [PATCH] t/feature3
```

其中：

- 标识“>”：(t/feature3 分支之前的大于号）用于标记当前所处的特性分支。
- 标记“0”：(t/feature2 分支前的数字 0）含义是该分支中没有提交，这是一个建立后尚未使用或废弃的分支。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 标记“D”：表明该分支处于过时（out-of-date）状态。可能是一个或多个依赖的分支包含了新的提交，尚未合并到此特性分支。可以用 tg info 命令看出到底是由于哪个依赖分支的改动导致该特性分支处于过时状态。
- 标记“B”：之前演示中出现过，表明该分支处于 Break 状态，即可能由于冲突未解决或其他原因导致该特性分支的基准分支（base）相对该分支的头（head）不匹配。例如 refs/top-bases 下的特性分支的基准分支迁移了，但是特性分支未完成迁移。
- 标记“!”：表明该特性分支所依赖的分支不存在。
- 标记“l”：表明该特性分支只存在于本地，不存在于远程跟踪服务器。
- 标记“r”：表明该特性分支既存在于本地，又存在于远程跟踪服务器，并且两者匹配。
- 标记“L”：表明该特性分支，本地的比远程跟踪服务器的新。
- 标记“R”：表明该特性分支，远程跟踪服务器的比本地的新。
- 如果没有出现“l/r/L/R”：表明该版本库尚未设置远程版本库。
- 一般带有标记“r”的是最常见的，也是最正常的。

下面要介绍的 tg remote 命令为测试版本库建立一个对应的远程版本库，然后就能在 tg summary 的输出中看到“l/r/L/R”等标识符了。

### 6. tg remote 命令

tg remote 命令用于为远程版本库增加 Topgit 的相关设置，以便在和该远程版本库进行 git fetch、git pull 等操作时能够同步 Topgit 的相关分支。命令用法如下：

```
tg [...] remote [--populate] [REMOTE]
```

其中 REMOTE 为远程版本库的名称，默认为 origin。执行 tg remote 命令会自动在版本的配置文件中增加的 refs/top-bases 下引用同步表达式。下面的示例中的最后一行就是执行 tg remote origin 后增加的设置。

```
[remote "origin"]
        url = /path/to/repos/tgtest.git
        fetch = +refs/heads/*:refs/remotes/origin/*
        fetch = +refs/top-bases/*:refs/remotes/origin/top-bases/*
```

如果使用 --populate 参数，除了会向上面那样设置默认的 Topgit 远程版本库外，还会自动执行 git fetch 命令，然后在本地建立 Topgit 特性分支和对应的基准分支。

当执行 tg 命令时，如果不用 -r remote 全局参数，则默认使用 origin 远程版本库。

下面为前面测试的 Topgit 版本库设置一个远程版本库，具体操作过程如下。

（1）先创建一个裸版本库 tgtest.git。

```
$ git init --bare /path/to/repos/tgtest.git
Initialized empty Git repository in /path/to/repos/tgtest.git/
```

（2）然后执行 git remote 命令，将刚刚创建的版本库以 origin 为名注册为远程版本库。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git remote add origin /path/to/repos/tgtest.git
```

（3）执行 git push，将当前版本库的 master 分支推送到刚刚创建的远程版本库。

```
$ git push origin master
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (7/7), 585 bytes, done.
Total 7 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
To /path/to/repos/tgtest.git
 * [new branch]      master -> master
```

（4）之后运行 tg remote 命令为远程版本库添加额外的配置，以便该远程版本库能够跟踪 Topgit 分支。

```
$ tg remote --populate origin
```

（5）执行了上面的 tg remote 命令后，会在当前版本库的 .git/config 文件中添加设置（以加号开头的行）：

```
 [remote "origin"]
        url = /path/to/repos/tgtest.git
        fetch = +refs/heads/*:refs/remotes/origin/*
+       fetch = +refs/top-bases/*:refs/remotes/origin/top-bases/*
+[topgit]
+       remote = origin
```

（6）这时再执行 tg summary 会看到分支前面都有标记“l”，即本地提交比远程版本库的新。

```
$ tg summary
  l     t/feature1                      [PATCH] t/feature1
 0l     t/feature2                      [PATCH] t/feature2
> l     t/feature3                      [PATCH] t/feature3
```

（7）执行 tg push 命令将特性分支 t/feature2 及其基准分支推送到远程版本库。

```
$ tg push t/feature2
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 457 bytes, done.
Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
To /path/to/repos/tgtest.git
 * [new branch]      t/feature2 -> t/feature2
 * [new branch]      refs/top-bases/t/feature2 -> refs/top-bases/t/feature2
```

（8）再来看看 tg summary 的输出，会看到 t/feature2 的标识变为“r”，即远程和本地同步。

```
$ tg summary
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
  l     t/feature1                    [PATCH] t/feature1
 0r     t/feature2                    [PATCH] t/feature2
> l     t/feature3                    [PATCH] t/feature3
```

（9）运行 tg push --all[①]，会将所有的 Topgit 分支推送到远程版本库。之后再来看 tg summary 的输出，会看到所有分支都带上了“r”的标识。

```
$ tg push --all
...
$ tg summary
  r     t/feature1                    [PATCH] t/feature1
 0r     t/feature2                    [PATCH] t/feature2
> r     t/feature3                    [PATCH] t/feature3
```

如果版本库设置了多个远程版本库，要针对每一个远程版本库执行 tg remote <REMOTE>，但只能有一个远程的源用 --populate 参数调用 tg remote 将其设置为默认的远程版本库。

### 7. tg push 命令

在前面 tg remote 的介绍中，已经看到过 tg push 命令。tg push 命令用于将 Topgit 特性分支及对应的基准分支推送到远程版本库。用法如下：

```
tg [...] push [--dry-run] [--no-deps] [--tgish-only] [--all|branch*]
```

tg push 命令后面的参数指定要推送给远程服务器的分支列表，如果省略则推送当前分支。改进的 tg push 命令支持通过 --all 参数将所有 Topgit 特性分支推送到远程版本库。

参数 --dry-run 用于测试推送的执行效果，而不是真正执行。参数 --no-deps 的含义是不推送依赖的分支，默认推送。参数 --tgish-only 的含义是只推送 Topgit 特性分支，默认推送指定的所有分支。

### 8. tg depend 命令

tg depend 命令目前仅实现了为当前的 Topgit 特性分支增加新的依赖。用法如下：

```
tg [...] depend add NAME
```

将 NAME 加入到文件 .topdeps 中，并将 NAME 分支向该特性分支及特性分支的基准分支进行合并操作。虽然 Topgit 可以检查到分支的循环依赖，但还是要注意合理地设置分支的依赖，合并重复的依赖。

### 9. tg base 命令

tg base 命令用于显示特性分支的基准分支的提交 ID（精简格式）。

① 改进后的 Topgit 才提供同步全部特性分支的功能。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 10. tg delete 命令

tg delete 命令用于删除 Topgit 特性分支及其对应的基准分支。用法如下：

```
tg [...] delete [-f] NAME
```

默认只删除没有改动的分支，即标记为“0”的分支，除非使用 -f 参数。

目前此命令尚不能自动清除其分支中对删除分支的依赖，还需要手工调整 .topdeps 文件，删除对不存在的分支的依赖。

### 11. tg patch 命令

tg patch 命令通过比较特性分支及其基准分支的差异，显示该特性分支的补丁。用法如下：

```
tg [...] patch [-i | -w] [NAME]
```

其中 -i 参数显示暂存区和基准分支的差异。-w 参数显示工作区和基准分支的差异。

tg patch 命令存在的一个问题是只能正确显示工作区中的根执行。这个缺陷已经在我改进的 Topgit 中被改正[①]。

### 12. tg export 命令

tg export 命令用于导出特性分支及其依赖，便于向上游贡献。可以导出 Quilt 格式的补丁列表，或者顺序提交到另外的分支中。用法如下：

```
tg [...] export ([--collapse] NEWBRANCH | [--all | -b BRANCH1,BRANCH2...] --quilt
DIRECTORY | --linearize NEWBRANCH)
```

这个命令有三种导出方法。

- 将所有的 Topgit 特性分支压缩为一个，提交到新的分支。

```
tg [...] export --collapse NEWBRAQNCH
```

- 将所有的 Topgit 特性分支按照线性顺序提交到一个新的分支中。

```
tg [...] export --linearize NEWBRANCH
```

- 将指定的 Topgit 分支（一个或多个）及其依赖分支转换为 Quilt 格式的补丁，保存到指定目录中。

```
tg [...] export -b BRANCH1,BRANCH2... --quilt DIRECTORY
```

在导出为 Quilt 格式补丁的时候，如果想将所有的分支导出，必须用 -b 参数将分支全部罗列（或者分支的依赖关系将所有分支囊括），这对于需要导出所有分支的操作非常不方便。我改进的 Topgit 通过 --all 参数实现了所有分支的导出。

### 13. tg import 命令

tg import 命令将分支的提交转换为 Topgit 特性分支，指定范围的每个提交都转换为

① 最新的 Topgit 重构了 tg-push 的代码，改正了这个缺陷，所以会看到最新改进版的 Topgit 该特性分支为空。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


一个特性分支，各个特性分支之间线性依赖。用法如下：

```
tg [...] import [-d BASE_BRANCH] {[-p PREFIX] RANGE...|-s NAME COMMIT}
```

如果不使用 -d 参数，特性分支则以当前分支为依赖。特性分支名称自动生成，使用约定俗成的 t/ 作为前缀，也可以通过 -p 参数指定其他前缀。可以通过 -s 参数设定特性分支的名称。

### 14. tg log 命令

tg log 命令显示特性分支的提交历史，并忽略合并引入的提交。

```
tg [...] log [NAME] [-- GIT LOG OPTIONS...]
```

tg log 命令实际是对 git log 命令的封装。这个命令通过 --no-merges 和 --first-parent 参数调用 git log，虽然屏蔽了大量因和依赖分支合并而引入的依赖分支的提交日志，但是同时也屏蔽了合并到该特性分支的其他贡献者的提交。

### 15. tg mail 命令

tg mail 命令将当前分支或指定特性分支的补丁以邮件的形式外发。用法如下：

```
tg [...] mail [-s SEND_EMAIL_ARGS] [-r REFERENCE_MSGID] [NAME]
```

tg mail 调用 git send-email 发送邮件，参数 -s 用于向该命令传递参数（需要用双引号括起来）。邮件中的目的地址从补丁文件头中的 To、Cc 和 Bcc 等字段获取。参数 -r 引用回复邮件的 ID 以便正确地生成 in-reply-to 邮件头。

注意：此命令可能会发送多封邮件，可以通过如下设置在调用 git send-email 命令发送邮件时进行确认。

```
$ git config sendemail.confirm always
```

### 16. tg files 命令

tg files 命令用于显示当前或指定的特性分支改动了哪些文件。

### 17. tg prev 命令

tg prev 命令用于显示当前或指定的特性分支所依赖的分支名。

### 18. tg next 命令

tg next 命令用于显示当前或指定的特性分支被其他哪些特性分支所依赖。

### 19. tg graph 命令

tg graph 命令并非官方提供的命令，而是源自一个补丁[①]，实现文本方式的 Topgit 分支图。当然这个文本分支图没有 tg summary --graphviz 生成的那么漂亮。

① *http://kerneltrap.org/mailarchive/git/2009/5/20/2922*



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 22.5 用 Topgit 方式改造 Topgit

在 Topgit 的使用中陆续发现了一些不合用的地方，于是便以 Topgit 特性分支的方式来对 Topgit 进行改造。在群英汇的博客上介绍了如下几个改造：

- TopGit 改进（1）：tg push 全部分支[①]
- TopGit 改进（2）：tg 导出全部分支[②]
- TopGit 改进（3）：更灵活的 tg patch[③]
- TopGit 改进（4）：tg 命令补齐[④]
- TopGit 改进（5）：tg summary 执行的更快[⑤]

下面就以 Topgit 改造过程为例，来介绍如何参与一个 Topgit 管理下的项目的开发。改造后的 Topgit 版本库地址为：`git://github.com/ossxp-com/topgit.git`。

首先克隆该版本库：

```
$ git clone git://github.com/ossxp-com/topgit.git
$ cd topgit
```

查看远程分支：

```
$ git branch -r
origin/HEAD -> origin/master
origin/master
origin/t/debian_locations
origin/t/export_quilt_all
origin/t/fast_tg_summary
origin/t/graphviz_layout
origin/t/tg_completion_bugfix
origin/t/tg_graph_ascii_output
origin/t/tg_patch_cdup
origin/t/tg_push_all
origin/tgmaster
```

看到远程分支中出现了熟悉的以 `t/` 为前缀的 Topgit 分支，说明这个版本库是一个由 Topgit 进行管理的版本库。为了能够获取 Topgit 各个特性分支的基准分支，需要用 `tg remote` 命令对默认的 `origin` 远程版本库注册一下。

```
$ tg remote --populate origin
tg: Remote origin can now follow TopGit topic branches.
tg: Populating local topic branches from remote 'origin'...
```

① *http://blog.ossxp.com/2010/01/247/*
② *http://blog.ossxp.com/2010/01/255/*
③ *http://blog.ossxp.com/2010/01/257/*
④ *http://blog.ossxp.com/2010/01/259/*
⑤ *http://blog.ossxp.com/2010/01/261/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
From git://github.com/ossxp-com/topgit
 * [new branch]      refs/top-bases/t/debian_locations ->
origin/top-bases/t/debian_locations
 * [new branch]      refs/top-bases/t/export_quilt_all ->
origin/top-bases/t/export_quilt_all
 * [new branch]      refs/top-bases/t/fast_tg_summary ->
origin/top-bases/t/fast_tg_summary
 * [new branch]      refs/top-bases/t/graphviz_layout ->
origin/top-bases/t/graphviz_layout
 * [new branch]      refs/top-bases/t/tg_completion_bugfix ->
origin/top-bases/t/tg_completion_bugfix
 * [new branch]      refs/top-bases/t/tg_graph_ascii_output ->
origin/top-bases/t/tg_graph_ascii_output
 * [new branch]      refs/top-bases/t/tg_patch_cdup ->
origin/top-bases/t/tg_patch_cdup
 * [new branch]      refs/top-bases/t/tg_push_all ->
origin/top-bases/t/tg_push_all
tg: Adding branch t/debian_locations...
tg: Adding branch t/export_quilt_all...
tg: Adding branch t/fast_tg_summary...
tg: Adding branch t/graphviz_layout...
tg: Adding branch t/tg_completion_bugfix...
tg: Adding branch t/tg_graph_ascii_output...
tg: Adding branch t/tg_patch_cdup...
tg: Adding branch t/tg_push_all...
tg: The remote 'origin' is now the default source of topic branches.
```

执行 `tg summary` 查看一下本地 Topgit 特性的分支状态。

```
$ tg summary
  r  !  t/debian_locations              [PATCH] make file locations Debian-compatible
  r  !  t/export_quilt_all              [PATCH] t/export_quilt_all
  r  !  t/fast_tg_summary               [PATCH] t/fast_tg_summary
  r  !  t/graphviz_layout               [PATCH] t/graphviz_layout
  r  !  t/tg_completion_bugfix          [PATCH] t/tg_completion_bugfix
  r     t/tg_graph_ascii_output         [PATCH] t/tg_graph_ascii_output
  r  !  t/tg_patch_cdup                 [PATCH] t/tg_patch_cdup
  r  !  t/tg_push_all                   [PATCH] t/tg_push_all
```

怎么？出现了感叹号？记得前面在介绍 `tg summary` 命令的章节中提到过，感叹号的出现说明该特性分支所依赖的分支丢失了。用 `tg info` 查看一下其中的某个特性分支。

```
$ tg info t/export_quilt_all
Topic Branch: t/export_quilt_all (6/4 commits)
Subject: [PATCH] t/export_quilt_all
Base: 8b0f1f9
Remote Mate: origin/t/export_quilt_all
Depends: tgmaster
MISSING: tgmaster
Up-to-date.
```

原来该特性分支依赖 `tgmaster` 分支，而不是 `master` 分支。远程存在 `tgmaster` 分

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


支而本地尚不存在。于是在本地建立 tgmaster 跟踪分支。

```
$ git checkout tgmaster①
Branch tgmaster set up to track remote branch tgmaster from origin.
Switched to a new branch 'tgmaster'
```

这回 tg summary 的输出正常了。

```
$ tg summary
  r     t/debian_locations          [PATCH] make file locations Debian-compatible
  r     t/export_quilt_all          [PATCH] t/export_quilt_all
  r     t/fast_tg_summary           [PATCH] t/fast_tg_summary
  r     t/graphviz_layout           [PATCH] t/graphviz_layout
  r     t/tg_completion_bugfix      [PATCH] t/tg_completion_bugfix
  r     t/tg_graph_ascii_output     [PATCH] t/tg_graph_ascii_output
  r     t/tg_patch_cdup             [PATCH] t/tg_patch_cdup
  r     t/tg_push_all               [PATCH] t/tg_push_all
```

通过下面的命令创建图形化的分支图。

```
$ tg summary --graphviz | dot -T png -o topgit.png
```

生成的特性分支关系图如图 22-5 所示。

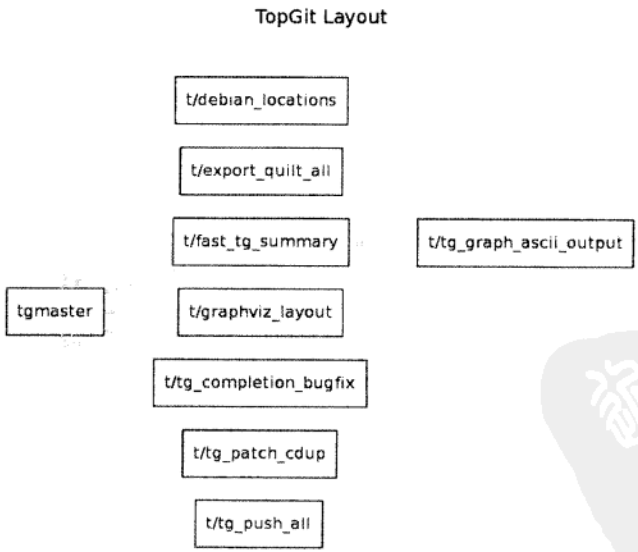



图 22-5 Topgit 改进项目的特性分支依赖关系图

① 如果 Git 版本号小于 1.6.6，则需要执行 git checkout -b tgmaster origin/tgmaster 命令以创建 tgmaster 分支。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


其中：

- 特性分支 t/export_quilt_all，为 tg export --quilt 命令增加 --all 选项，以便导出所有的特性分支。
- 特性分支 t/fast_tg_summary，主要是改进 tg 命令补齐时分支的显示速度，当特性分支接近上百个时差异非常明显。
- 特性分支 t/graphviz_layout，改进了分支的图形输出格式。
- 特性分支 t/tg_completion_bugfix，解决了命令补齐的一个 Bug。
- 特性分支 t/tg_graph_ascii_output，源自 Bert Wesarg 的贡献，非常巧妙地实现了文本化的分支图显示，展示了 gvpr 命令的强大功能。
- 特性分支 t/tg_patch_cdup，解决了在项目的子目录下无法执行 tg patch 的问题。
- 特性分支 t/tg_push_all，通过为 tg push 增加 --all 选项，解决了当 tg 从 0.7 版升级到 0.8 版后，无法批量向上游推送特性分支的问题。

下面展示一下如何跟踪上游的最新改动，并迁移到新的上游版本。分支 tgmaster 用于跟踪上游的 Topgit 分支，以 t/ 开头的分支是对 Topgit 改进的特性分支，而 master 分支实际上是导出 Topgit 补丁文件并负责编译特定 Linux 平台发行包的分支。具体操作过程如下：

（1）把官方的 Topgit 版本库以 upstream 的名称加入作为新的远程版本库。

```
$ git remote add upstream git://repo.or.cz/topgit.git
```

（2）然后将 upstream 远程版本的 master 分支合并到本地的 tgmaster 分支。

```
$ git pull upstream master:tgmaster
From git://repo.or.cz/topgit
   29ab4cf..8b0f1f9  master     -> tgmaster
```

（3）此时再执行 tg summary 会发现所有的 Topgit 分支都多了一个标记 D，表明因为依赖分支的更新而导致 Topgit 特性分支过时了。

```
$ tg summary
  r D   t/debian_locations                [PATCH] make file locations Debian-compatible
  r D   t/export_quilt_all                [PATCH] t/export_quilt_all
  r D   t/fast_tg_summary                 [PATCH] t/fast_tg_summary
  r D   t/graphviz_layout                 [PATCH] t/graphviz_layout
  r D   t/tg_completion_bugfix            [PATCH] t/tg_completion_bugfix
  r D   t/tg_graph_ascii_output           [PATCH] t/tg_graph_ascii_output
  r D   t/tg_patch_cdup                   [PATCH] t/tg_patch_cdup
  r D   t/tg_push_all                     [PATCH] t/tg_push_all
```

（4）依次对各个分支执行 tg update，完成对更新的依赖分支的合并。

```
$ tg update t/export_quilt_all
...
```

（5）对各个分支完成更新后，会发现 tg summary 的输出中，标识过时的 D 标记变为

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


L，即本地比远程服务器分支要新。

```
$ tg summary
  rL    t/debian_locations                [PATCH] make file locations Debian-compatible
  rL    t/export_quilt_all                [PATCH] t/export_quilt_all
  rL    t/fast_tg_summary                 [PATCH] t/fast_tg_summary
  rL    t/graphviz_layout                 [PATCH] t/graphviz_layout
  rL    t/tg_completion_bugfix            [PATCH] t/tg_completion_bugfix
  rL    t/tg_graph_ascii_output           [PATCH] t/tg_graph_ascii_output
  rL    t/tg_patch_cdup                   [PATCH] t/tg_patch_cdup
  rL    t/tg_push_all                     [PATCH] t/tg_push_all
```

（6）执行 tg push --all 就可以实现将所有的 Topgit 特性分支推送到远程服务器上，当然需要有提交权限才可以。

## 22.6 Topgit 使用中的注意事项

### 1. 经常运行 tg remote --populate 获取他人创建的特性分支

运行命令 git fetch 或命令 git pull 和远程版本库同步，只能将他人创建的 Topgit 特性分支在本地以远程分支（refs/remotes/origin/t/<branch-name>）的方式保存，而不能自动在本地建立分支。

如果确认版本库是使用 Topgit 维护的话，应该在和远程版本库同步的时候执行 tg remote --populate origin。这条命令会做两件事情：

- 自动调用 git fetch origin 获取远程 origin 版本库的新的提交和引用。
- 检查 refs/remotes/origin/top-bases/ 下的所有引用，如果是新的，在本地（refs/top-bases/）尚不存在，说明有其他人创建了新的特性分支。Topgit 会据此自动在本地创建新的特性分支。

### 2. 适时地调整特性分支的依赖关系

例如之前用于 Topgit 演示的版本库，各个特性分支的依赖文件内容如下。

- 分支 t/feature1 的 .topdeps 文件

```
master
```

- 分支 t/feature2 的 .topdeps 文件

```
master
```

- 分支 t/feature3 的 .topdeps 文件

```
t/feature1
t/feature2
```

如果分支 t/feature3 的 .topdeps 文件是这样的，可能就会存在问题。

```
master
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
t/feature1
t/feature2
```

问题在于 t/feature3 依赖的其他分支已经依赖了 master 分支，虽然不会造成致命的影响，但是在特定情况下这种重复会造成不便。例如在 master 分支更新后，可能由于代码重构的比较厉害，在特性分支迁移时会造成冲突，如在 t/feature1 分支中执行 tg update 会遇到冲突，当辛苦完成冲突解决并提交后，在 t/feature3 中执行 tg update 会因为先依赖的是 master 分支，所以先在 master 分支上对 t/feature3 分支进行合并，这样就肯定会遇到和 t/feature1 相同的冲突，还要再重复解决一次。

如果在 .topdeps 文件中删除了对 master 分支的重复的依赖，就不会出现上面的重复解决冲突的问题了。

同样的道理，如果 t/feature3 的 .topdeps 文件写成这样，效果也将不同：

```
t/feature2
t/feature1
```

依赖的顺序不同会造成合并的顺序也不同，同样也会产生重复的冲突解决。因此当发现重复的冲突时，可以取消合并操作，修改特性分支的 .topdeps 文件，调整文件内容（删除重复分支，调整分支顺序）并提交，然后再执行 tg update 继续合并操作。

### 3. Topgit 特性分支的里程碑和分支管理

Topgit 本身就是管理特性分支的软件。Topgit 某个时刻的开发状态是 Topgit 管理下的所有分支（包括基准分支）整体的状态。思考一下：能够用里程碑来标记 Topgit 管理的版本库的开发状态吗？

使用里程碑来管理是不可能的，因为 Git 里程碑只能针对一个分支做标记而不能标记所有的分支。使用克隆是唯一的方法。克隆不但用于标记 Topgit 版本库的开发状态，也可以用于 Topgit 版本库的分支管理。例如一旦上游出现新版本，就从当前版本库建立一个克隆，原来的版本库用于维护原有上游版本的定制开发，新的克隆版本库针对新的上游版本进行迁移，用于新的上游版本的特性开发。

也许还可以通过其他方法实现，例如将 Topgit 所有相关的分支都复制到一个特定的引用目录中或记录在文件中，以实现特性分支的状态记录。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 23 章　子模组协同模型

项目的版本库在某些情况下需要引用其他版本库中的文件，例如公司积累了一套常用的函数库，被多个项目调用，显然这个函数库的代码不能直接放到某个项目的代码中，而是要独立为一个代码库，那么其他项目要调用公共的函数库该如何处理呢？分别把公共函数库的文件拷贝到各自的项目中会造成冗余，丢弃了公共函数库的维护历史，这显然不是好的方法。本章要讨论的子模组协同模型，就是解决这个问题的一个方案。

熟悉 Subversion 的用户马上会想起 `svn:externals` 属性可以实现对外部代码库的引用。Git 的子模组（submodule）是类似的一种实现。不过因为 Git 的特殊性，二者的区别还是蛮大的，参见表 23-1。

表 23-1　SVN 和 Git 相似功能对照表

| | svn:externals | git submodule |
|---|---|---|
| 如何记录外部版本库的地址 | 目录的 `svn:externals` 属性 | 项目根目录下的 `.gitmodules` 文件 |
| 默认是否自动检出外部版本库 | 是<br>在使用 `svn checkout` 检出时，若使用参数 `--ignore-externals` 可以忽略对外部版本库的引用，不检出 | 否<br>默认不克隆外部版本库。若要克隆则用 `git submodule init` 及 `git submodule update` 命令 |
| 是否能部分引用外部版本库内容 | 是<br>因为 SVN 支持部分检出 | 否<br>必须克隆整个外部版本库 |
| 是否可以指向分支而随之改变 | 是 | 否<br>固定于外部版本库的某个提交 |

## 23.1　创建子模组

在演示子模组的创建和使用之前，先做些准备工作。先尝试建立两个公共函数库（libA.git 和 libB.git），以及一个引用函数库的主版本库（super.git）。

```
$ git --git-dir=/path/to/repos/libA.git init --bare
$ git --git-dir=/path/to/repos/libB.git init --bare
$ git --git-dir=/path/to/repos/super.git init --bare
```

向两个公共的函数库中填充些数据。这就需要在工作区克隆两个函数库，提交数据并推送。

克隆 libA.git 版本库，添加一些数据，然后提交并推送。说明：示例中显示为 `hack...` 的地方做了一些改动（如创建新文件等），并将改动添加到暂存区。

```
$ git clone /path/to/repos/libA.git /path/to/my/workspace/libA
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cd /path/to/my/workspace/libA
hack ...
$ git commit -m "add data for libA"
$ git push origin master
```

克隆 libB.git 版本库，添加一些数据，然后提交并推送。

```
$ git clone /path/to/repos/libB.git /path/to/my/workspace/libB
$ cd /path/to/my/workspace/libB
hack ...
$ git commit -m "add data for libB"
$ git push origin master
```

版本库 super 是准备在其中创建子模组的。super 版本库刚刚初始化还未包含提交，master 分支尚未有正确的引用。需要在 super 版本库中至少创建一个提交。下面就克隆 super 版本库，在其中完成一个提交（空提交即可），并推送。

```
$ git clone /path/to/repos/super.git /path/to/my/workspace/super
$ cd /path/to/my/workspace/super
$ git commit --allow-empty -m "initialized."
$ git push origin master
```

现在就可以在 super 版本库中使用 `git submodule add` 命令添加子模组了。

```
$ git submodule add /path/to/repos/libA.git lib/lib_a
$ git submodule add /path/to/repos/libB.git lib/lib_b
```

至此看一下 super 版本库工作区的目录结构。在根目录下多了一个 `.gitmodules` 文件，并且两个函数库分别被克隆到 `lib/lib_a` 目录和 `lib/lib_b` 目录下。

```
$ ls -aF
./  ../  .git/  .gitmodules  lib/
```

看看 `.gitmodules` 的内容：

```
$ cat .gitmodules
[submodule "lib/lib_a"]
        path = lib/lib_a
        url = /path/to/repos/libA.git
[submodule "lib/lib_b"]
        path = lib/lib_b
        url = /path/to/repos/libB.git
```

此时 super 的工作区尚未提交。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
#       new file:   .gitmodules
#       new file:   lib/lib_a
#       new file:   lib/lib_b
#
```

完成提交之后，子模组才算正式在 super 版本库中创立。运行 git push 把建立了新模组的本地版本库推送到远程版本库。

```
$ git commit -m "add modules in lib/lib_a and lib/lib_b."
$ git push
```

在提交过程中，发现作为子模组方式添加的版本库实际上并没有添加版本库的内容。实际上只是以 gitlink 的方式添加了一个链接。至于子模组的实际地址，是由文件 .gitmodules 指定的。

可以通过查看补丁的方式看到 lib/lib_a 和 lib/lib_b 子模组的存在方式（即 gitlink）。

```
$ git show HEAD

commit 19bb54239dd7c11151e0dcb8b9389e146f055ba9
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Fri Oct 29 10:16:59 2010 +0800

    add modules in lib/lib_a and lib/lib_b.

diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..60c7d1f
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,6 @@
+[submodule "lib/lib_a"]
+       path = lib/lib_a
+       url = /path/to/repos/libA.git
+[submodule "lib/lib_b"]
+       path = lib/lib_b
+       url = /path/to/repos/libB.git
diff --git a/lib/lib_a b/lib/lib_a
new file mode 160000
index 0000000..126b181
--- /dev/null
+++ b/lib/lib_a
@@ -0,0 +1 @@
+Subproject commit 126b18153583d9bee4562f9af6b9706d2e104016
diff --git a/lib/lib_b b/lib/lib_b
new file mode 160000
index 0000000..3b52a71
--- /dev/null
+++ b/lib/lib_b
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
@@ -0,0 +1 @@
+Subproject commit 3b52a710068edc070e3a386a6efcbdf28bf1bed5
```

## 23.2 克隆带子模组的版本库

之前的表 23-1 在对比 Subversion 的 svn:externals 属性和 Git 子模组实现差异时，提到过克隆带子模组的 Git 库，并不能自动将子模组的版本库克隆出来。对于只关心项目本身的数据，而不关心项目引用的外部项目数据的用户，这个功能非常好，数据没有冗余而且克隆的速度也更快。

下面在另外的位置克隆 super 版本库，会发现 lib/lib_a 和 lib/lib_b 并未克隆。

```
$ git clone /path/to/repos/super.git /path/to/my/workspace/super-clone
$ cd /path/to/my/workspace/super-clone
$ ls -aF
./  ../  .git/  .gitmodules  lib/
$ find lib
lib
lib/lib_a
lib/lib_b
```

这时如果运行 git submodule status 可以查看到子模组的状态。

```
$ git submodule status
-126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a
-3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b
```

可以看到，每个子模组的目录前面都是 40 位的提交 ID，最前面的是一个减号。减号的含义是该子模组尚未检出。

如果需要克隆出子模组形式引用的外部库，首先需要执行 git submodule init。

```
$ git submodule init
Submodule 'lib/lib_a' (/path/to/repos/libA.git) registered for path 'lib/lib_a'
Submodule 'lib/lib_b' (/path/to/repos/libB.git) registered for path 'lib/lib_b'
```

执行 git submodule init 实际上修改了 .git/config 文件，对子模组进行了注册。文件 .git/config 的修改示例如下（以加号开始的行代表新增的行）。

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        url = /path/to/repos/super.git
[branch "master"]
        remote = origin
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
          merge = refs/heads/master
+[submodule "lib/lib_a"]
+       url = /path/to/repos/libA.git
+[submodule "lib/lib_b"]
+       url = /path/to/repos/libB.git
```

然后执行 `git submodule update` 完成子模组版本库的克隆。

```
$ git submodule update
Initialized empty Git repository in
/path/to/my/workspace/super-clone/lib/lib_a/.git/
Submodule path 'lib/lib_a': checked out
'126b18153583d9bee4562f9af6b9706d2e104016'
Initialized empty Git repository in
/path/to/my/workspace/super-clone/lib/lib_b/.git/
Submodule path 'lib/lib_b': checked out
'3b52a710068edc070e3a386a6efcbdf28bf1bed5'
```

## 23.3 在子模组中修改和子模组的更新

执行 `git submodule update` 更新出来的子模组，都以某个具体的提交版本进行检出。进入某个子模组目录，会发现其处于非跟踪状态（分离头指针状态）。

```
$ cd /path/to/my/workspace/super-clone/lib/lib_a
$ git branch
* (no branch)
  master
```

显然这种情况下，如果修改 `lib/lib_a` 下的文件，提交就会丢失。下面介绍一下如何在检出的子模组中修改，以及如何更新子模组。

在子模组中切换到 master 分支（或其他想要修改的分支）后再进行修改。

（1）切换到 master 分支，然后在工作区做一些改动。

```
$ cd /path/to/my/workspace/super-clone/lib/lib_a
$ git checkout master
hack ...
```

（2）执行提交。

```
$ git commit
```

（3）查看状态，会看到相对于远程分支领先一个提交。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


在 git status 的状态输出中，可以看出新提交尚未推送到远程版本库。现在暂时不推送，看看在 super 版本库中执行 git submodule update 对子模组的影响，具体操作过程如下。

（4）先到 super-clone 版本库查看一下状态，可以看到子模组已修改，包含了更新的提交。

```
$ cd /path/to/my/workspace/super-clone/
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   lib/lib_a (new commits)
#
no changes added to commit (use "git add" and/or "git commit -a")
```

（5）通过 git submodule stauts 命令可以看出 lib/lib_a 子模组指向了新的提交 ID（前面有一个加号），而 lib/lib_b 子模组状态正常（提交 ID 前是一个空格，不是加号也不是减号）。

```
$ git submodule status
+5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9 lib/lib_a (heads/master)
 3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

（6）这时如果不小心执行了一次 git submodule update 命令，会将 lib/lib_a 重新切换到旧的指向。

```
$ git submodule update
Submodule path 'lib/lib_a': checked out '126b18153583d9bee4562f9af6b9706d2e104016'
```

（7）执行 git submodule status 命令查看子模组状态，可以看到 lib/lib_a 子模组被重置了。

```
$ git submodule status
 126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)
 3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

那么刚才在 lib/lib_a 中的提交丢失了么？实际上因为已经提交到了 master 主线，因此提交没有丢失，但是如果有数据没有提交，就会造成未提交数据的丢失。

（1）进到 lib/lib_a 目录，看到工作区再一次进入分离头指针状态。

```
$ cd lib/lib_a
$ git branch
* (no branch)
  master
```

（2）重新检出 master 分支找回之前的提交。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git checkout master
Previous HEAD position was 126b181... add data for libA
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
```

现在如果要将 lib/lib_a 目录下子模组的改动记录到父项目（super 版本库）中，就需要在父项目中进行一次提交才能实现。

（1）进入父项目根目录查看状态。因为 lib/lib_a 的提交已经恢复，因此再次显示为有改动。

```
$ cd /path/to/my/workspace/super-clone/
$ git status -s
 M lib/lib_a
```

（2）查看差异比较，会看到指向子模组的 gitlink 有改动。

```
$ git diff
diff --git a/lib/lib_a b/lib/lib_a
index 126b181..5dea269 160000
--- a/lib/lib_a
+++ b/lib/lib_a
@@ -1 +1 @@
-Subproject commit 126b18153583d9bee4562f9af6b9706d2e104016
+Subproject commit 5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9
```

（3）将 gitlink 的改动添加到暂存区，然后提交。

```
$ git add -u
$ git commit -m "submodule lib/lib_a upgrade to new version."
```

此时先不要忙着推送，因为如果此时执行 git push 将 super 版本库推送到远程版本库，会引发一个问题。即推送后的远程 super 版本库的子模组 lib/lib_a 指向了一个新的提交，而该提交还在本地的 lib/lib_a 版本库（尚未向上游推送），这会导致其他人克隆 super 版本库和更新模组时因为找不到该子模组版本库相应的提交而出错。下面就是这类错误的信息：

```
fatal: reference is not a tree: 5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9
Unable to checkout '5dea2693e5574a6e3b3a59c6b0c68cb08b2c07e9' in submodule path
'lib/lib_a'
```

为了避免这种可能性的发生，最好先推送 lib/lib_a 中的新提交，然后再向 super 版本库推送更新的子模组 gitlink 改动。即：

（1）先推送子模组。

```
$ cd /path/to/my/workspace/super-clone/lib/lib_a
$ git push
```

（2）再推送父版本库。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cd /path/to/my/workspace/super-clone/
$ git push
```

## 23.4 隐性子模组

我在开发备份工具 Gistore[①] 时遇到一个棘手的问题，就是隐性子模组的问题。Gistore 备份工具的原理是将要备份的目录都挂载（mount）在工作区中，然后执行 git add。但是如果有某个目录已经被 Git 化了，就只会以子模组的方式将该目录添加进来，而不会添加该目录下的文件。对于一个备份工具来说，这就意味着备份没有成功。具体操作过程如下：

（1）例如当前 super 版本库下有两个子模组：

```
$ cd /path/to/my/workspace/super-clone/
$ git submodule status
 126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)
 3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

（2）然后创建一个新目录 others，并把该目录用 Git 初始化，并做一次空的提交。

```
$ mkdir others
$ cd others
$ git init
$ git commit --allow-empty -m initial
[master (root-commit) 90364e1] initial
```

（3）在 others 目录下创建一个文件 newfile。

```
$ date > newfile
```

（4）回到上一级目录执行 git status，看到有一个 others 目录没有加入版本库控制，这很自然。

```
$ cd ..
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       others/
nothing added to commit but untracked files present (use "git add" to track)
```

（5）但是如果对 others 目录执行 git add 后，会发现奇怪的状态。

```
$ git add others
```

① 参见本书第 7 篇“第 37 章 Gistore”。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   others
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#   (commit or discard the untracked or modified content in submodules)
#
#       modified:   others (untracked content)
#
```

（6）看看 others 目录的添加方式，就会发现 others 目录以 gitlink 的方式添加到版本库中，而没有把该目录下的文件添加到版本库中。

```
$ git diff --cached
diff --git a/others b/others
new file mode 160000
index 0000000..90364e1
--- /dev/null
+++ b/others
@@ -0,0 +1 @@
+Subproject commit 90364e1331abc29cc63e994b4d2cfbf7c485e4ad
```

之所以上面的步骤（5）运行 git status 命令时 others 出现了两次，就是因为目录 others 被当作子模组添加到了父版本库中，而且由于 others 版本库本身“不干净”，存在尚未加入版本控制的文件，所以又在状态输出中显示了子模组包含改动的提示信息。

接下来执行提交，将 others 目录提交到版本库中。然后当执行 git submoudle status 命令时会报错。因为 others 作为子模组没有在 .gitmodules 文件中注册。

```
$ git commit -m "add others as submodule."
$ git submodule status
 126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)
 3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
No submodule mapping found in .gitmodules for path 'others'
```

那么如何在不破坏 others 版本库的前提下，把 others 目录下的文件加入版本库呢？即避免 others 以子模组形式添加入库，具体操作过程如下。

（1）先删除以 gitlink 形式入库的 others 子模组。

```
$ git rm --cached others
rm 'others'
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（2）查看当前状态。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    others
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       others/
```

（3）重新添加 others 目录，注意目录后面的斜线（即路径分隔符）非常重要。

```
$ git add others/
```

（4）再次查看状态，看到 others 下的文件被添加到 super 版本库中。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    others
#       new file:   others/newfile
#
```

（5）执行提交。

```
$ git commit -m "add contents in others/."
[master 1e0c418] add contents in others/.
 2 files changed, 1 insertions(+), 1 deletions(-)
 delete mode 160000 others
 create mode 100644 others/newfile
```

在上面的操作过程中，首先删除了库中的 others 子模组（使用 --cached 参数执行删除）；然后为了添加 others 目录下的文件使用了“others/”（注意 others 后面的路径分割符“/”）。现在查看一下子模组的状态，会看到只显示出了之前的两个子模组。

```
$ git submodule status
 126b18153583d9bee4562f9af6b9706d2e104016 lib/lib_a (remotes/origin/HEAD)
 3b52a710068edc070e3a386a6efcbdf28bf1bed5 lib/lib_b (heads/master)
```

## 23.5 子模组的管理问题

子模组最主要的一个问题是不能基于外部版本库的某一个分支进行创建，而使得更新

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


后子模组处于非跟踪状态，不便于在子模组中进行代码修改、提交和向外部推送。尤其对于因授权或其他原因将一个版本库拆分为子模组后，使用和管理非常不方便。在“第 25 章 Android 式多版本库协同”一章中可以看到管理多版本库的另外一个可行方案。

如果在局域网内维护的版本库所引用的子模组版本库存在于另外的服务器上，甚至互联网上，克隆子版本库就要浪费很多时间。而且如果子模组指向的版本库不在我们的掌控之内，一旦需要对其进行定制，就会因为无法向远程服务器推送提交而无法实现定制。下一章即“第 24 章 子树合并”会给出针对这个问题的解决方案。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第24章 子树合并

使用子树合并，同样可以实现在一个项目中引用其他项目的数据。但是和子模组方式不同的是，使用子树合并模式，外部的版本库会作为一个目录被整个复制到本版本库中，并且复制到本版本库中的子目录下的数据可以和原版本库数据建立跟踪关联。使用子树合并模式，对源自外部版本库的数据的访问和对本版本库数据的访问没有区别，也可以对其进行本地修改，并且能够以子树合并的方式将外部版本库的新的改动和本地的修改相合并。

## 24.1 引入外部版本库

为演示子树合并至少需要准备两个版本库，一个是将要被作为子目录引入的版本库util.git，另外一个是主版本库main.git。

```
$ git --git-dir=/path/to/repos/util.git init --bare
$ git --git-dir=/path/to/repos/main.git init --bare
```

在本地检出这两个版本库：

```
$ cd /path/to/my/workspace
$ git clone /path/to/repos/util.git
$ git clone /path/to/repos/main.git
```

需要为这两个空版本库添加些数据。非常简单，每个版本库下只创建两个文件：Makefile和version。当执行make命令时显示version文件的内容。对version文件多次提交以建立多个提交历史。别忘了在最后使用git push origin master将版本库推送到远程版本库中。

Makefile文件示例如下。注意第二行前面的空白是<TAB>字符，而非空格。

```
all:
	@cat version
```

在之前尝试的git fetch命令都是获取同一项目的版本库的内容。其实命令git fetch从哪个项目获取数据并没有什么限制，因为Git的版本库不像Subversion那样用一个唯一的UUID标识让Subversion的版本库之间势同水火。当然也可以用git pull来获取其他版本库中的提交，但是那样将把两个项目的文件彻底混杂在一起。对于这个示例来说，因为两个项目具有同样的文件Makefile和version，使用git pull将导致冲突。所以为了将不同项目的版本库引入，并在稍后以子树合并的方式添加到一个子目录中，需要用

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


git fetch 命令从其他版本库获取数据，具体操作过程如下。

（1）为了便于以后对外部版本库的跟踪，在使用 git fetch 前，先在 main 版本库中注册远程版本库 util.git。

```
$ cd /path/to/my/workspace/main
$ git remote add util /path/to/repos/util.git
```

（2）查看注册的远程版本库。

```
$ git remote -v
origin  /path/to/repos/main.git/ (fetch)
origin  /path/to/repos/main.git/ (push)
util    /path/to/repos/util.git (fetch)
util    /path/to/repos/util.git (push)
```

（3）执行 git fetch 命令获取 util.git 版本库的提交。

```
$ git fetch util
```

（4）查看分支，包括远程分支。

```
$ git branch -a
* master
  remotes/origin/master
  remotes/util/master
```

在不同的分支：master 分支和 remotes/util/master 分支中，文件 version 的内容并不相同，因为来自不同的上游版本库。

- 在 master 分支中执行 make 命令，显示的是 main.git 版本库中 version 文件的内容。

```
$ make
main v2010.1
```

- 从 util/master 远程分支创建一个本地分支 util-branch，并切换分支。

```
$ git checkout -b util-branch util/master
Branch util-branch set up to track remote branch master from util.
Switched to a new branch 'util-branch'
```

- 执行 make 命令，显示的是 util.git 版本库中 version 文件的内容。

```
$ make
util v3.0
```

像这样在 main.git 中引入 util.git 显然不能满足需要，因为在 main.git 的本地克隆版本库中，master 分支访问不到只有在 util-branch 分支中才出现的 util 版本库数据。这就需要做进一步的工作，将两个版本库的内容合并到一个分支中。即将 util-branch 分支的数据作为子目录加入到 master 分支中。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 24.2 子目录方式合并外部版本库

下面就用Git的底层命令git read-tree、git write-tree和git commit-tree子命令，实现将util-branch分支所包含的util.git版本库的目录树以子目录（lib/）的形式添加到master分支中。

先来看看util-branch分支当前的最新提交，记住最新提交所指向的目录树（tree），即tree id：0c743e4。

```
$ git cat-file -p util-branch
tree 0c743e49e11019678c8b345e667504cb789431ae
parent f21f9c10cc248a4a28bf7790414baba483f1ec15
author Jiang Xin <jiangxin@ossxp.com> 1288494998 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1288494998 +0800

util v2.0 -> v3.0
```

查看tree 0c743e4所包含的内容，会看到两个文件：Makefile和version。

```
$ git cat-file -p 0c743e4
100644 blob 07263ff95b4c94275f4b4735e26ea63b57b3c9e3    Makefile
100644 blob bebe6b10eb9622597dd2b641efe8365c3638004e    version
```

切换到master分支，以如下方式调用git read-tree将util-branch分支的目录树读取到当前分支lib目录下，具体操作过程如下。

（1）切换到master分支。

```
$ git checkout master
```

（2）执行git read-tree命令，将分支util-branch读取到当前分支的一个子目录下。

```
$ git read-tree --prefix=lib util-branch
```

（3）调用git read-tree只是更新了暂存区，所以查看工作区状态会看到工作区中还不存在lib目录下的两个文件。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   lib/Makefile
#       new file:   lib/version
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    lib/Makefile
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
#       deleted:    lib/version
#
```

（4）执行检出命令，将 lib 目录下的文件更新出来。

```
$ git checkout -- lib
```

（5）再次查看状态，会看到前面执行的 git read-tree 命令添加到了暂存区的文件中。

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   lib/Makefile
#       new file:   lib/version
#
```

现在还不能忙着提交，因为如果现在进行提交就体现不出两个分支的合并关系。需要使用 Git 底层的命令进行数据提交，具体操作过程如下。

（1）调用 git write-tree 将暂存区的目录树保存下来。

要记住调用 git write-tree 后形成的新的 tree-id：2153518。

```
$ git write-tree
2153518409d218609af40babededec6e8ef51616
```

（2）执行 git cat-file 命令显示这棵树的内容，会注意到其中 lib 目录的 tree-id 和之前查看过的 util-branch 分支最新的提交对应的 tree-id 一样都是 0c743e4。

```
$ git cat-file -p 2153518409d218609af40babededec6e8ef51616
100644 blob 07263ff95b4c94275f4b4735e26ea63b57b3c9e3    Makefile
040000 tree 0c743e49e11019678c8b345e667504cb789431ae    lib
100644 blob 638c7b7c6bdbde1d29e0b55b165f755c8c4332b5    version
```

（3）要手工创建一个合并提交，即新的提交要有两个父提交。这两个父提交分别是 master 分支和 util-branch 分支的最新提交。用下面的命令显示两个提交的提交 ID，并记下这两个提交 ID。

```
$ git rev-parse HEAD
911b1af2e0c95a2fc1306b8dea707064d5386c2e
$ git rev-parse util-branch
12408a149bfa78a4c2d4011f884aa2adb04f0934
```

（4）执行 git commit-tree 命令手动创建提交。新提交的目录树来自上面的 git write-tree 产生的目录树（tree-id 为 2153518），而新提交（合并提交）的两个父提交直接用上面 git rev-parse 显示的两个提交 ID 表示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ echo "subtree merge" | \
  git commit-tree 2153518409d218609af40babededec6e8ef51616 \
  -p 911b1af2e0c95a2fc1306b8dea707064d5386c2e \
  -p 12408a149bfa78a4c2d4011f884aa2adb04f0934
62ae6cc3f9280418bdb0fcf6c1e678905b1fe690
```

（5）执行 git commit-tree 命令的输出是提交之后产生的新提交的提交 ID。需要把当前的 master 分支重置到此提交 ID。

```
$ git reset 62ae6cc3f9280418bdb0fcf6c1e678905b1fe690
```

（6）查看一下提交日志及分支图，可以看到通过复杂的 git read-tree、git write-tree 和 git commit-tree 命令制造的合并提交，的确将两个不同的版本库合并到一起了。

```
$ git log --graph --pretty=oneline
*   62ae6cc3f9280418bdb0fcf6c1e678905b1fe690 subtree merge
|\
| * 12408a149bfa78a4c2d4011f884aa2adb04f0934 util v2.0 -> v3.0
| * f21f9c10cc248a4a28bf7790414baba483f1ec15 util v1.0 -> v2.0
| * 76db0ad729db9fdc5be043f3b4ed94ddc945cd7f util v1.0
* 911b1af2e0c95a2fc1306b8dea707064d5386c2e main v2010.1
```

（7）看看现在的 master 分支。

```
$ git cat-file -p HEAD
tree 2153518409d218609af40babededec6e8ef51616
parent 911b1af2e0c95a2fc1306b8dea707064d5386c2e
parent 12408a149bfa78a4c2d4011f884aa2adb04f0934
author Jiang Xin <jiangxin@ossxp.com> 1288498186 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1288498186 +0800

subtree merge
```

（8）看看目录树。

```
$ git cat-file -p 2153518409d218609af40babededec6e8ef51616
100644 blob 07263ff95b4c94275f4b4735e26ea63b57b3c9e3    Makefile
040000 tree 0c743e49e11019678c8b345e667504cb789431ae    lib
100644 blob 638c7b7c6bdbde1d29e0b55b165f755c8c4332b5    version
```

整个过程非常繁琐，但是不要太过担心，只需要对原理了解清楚就可以了，因为在后面会介绍一个 Git 插件，它封装了复杂的子树合并操作。

## 24.3 利用子树合并跟踪上游改动

如果子树（lib 目录）的上游（即 util.git）包含了新的提交，如何将 util.git

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的新提交合并过来呢？这就要用到名为 subtree 的合并策略[①]。

在执行子树合并之前，先切换到 util-branch 分支，获取远程版本库的改动。

```
$ git checkout util-branch
$ git pull
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From /path/to/repos/util
   12408a1..5aba14f  master     -> util/master
Updating 12408a1..5aba14f
Fast-forward
 version |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git checkout master
```

在切换回 master 分支后，如果这时执行 `git merge util-branch`，会将 uitl-branch 的数据直接合并到 master 分支的根目录下，而实际上是希望合并发生在 lib 目录中，这就需要以如下方式进行调用，以 subtree 策略进行合并。

如果 Git 的版本小于 1.7，直接使用 subtree 合并策略。

```
$ git merge -s subtree util-branch
```

如果 Git 的版本是 1.7 之后（含 1.7）的版本，则可以使用默认的 recursive 合并策略，通过参数 `-Xsubtree=<prefix>` 在合并时使用正确的子树进行匹配合并。避免了使用 subtree 合并策略时的猜测。

```
$ git merge -Xsubtree=lib util-branch
```

再来看看执行子树合并之后的分支图示。

```
$ git log --graph --pretty=oneline
*   f1a33e55eea04930a500c18a24a8bd009ecd9ac2 Merge branch 'util-branch'
|\
| * 5aba14fd347fc22cd8fbd086c9f26a53276f15c9 util v3.1 -> v3.2
| * a6d53dfcf78e8a874e9132def5ef87a2b2febfa5 util v3.0 -> v3.1
* |   62ae6cc3f9280418bdb0fcf6c1e678905b1fe690 subtree merge
|\ \
| |/
| * 12408a149bfa78a4c2d4011f884aa2adb04f0934 util v2.0 -> v3.0
| * f21f9c10cc248a4a28bf7790414baba483f1ec15 util v1.0 -> v2.0
| * 76db0ad729db9fdc5be043f3b4ed94ddc945cd7f util v1.0
* 911b1af2e0c95a2fc1306b8dea707064d5386c2e main v2010.1
```

① 参见第3篇第16章“16.6 合并策略”。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 24.4 子树拆分

既然可以将一个代码库通过子树合并的方式作为子目录加入到另外一个版本库中，反之也可以将一个代码库的子目录独立出来转换为另外的版本库。不过这个反向过程非常复杂。要将一个版本库的子目录作为顶级目录导出到另外的项目，潜藏的条件是要导出历史，因为如果不关心历史，直接拷贝文件重建项目就可以了。子树拆分的大致过程是：

（1）找到要导出的目录的提交历史，并反向排序。

（2）依次对每个提交执行下面的操作。

（3）找出提交中导出目录对应的 tree id。

（4）对该 tree id 执行 `git commit-tree`。

（5）执行 `git commit-tree` 要保持提交信息还要重新设置提交的 parents。

手工执行这个操作复杂且易出错，可以用下节介绍的 `git-subtree` 插件，或使用第 6 篇第 35 章的“35.4 Git 版本库整理”一节中介绍的 `git filter-branch` 子目录过滤器的技术。

## 24.5 git-subtree 插件

git-subtree 插件[①]用 shell 脚本开发，安装之后为 Git 提供了新的 `git subtree` 命令，支持前面介绍的子树合并和子树拆分。命令非常简单易用，将其他版本库以子树形式导入，再也不必和底层的 Git 命令打交道了。

安装 Git subtree 很简单，操作如下：

```
$ git clone git://github.com/apenwarr/git-subtree.git
$ cd git-subtree
$ make doc
$ make test
$ sudo make install
```

下面就来介绍 Git subtree 的常用命令。

### 1. git subtree add

命令 `git subtree add` 相当于将其他版本库以子树方式加入到当前版本库。用法如下：

```
git subtree add [--squash] -P <prefix> <commit>
git subtree add [--squash] -P <prefix> <repository> <refspec>
```

其中可选的 `--squash` 的含义是压缩为一个版本后再添加。

① *http://github.com/apenwarr/git-subtree/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


对于文中的示例，为了将 util.git 合并到 main.git 的 lib 目录，可以直接这样调用：

```
$ git subtree add -P lib /path/to/repos/util.git master
```

不过推荐的方法还是先在本地建立 util.git 版本库的追踪分支。

```
$ git remote add util /path/to/repos/util.git
$ git fetch util
$ git branch util-branch util/master
$ git subtree add -P lib util-branch
```

### 2. git subtree merge

命令 git subtree merge 相当于将子树对应的远程分支的更新重新合并到子树中，相当于完成了 git merge -s subtree 操作，用法如下：

```
git subtree merge [--squash] -P <prefix> <commit>
```

其中可选的 --squash 的含义是压缩为一个版本后再合并。

对于文章中的示例，为了将 util-branch 分支包含的上游的最新改动合并到 master 分支的 lib 目录中，可以直接这样调用：

```
$ git subtree merge -P lib util-branch
```

### 3. git subtree pull

命令 git subtree pull 相当于先对子树对应的远程版本库执行一次 git fetch 操作，然后再执行 git subtree merge。用法如下：

```
git subtree pull [--squash] -P <prefix> <repository> <refspec...>
```

对于文章中的示例，为了将 util.git 版本库的 master 分支包含的最新改动合并到 master 分支的 lib 目录中。可以直接这样调用：

```
$ git subtree pull -P lib /path/to/repos/util.git master
```

更喜欢用前面介绍的 git subtree merge 命令，因为 git subtree pull 存在版本库地址写错的风险。

### 4. git subtree split

命令 git subtree split 相当于将目录拆分为独立的分支，即子树拆分。拆分后形成的分支可以推送到一个新的版本库中，进而实现用原版本库的一个子目录为根目录创建出新的版本库。用法如下：

```
git subtree split -P <prefix> [--branch <branch>] [--onto ...] [--ignore-joins]
[--rejoin] <commit...>
```

说明：

- 该命令总是输出子树拆分后的最后一个 commit-id。这样可以通过管道方式传递给其

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


他命令，如 git subtree push 命令。

- 参数 --branch 提供拆分后创建的分支名称。如果不提供，只能通过 git subtree split 命令提供的提交 ID 得到拆分的结果。
- 参数 --onto 参数将目录拆分附加于已经存在的提交上。
- 参数 --ignore-joins 忽略对之前拆分历史的检查。
- 参数 --rejoin 会将拆分结果合并到当前分支，因为采用 ours 的合并策略，不会破坏当前分支。

### 5. git subtree push

命令 git subtree push 先执行子树拆分，再将拆分的分支推送到远程服务器。用法如下：

```
git subtree push -P <prefix> <repository> <refspec...>
```

该命令的用法和 git subtree split 的类似，这里就不再赘述。





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 25 章 Android 式多版本库协同

Android 是谷歌（Google）开发的适合手持设备的操作系统，提供了当前最吸引眼球的开源的手持设备操作平台，大有超越苹果（Apple.com）专有的 iOS 系统的趋势。而 Android 的源代码就是使用 Git 进行维护的。Android 项目在使用 Git 进行源代码管理上有两个伟大的创造，一个是用 Java 开发的名为 Gerrit 的代码审核服务器（将在第 5 篇第 32 章专题介绍），另外一个就是本章要重点介绍的 repo。

repo 是一个用 Python 语言开发的命令行工具，可以更方便地进行多版本库的管理。先来看看 Android 到底包含了多少个 Git 库：

- Android 的版本库管理工具 repo。

```
git://android.git.kernel.org/tools/repo.git
```

- 保存 GPS 配置文件的版本库。

```
git://android.git.kernel.org/device/common.git
```

- 160 多个其他的版本库（截至 2010 年 10 月）。

如果把 160 多个版本库都列在这里，恐怕各位的下巴都会掉下来。那么为什么 Android 的版本库会有这么多呢？怎么管理这么复杂的版本库呢？

Android 版本库众多的原因，主要是版本库太大，以及 Git 不能部分检出。Android 的版本库有接近 2 个 GB 之大。如果把所有的东西都放在一个库中，而某个开发团队感兴趣的可能就是某个驱动，或者是某个应用，却要下载如此庞大的版本库，是有些说不过去。

好了，既然接受了 Android 有多达 160 多个版本库这一事实，那么 Android 是不是用之前介绍的“子模组”方式组织起来的呢？如果真的用“子模组”方式来管理这 160 多个代码库，可能就需要如此管理：

- 建立一个索引版本库，在该版本库中，通过子模组方式，将目录一个一个地对应到这 160 多个版本库。
- 对此索引版本库执行克隆操作后，再执行 `git submodule init` 命令。
- 当执行 `git submodule update` 命令时，开始分别克隆这 160 多个版本库。
- 如果想修改某个版本库中的内容，需要进入到相应的子模组目录，执行切换分支的操作。因为子模组是以某个固定提交的状态存在的，是不能更改的，必须先切换到某个工作分支后，才能进行修改和提交。
- 如果要将所有的子模组都切换到某个分支（如 master）进行修改，必须自己通过脚本对这 160 多个版本库一一进行切换。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- Android 有多个版本：android-1.0、android-1.5、……、android-2.2_r1.3……如何维护这么多的版本呢？也许索引库要通过分支和里程碑，与子模组的各个不同的提交状态进行对应。但是由于子模组的状态只是一个提交 ID，如何能够动态地指定到分支，真的给不出答案。

幸好上面只是假设。聪明的 Android 程序设计师一早就考虑到了 Git 子模组的局限性，以及多版本库管理的问题，开发出了 repo 这一工具。

关于 repo 有这么一则小故事：Android 之父安迪·鲁宾在回应乔布斯关于 Android 太开放导致开发维护更麻烦的言论时，在 Twitter[①]上留了下面这段简短的话：

```
the definition of open: "mkdir android ; cd android ; repo init -u
git://android.git.kernel.org/platform/manifest.git ; repo sync ; make"
```

是的，就是 repo 让 Android 的开发变得如此简单。

## 25.1 关于 repo

repo 是 Google 开发的用于管理 Android 版本库的一个工具。repo 并不是用于取代 Git，而是用 Python 对 Git 进行了一定的封装，简化了对多个 Git 版本库的管理。对于 repo 管理的任何一个版本库，都需要使用 Git 命令进行操作。

repo 的使用过程大致如下：

（1）运行 `repo init` 命令，克隆 Android 的一个清单库。这个清单库和前面假设的“子模组”方式工作的索引库不同，是通过 XML 技术建立的版本库清单。

（2）清单库中的 `manifest.xml` 文件，列出了 160 多个版本库的克隆方式。包括版本库的地址和工作区地址的对应关系，以及分支的对应关系。

（3）运行 `repo sync` 命令，开始同步，即分别克隆这 160 多个版本库到本地的工作区中。

（4）同时对 160 多个版本库执行切换分支操作，切换到某个分支。

## 25.2 安装 repo

首先下载 repo 的引导脚本，可以使用 `wget`、`curl` 甚至浏览器从 *http://android.git.kernel.org/repo* 上下载。把 repo 脚本设置为可执行，并复制到可执行的路径中。在 Linux 上可以用下面的指令将 repo 下载并复制到用户主目录的 bin 目录下。

```
$ curl -L -k http://android.git.kernel.org/repo >~/bin/repo
$ chmod a+x ~/bin/repo
```

为什么说下载的 repo 只是一个引导脚本（bootstrap）而不是直接称为 repo 呢？因为 repo 的大部分功能代码不在其中，下载的只是一个帮助完成整个 repo 程序继续下载和加载

① *http://twitter.com/Arubin*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的工具。如果您是一个程序员，对 repo 的执行比较好奇，可以一起来分析一下 repo 引导脚本。否则可以直接跳到下一节。

看看 repo 引导脚本的前几行（为方便描述，把注释和版权信息过滤掉了），会发现一个神奇的魔法：

```
1  #!/bin/sh
2
3  REPO_URL='git://android.git.kernel.org/tools/repo.git'
4  REPO_REV='stable'
5
6  magic='--calling-python-from-/bin/sh--'
7  """exec" python -E "$0" "$@" """#$magic"
8  if __name__ == '__main__':
9    import sys
10   if sys.argv[-1] == '#%s' % magic:
11     del sys.argv[-1]
12 del magic
```

repo 引导脚本是用什么语言开发的？这是一个问题。

- 第 1 行，有经验的 Linux 开发者会知道此脚本是用 Shell 脚本语言开发的。
- 第 7 行，是这个魔法的最神奇之处。既是一条合法的 shell 语句，又是一条合法的 python 语句。
- 第 7 行如果作为 shell 语句，执行 exec，用 python 调用本脚本，并替换本进程。三引号在这里相当于一个空字串和一个单独的引号。
- 第 7 行如果作为 python 语句，三引号定义的是一个字符串，字符串后面是一个注释。
- 实际上第 1 行到第 7 行，既是合法的 shell 语句又是合法的 python 语句。从第 8 行开始后面都是 python 脚本了。
- repo 引导脚本无论是用 shell 执行，还是用 python 执行，效果都相当于使用 python 执行此脚本。

repo 脚本的真正位置在哪里？可以通过分析引导脚本 repo 得到。在引导脚本 repo 的 main 函数中，首先调用 _FindRepo 函数，从当前目录开始依次向上递归查找 .repo/repo/main.py 文件。

```
def main(orig_args):
  main, dir = _FindRepo()
```

函数 _FindRepo 返回找到的 .repo/repo/main.py 脚本文件，以及包含 repo/main.py 的 .repo 目录。如果找到了 .repo/repo/main.py 脚本，则把程序的控制权交给 .repo/repo/main.py 脚本（省略了在 repo 开发库中执行情况的判断）。

在下载 repo 引导脚本后，没有初始化之前，当然不会存在 .repo/repo/main.py 脚本，这时必须进行初始化操作。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 25.3 repo 和清单库的初始化

下载并保存 repo 引导脚本后，建立一个工作目录，这个工作目录将作为 Android 的工作区目录。在工作目录中执行 repo init -u <url>，完成 repo 完整的下载及项目清单版本库（manifest.git）的下载。

```
$ mkdir working-directory-name
$ cd working-directory-name
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

命令 repo init 要完成如下操作：

- 完成 repo 这一工具的完整下载，因为现在有的不过是 repo 的引导程序。
  初始化操作会从 android 的代码中克隆 repo.git 库到当前目录下的 .repo/repo 目录下。在完成 repo.git 克隆之后，repo init 命令会将控制权交给工作区的 .repo/repo/main.py，这个刚刚从 repo.git 库克隆来的脚本文件，继续进行初始化。
- 克隆 android 的清单库 manifest.git（地址来自于 -u 参数）。
- 克隆的清单库位于 .repo/manifests.git 中，本地克隆到 .repo/manifests。清单文件 .repo/manifest.xml 只是符号链接，它指向 .repo/manifests/default.xml。
- 询问用户的姓名和邮件地址，如果和 Git 默认的用户名、邮件地址不同，则记录在 .repo/manifests.git 库的 config 文件中。
- 命令 repo init 还可以附带 --mirror 参数，以建立和上游 Android 的版本库一模一样的镜像。这会在后面的章节介绍。

### 1. 从哪里下载 repo.git？

在 repo 引导脚本的前几行，定义了默认的 repo.git 的版本库位置及要检出的默认分支。

```
REPO_URL='git://android.git.kernel.org/tools/repo.git'
REPO_REV='stable'
```

如果不想从默认 URL 地址中获取 repo，或者不想获取稳定版（stable 分支）的 repo，可以在 repo init 子命令中通过下面的参数覆盖默认的设置，从指定的源地址克隆 repo 代码库：

- 参数 --repo-url，用于设定 repo 的版本库地址。
- 参数 --repo-branch，用于设定要检出的分支。
- 参数 --no-repo-verify，设定不要对 repo 的里程碑签名进行严格的验证。

实际上，完成 repo.git 版本库的克隆，这个 repo 引导脚本就江郎才尽了，repo init 命令的后续处理（以及其他子命令）都交给刚刚克隆出来的 .repo/repo/main.py 来继续执行。

### 2. 清单库是什么？从哪里下载？

清单库实际上只包含一个 default.xml 文件。这个 XML 文件定义了多个版本库和本



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


地地址的映射关系，是 repo 工作的指引文件。所以在使用 repo 引导脚本进行初始化的时候，必须通过 -u 参数指定清单库的源地址。

清单库的下载，是通过 repo init 命令初始化时，用 -u 参数指定清单库的位置。例如 repo 针对 Android 代码库进行初始化时执行的命令：

```
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

repo 引导脚本的 init 子命令可以使用下列和清单库相关的参数：

- 参数 -u（--manifest-url）：设定清单库的 Git 服务器地址。
- 参数 -b（--manifest-branch）：检出清单库的特定分支。
- 参数 --mirror：只在 repo 第一次初始化的时候使用，以和 Android 服务器同样的结构在本地建立镜像。
- 参数 -m（--manifest-name）：当有多个清单文件时，可以指定清单库的某个清单文件为有效的清单文件。默认为 default.xml。

repo 初始化命令（repo init）可以执行多次：

- 不带参数地执行 repo init，从上游的清单库获取新的清单文件 default.xml。
- 使用参数 -u（--manifest-url）执行 repo init，会重新设定上游的清单库地址，并重新同步。
- 使用参数 -b（--manifest-branch）执行 repo init，会使用清单库的不同分支，以便在使用 repo sync 时将项目同步到不同的里程碑。
- 但是不能使用 --mirror 命令，该命名只能在第一次初始化时执行。那么如何将已经按照工作区模式同步的版本库转换为镜像模式呢？后面会看到一个解决方案。

## 25.4 清单库和清单文件

执行完 repo init 之后，工作目录内空空如也。实际上有一个 .repo 目录。在该目录下除了一个包含 repo 实现的 repo 库克隆外，就是 manifest 库的克隆，以及一个符号链接，链接到清单库中的 default.xml 文件。

```
$ ls -lF .repo/
drwxr-xr-x 3 jiangxin jiangxin 4096 2010-10-11 18:57 manifests/
drwxr-xr-x 8 jiangxin jiangxin 4096 2010-10-11 10:08 manifests.git/
lrwxrwxrwx 1 jiangxin jiangxin   21 2010-10-11 10:07 manifest.xml ->
manifests/default.xml
drwxr-xr-x 7 jiangxin jiangxin 4096 2010-10-11 10:07 repo/
```

在工作目录下的 .repo/manifest.xml 文件就是 Android 项目的众多版本库的清单文件。repo 命令的操作都要参考这个清单文件。

打开清单文件会看到如下内容：

```
1  <?xml version="1.0" encoding="UTF-8"?>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 2  <manifest>
 3    <remote  name="korg"
 4             fetch="git://android.git.kernel.org/"
 5             review="review.source.android.com" />
 6    <default revision="master"
 7             remote="korg" />
 8
 9    <project path="build" name="platform/build">
10      <copyfile src="core/root.mk" dest="Makefile" />
11    </project>
12
13    <project path="bionic" name="platform/bionic" />

      ...

181  </manifest>
```

这个文件不太复杂，是吗？

- 这个 XML 的顶级元素是 manifest，见第 2 行和第 181 行。
- 第 3 行通过一个 remote 元素，定义了名为 korg（kernel.org 缩写）的远程版本库，其 Git 库的基址为 git://android.git.kernel.org/。还定义了代码审核服务器的地址 review.source.android.com。还可以定义更多的 remote 元素，这里只定义了一个。
- 第 6 行用于设置各个项目默认的远程版本库（remote）为 korg，默认的分支为 master。当然各个项目（project 元素）可以定义自己的 remote 和 revision 覆盖该默认配置。
- 第 9 行定义了一个项目，该项目的远程版本库相对路径为：platform/build，在工作区克隆的位置为：build。
- 第 10 行，即 project 元素的子元素 copyfile，定义了项目克隆后的一个附加动作：从 core/root.mk 拷贝文件至 Makefile。
- 第 13 行后续的 100 多行定义了其他 160 个项目，都是采用类似的 project 元素语法。name 参数定义远程版本库的相对路径，path 参数定义克隆到本地工作区的路径。
- 还可以出现 manifest-server 元素，其 url 属性定义了通过 XMLRPC 提供实时更新清单的服务器 URL。只有当执行 repo sync --smart-sync 的时候才会检查该值，并用动态获取的 manifest 覆盖掉默认的清单。

## 25.5 同步项目

在工作区执行下面的命令，会参照 .repo/manifest.xml 清单文件，将项目所有相关的版本库全部克隆出来。不过最好请在读完本节内容之后再尝试执行这条命令。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ repo sync
```

对于 Android，这个操作需要通过网络传递接近 2 个 GB 的内容，如果带宽不是很高的话，可能会花掉几个小时甚至是一天的时间。

也可以仅克隆感兴趣的项目，在 repo sync 后面跟上项目的名称。项目的名称来自于 .repo/manifest.xml 这个 XML 文件中 project 元素的 name 属性值。例如克隆 platform/build 项目：

```
$ repo sync platform/build
```

repo 有一个功能可以在这里展示，就是 repo 支持通过本地清单对默认的清单文件进行补充及覆盖。即可以在 .repo 目录下创建 local_manifest.xml 文件，其内容会和 .repo/manifest.xml 文件的内容进行合并。

在工作目录下运行下面的命令可以创建一个本地清单文件。这个本地定制的清单文件来自默认文件，但是删除了 remote 元素和 default 元素，并将所有的 project 元素都重命名为 remove-project 元素。这实际相当于对原有的清单文件“取反”。

```
$ curl -L -k  \
          http://www.ossxp.com/doc/gotgit/download/ch25/manifest-revert.xslt  \
          > manifest-revert.xslt
$ xsltproc manifest-revert.xslt .repo/manifest.xml > .repo/local_manifest.xml
```

用下面的这条命令可以看到 repo 运行时实际获取到的清单。这个清单来自于 .repo/manifest.xml 和 .repo/local_manifest.xml 两个文件的汇总。

```
$ repo manifest -o -
```

当执行 repo sync 命令时，实际上就是依据合并后的清单文件进行同步。如果清单中的项目被自定义清单全部“取反”，执行同步就不会同步任何项目，甚至会删除已经完成同步的项目。

本地定制的清单文件 local_manifest.xml 支持前面介绍的清单文件的所有语法，需要注意的是：

- 不能出现重复定义的 remote 元素。这就是为什么上面的脚本要删除来自默认 manifest.xml 的 remote 元素。
- 不能出现 default 元素，因为全局只能有一个。
- 不能出现重复的 project 定义（name 属性不能相同），但是可以通过 remove-project 元素将默认清单中定义的 project 删除然后再重新定义。

试着编辑 .repo/local_manifest.xml，在其中再添加几个 project 元素，然后试着用 repo sync 命令进行同步。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 25.6 建立 Android 代码库本地镜像

Android 为企业提供一个新的市场，无论企业大小都处于同一个起跑线上。研究 Android 尤其是 Android 系统核心或驱动的开发，首先要做的就是通过本地克隆建立一套 Android 版本库管理机制。因为 Android 的代码库是那么庞杂，如果一个开发团队每个人都去执行 repo init -u，再执行 repo sync 从 Android 服务器克隆版本库的话，多大的网络带宽恐怕都不够用。唯一的办法是在本地建立一个 Android 版本库的镜像。

建立本地镜像非常简单，就是在执行 repo init -u 初始化的时候，附带上 --mirror 参数。

```
$ mkdir android-mirror-dir
$ cd android-mirror-dir
$ repo init --mirror -u git://android.git.kernel.org/platform/manifest.git
```

之后执行 repo sync 就可以从安装 Android 的 Git 服务器方式来组织版本库，创建一个 Android 版本库镜像了。

实际上附带了 --mirror 参数执行 repo init -u 命令，会在克隆的 .repo/manifests.git 下的 config 中记录配置信息：

```
[repo]
    mirror = true
```

### 1. 从 Android 的工作区到代码库镜像

在初始化 repo 工作区时，使用不带 --mirror 参数的 repo init -u，并完成代码同步后，如果再次执行 repo init 并附带了 --mirror 参数，repo 会报错退出："fatal: --mirror not supported on existing client"。实际上 --mirror 参数只能对尚未初始化的 repo 工作区执行。

那么如果之前没有用镜像的方法同步 Android 版本库，难道要为创建代码库镜像再重新执行一次 repo 同步吗？要知道重新同步一份 Android 版本库是非常慢的。我就遇到了这个问题。

不过既然有 manifest.xml 文件，完全可以对工作区进行反向操作，将工作区转换为镜像服务器的结构。下面就是一个示例脚本，可以从本书在 Github 上的相关版本库[①]下载。这个脚本利用了已有的 repo 代码进行实现，所以看着很简洁。 8-)

脚本 work2mirror.py 如下：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import os, sys, shutil

cwd = os.path.abspath( os.path.dirname( __file__ ) )
```

① *https://github.com/ossxp-com/gotgit/raw/master/download/ch25/work2mirror.py*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
repodir = os.path.join( cwd, '.repo' )
S_repo = 'repo'
TRASHDIR = 'old_work_tree'

if not os.path.exists( os.path.join(repodir, S_repo) ):
    print >> sys.stderr, "Must run under repo work_dir root."
    sys.exit(1)

sys.path.insert( 0, os.path.join(repodir, S_repo) )
from manifest_xml import XmlManifest

manifest = XmlManifest( repodir )

if manifest.IsMirror:
    print >> sys.stderr, "Already mirror, exit."
    sys.exit(1)

trash = os.path.join( cwd, TRASHDIR )

for project in manifest.projects.itervalues():
    # 将旧的版本库路径移动到镜像模式下新的版本库路径
    newgitdir = os.path.join( cwd, '%s.git' % project.name )
    if os.path.exists( project.gitdir ) and project.gitdir != newgitdir:
        if not os.path.exists( os.path.dirname(newgitdir) ):
            os.makedirs( os.path.dirname(newgitdir) )
        print "Rename %s to %s." % (project.gitdir, newgitdir)
        os.rename( project.gitdir, newgitdir )

    # 将工作区移动到待删除目录
    if project.worktree and os.path.exists( project.worktree ):
        newworktree = os.path.join( trash, project.relpath )
        if not os.path.exists( os.path.dirname(newworktree) ):
            os.makedirs( os.path.dirname(newworktree) )
        print "Move old worktree %s to %s." % (project.worktree, newworktree )
        os.rename( project.worktree, newworktree )

    if os.path.exists ( os.path.join( newgitdir, 'config' ) ):
        # 修改版本库的配置
        os.chdir( newgitdir )
        os.system( "git config core.bare true" )
        os.system( "git config remote.korg.fetch '+refs/heads/*:refs/heads/*'" )

        # 删除 remotes 分支，因为作为版本库镜像不需要 remote 分支
        if os.path.exists ( os.path.join( newgitdir, 'refs', 'remotes' ) ):
            print "Delete " + os.path.join( newgitdir, 'refs', 'remotes' )
            shutil.rmtree( os.path.join( newgitdir, 'refs', 'remotes' ) )

# 设置 menifest 为镜像
mp = manifest.manifestProject
mp.config.SetString('repo.mirror', 'true')
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


使用方法很简单，只要将脚本放在 Android 工作区下执行就可以了。执行完毕会将原有工作区的目录移动到 old_work_tree 子目录下，在确认原有工作区没有未提交的数据后，直接删除 old_work_tree 即可。

```
$ python work2mirror.py
```

### 2. 创建新的清单库，或者修改原有清单库

建立了 Android 代码库的本地镜像后，如果不对 manifest 清单版本库进行定制，在使用 repo sync 同步代码的时候，仍然使用 Android 官方的代码库同步代码，就会使得本地的镜像版本库形同虚设。解决办法是创建一个自己的 manifest 库，或者在原有清单库中建立一个分支加以修改。如果创建新的清单库，参考 Android 上游的 manifest 清单库进行创建。

## 25.7 repo 的命令集

repo 子命令实际上是 Git 命令的或简单或复杂的封装。每一个 repo 子命令都对应于 repo 源码树中 subcmds 目录下的一个同名的 Python 文件。每一个 repo 子命令都可以通过下面的命令获得帮助。

```
repo help <command>
```

通过阅读代码，可以更加深入地了解 repo 子命令的封装。

### 1. repo init 命令

repo init 子命令主要完成检出清单版本库（manifest.git），以及配置 Git 用户的用户名和邮件地址的工作。

实际上，完全可以进入到 .repo/manifests 目录，用 Git 命令操作清单库。对 manifests 的修改不会因为执行 repo init 而丢失，除非是处于未跟踪状态。

### 2. repo sync 命令

repo sync 子命令用于参照清单文件克隆或同步版本库。如果某个项目版本库尚不存在，则执行 repo sync 命令相当于执行 git clone。如果项目版本库已经存在，则相当于执行下面的两个命令：

- git remote update

  相当于对每一个 remote 源执行 fetch 操作。

- git rebase origin/branch

  针对当前分支的跟踪分支执行 rebase 操作。不采用 merge 而是采用 rebase，目的是减少提交数量、方便评审（Gerrit）。

### 3. repo start 命令

repo start 子命令实际上是对 git checkout -b 命令的封装。为指定的项目或所有项



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


目（若使用 --all 参数），以清单文件中为项目设定的分支或里程碑为基础，创建特性分支。特性分支的名称由命令的第一个参数指定。相当于执行 checkout -b 。

用法如下：

```
repo start <newbranchname> [--all | <project>...]
```

#### 4. repo status 命令

repo status 子命令实际上是对 git diff-index、git diff-files 命令的封装，同时显示暂存区的状态和本地文件修改的状态。

用法如下：

```
repo status [<project>...]
```

示例输出：

```
project repo/                                   branch devwork
 -m     subcmds/status.py
 ...
```

上面的示例输出显示了 repo 项目的 devwork 分支的修改状态。

- 每个小节的首行显示项目的名称，以及所在分支的名称。
- 之后显示该项目中文件的变更状态。头两个字母显示变更状态，后面显示文件名或其他变更信息。
- 第一个字母表示暂存区的文件修改状态。

  其实是 git-diff-index 命令输出中的状态标识，用大写显示。
  - -：没有改变
  - A：添加（不在 HEAD 中，在暂存区）
  - M：修改（在 HEAD 中，在暂存区，内容不同）
  - D：删除（在 HEAD 中，不在暂存区）
  - R：重命名（不在 HEAD 中，在暂存区，路径修改）
  - C：拷贝（不在 HEAD 中，在暂存区，从其他文件拷贝）
  - T：文件状态改变（在 HEAD 中，在暂存区，内容相同）
  - U：未合并，需要冲突解决
- 第二个字母表示工作区文件的更改状态。

  其实是 git-diff-files 命令输出中的状态标识，用小写显示。
  - -：新 / 未知（不在暂存区，在工作区）
  - m：修改（在暂存区，在工作区，被修改）
  - d：删除（在暂存区，不在工作区）
- 两个表示状态的字母后面，显示文件名信息。如果有文件重命名还会显示改变前后的文件名及文件的相似度。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 5. repo checkout 命令

repo checkout 子命令实际上是对 `git checkout` 命令的封装。检出之前由 repo start 创建的分支。

用法如下：

```
repo checkout <branchname> [<project>...]
```

### 6. repo branches 命令

repo branches 读取各个项目的分支列表并汇总显示。该命令实际上通过直接读取 `.git/refs` 目录下的引用来获取分支列表，以及分支的发布状态等。

用法如下：

```
repo branches [<project>...]
```

输出示例：

```
*P nocolor                    | in repo
   repo2                      |
```

- 第一个字段显示分支的状态：是否当前分支，分支是否发布到代码审核服务器上？
- 第一个字母若显示星号（*），则含义是此分支为当前分支。
- 第二个字母若为大写字母 P，则含义是分支的所有提交都发布到代码审核服务器上了。
- 第二个字母若为小写字母 p，则含义是只有部分提交被发布到代码审核服务器上。
- 若不显示 P 或 p，则表明分支尚未发布。
- 第二个字段为分支名。
- 第三个字段为以竖线（|）开始的字符串，表示该分支存在于哪些项目中。
  - in all projects
    该分支处于所有项目中。
  - in project1 project2
    该分支只在特定项目中定义。如：project1、project2。
  - not in project1
    该分支不存在于这些项目中。即除了 project1 项目外，其他项目都包含此分支。

### 7. repo diff 命令

repo diff 子命令实际上是对 `git diff` 命令的封装，用于分别显示各个项目工作区下的文件差异。

用法如下：

```
repo diff [<project>...]
```

### 8. repo stage 命令

repo stage 子命令实际上是对 `git add --interactive` 命令的封装，用于挑选各个

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


项目工作区中的改动（修改、添加等）以加入暂存区。

用法如下：

```
repo stage -i [<project>...]
```

### 9. repo upload 命令

repo upload 相当于 git push，但是又有很大的不同。执行 repo upload 不是将版本库改动推送到克隆时的远程服务器，而是推送到代码审查服务器（由 Gerrit 软件架设）的特殊引用上，使用的是 SSH 协议（特殊端口）。代码审核服务器会对推送的提交进行特殊处理，将新的提交显示为一个待审核的修改集，并进入代码审查流程。只有当审核通过后，才会合并到官方正式的版本库中。

用法如下：

```
repo upload [--re --cc] {[<project>]... | --replace <project>}

参数：
  -h, --help              显示帮助信息。
  -t                      发送本地分支名称到 Gerrit 代码审核服务器。
  --replace               发送此分支的更新补丁集。注意使用该参数，只能指定一个项目。
  --re=REVIEWERS, --reviewers=REVIEWERS
                          要求由指定的人员进行审核。
  --cc=CC                 同时发送通知到如下邮件地址。
```

**确定推送服务器的端口**

分支改动的推送是发给代码审核服务器，而不是下载代码的服务器。使用的协议是 SSH 协议，但是使用的并非标准端口。如何确认代码审核服务器上提供的特殊 SSH 端口呢？

在执行 repo upload 命令时，repo 会通过访问代码审核 Web 服务器的 /ssh_info 的 Url 获取 SSH 服务端口，默认为 29418。这个端口，就是 repo upload 发起推送的服务器的 SSH 服务端口。

**修订集修改后重新传送**

只有已经通过 repo upload 命令在代码审查服务器上提交了一个修订集，才会得到一个修订号。关于此次修订的相关讨论会发送到提交者的邮箱中。如果修订集有误没有通过审核，可以重新修改代码，再次向代码审核服务器上传修订集。

一个修订集修改后再次上传，如果修订集的 ID 不变那将是非常有用的，因为这样相关的修订集都在代码审核服务器的同一个界面中显示。

在执行 repo upload 时会弹出一个编辑界面，提示在方括号中输入修订集编号，否则会在代码审查服务器上创建新的 ID。有一个办法可以不用手工输入修订集，如下：

```
repo upload --replace project_name
```

当使用 --replace 参数后，repo 会检查本地版本库名为 refs/published/branch_name 的特殊引用（上一次提交的修订），获得其对应的提交 SHA1 哈希值。然后在代码审核服务器的 refs/changes/ 命名空间下的特殊引用中寻找和提交 SHA1 哈希值

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


匹配的引用，找到的匹配引用其名称中就所包含有变更集 ID，直接用此变更集 ID 作为新的变更集 ID 提交到代码审核服务器。

**Gerrit 服务器魔法**

repo upload 命令执行推送，实际上会以类似如下的命令行格式进行调用：

```
git push --receive-pack='gerrit receive-pack --reviewer charlie@example.com' \
        ssh://review.example.com:29418/project HEAD:refs/for/master
```

Gerrit 服务器接收到 git push 请求后，会自动将对分支的提交转换为修订集，显示于 Gerrit 的提交审核界面中。Gerrit 的魔法破解的关键点就在于 `git push` 命令的 `--receive-pack` 参数。即交由 `gerrit-receive-pack` 命令执行提交，进入非标准的 git 处理流程，将提交转换为在 `refs/changes` 命名空间下的引用，而不在 `refs/for` 命名空间下创建引用。

### 10. repo download 命令

`repo download` 命令主要用于代码审核者下载和评估贡献者提交的修订。贡献者的修订在 Git 版本库中以 `refs/changes/<changeid>/<patchset>` 引用方式命名（默认的 patchset 为 1），和其他 Git 引用一样，用 `git fetch` 获取，该引用所指向的最新的提交就是贡献者待审核的修订。使用 repo download 命令实际上就是用 `git fetch` 获取到对应项目的 `refs/changes/<changeid>/patchset>` 引用，并自动切换到对应的引用上。

用法如下：

```
repo download {project change[/patchset]}...
```

### 11. repo rebase 命令

`repo rebase` 子命令实际上是对 `git rebase` 命令的封装，该命令的参数也作为 `git rebase` 命令的参数，但 `-i` 参数仅当对一个项执行时才有效。

用法如下：

```
命令行：repo rebase {[<project>...] | -i <project>...}

参数：
  -h, --help            显示帮助并退出
  -i, --interactive     交互式的变基（仅对一个项目时有效）
  -f, --force-rebase    向 git rebase 命令传递 --force-rebase 参数
  --no-ff               向 git rebase 命令传递 -no-ff 参数
  -q, --quiet           向 git rebase 命令传递 --quiet 参数
  --autosquash          向 git rebase 命令传递 --autosquash 参数
  --whitespace=WS       向 git rebase 命令传递 --whitespace 参数
```

### 12. repo prune 命令

`repo prune` 子命令实际上是对 `git branch -d` 命令的封装，该命令用于扫描项目的各个分支，并删除已经合并的分支。

用法如下：

```
repo prune [<project>...]
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 13. repo abandon 命令

相比 repo prune 命令，repo abandon 命令更具破坏性，因为 repo abandon 是对 git branch -D 命令的封装。该命令非常危险，将直接删除分支，请慎用。

用法如下：

```
repo abandon <branchname> [<project>...]
```

### 14. 其他命令

- repo grep

  相当于对 git grep 的封装，用于在项目文件中进行内容查找。

- repo smartsync

  相当于用 -s 参数执行 repo sync。

- repo forall

  迭代器，可以对 repo 管理的项目进行迭代。

- repo manifest

  显示 manifest 文件内容。

- repo version

  显示 repo 的版本号。

- repo selfupdate

  用于 repo 自身的更新。如果提供 --repo-upgraded 参数，还会更新各个项目的钩子脚本。

## 25.8 repo 命令的工作流

图 25-1 是 repo 的工作流，每一个代码贡献都起始于 repo start 创建的本地工作分支，最终都以 repo upload 命令将代码补丁发布到代码审核服务器。

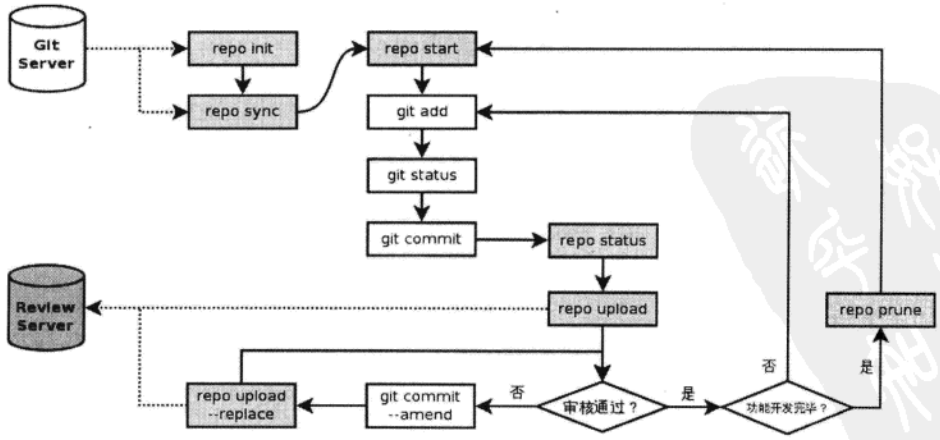



图 25-1 repo 工作流

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 25.9 好东西不能 Android 独享

通过前面的介绍能够体会到 repo 的精巧——repo 巧妙地实现了多 Git 版本库的管理。因为 repo 使用了清单版本库，所以 repo 这一工具并没有被局限于 Android 项目，可以在任何项目中使用。下面就介绍三种 repo 的使用模式，将 repo 引入自己的项目（非 Android 项目）中，其中第三种 repo 使用模式是用我改造后的 repo，实现了脱离 Gerrit 服务器进行推送。

### 25.9.1 repo+Gerrit 模式

repo 和 Gerrit 是 Android 代码管理的两大支柱。正如前面在 repo 工作流中介绍的，部分的 repo 命令从 Git 服务器读取，这个 Git 服务器可以是只读的版本库控制服务器，还有部分 repo 命令（repo upload、repo download）访问的则是代码审核服务器，其中 repo upload 命令还要向代码审核服务器进行 git push 操作。

在使用未经改动的 repo 来维护自己的项目（多个版本库组成）时，必须搭建 Gerrit 代码审核服务器。

搭建项目的版本控制系统环境的一般方法为：

- 用 Git 协议或 HTTP 协议搭建 Git 服务器。具体搭建方法参见第 5 篇“搭建 Git 服务器”的相关章节。
- 导入 repo.git 工具库。非必须，只是为了减少不必要的互联网操作。
- 还可以在内部 HTTP 服务器维护一个定制的 repo 引导脚本，非必须。
- 建立 Gerrit 代码审核服务器。会在第 5 篇“第 32 章 Gerrit 代码审核服务器”中介绍 Gerrit 的安装和使用。
- 一一创建相关的子项目代码库。
- 建立一个 manifest.git 清单库，其中 remote 元素的 fetch 属性指向只读 Git 服务器地址，review 属性指向代码审核服务器地址。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote  name="example"
           fetch="git://git.example.net/"
           review="review.example.net" />
  <default revision="master"
           remote="example" />

  ...
```

### 25.9.2 repo 无审核模式

Gerrit 代码审核服务器部署比较麻烦，更不要说因为 Gerrit 用户界面的学习和用户使用习惯的更改而带来的困难了。在一个固定的团队内部使用 repo 可能真的没有必要使用 Gerrit，因为团队成员都应该熟悉 Git 的操作，团队成员的编程能力都可信，单元测试质量由



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


提交者保证，集成测试由单独的测试团队进行，即团队拥有一套完整、成型的研发工作流，引入 Gerrit 并非必要。

脱离了 Gerrit 服务器，直接跟 Git 服务器打交道，repo 可以工作吗？是的，可以利用 repo forall 迭代器实现多项目代码的 PUSH，其中有如下关键点需要重点关注。

- repo start 命令创建本地分支时，需要使用和上游同样的分支名。
  如果使用不同的分支名，上传时需要提供复杂的引用描述。下面的示例先通过 repo manifest 命令确认上游清单库默认的分支名为 master，再使用该分支名（master）作为本地分支名执行 repo start。示例如下：

```
$ repo manifest -o - | grep default
  <default remote="bj" revision="master"/>

$ repo start master --all
```

- 推送不能使用 repo upload，而需要使用 git push 命令。
  可以利用 repo forall 迭代器实现批命令方式执行。例如：

```
$ repo forall -c git push
```

- 如果清单库中的上游 git 库地址用的是只读地址，需要为本地版本库一一更改上游版本库地址。
  可以使用 forall 迭代器，批量为版本库设置 git push 时的版本库地址。下面的命令使用的环境变量 $REPO_PROJECT 是实现批量设置的关键。

```
$ repo forall -c \
  'git remote set-url --push bj android@bj.ossxp.com:android/${REPO_PROJECT}.git'
```

### 25.9.3 改进的 repo 无审核模式

前面介绍的使用 repo forall 迭代器实现在无审核服务器情况下向上游推送提交，只是权宜之计，尤其是用 repo start 建立工作分支要求和上游一致，实在是有点强人所难。

我改造了 repo，增加了两个新的子命令 repo config 和 repo push，让 repo 可以脱离 Gerrit 服务器直接向上游推送。代码托管在 Github 上：http://github.com/ossxp-com/repo 。下面简单地介绍一下如何使用改造之后的 repo。

#### 1. 下载改造后的 repo 引导脚本

建议使用改造后的 repo 引导脚本替换原脚本，否则在执行 repo init 命令时需要提供额外的 --no-repo-verify 参数、--repo-url 和 --repo-branch 参数。

```
$ curl -L -k http://github.com/ossxp-com/repo/raw/master/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

#### 2. 用 repo 从 Github 上检出测试项目

如果安装了改造后的 repo 引导脚本，使用下面的命令初始化 repo 及清单库。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ mkdir test
$ cd test
$ repo init -u git://github.com/ossxp-com/manifest.git
$ repo sync
```

如果用的是标准的（未经改造的）repo 引导脚本，使用下面的命令。

```
$ mkdir test
$ cd test
$ repo init --repo-url=git://github.com/ossxp-com/repo.git \
  --repo-branch=master --no-repo-verify \
  -u git://github.com/ossxp-com/manifest.git
$ repo sync
```

当子项目代码全部同步完成后，执行 make 命令。可以看到各个子项目的版本及清单库的版本。

```
$ make
Version of test1:    1:0.2-dev
Version of test2:    2:0.2
Version of manifest: current
```

### 3. 用 repo config 命令设置 pushurl

现在如果进入到各个子项目目录，是无法成功执行 git push 命令的，因为上游 Git 库的地址是一个只读访问的 URL，无法提供写服务。可以用新增的 repo config 命令设置执行 git push 时的 URL 地址。

```
$ repo config repo.pushurl ssh://git@github.com/ossxp-com/
```

设置成功后，可以使用 repo config repo.pushurl 查看设置。

```
$ repo config repo.pushurl
ssh://git@github.com/ossxp-com/
```

### 4. 创建本地工作分支

使用下面的命令创建一个工作分支 jiangxin。

```
$ repo start jiangxin --all
```

使用 repo branches 命令可以查看当前所有的子项目都属于 jiangxin 分支。

```
$ repo branches
*  jiangxin                 | in all projects
```

参照下面的方法修改 test/test1 子项目。对 test/test2 项目也作类似修改。

```
$ cd test/test1
$ echo "1:0.2-jiangxin" > version
$ git diff
diff --git a/version b/version
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
index 37c65f8..a58ac04 100644
--- a/version
+++ b/version
@@ -1 +1 @@
-1:0.2-dev
+1:0.2-jiangxin
$ repo status
# on branch jiangxin
project test/test1/                                branch jiangxin
 -m     version
$ git add -u
$ git commit -m "0.2-dev -> 0.2-jiangxin"
```

执行 make 命令，看看各个项目的改变。

```
$ make
Version of test1:    1:0.2-jiangxin
Version of test2:    2:0.2-jiangxin
Version of manifest: current
```

### 5. PUSH 到远程服务器

直接执行 repo push 就可以推送各个项目的改动。

```
$ repo push
```

如果有多个项目同时进行了改动，为了避免出错，会弹出编辑器显示因为包含改动而需要推送的项目列表。

```
# Uncomment the branches to upload:
#
# project test/test1/:
#  branch jiangxin ( 1 commit, Mon Oct 25 18:04:51 2010 +0800):
#         4f941239 0.2-dev -> 0.2-jiangxin
#
# project test/test2/:
#  branch jiangxin ( 1 commit, Mon Oct 25 18:06:51 2010 +0800):
#         86683ece 0.2-dev -> 0.2-jiangxin
```

每行前面的井号都是注释会被忽略。将希望推送的分支前的注释去掉，就可以将该项目的分支执行推送动作。下面的操作中把其中的两个分支的注释都去掉了，这两个项目当前分支的改动会 push 到上游服务器。

```
# Uncomment the branches to upload:
#
# project test/test1/:
branch jiangxin ( 1 commit, Mon Oct 25 18:04:51 2010 +0800):
#         4f941239 0.2-dev -> 0.2-jiangxin
#
# project test/test2/:
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
branch jiangxin ( 1 commit, Mon Oct 25 18:06:51 2010 +0800):
#         86683ece 0.2-dev -> 0.2-jiangxin
```

保存退出（如果使用 vi 编辑器，输入 <ESC>:wq 执行保存退出）后，马上开始对选择的各个项目执行 git push。

```
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 293 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test1.git
   27aee23..4f94123  jiangxin -> master
Counting objects: 5, done.
Writing objects: 100% (3/3), 261 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test2.git
   7f0841d..86683ec  jiangxin -> master

------------------------------------------
[OK    ] test/test1/     jiangxin
[OK    ] test/test2/     jiangxin
```

从推送的命令输出可以看出来，本地的工作分支 jiangxin 的改动被推送到远程服务器的 master 分支（本地工作分支跟踪的上游分支）上。

再次执行 repo push，会显示没有项目需要推送。

```
$ repo push
no branches ready for upload
```

### 6. 在远程服务器创建新分支

如果想在服务器上创建一个新的分支，该如何操作呢？如下使用 --new_branch 参数调用 repo push 命令。

```
$ repo start feature1 --all
$ repo push --new_branch
```

经过同样的编辑操作之后自动调用 git push，在服务器上创建新分支 feature1。

```
Total 0 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test1.git
 * [new branch]      feature1 -> feature1
Total 0 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test2.git
 * [new branch]      feature1 -> feature1

------------------------------------------
[OK    ] test/test1/     feature1
[OK    ] test/test2/     feature1
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


用 git ls-remote 命令查看远程版本库的分支，会发现远程版本库中已经建立了新的分支。

```
$ git ls-remote git://github.com/ossxp-com/test1.git refs/heads/*
4f9412399bf8093e880068477203351829a6b1fb        refs/heads/feature1
4f9412399bf8093e880068477203351829a6b1fb        refs/heads/master
b2b246b99ca504f141299ecdbadb23faf6918973        refs/heads/test-0.1
```

注意到 feature1 和 master 分支引用指向了相同的 SHA1 哈希值，这是因为 feature1 分支是直接从 master 分支创建的。

**7. 通过不同的清单库版本，切换到不同分支**

换用不同的清单库，需要建立新的工作区，并且在执行 repo init 时，通过 -b 参数指定清单库的分支。

```
$ mkdir test-0.1
$ cd test-0.1
$ repo init -u git://github.com/ossxp-com/manifest.git -b test-0.1
$ repo sync
```

当子项目代码全部同步完成后执行 make 命令。可以看到各个子项目的版本及清单库的版本不同于之前的输出。

```
$ make
Version of test1:    1:0.1.4
Version of test2:    2:0.1.3-dev
Version of manifest: current-2-g12f9080
```

可以用 repo manifest 命令来查看清单库。

```
$ repo manifest -o -
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote fetch="git://github.com/ossxp-com/" name="github"/>

  <default remote="github" revision="refs/heads/test-0.1"/>

  <project name="test1" path="test/test1">
    <copyfile dest="Makefile" src="root.mk"/>
  </project>
  <project name="test2" path="test/test2"/>
</manifest>
```

仔细看看上面的清单文件，可以注意到默认的版本指向到了 refs/heads/test-0.1 引用所指向的分支 test-0.1。

如果在子项目中修改、提交，然后使用 repo push 会将改动推送到远程版本库的 test-0.1 分支中。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 8. 切换到清单库里程碑版本

执行如下命令可以查看清单库包含的里程碑版本：

```
$ git ls-remote --tags git://github.com/ossxp-com/manifest.git
43e5783a58b46e97270785aa967f09046734c6ab        refs/tags/current
3a6a6da36840e716a14d52252e7b40e6ba6cbdea        refs/tags/current^{}
4735d32613eb50a6c3472cc8087ebf79cc46e0c0        refs/tags/v0.1
fb1a1b7302a893092ce8b356e83170eee5863f43        refs/tags/v0.1^{}
b23884d9964660c8dd34b343151aaf968a744400        refs/tags/v0.1.1
9c4c287069e29d21502472acac34f28896d7b5cc        refs/tags/v0.1.1^{}
127d9789cd4312ed279a7fa683c43eec73d2b28b        refs/tags/v0.1.2
47aaa83866f6d910a118a9a19c2ac3a2a5819b3e        refs/tags/v0.1.2^{}
af3abb7ed0a9ef7063e9d814510c527287c92ef6        refs/tags/v0.1.3
99c69bcfd7e2e7737cc62a7d95f39c6b9ffaf31a        refs/tags/v0.1.3^{}
```

可以从任意里程碑版本的清单库初始化整个项目。

```
$ mkdir v0.1.2
$ cd v0.1.2
$ repo init -u git://github.com/ossxp-com/manifest.git -b refs/tags/v0.1.2
$ repo sync
```

当子项目代码全部同步完成后执行 make 命令。可以看到各个子项目的版本及清单库的版本不同于之前的输出。

```
$ make
Version of test1:    1:0.1.2
Version of test2:    2:0.1.2
Version of manifest: v0.1.2
```





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 26 章 Git 和 SVN 协同模型

在本篇的最后，将会从另外一个角度来看版本库协同。不是不同的用户在使用 Git 版本库时如何协同，也不是一个项目包含多个 Git 版本库时如何协同，而是当版本控制系统不是 Git（如 Subversion）时，如何能够继续以 Git 的方式进行操作。

Subversion 会在商业软件开发中占有一席之地，因为依然会有公司需要严格而复杂的源代码授权。对于熟悉了 Git 的用户，一定会对 Subversion 的那种一旦脱离网络和服务器便寸步难行的工作模式厌烦透顶。实际上对 Subversion 的集中式版本控制的不满和改进在 Git 诞生之前就发生了，这就是 SVK[①]。

在 2003 年（Git 诞生的前两年），台湾的高嘉良就开发了 SVK，用分布式版本控制的方法操作 SVN。其设计思想非常朴素，既然 SVN 的用户可以看到有访问权限数据的全部历史，那么也应该能够依据历史重建一个本地的 SVN 版本库，这样很多 SVN 操作都可以通过本地的 SVN 进行，从而脱离网络。当对本地版本库的修改感到满意后，通过本地 SVN 版本和服务器 SVN 版本库之间的双向同步，将改动归并到服务器上。这种工作方式真的非常酷。

不必为 SVK 的文档缺乏及不再维护而感到惋惜，因为有更强的工具登场了，这就是 git-svn。git-svn 是 Git 软件包的一部分，用 Perl 语言开发。它的工作原理是：

- 将 Subversion 版本库在本地转换为一个 Git 库。
- 转换可以基于 Subversion 的某个目录，或者基于某个分支，或者整个 Subversion 代码库的所有分支和里程碑。
- 远程的 Subversion 版本库可以和本地的 Git 双向同步。Git 本地库修改推送到远程 Subversion 版本库，反之亦然。

git-svn 作为 Git 软件包的一部分，当 Git 从源码包进行安装时会默认安装，提供 `git svn` 命令。而几乎所有的 Linux 发行版都将 git-svn 作为一个独立的软件单独发布，因此需要单独安装。例如 Debian 和 Ubuntu 运行下面的命令安装 git-svn。

```
$ sudo aptitude install git-svn
```

将 git-svn 独立安装是因为 git-svn 软件包有着特殊的依赖，即依赖 Subversion 的 Perl 语言绑定接口，Debian/Ubuntu 上由 libsvn-perl 软件包提供。

当 git-svn 正确安装后，就可以使用 `git svn` 命令了。但如果在执行 `git svn --version` 时遇到下面的错误，则说明 Subversion 的 Perl 语言绑定没有正确安装。

---

① *http://svk.bestpractical.com/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git svn --version
Can't locate loadable object for module SVN::_Core in @INC (@INC contains:
/usr/share/perl/5.10.1 /etc/perl /usr/local/lib/perl/5.10.1
/usr/local/share/perl/5.10.1 /usr/lib/perl5 /usr/share/perl5
/usr/lib/perl/5.10 /usr/share/perl/5.10 /usr/local/lib/site_perl
/usr/local/lib/perl/5.10.0 /usr/local/share/perl/5.10.0 .) at
/usr/lib/perl5/SVN/Base.pm line 59
BEGIN failed--compilation aborted at /usr/lib/perl5/SVN/Core.pm line 5.
Compilation failed in require at /usr/lib/git-core/git-svn line 41.
```

遇到上面的情况，需要检查本机是否正确安装了 Subversion 及 Subversion 的 Perl 语言绑定。

为了便于对 git-svn 的介绍和演示，要有一个 Subversion 版本库，并且要有提交权限以便演示如何用 Git 向 Subversion 进行提交。下面就在本地创建一个 Subversion 版本库。

```
$ svnadmin create /path/to/svn/repos/demo

$ svn co file:///path/to/svn/repos/demo svndemo
取出版本 0

$ cd svndemo

$ mkdir trunk tags branches
$ svn add *
A         branches
A         tags
A         trunk

$ svn ci -m "initialized."
增加            branches
增加            tags
增加            trunk

提交后的版本为 1。
```

再向 Subversion 开发主线 trunk 中添加些数据。

```
$ echo hello > trunk/README
$ svn add trunk/README
A         trunk/README
$ svn ci -m "hello"
增加            trunk/README
传输文件数据 .
提交后的版本为 2。
```

建立分支：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ svn up
$ svn cp trunk branches/demo-1.0
A         branches/demo-1.0
$ svn ci -m "new branch: demo-1.0"
增加            branches/demo-1.0

提交后的版本为 3。
```

建立里程碑：

```
$ svn cp -m "new tag: v1.0" trunk file:///path/to/svn/repos/demo/tags/v1.0

提交后的版本为 4。
```

## 26.1 使用 git-svn 的一般流程

使用 git-svn 的一般流程参见图 26-1。

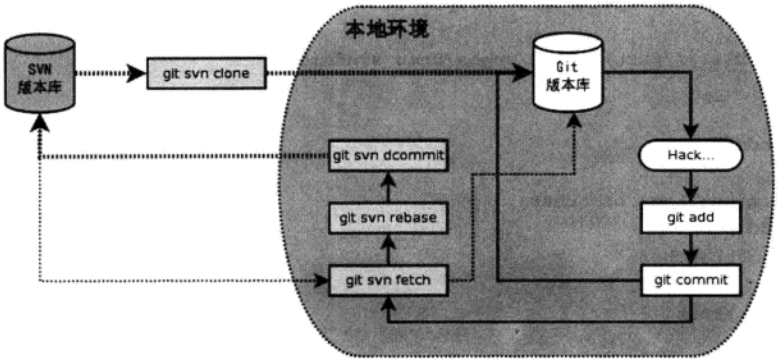



图 26-1 git-svn 工作流

首先用 `git svn clone` 命令对 Subversion 进行克隆，创建一个包含 git-svn 扩展的本地 Git 库。在下面的示例中，使用 Subversion 的本地协议（file://）来访问之前创立的 Subversion 示例版本库，实际上 git-svn 可以使用任何 Subversion 可用的协议，并可以对远程版本库进行操作。

```
$ git svn clone -s file:///path/to/svn/repos/demo git-svn-demo
Initialized empty Git repository in /path/to/my/workspace/git-svn-demo/.git/
r1 = 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4 (refs/remotes/trunk)
        A       README
r2 = 1863f91b45def159a3ed2c4c4c9428c25213f956 (refs/remotes/trunk)
Found possible branch point: file:///path/to/svn/repos/demo/trunk =>
file:///path/to/svn/repos/demo/branches/demo-1.0, 2
Found branch parent: (refs/remotes/demo-1.0)
1863f91b45def159a3ed2c4c4c9428c25213f956
Following parent with do_switch
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Successfully followed parent
r3 = 1adcd5526976fe2a796d932ff92d6c41b7eedcc4 (refs/remotes/demo-1.0)
Found possible branch point: file:///path/to/svn/repos/demo/trunk =>
file:///path/to/svn/repos/demo/tags/v1.0, 2
Found branch parent: (refs/remotes/tags/v1.0)
1863f91b45def159a3ed2c4c4c9428c25213f956
Following parent with do_switch
Successfully followed parent
r4 = c12aa40c494b495a846e73ab5a3c787ca1ad81e9 (refs/remotes/tags/v1.0)
Checked out HEAD:
  file:///path/to/svn/repos/demo/trunk r2
```

从上面的输出可以看出，当执行了 `git svn clone` 之后，在本地工作目录创建了一个 Git 库（`git-svn-demo`），并将 Subversion 的每一个提交都转换为 Git 库中的提交。进入 `git-svn-demo` 目录，看看用 git-svn 克隆出来的版本库。

```
$ cd git-svn-demo/
$ git branch -a
* master
  remotes/demo-1.0
  remotes/tags/v1.0
  remotes/trunk
$ git log
commit 1863f91b45def159a3ed2c4c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

    hello

    git-svn-id: file:///path/to/svn/repos/demo/trunk@2
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

    initialized.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@1
f79726c4-f016-41bd-acd5-6c9acb7664b2
```

看到 Subversion 版本库的分支和里程碑都被克隆出来，并保存在 `refs/remotes` 下的引用中。在 `git log` 的输出中，可以看到 Subversion 的提交的确被转换为 Git 的提交。

下面就可以在 Git 库中进行修改，并在本地提交（用 `git commit` 命令）。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cat README
hello
$ echo "I am fine." >> README
$ git add -u
$ git commit -m "my hack 1."
[master 55e5fd7] my hack 1.
 1 files changed, 1 insertions(+), 0 deletions(-)
$ echo "Thank you." >> README
$ git add -u
$ git commit -m "my hack 2."
[master f1e00b5] my hack 2.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

对工作区中的 README 文件修改了两次，并进行了本地的提交。查看这时的提交日志，会发现最新的两个提交和之前的历次提交略有不同，最新的两个提交的提交说明中不包含 git-svn-id: 标记。

```
$ git log
commit f1e00b52209f6522dd8135d27e86370de552a7b6
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Nov 4 15:05:47 2010 +0800

    my hack 2.

commit 55e5fd794e6208703aa999004ec2e422b3673ade
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Nov 4 15:05:32 2010 +0800

    my hack 1.

commit 1863f91b45def159a3ed2c4c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

    hello

    git-svn-id: file:///path/to/svn/repos/demo/trunk@2
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

    initialized.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@1
f79726c4-f016-41bd-acd5-6c9acb7664b2
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


现在就可以向 Subversion 服务器推送改动了。但在真实的环境中，往往在向服务器推送时，已经有其他用户在服务器上进行了提交，而且更糟的是，先于我们的提交会造成我们的提交冲突！现在就人为地制造一个冲突：使用 svn 命令在 Subversion 版本库中执行一次提交。

```
$ svn checkout file:///path/to/svn/repos/demo/trunk demo
A    demo/README
取出版本 4。
$ cd demo/
$ cat README
hello
$ echo "HELLO." > README
$ svn commit -m "hello -> HELLO."
正在发送        README
传输文件数据 .
提交后的版本为 5。
```

好的，已经模拟了一个用户先于我们更改了 Subversion 版本库。现在回到用 git-svn 克隆的本地版本库，执行 `git svn dcommit` 操作，将 Git 中的提交推送到 Subversion 版本库中。

```
$ git svn dcommit
Committing to file:///path/to/svn/repos/demo/trunk ...
事务过时：文件 "/trunk/README" 已经过时 at /usr/lib/git-core/git-svn line 572
```

显然，由于 Subversion 版本库中包含了新的提交，导致执行 `git svn dcommit` 出错。这时须执行 `git svn fetch` 命令，以从 Subversion 版本库获取更新。

```
$ git svn fetch
        M       README
r5 = fae6dab863ed2152f71bcb2348d476d47194fdd4 (refs/remotes/trunk)
$ git status
# On branch master
nothing to commit (working directory clean)
```

当获取了新的 Subversion 提交之后，需要执行 `git svn rebase` 将 Git 中未推送到 Subversion 的提交通过变基操作转化为继 Subversion 最新提交的线性提交。这是因为 Subversion 的提交都是线性的。

```
$ git svn rebase
First, rewinding head to replay your work on top of it...
Applying: my hack 1.
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging README
CONFLICT (content): Merge conflict in README
Failed to merge in the changes.
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Patch failed at 0001 my hack 1.

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".

rebase refs/remotes/trunk: command returned error: 1
```

果不其然，变基时发生了冲突，这是因为 Subversion 中他人的修改和我们在 Git 库中的修改都改动了同一个文件，并且改动了相近的行。下面按照 git rebase 冲突解决的标准步骤进行，直到成功完成变基操作，具体操作步骤如下。

（1）先编辑 README 文件以解决冲突。

```
$ git status
# Not currently on any branch.
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      README
#
no changes added to commit (use "git add" and/or "git commit -a")

$ vi README
```

（2）处于冲突状态的 REAEME 文件的内容。

```
<<<<<<< HEAD
HELLO.
=======
hello
I am fine.
>>>>>>> my hack 1.
```

（3）下面是修改后的内容，保存退出。

```
HELLO.
I am fine.
```

（4）执行 git add 命令解决冲突。

```
$ git add README
```

（5）调用 git rebase --continue 完成变基操作。

```
$ git rebase --continue
Applying: my hack 1.
Applying: my hack 2.
Using index info to reconstruct a base tree...
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Falling back to patching base and 3-way merge...
Auto-merging README
```

（6）看看变基之后的 Git 版本库日志：

```
$ git log
commit e382f2e99eca07bc3a92ece89f80a7a5457acfd8
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Nov 4 15:05:47 2010 +0800

    my hack 2.

commit 6e7e0c7dccf5a072404a28f06ce0c83d77988b0b
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Nov 4 15:05:32 2010 +0800

    my hack 1.

commit fae6dab863ed2152f71bcb2348d476d47194fdd4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Thu Nov 4 07:15:58 2010 +0000

    hello -> HELLO.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@5
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit 1863f91b45def159a3ed2c4c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

    hello

    git-svn-id: file:///path/to/svn/repos/demo/trunk@2
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

    initialized.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@1
f79726c4-f016-41bd-acd5-6c9acb7664b2
```

（7）当变基操作成功完成后，再执行 `git svn dcommit` 向 Subversion 推送 Git 库中的两个新提交。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git svn dcommit
Committing to file:///path/to/svn/repos/demo/trunk ...
        M       README
Committed r6
        M       README
r6 = d0eb86bdfad4720e0a24edc49ec2b52e50473e83 (refs/remotes/trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
Unstaged changes after reset:
M       README
        M       README
Committed r7
        M       README
r7 = 69f4aa56eb96230aedd7c643f65d03b618ccc9e5 (refs/remotes/trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

（8）推送之后本地 Git 库中最新的两个提交的提交说明中也嵌入了 git-svn-id: 标签。这个标签的作用非常重要，在下一节会予以介绍。

```
$ git log -2
commit 69f4aa56eb96230aedd7c643f65d03b618ccc9e5
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Thu Nov 4 07:56:38 2010 +0000

    my hack 2.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@7
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit d0eb86bdfad4720e0a24edc49ec2b52e50473e83
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Thu Nov 4 07:56:37 2010 +0000

    my hack 1.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@6
f79726c4-f016-41bd-acd5-6c9acb7664b2
```

## 26.2 git-svn 的奥秘

通过上面对 git-svn 的工作流程的介绍，相信您已经能够体会到 git-svn 的强大。那么 git-svn 是怎么做到的呢？

git-svn 只是在本地 Git 库中增加了一些附加的设置和特殊的引用，并引入了附加的可重建的数据库实现对 Subversion 版本库的跟踪。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 26.2.1 Git 库配置文件的扩展及分支映射

当执行 `git svn init` 或 `git svn clone` 时，git-svn 会通过在 Git 库的配置文件中增加一个小节，记录 Subversion 版本库的 URL，以及 Subversion 分支 / 里程碑和本地 Git 库的引用之间的对应关系。

例如：当执行 `git svn clone -s file:///path/to/svn/repos/demo` 指令时，会在创建的本地 Git 库的配置文件 `.git/config` 中引入下面的新配置：

```
[svn-remote "svn"]
        url = file:///path/to/svn/repos/demo
        fetch = trunk:refs/remotes/trunk
        branches = branches/*:refs/remotes/*
        tags = tags/*:refs/remotes/tags/*
```

默认 svn-remote 的名字为 `svn`，所以新增的配置小节的名字为：`[svn-remote "svn"]`。在 git-svn 克隆时，可以使用 `--remote` 参数设置不同的 svn-remote 名称，但是并不建议使用。因为一旦使用 `--remote` 参数更改 svn-remote 名称，必须在 git-svn 的其他命令中都使用 `--remote` 参数，否则报告 `[svn-remote "svn"]` 配置小节未找到。

该小节中主要的配置有：

- url = <URL>

  设置 Subversion 版本库的地址。
- fetch = <svn-path>:<git-refspec>

  Subversion 的开发主线和 Git 版本库引用的对应关系。

  在上例中 Subversion 的 trunk 目录对应于 Git 的 refs/remotes/trunk 引用。
- branches = <svn-path>:<git-refspec>

  Subversion 的开发分支和 Git 版本库引用的对应关系。可以包含多条 branches 的设置，以便将分散在不同目录下的分支汇总。

  在上例中 Subversion 的 branches 子目录的下一级子目录（branches/*）所代表的分支在 Git 的 refs/remotes/ 下建立引用。
- tags = <svn-path>:<git-refspec>

  Subversion 的里程碑和 Git 版本库引用的对应关系，可以包含多条 tags 的设置，以便将分散在不同目录下的里程碑汇总。

  在上例中 Subversion 的 tags 子目录的下一级子目录（tags/*）所代表的里程碑在 Git 的 refs/remotes/tags 下建立引用。

可以看到 Subversion 的主线和分支默认都直接被映射到 `refs/remotes/` 下。如 trunk 主线对应于 `refs/remotes/trunk`，分支 demo-1.0 对应于 `refs/remotes/demo-1.0`。Subversion 的里程碑因为有可能和分支同名，因此被映射到 `refs/remotes/tags/` 之下，这样里程碑和分支的映射就放到了不同的目录下，不会互相影响。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 26.2.2 Git 工作分支和 Subversion 如何对应

Git 默认的工作分支是 master，而看到上例中的 Subversion 主线在 Git 中对应的远程分支为 refs/remotes/trunk。那么在执行 git svn rebase 时，git-svn 是如何知道当前的 HEAD 对应的分支是基于哪个 Subversion 跟踪分支进行的变基呢？还有就是执行 git svn dcommit 时，当前的工作分支应该将改动推送到哪个 Subversion 分支中去呢？

很自然地会按照 Git 的方式进行思考，期望在 .git/config 配置文件中找到类似 [branch master] 之类的配置小节。实际上，在 git-svn 的 Git 库的配置文件中可能根本就不存在 [branch ...] 小节。那么 git-svn 是如何确定当前 Git 工作分支和远程 Subversion 版本库的分支建立了对应呢？

其实奥秘就在 Git 的日志中。当在工作区执行 git log 时，会看到包含 git-svn-id: 标识的特殊日志。发现的最近的一个 git-svn-id: 标识会确定当前分支提交的 Subversion 分支。

下面继续上一节的示例，先切换到分支，并将提交推送到 Subversion 的分支 demo-1.0 中，具体操作过程如下。

（1）首先在 Git 库中会看到有一个对应于 Subversion 分支的远程分支和一个对应于 Subversion 里程碑的远程引用。

```
$ git branch -r
  demo-1.0
  tags/v1.0
  trunk
```

（2）然后基于远程分支 demo-1.0 建立本地工作分支 myhack。

```
$ git checkout -b myhack refs/remotes/demo-1.0
Switched to a new branch 'myhack'
$ git branch
  master
* myhack
```

（3）在 myhack 分支做一些改动并提交。

```
$ echo "Git" >> README
$ git add -u
$ git commit -m "say hello to Git."
[myhack d391fd7] say hello to Git.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

（4）下面看看 Git 的提交日志。

```
$ git log --first-parent
commit d391fd75c33f62307c3add1498987fa3eb70238e
Author: Jiang Xin <jiangxin@ossxp.com>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Date:   Fri Nov 5 09:40:21 2010 +0800

    say hello to Git.

commit 1adcd5526976fe2a796d932ff92d6c41b7eedcc4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:54:19 2010 +0000

    new branch: demo-1.0

    git-svn-id: file:///path/to/svn/repos/demo/branches/demo-1.0@3
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit 1863f91b45def159a3ed2c4c4c9428c25213f956
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:49:41 2010 +0000

    hello

    git-svn-id: file:///path/to/svn/repos/demo/trunk@2
f79726c4-f016-41bd-acd5-6c9acb7664b2

commit 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
Author: jiangxin <jiangxin@f79726c4-f016-41bd-acd5-6c9acb7664b2>
Date:   Mon Nov 1 05:47:03 2010 +0000

    initialized.

    git-svn-id: file:///path/to/svn/repos/demo/trunk@1
f79726c4-f016-41bd-acd5-6c9acb7664b2
```

（5）看到了上述 Git 日志中出现的第一个 `git-svn-id:` 标识的内容为：

```
git-svn-id: file:///path/to/svn/repos/demo/branches/demo-1.0@3
f79726c4-f016-41bd-acd5-6c9acb7664b2
```

这就是说，当需要将 Git 提交推送给 Subversion 服务器时，需要推送到地址：`file:///path/to/svn/repos/demo/branches/demo-1.0`。

（6）执行 `git svn dcommit`，果然是推送到 Subversion 的 demo-1.0 分支。

```
$ git svn dcommit
Committing to file:///path/to/svn/repos/demo/branches/demo-1.0 ...
        M       README
Committed r8
        M       README
r8 = a8b32d1b533d308bef59101c1f2c9a16baf91e48 (refs/remotes/demo-1.0)
No changes between current HEAD and refs/remotes/demo-1.0
Resetting to the latest refs/remotes/demo-1.0
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 26.2.3 其他辅助文件

在 Git 版本库中，git-svn 在 .git/svn 目录下保存了一些索引文件，便于 git-svn 更加快速地执行。

.git/svn/.metadata 文件是类似于 .git/config 文件一样的 INI 文件，其中保存了版本库的 URL、版本库 UUID、分支和里程碑的最后获取的版本号等。

```
; This file is used internally by git-svn
; You should not have to edit it
[svn-remote "svn"]
        reposRoot = file:///path/to/svn/repos/demo
        uuid = f79726c4-f016-41bd-acd5-6c9acb7664b2
        branches-maxRev = 8
        tags-maxRev = 8
```

在 .git/svn/refs/remotes 目录下，以各个分支和里程碑为名的各个子目录下都包含了一个名为 .rev_map.<SVN-UUID> 的索引文件，这个文件用于记录 Subversion 的提交 ID 和 Git 的提交 ID 的映射。

目录 .git/svn 的辅助文件由 git-svn 维护，不要手工修改否则会造成 git-svn 不能正常工作。

## 26.3 多样的 git-svn 克隆模式

在前面的 git-svn 示例中，使用 git svn clone 命令完成对远程版本库的克隆，实际上 git svn clone 相当于两条命令，即：

```
git svn clone = git svn init + git svn fetch
```

命令 git svn init 只完成两个工作。一是在本地建立一个空的 Git 版本库，另外是修改 .git/config 文件，在其中建立 Subversion 和 Git 之间的分支映射关系。在实际使用中，我更喜欢使用 git svn init 命令，因为这样可以对 Subversion 和 Git 的分支映射进行手工修改。该命令的用法是：

```
用法：git svn init [options] <subversion-url> [local-dir]

可选的主要参数有：

    --stdlayout, -s
    --trunk, -T <arg>
    --branches, --b=s@
    --tags, --t=s@
    --config-dir <arg>
    --ignore-paths <arg>
    --prefix <arg>
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    --username <arg>
```

其中 --username 参数用于设定远程 Subversion 服务器认证时提供的用户名。参数 --prefix 用于设置在 Git 的 refs/remotes 下保存引用时使用的前缀。参数 --ignore-paths 后面跟一个正则表达式定义忽略的文件列表，这些文件将不予克隆。

最常用的参数是 -s。该参数和前面演示的 git clone 命令中的一样，即使用标准的分支 / 里程碑部署方式克隆 Subversion 版本库。Subversion 约定俗成使用 trunk 目录跟踪主线的开发，使用 branches 目录保存各个分支，使用 tags 目录来记录里程碑。

即命令：

```
$ git svn init -s file:///path/to/svn/repos/demo
```

和下面的命令等效：

```
$ git svn init -T trunk -b branches -t tags file:///path/to/svn/repos/demo
```

有的 Subversion 版本库的分支可能分散于不同的目录下，例如有的位于 branches 目录，有的位于 sandbox 目录，可以使用下面的命令：

```
$ git svn init -T trunk -b branches -b sandbox -t tags
file:///path/to/svn/repos/demo git-svn-test
Initialized empty Git repository in /path/to/my/workspace/git-svn-test/.git/
```

查看本地克隆版本库的配置文件：

```
$ cat git-svn-test/.git/config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[svn-remote "svn"]
        url = file:///path/to/svn/repos/demo
        fetch = trunk:refs/remotes/trunk
        branches = branches/*:refs/remotes/*
        branches = sandbox/*:refs/remotes/*
        tags = tags/*:refs/remotes/tags/*
```

可以看到在 [svn-remote "svn"] 小节中包含了两条 branches 配置，这就会实现将 Subversion 分散于不同目录的分支都克隆出来。如果担心 Subversion 的 branches 目录和 sandbox 目录下出现同名的分支，从而导致在 Git 库的 refs/remotes/ 下造成覆盖，可以在版本库尚未执行 git svn fetch 之前编辑 .git/config 文件，避免可能出现的覆盖。例如编辑后的 [svn-remote "svn"] 配置小节：

```
[svn-remote "svn"]
        url = file:///path/to/svn/repos/demo
        fetch = trunk:refs/remotes/trunk
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        branches = branches/*:refs/remotes/branches/*
        branches = sandbox/*:refs/remotes/sandbox/*
        tags = tags/*:refs/remotes/tags/*
```

如果项目的分支或里程碑非常多，也可以修改[svn-remote "svn"]配置小节中的版本号通配符，使得只获取部分分支或里程碑。例如下面的配置小节：

```
[svn-remote "svn"]
        url = http://server.org/svn
        fetch = trunk/src:refs/remotes/trunk
        branches = branches/{red,green}/src:refs/remotes/branches/*
        tags = tags/{1.0,2.0}/src:refs/remotes/tags/*
```

如果只关心Subversion的某个分支甚至某个子目录，而不关心其他分支或目录，那就更简单了，不带参数地执行git svn init针对Subversion的某个具体路径执行初始化就可以了。

```
$ git svn init file:///path/to/svn/repos/demo/trunk
```

有的情况下，版本库太大，而且对历史不感兴趣，可以只克隆最近的部分提交。这时可以通过git svn fetch命令的-r参数实现部分提交的克隆。

```
$ git svn init file:///path/to/svn/repos/demo/trunk git-svn-test
Initialized empty Git repository in /path/to/my/workspace/git-svn-test/.git/
$ cd git-svn-test
$ git svn fetch -r 6:HEAD
        A       README
r6 = 053b641b7edd2f1a59a007f27862d98fe5bcda57 (refs/remotes/git-svn)
        M       README
r7 = 75c17ea61d8527334855a51e65ac98c981f545d7 (refs/remotes/git-svn)
Checked out HEAD:
  file:///path/to/svn/repos/demo/trunk r7
```

当然也可以使用git svn clone命令实现部分克隆：

```
$ git svn clone -r 6:HEAD \
      file:///path/to/svn/repos/demo/trunk git-svn-test
Initialized empty Git repository in /path/to/my/workspace/git-svn-test/.git/
        A       README
r6 = 053b641b7edd2f1a59a007f27862d98fe5bcda57 (refs/remotes/git-svn)
        M       README
r7 = 75c17ea61d8527334855a51e65ac98c981f545d7 (refs/remotes/git-svn)
Checked out HEAD:
  file:///path/to/svn/repos/demo/trunk r7
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 26.4 共享 git-svn 的克隆库

当一个 Subversion 版本库非常庞大而且还不在同一个局域网内，执行 git svn clone 可能需要花费很多时间。为了避免因重复执行 git svn clone 而导致时间上的浪费，可以将一个已经使用 git-svn 克隆出来的 Git 库共享，其他人基于此 Git 进行克隆，然后再用特殊的方法重建和 Subversion 的关联。还记得之前提到过 .git/svn 目录下的辅助文件可以重建吗？

例如通过工作区中已经存在的 git-svn-demo 执行克隆。

```
$ git clone git-svn-demo myclone
Initialized empty Git repository in /path/to/my/workspace/myclone/.git/
```

进入新的克隆中，会发现新的克隆缺乏跟踪 Subversion 分支的引用，即 refs/remotes/trunk 等。

```
$ cd myclone/
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/myhack
```

这是因为 Git 克隆默认不复制远程版本库的 refs/remotes/ 下的引用。可以用 git fetch 命令获取 refs/remotes 的引用。

```
$ git fetch origin refs/remotes/*:refs/remotes/*
From /path/to/my/workspace/git-svn-demo
 * [new branch]      demo-1.0   -> demo-1.0
 * [new branch]      tags/v1.0  -> tags/v1.0
 * [new branch]      trunk      -> trunk
```

现在这个从 git-svn 库中克隆出来的版本库已经有了相同的 Subversion 跟踪分支，但是 .git/config 文件还缺乏相应的 [svn-remote "svn"] 配置。可以通过使用同样的 git svn init 命令实现。

```
$ pwd
/path/to/my/workspace/myclone

$ git svn init -s file:///path/to/svn/repos/demo

$ git config --get-regexp 'svn-remote.*'
svn-remote.svn.url file:///path/to/svn/repos/demo
svn-remote.svn.fetch trunk:refs/remotes/trunk
svn-remote.svn.branches branches/*:refs/remotes/*
svn-remote.svn.tags tags/*:refs/remotes/tags/*
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


但是克隆版本库相比用 git-svn 克隆的版本库还缺乏 .git/svn 下的辅助文件。实际上可以用 git svn rebase 命令重建，同时这条命令也可以变基到 Subversion 相应分支的最新提交上。

```
$ git svn rebase
Rebuilding .git/svn/refs/remotes/trunk/.rev_map.f79726c4-f016-41bd-acd5-6c9ac
b7664b2 ...
r1 = 2c73d657dfc3a1ceca9d465b0b98f9e123b92bb4
r2 = 1863f91b45def159a3ed2c4c4c9428c25213f956
r5 = fae6dab863ed2152f71bcb2348d476d47194fdd4
r6 = d0eb86bdfad4720e0a24edc49ec2b52e50473e83
r7 = 69f4aa56eb96230aedd7c643f65d03b618ccc9e5
Done
rebuilding .git/svn/refs/remotes/trunk/.rev_map.f79726c4-f016-41bd-acd5-6c9ac
b7664b2
Current branch master is up to date.
```

如果执行 git svn fetch 则会对所有的分支都进行重建。

```
$ git svn fetch
Rebuilding .git/svn/refs/remotes/demo-1.0/.rev_map.f79726c4-f016-41bd-acd5-6c
9acb7664b2 ...
r3 = 1adcd5526976fe2a796d932ff92d6c41b7eedcc4
r8 = a8b32d1b533d308bef59101c1f2c9a16baf91e48
Done
rebuilding .git/svn/refs/remotes/demo-1.0/.rev_map.f79726c4-f016-41bd-acd5-6c
9acb7664b2
Rebuilding .git/svn/refs/remotes/tags/v1.0/.rev_map.f79726c4-f016-41bd-acd5-6c
9acb7664b2 ...
r4 = c12aa40c494b495a846e73ab5a3c787ca1ad81e9
Done
rebuilding .git/svn/refs/remotes/tags/v1.0/.rev_map.f79726c4-f016-41bd-acd5-6
c9acb7664b2
```

至此，从 git-svn 克隆库二次克隆的 Git 库，已经和原生的 git-svn 库一样使用 git-svn 命令了。

## 26.5 git-svn 的局限

Subversion 和 Git 的分支实现有着巨大的不同。Subversion 的分支和里程碑，是用轻量级拷贝实现的，虽然创建分支和里程碑的速度也很快，但是很难维护。即使 Subversion 在 1.5 之后引入了 svn:mergeinfo 属性对合并过程进行标记，但是也不可能让 Subversion 的分支逻辑更清晰。git-svn 无须利用 svn:mergeinfo 属性也可实现对 Subversion 合并的追踪，在合并的时候也不会对 svn:mergeinfo 属性进行更改，因此在使用 git-svn 操作时，如果在不同分支间进行合并，会导致 Subversion 的 svn:mergeinfo 属性没有相应的更新，

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


从而导致 Subversion 用户进行合并时因为重复合并而冲突。

简而言之，在使用 git-svn 时尽量不要在不同的分支之间进行合并，而是尽量在一个分支下进行线性的提交。这种线性的提交会很好地推送到 Subversion 服务器中。

如果真的需要在不同的 Subversion 分支之间合并，尽量使用 Subversion 的客户端（svn 1.5 版本或以上）执行，因为这样可以正确地记录 `svn:mergeinfo` 属性。当 Subversion 完成分支合并后，在 git-svn 的克隆库中执行 `git svn rebase` 命令获取最新的 Subversion 提交并变基到相应的跟踪分支中。





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第5篇 搭建Git服务器

如果不是要和他人协同开发，Git 根本就不需要架设服务器，因为 Git 可以直接使用本地路径操作本地的版本库及完成本地版本库间的协同。

但是如果需要和他人分享版本库、协作开发，或者想要通过网络为个人的版本库建立一个远程容灾备份，就会涉及服务器搭建，以及使用特定的网络协议操作 Git 库。

Git 支持的协议很丰富，架设服务器的选择也很多，不同的方案有着各自的优缺点，如下表所示。

**Git 服务器架设方案对照表**

| | HTTP | Git-daemon | SSH | Gitosis/Gitolite |
|---|---|---|---|---|
| 服务架设 | 简单 | 中等 | 简单 | 复杂 |
| 匿名读取 | 支持 | 支持 | 否* | 否* |
| 身份认证 | 支持 | 否 | 支持 | 支持 |
| 版本库写操作 | 支持 | 否 | 支持 | 支持 |
| 企业级授权 | 否 | 否 | 否 | 支持 |
| 远程建库 | 否 | 否 | 否 | 支持 |

注：*SSH 协议和基于 SSH 的 Gitolite 等可以通过空口令账号实现匿名访问。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 27 章　使用 HTTP 协议

HTTP 协议是版本控制非常重要的一种协议，具有安全（使用 HTTPS 的情况下）、方便（可以跨越防火墙）等优点。Git 在 1.6.6 版本之前对 HTTP 协议的支持有限，是哑协议，访问效率低，但是在 1.6.6 之后，通过一个 CGI 实现了智能的 HTTP 协议支持。

## 27.1　哑传输协议

在 Git 1.6.6 之前，若要通过 HTTP 协议提供 Git 服务，简单到直接拷贝 Git 版本库到 Web 服务器中就可以了，即将 Git 的裸版本库（不带工作区）作为一个 Web 静态目录直接开放给用户即可[①]。

### 1. 只读的 HTTP 哑协议

如下的 Apache 配置，提供了 Git 版本库的只读访问服务：

```
Alias /git /path/to/repos

<Directory /path/to/repos>
    Options FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

当用户执行 `git clone http://server/git/myrepo.git` 时，实际访问的是服务器端 `/path/to/repos/myrepo.git` 路径中的版本库。

以 Web 服务器静态目录方式提供 Git 服务，对 Git 版本库有一个特别的要求，即版本库目录下必须存在两个索引文件。其中一个是文件 `.git/info/refs`，包含了版本库中所有引用的列表，且包含引用对应的 SHA1 哈希值。另外一个文件是 `.git/objects/info/packs`，包含了所有打包文件的路径，以便在对象库打包后，能够通过该索引文件找到打包文件。

这是因为在 Web 哑协议下，作为客户端的 Git 类似于一个 Web 浏览器，远程的 Web 服务器只提供 Git 版本库文件的静态访问，而不像其他 Git 智能服务器那样能够提供帮助以获取 Git 版本库的引用和对象库文件。即作为 Web 客户端的 Git 仅凭一己之力主动到服务器上抓取文件。一旦严格配置的 Web 服务器禁止目录浏览（最安全和最常用的配置），如果不通

① 实际上还需要执行 git update-server-info 命令，以更新版本库中的几个索引文件。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


过事先约定的索引文件（.git/info/refs 和 .git/objects/info/packs 文件），Git 作为 Web 客户端是无法获得版本库的引用列表和打包文件列表的。

在 Git 版本库中创建这两个索引文件很简单，只要执行 git update-server-info 命令即可。但是要随着版本库的更新不断地维护 .git/info/refs 和 .git/objects/info/packs 这几个文件，以便使用哑协议的 Git 客户端始终能够正常地访问 Git 版本库可不是一件简单的工作。幸好 Git 提供了钩子脚本扩展机制，可以实现随着 Git 版本库的更新同步地更改这几个文件。

在版本库的钩子脚本目录 .git/hooks 中，创建一个名为 post-update 的钩子脚本。实际上在钩子脚本的目录中已经存在一个示例脚本 post-update.sample，直接将其重命名为 post-update，并将其设置为可执行即可。脚本 post-update 非常简单，内容如下：

```
#!/bin/sh

exec git update-server-info
```

### 2. 可读写的 HTTP 哑协议

上面配置的 HTTP 哑协议只提供 Git 版本库的只读访问，如果需要提供可写的 Git 版本库服务，即允许远程客户端推送，那就还需要在 Apache 中为版本库逐一启用 WebDAV 支持。例如 Apache 中的配置如下：

```
Alias /git /path/to/repos

<Directory /path/to/repos>
    Options FollowSymLinks
    AllowOverride None
    Order allow,deny
    allow from all
</Directory>

<Location /git/myrepos.git>
    DAV on
    AuthType Basic
    AuthName "Git"
    AuthBasicProvider file
    AuthUserFile /path/to/file/passwd
    AuthGroupFile /path/to/file/group
    Require group git
    Satisfy All
</Location>
```

无论是只读还是支持写入，以 HTTP 哑传输协议配置的 Git 服务器都有着非常明显的缺点：

（1）数据传输效率低。当版本库经过整理，各个零散的提交文件被打包后，只获取某一个或某几个提交也需要对整个打包文件进行传输！

（2）传输过程无进度显示。哑协议在 Git 操作过程中不能像其他协议那样显示进度，在

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


操作大的版本库时非常不便。

（3）为版本库提供写操作需要对每个版本库进行单独配置。缺乏类似Subversion的WebDAV插件，使得需要为每个Git库逐一进行设置。

（4）需要在服务器端手工创建版本库，而不能通过远程推送操作来实现在客户端发起版本库的创建。

（5）Git客户端不像Subversion那样提供口令缓存机制，如果要避免每次操作时频繁地输入口令，需要在URL中记录明文口令。

```
git clone https://username:password@server/path/to/repos/myrepo.git
```

下面将要介绍的智能HTTP协议则没有上面所述的大多数1、2和3缺点，在很大程度上完善了在HTTP协议上架设Git版本库的用户体验。

## 27.2 智能HTTP协议

Git 1.6.6之后的版本提供了针对HTTP协议的CGI程序git-http-backend，实现了智能的HTTP协议支持。但同时要求Git客户端的版本不低于1.6.6。

查看文件git-http-backend的安装位置，可以用如下命令。

```
$ ls $(git --exec-path)/git-http-backend
/usr/lib/git-core/git-http-backend
```

更改Apache的配置文件，以使用CGI提供智能Git访问服务。相关的Apache配置如下：

```
SetEnv GIT_PROJECT_ROOT /path/to/repos
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

说明：

- 第一行设置版本库的根目录为/path/to/repos。
- 第二行设置所有版本库均可访问，无论在版本库中是否存在git-daemon-export-ok文件。

  默认只有在版本库目录中存在git-daemon-export-ok文件时，该版本库才可以访问。这个文件是git-daemon服务的一个特性。
- 第三行，就是使用名为git-http-backend的CGI脚本来响应客户端的请求。

  当访问http://server/git/myrepos.git时，即由此CGI提供服务。

### 1. 写操作授权

上面的配置只能提供版本库的读取服务，若想提供基于HTTP协议的写操作，必须添加认证配置指令。用户通过认证后才能对版本库进行写操作。

下面的Apache配置中，在前面配置的基础上为Git写操作提供授权：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
<LocationMatch "^/git/.*/git-receive-pack$">
  AuthType Basic
  AuthName "Git Access"
  AuthBasicProvider file
  AuthUserFile /path/to/passwd/file
  ...
</LocationMatch>
```

### 2. 读和写均需授权

如果需要对读操作也进行授权，那就更简单了。下面的配置通过一个 Location 语句实现了对路径 /path/private 下版本库的读写授权。

```
<Location /git/private>
  AuthType Basic
  AuthName "Git Access"
  AuthBasicProvider file
  AuthUserFile /path/to/passwd/file
  ...
</Location>
```

### 3. 对静态文件的直接访问

如果对静态文件的访问不经过 CGI 程序，直接由 Apache 提供服务，会提高访问性能。

下面的 Apache 配置设置了直接通过 Apache（绕过 CGI）访问 Git 版本库中对象库下的文件。

```
SetEnv GIT_PROJECT_ROOT /path/to/repos

AliasMatch ^/git/(.*/objects/[0-9a-f]{2}/[0-9a-f]{38})$          /path/to/repos/$1
AliasMatch ^/git/(.*/objects/pack/pack-[0-9a-f]{40}.(pack|idx))$ /path/to/repos/$1
ScriptAlias /git/ /usr/libexec/git-core/git-http-backend/
```

Git 的智能 HTTP 服务彻底打破了以前哑传输协议给 HTTP 协议带来的恶劣印象，让 HTTP 协议成为 Git 服务的一个重要选项。但是在授权的管理上，智能 HTTP 服务仅仅依赖 Apache 自身的授权模型，相比后面要介绍的 Gitosis 和 Gitolite，可管理性要弱得多。表现如下：

- 创建版本库只能在服务器端进行，不能通过远程客户端进行。
- 配置认证和授权，也只能在服务器端进行，不能在客户端远程配置。
- 版本库的写操作授权只能进行非 0 即 1 的授权，不能针对分支甚至路径进行授权。

如果需要企业级的版本库管理，可以考虑后面介绍的基于 SSH 协议的 Gitolite 或 Gitosis。

## 27.3 Gitweb 服务器

前面介绍的 HTTP 哑协议和智能协议服务架设，都可以用于提供 Git 版本库的读写服务，而本节介绍的 Gitweb 作为一个 Web 应用，只提供版本库的图形化浏览功能，而不能提供版



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


本库读写访问的功能。

Gitweb 是用 Perl 语言开发的 CGI 脚本，既可以通过配置架设于 Web 服务器下，也可以无须任何配置针对单独 Git 版本库即时启动。Gitweb 支持多个版本库，可以对版本库进行目录浏览（包括历史版本），可以查看文件内容，查看提交历史，提供搜索及 RSS feed 支持，也可以提供目录文件的打包下载等。图 27-1 就是 kernel.org 上的 Gitweb 示例。

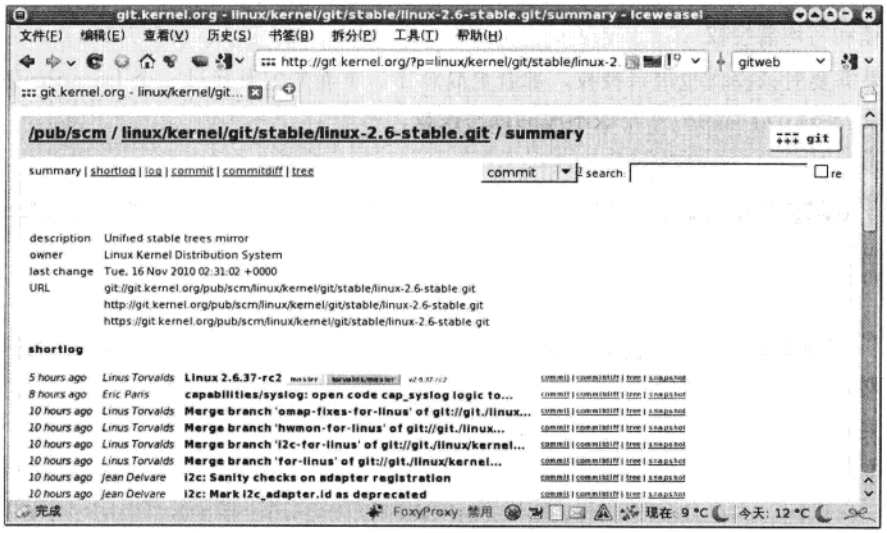



图 27-1 Gitweb 界面（kernel.org）

## 27.3.1 Gitweb 的安装

### 1. 使用包管理器安装

各个 Linux 平台大都提供了 Gitweb 软件包，可以使用相应的包管理器进行安装。例如在 Debian/Ubuntu 上安装 Gitweb：

```
$ sudo aptitude install gitweb
```

安装 Gitweb 后会在系统中创建下列文件：

- 配置文件：/etc/gitweb.conf
- Apache 配置文件：/etc/apache2/conf.d/gitweb
- CGI 脚本：/usr/share/gitweb/index.cgi
- 其他附属文件：/usr/share/gitweb/*
- 图片和 css 等

其中配置文件 /etc/apache2/conf.d/gitweb 用于完成和 Web 服务器 Apache2 的整合。当 Apache2 重启后，就可以通过 URL 地址 *http://server/gitweb* 访问 Gitweb 服务。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 2. 使用 Git 源码安装

Gitweb 的代码位于 Git 的源码库中，如果 Git 是从源码进行安装的，那么 Gitweb 应该已经安装好了。通过下面的命令可以查看 Gitweb 的安装位置：

```
$ ls -F $(dirname $(dirname $(git --html-path)))/gitweb
gitweb.cgi* static/
$ echo $(dirname $(dirname $(git --html-path)))/gitweb
/usr/share/gitweb
```

Gitweb 虽然已经安装，但是尚未和 Web 服务器整合。在 Apache 的配置文件中添加如下配置，重启 Apache 后，即可以用地址 /gitweb 来访问 Gitweb 服务。

```
Alias /gitweb /usr/share/gitweb

<Directory /usr/share/gitweb>
  Options FollowSymLinks +ExecCGI
  AddHandler cgi-script .cgi
  DirectoryIndex index.cgi gitweb.cgi
  Order Allow,Deny
  Allow from all
</Directory>
```

## 27.3.2 Gitweb 的配置

编辑 /etc/gitweb.conf，更改 Gitweb 的默认设置：

- 版本库的根目录。

```
$projectroot = "/var/cache/git";
```

- 设置版本库访问 URL。

  Gitweb 可以为每个版本库显示克隆该版本库的 URL 地址，可以设置多个。

```
@git_base_url_list = ("git://bj.ossxp.com/git", "http://bj.ossxp.com/git");
```

- 设置首页模板文件。该文件为 HTML 格式，其内容将显示在首页上。如果使用相对路径，则相对于 CGI 脚本所在的目录。

```
$home_text = "indextext.html";
```

- 定制首页模板。下面是我公司内部使用的 Gitweb 首页模板。

```
<html>
<head>
</head>
<body>
<h2> 北京群英汇信息技术有限公司 - git 代码库 </h2>
<ul>
  <li> 点击版本库，进入相应的版本库页面，有 URL 指向一个 git://... 的检出链接 </li>
  <li> 使用命令 git clone git://... 来克隆一个版本库 </li>
  <li> 对于名称中含有 <i>-gitsvn</i> 字样的代码库，是用 git-svn 从 svn 代码库镜像而来的。
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
对于它们的镜像，需要做进一步的工作。
        <ul>
          <li>要将 git 库的远程分支（.git/ref/remotes/*）也同步到本地！
            <pre>
            $ git config --add remote.origin.fetch '+refs/remotes/*:refs/remotes/*'
            $ git fetch
            </pre>
          </li>
          <li>如果需要克隆库和 Subversion 同步。用 git-svn 初始化代码库，并使得相关配置和源保持一致 </li>
        </ul>
      </li>
    </ul>
    </body>
    </html>
```

- 版本库列表。

  默认扫描版本库根目录查找版本库。如果版本库非常多，这个查找过程可能很耗时，可以提供一个文本文件包含版本库的列表加速 Gitweb 显示初始化。

  ```
  $projects_list = "/home/git/gitosis/projects.list";
  ```

  后面介绍的 Gitosis 和 Gitolite 都可以自动生成这么一个版本库列表，供 Gitweb 使用。

- Gitweb 菜单定制。

  Git 菜单定制项很多，下面选取几个典型配置进行介绍。

  - 在 tree view 文件的旁边显示追溯（blame）链接。

    ```
    $feature{'blame'}{'default'} = [1];
    $feature{'blame'}{'override'} = 1;
    ```

  - 通过版本库的配置文件 config 对版本库在 Gitweb 中是否显示追溯进行单独设置。下面的设置覆盖 Gitweb 的全局设置，不对该项目显示 blame 菜单。

    ```
    [gitweb]
    blame = 0
    ```

  - 为每个 tree 添加快照（snapshot）下载链接。

    ```
    $feature{'snapshot'}{'default'} = ['zip', 'tgz'];
    $feature{'snapshot'}{'override'} = 1;
    ```

### 27.3.3 版本库的 Gitweb 相关设置

可以通过 Git 版本库下的配置文件，定制版本库在 Gitweb 下的显示。

- 文件 description。

  提供一行简短的 Git 库描述，显示在 Gitweb 版本库列表中。

  也可以通过 config 配置文件中的 gitweb.description 进行设置，但是文件优先。

- 文件 README.html。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


提供更详细的项目描述，显示在 Gitweb 项目页面中。

- 文件 cloneurl。
  版本库访问的 URL 地址，一行一个。
- 文件 config。
  通过 [gitweb] 小节的配置，覆盖 Gitweb 的全局设置：
  - gitweb.owner 用于显示版本库的创建者。
  - gitweb.description 显示项目的简短描述，也可以通过 description 文件来提供。（文件优先）
  - gitweb.url 显示项目的 URL 列表，也可以通过 cloneurl 文件来提供。（文件优先）

### 27.3.4 即时 Gitweb 服务

如果安装有 lighttpd[①]——轻量级 Web 服务器，甚至可以无须任何配置，当在工作区中运行下面的命令时，自动用 Web 浏览器打开 URL 地址：*http://127.0.0.1:1234/*，访问即时启动的 Gitweb 服务。

```
$ git instaweb
```

可以通过 Git 配置变量对即时启动的 Gitweb 服务进行设置，例如用 instaweb.port 配置变量设置默认 Web 服务端口（默认为 1234）。



① *http://www.lighttpd.net/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 28 章 使用 Git 协议

Git 协议是提供 Git 版本库只读服务的最常用的协议，也是非常易用和易于配置的协议。该协议的缺点就是不能提供身份认证，而且一般也不提供写入服务。

## 28.1 Git 协议语法格式

Git 协议的语法格式如下。

```
语法：git://<server>[:<port>]/path/to/repos.git/
```

说明：

- 端口为可选项，默认端口为 9418。
- 版本库路径 /path/to/repos.git 的根目录并不一定是系统的根目录，可以在 git-daemon 启动时用参数 --base-path 来指定根目录。如果 git-daemon 没有设置根目录，则对应于系统的根目录。

## 28.2 Git 服务软件

Git 服务由名为 git-daemon 的服务软件提供。虽然 git-daemon 也可以支持写操作，但因为 git-daemon 没有提供认证支持，因此很少有人胆敢配置 git-daemon 来提供匿名的写服务。使用 git-daemon 提供的 Git 版本库只读服务效率很高，而且是一种智能协议，操作过程有进度显示，远比 HTTP 哑通信协议方便（Git 1.6.6 之后的版本已经支持智能 HTTP 通信协议）。因此 git-daemon 很久以来，一直是 Git 版本库只读服务的首选。

Git 软件包本身提供了 git-daemon，因此只要安装了 Git，一般就已经安装了 git-daemon。默认 git-daemon 并没有运行，需要对其进行配置，以服务的方式运行。下面介绍两种不同的配置运行方式。

## 28.3 以 inetd 方式配置运行

最简单的方式是以 inetd 服务的方式运行 git-daemon。在配置文件 /etc/inetd.conf 中添加如下设置：

```
git stream tcp nowait nobody  /usr/bin/git
    git daemon --inetd --verbose --export-all
    /gitroot/foo /gitroot/bar
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


说明：

- 以 nobody 用户身份执行 git daemon 服务。
- 默认 git-daemon 只对包含文件 git-daemon-export-ok 的版本库提供服务。使用参数 --export-all 后，无论版本库是否存在标识文件 git-daemon-export-ok，都对版本库提供 Git 访问服务。
- 后面的两个参数是版本库根目录，用户只可以访问指定目录下的 Git 版本库。例如可以访问 git://server/gitroot/foo/project1.git 和 git://server/gitroot/bar/project2.git，但是 git://server/others/project3.git 是访问不到的。

如果版本库的路径比较深，有什么办法能在用户访问时提供短一些的 URL 地址呢？可以使用参数 --base-path=<path> 建立版本库根目录映射，例如下面的 inetd 的配置：

```
git stream tcp nowait nobody  /usr/bin/git
    git daemon --inetd --verbose --export-all
    --base-path=/var/cache /var/cache/git
```

在上面的配置中，设置提供版本库服务的路径为 /var/cache/git，但因为配置了 --base-path=/var/cache 参数，在实际访问时用户所请求的 Git 版本库路径都会添加这个前缀，然后再到指定的目录中去寻找。例如当用户访问 git://server/git/myrepos.git 时，实际访问的路径是 /var/cache/git/myrepos.git。

## 28.4 以 runit 方式配置运行

runit[①]是类似于 sysvinit 的服务管理进程，但是更为简单。在 Debian/Ubuntu 上的软件包 git-daemon-run 就是基于 runit 启动 git-daemon 服务。

- 安装 git-daemon-run：

```
$ sudo aptitude install git-daemon-run
```

- 配置 git-daemon-run：

  默认的服务配置文件：/etc/sv/git-daemon/run。和之前的 inetd 运行方式相比，以独立的服务进程启动，速度更快。

```
#!/bin/sh
exec 2>&1
echo 'git-daemon starting.'
exec chpst -ugitdaemon \
  "$(git --exec-path)"/git-daemon --verbose --export-all \
  --base-path=/var/cache /var/cache/git
```

默认版本库中需要存在文件 git-daemon-export-ok，git-daemon 才对此版本库提供

① *http://smarden.org/runit/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


服务。不过可以通过启动 git-daemon 时提供的参数 `--export-all`，无论版本库是否存在标识文件 `git-daemon-export-ok`，都对版本库提供 git 访问服务。

git-daemon 提供很多参数，在此没有一一介绍，可以运行 `git help daemon` 在控制台查看 git-daemon 帮助，或者运行 `git help --web daemon` 查看 HTML 格式的帮助。

通过 git-daemon 提供的 Git 访问协议存在着局限性：

- 不支持认证。管理员可以做的大概只是配置防火墙，限制某个网段用户的使用。
- 只能提供匿名的版本库读取服务。因为写操作没有授权控制，因此一般不用来提供写操作。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 29 章　使用 SSH 协议

SSH 协议用于为 Git 提供远程读写操作，是远程写操作的标准服务，在智能 HTTP 协议出现之前，甚至是写操作的唯一标准服务。

## 29.1　SSH 协议语法格式

对于拥有 shell 登录权限的用户账号，可以用下面的语法访问 Git 版本库：

```
语法 1：ssh://[<username>@]<server>[:<port>]/path/to/repos/myrepo.git
语法 2：[<username>@]<server>:/path/to/repos/myrepo.git
```

说明：

- SSH 协议地址格式可以使用两种不同的写法，第一种是使用 ssh:// 开头的标准的 SSH 协议 URL 写法，另外一种是 SCP 格式的写法。
  两种写法均可，SSH 协议标准 URL 写法稍嫌复杂，但是对于非标准 SSH 端口（非 22 端口），可以通过 URL 给出端口号。
- <username> 是服务器 <server> 上的用户账号。
  如果省略用户名，则默认使用当前登录用户名（配置和使用了主机别名的除外）。
- <port> 为 SSH 协议端口，默认为 22。
  因为只有语法 1 才能在 URL 中提供端口，因此如果使用非默认端口 22，最好使用语法 1。当然使用语法 2 也可以实现，但是要通过 `~/.ssh/config` 配置文件设置主机别名。
- 路径 `/path/to/repos/myrepo.git` 是服务器中版本库的绝对路径。若用相对路径则是相对于 username 用户的主目录而言的。
- 如果采用口令认证，不能像 HTTPS 协议那样可以在 URL 中同时给出登录名和口令，必须在每次连接时输入口令。
- 如果采用公钥认证，则无须输入口令。

## 29.2　服务架设方式比较

SSH 协议有两种方式来实现 Git 服务。第一种是用标准的 SSH 账号访问版本库。即用户账号可以直接登录到服务器获得 shell。对于这种使用标准 SSH 账号的方式，直接使用标准的 SSH 服务就可以了，无须赘述。

第二种实现方式是所有用户都使用同一个专用的 SSH 账号访问版本库，访问时通过公

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


钥认证的方式。虽然所有用户用同一个账号访问，但可以通过在建立连接时所用的不同公钥来区分不同的用户身份。Gitosis 和 Gitolite 就是实现该方式的两个服务器软件。

标准 SSH 账号和专用 SSH 账号这两种实现方式的区别见表 29-1。

**表 29-1 不同 SSH 服务架设方式对照表**

| | 标准 SSH | Gitosis/Gitolite |
|---|---|---|
| 账号 | 每个用户一个账号 | 所有用户共用同一个账号 |
| 认证方式 | 口令或公钥认证 | 公钥认证 |
| 登录到 shell | 是 | 否 |
| 安全性 | 差 | 好 |
| 管理员需要 shell | 是 | 否 |
| 版本库路径 | 相对路径或绝对路径 | 相对路径 |
| 授权方式 | 操作系统中用户组和目录权限 | 通过配置文件授权 |
| 分支写授权 | 否 | Gitolite |
| 路径写授权 | 否 | Gitolite |
| 架设难易度 | 简单 | 复杂 |

实际上，标准 SSH 也可以用公钥认证的方式实现所有用户共用同一个账号，不过这类似于把一个公共账号的登录口令同时告诉给多个人。具体操作过程如下：

（1）在服务器端（server）创建一个公共账号，例如 anonymous。

（2）管理员收集需要访问 git 服务的用户公钥。如：user1.pub、user2.pub。

（3）使用 ssh-copy-id 命令将各个 git 用户的公钥远程加入服务器（server）的公钥认证列表中。

❑ 远程操作，可以使用 ssh-copy-id 命令。

```
$ ssh-copy-id -i user1.pub anonymous@server
$ ssh-copy-id -i user2.pub anonymous@server
```

❑ 如果直接在服务器上操作，则直接将文件追加到 authorized_keys 文件中。

```
$ cat /path/to/user1.pub >> ~anonymous/.ssh/authorized_keys
$ cat /path/to/user2.pub >> ~anonymous/.ssh/authorized_keys
```

（4）在服务器端的 anonymous 用户主目录下建立 git 库，就可以实现多个用户利用同一个系统账号（anonymous）访问 Git 服务了。

这样做除了不必逐一设置账号，以及用户无须口令认证之外，标准 SSH 部署 Git 服务的缺点一个也不少，而且因为用户之间无法区分，更无法针对用户进行授权。

下面重点介绍一下 SSH 公钥认证，因为它们是后面即将介绍的 Gitosis 和 Gitolite 服务

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


器软件的基础。

## 29.3 关于 SSH 公钥认证

SSH 公钥认证是一种非常安全且免口令的认证方式。关于公钥认证的原理，维基百科上的这个条目是一个很好的起点：

*http://en.wikipedia.org/wiki/Public-key_cryptography*

为实现公钥认证，作为认证的客户端一方需要拥有两个文件，即公钥 / 私钥对。一般公钥 / 私钥对文件创建在用户的主目录下的 .ssh 目录中。如果用户主目录下不存在 .ssh 目录，说明 SSH 公钥 / 私钥对尚未创建。可以用下面的这个命令创建：

```
$ ssh-keygen
```

该命令会在用户主目录下创建 .ssh 目录，并在其中创建两个文件：

- id_rsa

  私钥文件。是基于 RSA 算法创建的。该私钥文件要妥善保管不要泄漏。

- id_rsa.pub

  公钥文件。和 id_rsa 文件是一对儿，该文件作为公钥文件可以公开。

创建了自己的公钥 / 私钥对后，就可以使用下面的命令，实现无口令登录远程服务器，即用公钥认证取代口令认证。

```
$ ssh-copy-id -i .ssh/id_rsa.pub <user>@<server>
```

说明：

- 该命令会提示输入用户 user 在 server 上的 SSH 登录口令。
- 此命令执行成功后，再以 user 用户用 ssh 命令登录 server 远程主机时，不必输入口令可直接登录。
- 该命令实际上是将 .ssh/id_rsa.pub 公钥文件追加到远程主机 server 的 user 主目录下的 .ssh/authorized_keys 文件中。

检查公钥认证是否生效，通过 ssh 命令连接远程主机，正常的话应该直接登录成功。如果要求输入口令则表明公钥认证配置存在问题。如果 SSH 登录存在问题，可以通过查看服务器端的 /var/log/auth.log 日志文件进行诊断。

## 29.4 关于 SSH 主机别名

在实际应用中，有时需要使用多套公钥 / 私钥对，例如：

- 使用默认的公钥访问服务器的 git 账号，可以执行 git 命令，但不能进行 shell 登录。
- 使用特别创建的公钥访问服务器的 git 账号，能够获取 shell，登录后可以对 Git 服务



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


器软件进行升级、维护等工作。

- 访问Github（免费的Git服务托管商）使用其他公钥（非默认公钥）。

从上面的说明中可以看出，用户可能拥有不只一套公钥/私钥对。为了创建不同的公钥/私钥对，在使用ssh-keygen命令时就需要通过-f参数指定不同的私钥名称。用法如下：

```
$ ssh-keygen -f ~/.ssh/<filename>
```

请将<filename>替换为有意义的名称。命令执行完毕后，会在~/.ssh目录下创建指定的公钥/私钥对：文件<filename>是私钥，文件<filename>.pub是公钥。

将新生成的公钥添加到远程主机登录用户主目录下的.ssh/authorized_keys文件中，就可以使用新创建的公钥建立到远程主机<server>的<user>账户的无口令登录（采用公钥认证）。操作如下：

```
$ ssh-copy-id -i .ssh/<filename>.pub <user>@<server>
```

现在用户存在多个公钥/私钥对，那么当执行下面的ssh登录指令时，用到的是哪个公钥呢？

```
$ ssh <user>@<server>
```

当然是默认公钥~/.ssh/id_rsa.pub。那么如何用新建的公钥连接server呢？

SSH的客户端配置文件~/.ssh/config可以通过创建主机别名，在连接主机时选择使用特定的公钥。例如~/.ssh/config文件中的下列配置：

```
host bj
  user git
  hostname bj.ossxp.com
  port 22
  identityfile ~/.ssh/jiangxin
```

执行下面的SSH登录命令：

```
$ ssh bj
```

或者执行下面的Git命令：

```
$ git clone bj:path/to/repos/myrepo.git
```

虽然这两条命令各不相同，但是都使用了SSH协议，以及相同的主机别名：bj。参考上面在~/.ssh/config文件中建立的主机别名，可以做出如下判断：

- 登录的SSH主机名为bj.ossxp.com。
- 登录时使用的用户名为git。
- 认证时使用的公钥文件为~/.ssh/jiangxin.pub。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 30 章 Gitolite 服务架设

Gitolite 是一款 Perl 语言开发的 Git 服务管理工具，通过公钥对用户进行认证，并能够通过配置文件对写操作进行基于分支和路径的精细授权。Gitolite 采用的是 SSH 协议并且使用 SSH 公钥认证，因此无论是管理员还是普通用户，都需要对 SSH 非常熟悉。在开始之前，请确认您已经通读过第 29 章“使用 SSH 协议”。

Gitolite 的官方网址是：*http://github.com/sitaramc/gitolite*。从提交日志里可以看出作者是 Sitaram Chamarty，最早的提交开始于 2009 年 8 月。作者是受到了 Gitosis 的启发，开发了这款功能更为强大和易于安装的软件。Gitolite 的命名，作者的原意是 Gitosis 和 lite 的组合，不过因为 Gitolite 的功能越来越强大，已经超越了 Gitosis，因此作者笑称 Gitolite 可以看作是 Github-lite——轻量级的 Github。

我是在 2010 年 8 月才发现 Gitolite 这个项目的，并尝试将公司基于 Gitosis 的管理系统迁移至 Gitolite。在迁移和使用过程中，增加和改进了一些实现，如：通配符版本库的创建过程，对创建者的授权，版本库名称映射等。本文关于 Gitolite 的介绍也是基于我改进的版本[①]。

- 原作者的版本库地址：
  *http://github.com/sitaramc/gitolite*
- 笔者改进后的 Gitolite 分支：
  *http://github.com/ossxp-com/gitolite*
  Gitolite 的实现机制和使用特点概述如下：
- Gitolite 安装在服务器（`server`）的某个账号之下，例如 git 账号。
- 管理员通过 git 命令检出名为 gitolite-admin 的版本库。

```
$ git clone git@server:gitolite-admin.git
```

- 管理员将 Git 用户的公钥保存在 gitolite-admin 库的 `keydir` 目录下，并编辑 `conf/gitolite.conf` 文件为用户授权。
- 当管理员提交对 gitolite-admin 库的修改并推送到服务器之后，服务器上 gitolite-admin 版本库的钩子脚本将执行相应的设置工作。
  - 新用户的公钥自动追加到服务器端安装账号主目录下的 `.ssh/authorized_keys` 文件中，并设置该用户的 shell 为 gitolite 的一条命令 `gl-auth-command`。在 `.ssh/authorized_keys` 文件中增加的内容示例如下：

① 对 Gitolite 的各项改动采用了 Topgit 特性分支进行维护，以便与上游的最新代码同步更新。还要注意，如果使用 Gitolite 时发现问题，要区分是由上游软件引发的，还是因为我的改动引起的，不要把我的错误算在 Sitaram 头上。;-)

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
command="/home/git/.gitolite/src/gl-auth-command
jiangxin",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-
pty ssh-rsa AAAAB3NzaC1yc2...(公钥内容来自于 jiangxin.pub)...①
```

- 更新服务器端的授权文件 ~/.gitolite/conf/gitolite.conf。
- 编译授权文件为 ~/.gitolite/conf/gitolite.conf-compiled.pm。

❑ 若用 ssh 命令登录服务器（以 Git 用户登录）时，因为公钥认证的相关设置（使用 gl-auth-command 作为 shell），不能进入 shell 环境，而是打印服务器端 Git 库授权信息后马上退出。即用户不会通过 Git 用户进入服务器的 shell，也不会对系统的安全造成威胁。

```
$ ssh git@bj
hello jiangxin, the gitolite version here is v1.5.5-9-g4c11bd8
the gitolite config gives you the following access:
     R          gistore-bj.ossxp.com/.*$
  C  R  W       ossxp/.*$
 @C @R  W       users/jiangxin/.+$
Connection to bj closed.
```

❑ 用户可以用 git 命令访问授权的版本库。

❑ 若管理员授权，用户可以远程在服务器上创建新版本库。

下面介绍 Gitolite 的部署和使用。在下面的示例中约定：服务器的名称为 server，Gitolite 的安装账号为 git，管理员的 ID 为 admin。

## 30.1 安装 Gitolite

Gitolite 要求 Git 的版本必须是 1.6.2 或以上的版本，并且服务器要提供 SSH 服务。下面是 Gitolite 的安装过程。

### 30.1.1 服务器端创建专用账号

安装 Gitolite，首先要在服务器端创建专用账号，所有用户都通过此账号访问 Git 库。一般为方便易记，选择 git 作为专用账号名称。

```
$ sudo adduser --system --shell /bin/bash --group git
```

创建用户 git，并设置用户的 shell 为可登录的 shell，如 /bin/bash，同时添加同名的用户组。

有的系统，只允许特定用户组（如 ssh 用户组）的用户才可以通过 SSH 协议登录，这就需要将新建的 git 用户同时也添加到该特定的用户组中。执行下面的命令可以将 git 用户添加到 ssh 用户组。

```
$ sudo adduser git ssh
```

① 本段内容为一整行，因排版需要做了换行处理。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


为 git 用户设置口令。当整个 git 服务配置完成，运行正常后，建议取消 git 的口令，只允许公钥认证。

```
$ sudo passwd git
```

管理员在客户端使用下面的命令，建立无口令登录：

```
$ ssh-copy-id git@server
```

至此，已经完成了安装 git 服务的准备工作，可以开始安装 Gitolite 服务软件了。

### 30.1.2 Gitolite 的安装 / 升级

本节的标题为安装 / 升级，是因为 Gitolite 的安装和升级可以采用同样的步骤。

Gitolite 安装可以在客户端执行，而不需要在服务器端操作，非常方便。远程安装 Gitolite 的前提是：

- 已经在服务器端创建了专有账号，如 git。
- 管理员能够以 git 用户的身份通过公钥认证以无口令方式登录服务器。

安装和升级都可以按照下面的步骤进行：

（1）使用 git 下载 Gitolite 的源代码。

```
$ git clone git://github.com/ossxp-com/gitolite.git
```

（2）进入 gitolite/src 目录，执行安装。

```
$ cd gitolite/src
$ ./gl-easy-install git server admin
```

命令 gl-easy-install 的第一个参数 git 是服务器上创建的专用账号 ID，第二个参数 server 是服务器 IP 或域名，第三个参数 admin 是管理员 ID。

（3）首先显示版本信息。

```
------------------------------------------------------------------------

you are upgrading      (or installing first-time)      to v1.5.4-22-g4024621

Note: getting '(unknown)' for the 'from' version should only happen once.
Getting '(unknown)' for the 'to' version means you are probably installing
from a tar file dump, not a real clone.  This is not an error but it's nice to
have those version numbers in case you need support.  Try and install from a clone
```

（4）自动创建名为 admin 的私钥 / 公钥对。创建的公钥 / 私钥对的名称来自于 gl-easy-install 命令的最后一个参数 admin。

```
------------------------------------------------------------------------

the next command will create a new keypair for your gitolite access
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
The pubkey will be /home/jiangxin/.ssh/admin.pub.  You will have to choose a
passphrase or hit enter for none.  I recommend not having a passphrase for
now, *especially* if you do not have a passphrase for the key which you are
already using to get server access!

Add one using 'ssh-keygen -p' after all the setup is done and you've
successfully cloned and pushed the gitolite-admin repo.  After that, install
'keychain' or something similar, and add the following command to your bashrc
(since this is a non-default key)

    ssh-add $HOME/.ssh/admin

This makes using passphrases very convenient.
```

（5）如果公钥已经存在，会弹出警告。

```
------------------------------------------------------------------------

Hmmm... pubkey /home/jiangxin/.ssh/admin.pub exists; should I just (re-)use it?

IMPORTANT: once the install completes, *this* key can no longer be used to get
a command line on the server -- it will be used by gitolite, for git access
only.  If that is a problem, please ABORT now.

doc/6-ssh-troubleshooting.mkd will explain what is happening here, if you need
more info.
```

（6）自动修改客户端的 .ssh/config 文件，增加名为 gitolite 的别名主机。

即当访问主机 gitolite 时，会自动用名为 admin.pub 的公钥，以 git 用户的身份连接服务器。

```
------------------------------------------------------------------------

creating settings for your gitolite access in /home/jiangxin/.ssh/config;
these are the lines that will be appended to your ~/.ssh/config:

host gitolite
     user git
     hostname server
     port 22
     identityfile ~/.ssh/admin
```

（7）上传脚本文件到服务器，完成服务器端软件的安装。

```
gl-dont-panic
100% 3106     3.0KB/s   00:00
gl-conf-convert
100% 2325     2.3KB/s   00:00
gl-setup-authkeys
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
100% 1572     1.5KB/s   00:00
...
gitolite-hooked
100%    0     0.0KB/s   00:00
update
100% 4922     4.8KB/s   00:00


-------------------------------------------------------------------------

the gitolite rc file needs to be edited by hand.  The defaults are sensible,
so if you wish, you can just exit the editor.

Otherwise, make any changes you wish and save it.  Read the comments to
understand what is what -- the rc file's documentation is inline.

Please remember this file will actually be copied to the server, and that all
the paths etc. represent paths on the server!
```

（8）自动调用 vi 编辑器打开 .gitolite.rc 文件，编辑结束后上传到服务器。

该配置文件为 Perl 语法，注意保持文件格式和语法。退出 vi 编辑器，输入“<ESC>:q”（不带引号）。以下为该配置文件中比较重要的设置，一般无须改变默认的配置。

- $REPO_BASE="repositories";

  用于设置 Git 服务器的根目录，默认是 Git 用户主目录下的 repositories 目录，可以使用绝对路径。所有 Git 库都将部署在该目录下。
- $REPO_UMASK = 0007; # gets you 'rwxrwx---'

  版本库创建使用的掩码。即新建立的版本库的权限为 'rwxrwx---'。
- $GL_BIG_CONFIG = 0;

  如果授权文件非常复杂，更改此项配置为 1，以免产生庞大的授权编译文件。
- $GL_WILDREPOS = 1;

  默认支持通配符版本库授权。

（9）至此完成安装。

### 30.1.3 关于 SSH 主机别名

在安装过程中，gitolite 创建了名为 admin 的公钥 / 私钥对，以名为 admin.pub 的公钥连接服务器的 git 账户，使用由 gitolite 提供的 Git 服务。但是如果直接连接服务器，使用的是默认的公钥，会直接进入 shell。

那么如何能够根据需要选择不同的公钥来连接 git 服务器呢？

别忘了在前面介绍过的 SSH 主机别名。实际上刚刚在安装 gitolite 的时候，就已经自动地创建了一个主机别名。打开 ~/.ssh/config 文件可以看到类似内容，如果对主机别名不满意可以修改。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
host gitolite
     user git
     hostname server
     port 22
     identityfile ~/.ssh/admin
```

即：

- 像下面这样输入 SSH 命令会直接进入 shell，因为使用的是默认的公钥。

```
$ ssh git@server
```

- 像下面这样输入 SSH 命令则不会进入 shell。因为使用名为 admin.pub 的公钥，会显示 Git 授权信息并马上退出。

```
$ ssh gitolite
```

### 30.1.4 其他的安装方法

上面介绍的是在客户端远程安装 Gitolite，是最常用和推荐的方法。当然还可以直接在服务器上安装，具体操作过程如下。

（1）首先也要在服务器端先创建一个专用的账号，如 git 。

```
$ sudo adduser --system --shell /bin/bash --group git
```

（2）将管理员公钥复制到服务器上。

管理员在客户端执行下面的命令：

```
$ scp ~/.ssh/id_rsa.pub server:/tmp/admin.pub
```

（3）服务器端安装 Gitolite（源码方式安装）。

推荐采用源码方式安装，因为如果以平台自带软件包模式安装 Gitolite，那么其中就不包含我对 Gitolite 的改进。

- 使用 git 下载 Gitolite 的源代码。

```
$ git clone git://github.com/ossxp-com/gitolite.git
```

- 创建目录。

```
$ sudo mkdir -p /usr/local/share/gitolite/conf \
                /usr/local/share/gitolite/hooks
```

- 进入 gitolite/src 目录，执行安装。

```
$ cd gitolite/src
$ sudo ./gl-system-install /usr/local/bin \
       /usr/local/share/gitolite/conf\
       /usr/local/share/gitolite/hooks
```

安装完毕跳到步骤 5。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（4）服务器端安装 Gitolite（平台包管理器安装）。

如果不选择从源代码进行安装（如步骤 3），也可以使用当前平台的包管理器进行安装。例如在 Debian/Ubuntu 平台执行下面的命令：

```
$ sudo aptitude install gitolite
```

（5）在服务器端以专用账号执行安装脚本。

例如服务器端的专用账号为 git，先执行 su 命令，临时切换到该用户，继续下面的安装。

```
$ sudo su - git
$ gl-setup /tmp/admin.pub
```

（6）管理员在客户端克隆 gitolite-admin 库。

```
$ git clone git@server:gitolite-admin
```

（7）在克隆出来的 gitolite-admin 工作区中，以 Git 的方式管理 gitolite。如添加、删除用户账号，设置用户权限。

升级 Gitolite 只需要执行上面的步骤 3 或步骤 4 即可完成升级。如果还修改或增加了新的钩子脚本，还需要重新执行步骤 5。Gitolite 的升级有可能要求修改配置文件：`~/.gitolite.rc`。

## 30.2 管理 Gitolite

### 30.2.1 管理员克隆 gitolite-admin 管理库

当 Gitolite 安装完成后，在服务器端自动创建了一个用于 Gitolite 自身管理的 Git 库：`gitolite-admin.git`。

克隆 `gitolite-admin.git` 库。别忘了使用 SSH 主机别名：

```
$ git clone gitolite:gitolite-admin.git
Initialized empty Git repository in /data/tmp/gitolite-admin/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.

$ cd gitolite-admin/

$ ls -F
conf/  keydir/

$ ls conf
gitolite.conf

$ ls keydir/
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
admin.pub
```

可以看出 gitolite-admin 目录下有两个目录 conf/ 和 keydir/。

- keydir/admin.pub 文件

  目录 keydir 下初始时只有一个用户公钥，即 amdin 用户的公钥。

- conf/gitolite.conf 文件

  该文件为授权文件。初始内容为：

```
#gitolite conf
# please see conf/example.conf for details on syntax and features

repo gitolite-admin
    RW+                 = admin

repo testing
    RW+                 = @all
```

  默认授权文件中只设置了两个版本库的授权：

- gitolite-admin

  即本版本库（gitolite 管理版本库）中只有 admin 用户有读写和强制更新的权限。

- testing

  默认设置的测试版本库，设置为任何人都可以读写及强制更新。

## 30.2.2 增加新用户

增加新用户，就是允许新用户能够通过其公钥访问 Git 服务。只要将新用户的公钥添加到 gitolite-admin 版本库的 keydir 目录下，即完成新用户的添加，具体操作过程如下。

（1）管理员从用户获取公钥，并将公钥按照 username.pub 格式进行重命名。

- 用户可以通过邮件或其他方式将公钥传递给管理员，切记不要将私钥误传给管理员。如果发生私钥泄漏，马上重新生成新的公钥 / 私钥对，并将新的公钥传递给管理员，并申请将旧的公钥作废。
- 用户从不同的客户端主机访问有着不同的公钥，如果希望使用同一个用户名进行授权，可以按照 username@host.pub 的方式命名公钥文件，和名为 username.pub 的公钥指向同一个用户 username 。
- Gitolite 也支持邮件地址格式的公钥，即形如 username@gmail.com.pub 的公钥。Gitolite 能够很智能地区分是以邮件地址命名的公钥还是相同用户在不同主机上的公钥。如果是邮件地址命名的公钥，将以整个邮件地址作为用户名。

（2）管理员进入 gitolite-admin 本地克隆版本库中，复制新用户公钥到 keydir 目录。

```
$ cp /path/to/dev1.pub keydir/
$ cp /path/to/dev2.pub keydir/
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cp /path/to/jiangxin.pub keydir/
```

(3) 执行 git add 命令，将公钥添加到版本库。

```
$ git add keydir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   keydir/dev1.pub
#       new file:   keydir/dev2.pub
#       new file:   keydir/jiangxin.pub
#
```

(4) 执行 git commit，完成提交。

```
$ git commit -m "add user: jiangxin, dev1, dev2"
[master bd81884] add user: jiangxin, dev1, dev2
 3 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 keydir/dev1.pub
 create mode 100644 keydir/dev2.pub
 create mode 100644 keydir/jiangxin.pub
```

(5) 执行 git push，同步到服务器，才真正完成新用户的添加。

```
$ git push
Counting objects: 8, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.38 KiB, done.
Total 6 (delta 0), reused 0 (delta 0)
remote: Already on 'master'
remote:
remote:                  ***** WARNING *****
remote:          the following users (pubkey files in parens) do not appear
in the config file:
remote: dev1(dev1.pub),dev2(dev2.pub),jiangxin(jiangxin.pub)
```

如果这时查看服务器端 git 用户主目录下的 .ssh/authorized_keys 文件，会发现新增的用户公钥也附加在其中：

```
$ cat ~git/.ssh/authorized_keys
# gitolite start
command="/home/git/.gitolite/src/gl-auth-command
admin",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty    < 用户
admin 的公钥 ...>
command="/home/git/.gitolite/src/gl-auth-command
dev1",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty     < 用户
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
dev1 的公钥 ...>
command="/home/git/.gitolite/src/gl-auth-command
dev2",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty      < 用户
dev2 的公钥 ...>
command="/home/git/.gitolite/src/gl-auth-command
jiangxin",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty < 用户
jiangxin 的公钥 ...>
# gitolite end
```

在之前执行 git push 后的输出中，以 remote 标识的输出是服务器端执行 post-update 钩子脚本的输出。其中的警告是说新添加的三个用户在授权文件中没有被引用。接下来便看看如何修改授权文件，以及如何为用户添加授权。

### 30.2.3 更改授权

新用户添加完毕，可能需要重新进行授权。更改授权的方法也非常简单，即修改 conf/gitolite.conf 配置文件，提交并推送，具体操作过程如下。

（1）管理员进入 gitolite-admin 本地克隆版本库中，编辑 conf/gitolite.conf 。

```
$ vi conf/gitolite.conf
```

（2）授权指令比较复杂，先通过建立新用户组尝试一下更改授权文件。

考虑到之前增加了三个用户公钥，服务器端发出了用户尚未在授权文件中出现的警告。现在就在这个示例中解决这个问题。

- 可以在其中加入用户组 @team1，将新添加的用户 jiangxin、dev1、dev2 都归属到这个组中。只需要在 conf/gitolite.conf 文件的文件头加入如下指令即可。用户名之间用空格分隔。

```
@team1 = dev1 dev2 jiangxin
```

- 还修改了版本库 testing 的授权，将 @all 用户组改为新建立的 @team1 用户组。从编辑完毕后的文件差异输出可以看到相关改动。

```
$ git diff
diff --git a/conf/gitolite.conf b/conf/gitolite.conf
index 6c5fdf8..f983a84 100644
--- a/conf/gitolite.conf
+++ b/conf/gitolite.conf
@@ -1,10 +1,12 @@
 #gitolite conf
 # please see conf/example.conf for details on syntax and features

+@team1 = dev1 dev2 jiangxin
+
 repo gitolite-admin
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        RW+                     = admin

    repo testing
-       RW+                     = @all
+       RW+                     = @team1
```

（3）编辑结束，提交改动。

```
$ git add conf/gitolite.conf
$ git commit -q -m "new team @team1 auth for repo testing."
```

（4）执行 git push，同步到服务器，授权文件的更改才真正生效。可以注意到，推送后的输出中没有了警告。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 398 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Already on 'master'
To gitadmin.bj:gitolite-admin.git
   bd81884..79b29e4  master -> master
```

## 30.3 Gitolite 授权详解

### 30.3.1 授权文件的基本语法

下面看一个不那么简单的授权文件。为方便描述添加了行号。

```
1   @admin = jiangxin wangsheng
2
3   repo gitolite-admin
4       RW+                     = jiangxin
5
6   repo ossxp/.+
7       C                       = @admin
8       RW                      = @all
9
10  repo testing
11      RW+                             =   @admin
12      RW      master                  =   junio
13      RW+     pu                      =   junio
14      RW      cogito$                 =   pasky
15      RW      bw/                     =   linus
16      -                               =   somebody
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
17      RW      tmp/                =   @all
18      RW      refs/tags/v[0-9]    =   junio
```

在上面的示例中，演示了很多授权指令：

- 第1行，定义了用户组@admin，包含两个用户jiangxin和wangsheng。
- 第3～4行，定义了版本库gitolite-admin。并指定只有用户jiangxin才能够访问，并拥有读（R）写（W）和强制更新（+）的权限。
- 第6行，通过正则表达式定义了一组版本库，即在ossxp/目录下的所有版本库。
- 第7行，用户组@admin中的用户，可以在ossxp/目录下创建版本库。创建版本库的用户，具有对版本库操作的所有权限。
- 第8行，所有用户都可以读写ossxp目录下的版本库，但不能强制更新。
- 第10行开始，定义的testing版本库授权使用了引用授权语法。
- 第11行，用户组@admin对所有的分支和里程碑拥有读写、重置、添加和删除的授权。
- 第12行，用户junio可以读写master分支。（还包括名字以master开头的其他分支，如果有的话。）
- 第13行，用户junio可以读写、强制更新、创建及删除pu开头的分支。
- 第14行，用户pasky可以读写cogito分支。（仅此分支，精确匹配）。

### 30.3.2 定义用户组和版本库组

在conf/gitolite.conf授权文件中，可以定义用户组或版本库组。组名称以@字符开头，可以包含一个或多个成员。成员之间用空格分开。

- 例如定义管理员组：

```
@admin = jiangxin wangsheng
```

- 组可以嵌套：

```
@staff = @admin @engineers tester1
```

- 除了作为用户组外，同样的语法也适用于版本库组。

  版本库组和用户组的定义没有任何区别，只是在版本库授权指令中处于不同的位置。即位于授权指令中的版本库位置代表版本库组，位于授权指令中的用户位置代表用户组。

### 30.3.3 版本库ACL

一个版本库可以包含多条授权指令，这些授权指令组成了一个版本库的权限控制列表（ACL）。例如：

```
repo testing
    RW+                 = jiangxin @admin
    RW                  = @dev @test
    R                   = @all
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 1. 版本库

每一个版本库授权都以一条 repo 指令开始。

- 指令 repo 后面是版本库列表，版本之间用空格分开，还可以包括版本库组。
  注意：版本库名称不要添加 .git 后缀。在版本库创建过程中会自动添加 .git 后缀。

```
repo sandbox/test1 sandbox/test2 @test_repos
```

- 用 repo 指令设置的版本库会自动在服务器上创建，但是如果 repo 指令后面的版本库名称中包含通配符，则不会自动创建。
- repo 指令后面的版本库名称中可以使用正则表达式，这种用正则表达式定义的版本库称为通配符版本库 。

在 Gitolite 对用户访问版本库名称进行匹配时，会自动给看似通配符版本库的名称加上前缀 ^ 和后缀 $。这一点和后面将要介绍的正则引用（refex）大不一样。

```
repo ossxp/.+
```

不过有时候使用了过于简单的正则表达式，如“myrepo.”，有可能会产生歧义，让 Gitolite 将希望用正则表达式表示的通配符版本库误判为普通版本库名称，在服务器端自动创建名为 myrepo..git 的版本库。解决歧义的一个办法是：在正则表达式的前面明确地插入 ^ 符号，或者在表达式后面添加 $ 符号，形如：“^myrepo.”、“myrepo.$”，或“^myrepo.$”。

### 2. 授权指令

在 repo 指令之后是缩进的一条或多条授权指令。授权指令的语法如下：

```
< 权限 >  [ 零个或多个正则表达式匹配的引用 ] = <user> [<user> ...]
```

- 每条指令必须指定一个权限。权限可以用下面任意一个权限关键字：

  C、R、RW、RW+、RWC、RW+C、RWD、RW+D、RWCD、RW+CD 。

- 权限后面包含一个可选的正则引用（refex）列表。

正则表达式格式的引用，简称正则引用（refex），对 Git 版本库的引用（分支、里程碑等）进行匹配。

如果在授权指令中省略正则引用，则意味着对全部的 Git 引用（分支、里程碑等）都有效。

正则引用如果不以 refs/ 开头，会自动添加 refs/heads/ 作为前缀。

正则引用如果不以 $ 结尾，则意味着后面可以匹配任意字符，相当于添加 .*$ 作为后缀。

- 权限后面也可以包含一个以 NAME/ 为前缀的路径列表，进行基于路径的授权。
- 授权指令以等号（=）为标记分为前后两段，等号后面的是用户列表。用户之间用空格分隔，并且可以使用用户组。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


3. 授权关键字

不同的授权关键字有不同的含义，有的授权关键字只用在特定的场合。

- C

  C 代表创建。仅在通配符版本库授权时可以使用。用于指定谁可以创建与通配符匹配的版本库。

- R、RW 和 RW+

  R 为只读。RW 为读写权限。RW+ 含义为除了具有读写权限外，还可以强制执行非快进式推送。

- RWC、RW+C

  只有当授权指令中定义了正则引用（正则表达式定义的分支、里程碑等）时，才可以使用该授权指令。其中 C 的含义是允许创建和正则表达式匹配的引用（分支或里程碑等）。加号（+）的含义是允许强制推送。

- RWD、RW+D

  只有当授权指令中定义了正则引用（正则表达式定义的分支、里程碑等）时，才可以使用该授权指令。其中 D 的含义是允许删除和正则表达式匹配的引用（分支或里程碑等），加号（+）的含义是允许强制推送。

- RWCD、RW+CD

  只有当授权指令中定义了正则引用（正则表达式定义的分支、里程碑等）时，才可以使用该授权指令。其中 C 的含义是允许创建和正则表达式匹配的引用（分支或里程碑等），D 的含义是允许删除和正则表达式匹配的引用（分支或里程碑等），加号（+）的含义是允许强制推送。

### 30.3.4 Gitolite 授权机制

Gitolite 的授权实际分为两个阶段，第一个阶段称为前 Git 阶段，即在 Git 命令执行前，由 SSH 连接触发的 `gl-auth-command` 命令执行的授权检查。包括：

- 版本库的读。

  如果用户拥有版本库（或至少一个分支）的下列权限之一：R、RW 或 RW+，则整个版本库（包含所有分支）对用户均可读。

  实际上为用户设置某个分支的 R 权限的含义并非其他分支不可读，而是此分支不可写。之所以 Gitolite 对读授权不能细化到分支甚至目录，只能粗放地对整个版本库进行读授权，是因为读授权只在版本库授权的第一个阶段进行检查，而在此阶段还获取不到版本库的分支。

- 版本库的写。

  版本库的写授权实际上要在两个阶段分别进行检查。第一阶段仅检查用户是否拥有下列权限之一：RW、RW+ 或 C 授权，具有这些授权则通过第一阶段的写权限检查。至

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


于要在第二个阶段进行基于分支和路径的写操作授权，以及对分支创建、删除和是否可强制更新进行判断，则参见后面对第二阶段授权过程的描述。

- 版本库的创建。

  仅对正则表达式定义的通配符版本库有效。即拥有 C 授权的用户可以创建和对应正则表达式匹配的版本库。同时该用户也拥有对版本库的读写权限。

Gitolite 对授权的第二个阶段的检查，实际上是通过 update 钩子脚本进行的。

- 因为版本库的读操作不执行 update 钩子，所以读操作只在授权的第一个阶段（前 Git 阶段）就完成了检查，授权的第二个阶段对版本库的读授权无任何影响。
- 钩子脚本 update 针对推送操作的各个分支进行逐一检查，因此第二个阶段可以进行针对分支写操作的精细授权。
- 在这个阶段可以获取到要更新的新、老引用的 SHA1 哈希值，因此可以判断出是否发生了非快进式推送、是否有新分支创建，以及是否发生了分支的删除，因此可以针对这些操作进行精细的授权。
- 基于路径的写授权也是在这个阶段进行的。

## 30.4 版本库授权案例

Gitolite 的授权非常强大也非常复杂，因此从版本库授权的实际案例来学习是非常有效的方式。

### 30.4.1 对整个版本库进行授权

授权文件如下：

```
1  @admin = jiangxin
2  @dev   = dev1 dev2 badboy jiangxin
3  @test  = test1 test2
4
5  repo testing
6      R = @test
7      - = badboy
8      RW = @dev test1
9      RW+ = @admin
```

说明：

- 用户 test1 对版本库具有写的权限。

  第 6 行定义了 test1 所属的用户组 @test 具有只读权限。第 8 行定义了 test1 用户具有读写权限。Gitolite 的实现是对读权限和写权限分别进行判断并汇总（并集），从而 test1 用户具有读写权限。

- 用户 jiangxin 对版本库具有写的权限，并能够强制推送。

  第 9 行授权指令中加号（+）的含义是允许强制推送。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


❑ 禁用指令，让用户 badboy 只对版本库具有读操作的权限。

第 7 行的指令以减号（-）开始，是一条禁用指令。禁用指令只在授权的第二阶段起作用，即只对写操作起作用，不会对 badboy 用户的读权限施加影响。

在第 8 行的指令中，badboy 所在的 @dev 组拥有读写权限。但禁用规则会对写操作起作用，导致 badboy 只有读操作权限，而没有写操作。

## 30.4.2 通配符版本库的授权

授权文件如下：

```
1  @administrators = jiangxin admin
2  @dev    = dev1 dev2 badboy
3  @test   = test1 test2
4
5  repo sandbox/.+$
6      C = @administrators
7      R = @test
8      - = badboy
9      RW = @dev test1
```

这个授权文件的版本库名称中使用了正则表达式，匹配在 sandbox 下的任意版本库。

正则表达式末尾的 $ 有着特殊的含义，代表匹配字符串的结尾，明确告诉 Gitolite 这个版本库是通配符版本库。因为加号 + 既可以作为普通字符出现在版本库的命名中，又可以作为正则表达式中特殊含义的字符，如果 Gitolite 将授权文件中的通配符版本库误判为普通版本库，就会自动在服务器端创建该版本库，这可不是管理员希望发生的。

我修改了 Gitolite 的代码，能正确判断部分正则表达式，但是最好还是对简单的正则表达式添加 ^ 作为前缀或 $ 作为后缀，以避免误判。

正则表达式定义的通配符版本库不会自动创建，需要管理员手动创建。

Gitolite 原来对通配符版本库的实现是克隆即创建，但是这样很容易因为录入错误而导致错误的版本库被意外创建。我改进的 Gitolite 需要通过推送来创建版本库。

下面的示例通过推送操作（以 admin 用户身份），远程创建版本库 sandbox/repos1.git。

```
$ git push gitolite①:sandbox/repos1.git master
```

创建完毕后对各个用户的权限进行测试会发现：

❑ 用户 admin 对版本库具有写的权限。

这并不是因为第 6 行的授权指令为 @administrators 授予了 C 的权限。而是因为该版本库是由 admin 用户创建的，创建者具有对版本库完全的读写权限。

服务器端该版本库目录自动生成的 gl-creator 文件记录了创建者的 ID 为 admin。

❑ 用户 jiangxin 对版本库没有读写权限。

① gitolite 是安装 Gitolite 过程中创建的主机别名，是以 admin 用户身份连接 Git 服务器。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


虽然用户 jiangxin 和用户 admin 一样都可以在 sandbox/ 下创建版本库，但是由于 sandbox/repos1.git 已经存在并且不是 jiangxin 用户创建的，所以 jiangxin 用户没有任何权限，不能读写。

- 和之前的例子相同的是：
  - 用户 test1 对版本库具有写的权限。
  - 禁用指令让用户 badboy 对版本库只具有读操作的权限。
- 版本库的创建者还可以使用 setperms 命令为版本库添加授权。具体用法参见下面的示例。

### 30.4.3 用户自己的版本库空间

授权文件如下：

```
1  @administrators = jiangxin admin
2
3  repo users/CREATOR/.+$
4      C = @all
5      R = @administrators
```

说明：

- 第 5 条指令，设置管理员组对任何用户在 users/ 目录下创建的版本库都有只读权限。
- 第 4 条指令，设置用户可以在自己的名字空间（/usrs/<userid>/）下，自己创建版本库。例如下面就是用户 dev1 在服务器端自己的名字空间下创建版本库。

```
$ git push dev1-server[①]:users/dev1/repos1.git master
```

- 用户 dev1 可以通过 ssh 连接服务器，使用 setperms 命令为自己的版本库进行二次授权。当 setperms 指令执行时，会启用编辑界面，授权指令录入完毕后，输入 ^D（Ctrl+D）结束编辑。如下所示：

```
$ ssh dev1-server setperms users/dev1/repos1.git
    R = dev2
    RW = jiangxin
    ^D
```

- 在执行 setperms 进行授权时，也可以预先将授权写入文件，再使用输入重定向，通过 setperms 命令加载，如下所示。

```
$ cat > perms << EOF
    R = dev2
    RW = jiangxin
    EOF

$ ssh dev1@server setperms < perms
```

① dev1-server 是别名主机，是用 dev1 用户的公钥访问 server。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 用户可以使用 getperms 查看为自己的版本库建立的授权。

```
$ ssh dev1@server getperms users/dev1/repos1.git
    R = dev2
    RW = jiangxin
```

### 30.4.4 对引用的授权：传统模式

传统的引用授权指的是授权指令中不包含 RWC、RWD、RWCD、RW+C、RW+D、RW+CD 授权关键字，只采用 RW 和 RW+ 的传统授权关键字。

在只使用传统的授权关键字的情况下，有如下注意事项：

- 非快进式推送必须拥有 + 的授权。
- 创建引用必须拥有 W 的授权。
- 删除引用必须拥有 + 的授权。
- 如果没有在授权指令中提供引用相关的参数，相当于提供 refs/.* 作为引用的参数，意味着对所有引用均有效。

授权文件：

```
1  @administrators = jiangxin admin
2  @dev    = dev1 dev2 badboy
3
4  repo test/repo1
5      RW+ = @administrators
6      RW master refs/heads/feature/ = @dev
7      R   = @test
```

说明：

- 第 5 行，对于版本库 test/repo1，管理员组用户 jiangxin 和 admin 可以任意创建和删除引用，并且可以强制推送。
- 第 6 行的规则看似是只对 master 和 refs/heads/feature/* 的引用授权，实际上 @dev 可以读取所有名字空间的引用。这是因为读取操作无法获得引用相关的内容。即用户组 @dev 的用户只能对 master 分支，以及以 feature/ 开头的分支进行写操作，但不能强制推送和删除。至于其他分支和里程碑，则只能读不能写。
- 至于用户组 @test 的用户，因为使用了 R 授权指令，所以不涉及分支的写授权。

### 30.4.5 对引用的授权：扩展模式

扩展模式的引用授权，指的是该版本库的授权指令出现了下列授权关键字中的一个或多个：RWC、RWD、RWCD、RW+C、RW+D、RW+CD。则 Gitolite 对授权采用新的判定方式。

- 非快进式推送必须拥有 + 的授权。
- 创建引用必须拥有 C 的授权。
- 删除引用必须拥有 D 的授权。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


即引用的创建和删除使用了单独的授权关键字，和写权限和强制推送权限分开。

下面是一个采用扩展授权关键字的授权文件：

```
repo test/repo2
    RW+C = @administrators
    RW+  = @dev
    RW   = @test

repo test/repo3
    RW+CD = @administrators
    RW+C  = @dev
    RW    = @test
```

通过上面的配置文件，对于版本库 test/repo2.git 具有如下的授权：

- 用户组 @administrators 中的用户，具有创建和删除引用的权限，并且能够强制推送。
- 用户组 @dev 中的用户，不能创建引用，但可以删除引用，并且可以强制推送。
- 用户组 @test 中的用户，可以推送到任何引用，但是不能创建引用，不能删除引用，也不能强制推送。

通过上面的配置文件，对于版本库 test/repo3.git 具有如下的授权：

- 用户组 @administrators 中的用户，具有创建和删除引用的权限，并且能够强制推送。
- 用户组 @dev 中的用户，可以创建引用，并能够强制推送，但不能删除引用。
- 用户组 @test 中的用户，可以推送到任何引用，但是不能创建引用，不能删除引用，也不能强制推送。



## 30.4.6 对引用的授权：禁用规则的使用

授权文件：

```
 1 repo testing

       ...

12     RW      refs/tags/v[0-9]        =   jiangxin
13     -       refs/tags/v[0-9]        =   dev1 dev2 @others
14     RW      refs/tags/              =   jiangxin dev1 dev2 @others
```

说明：

- 用户 jiangxin 可以写任何里程碑，包括以 v 加上数字开头的里程碑。
- 用户 dev1、dev2 和 @others 组，只能写除了以 v 加上数字开头之外的其他里程碑。
- 其中以 - 开头的授权指令建立禁用规则。禁用规则只在授权的第二阶段有效，因此不能限制用户的读取权限！

## 30.4.7 用户分支

和创建用户空间（使用了 CREATOR 关键字）的版本库类似，还可以在一个版本库内允

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


许管理自己名字空间（USER关键字）下的分支。在正则引用的参数中出现的USER关键字会被替换为用户的ID。

授权文件：

```
repo test/repo4
    RW+CD = @administrators
    RW+CD refs/personal/USER/   = @all
    RW+     master = @dev
```

说明：

- 用户组@administrators中的用户，对所有引用具有创建和删除的权限，并且能强制推送。
- 所有用户都可以在refs/personal/<userid>/（自己的名字空间）下创建和删除引用。但是不能修改其他人的引用。
- 用户组@dev中的用户对master分支具有读写和强制更新的权限，但是不能删除。

### 30.4.8 对路径的写授权

Gitolite也实现了对路径的写操作的精细授权，并且非常巧妙的是：在实现上增加的代码可以忽略不计。这是因为Gitolite把路径当作是特殊格式的引用的授权。

在授权文件中，如果一个版本库的授权指令中的正则引用字段出现了以NAME/开头的引用，则表明该授权指令是针对路径进行的写授权，并且该版本库要进行基于路径的写授权判断。

示例：

```
1  repo foo
2      RW                  =   @junior_devs @senior_devs
3
4      RW  NAME/           =   @senior_devs
5      -   NAME/Makefile   =   @junior_devs
6      RW  NAME/           =   @junior_devs
```

说明：

- 第2行，初级程序员@junior_devs和高级程序员@senior_devs可以对版本库foo进行读写操作。
- 第4行，设定高级程序员@senior_devs对所有文件（NAME/）进行写操作。
- 第5行和第6行，设定初级程序员@junior_devs对除了根目录的Makefile文件外的其他文件具有写权限。

## 30.5 创建新版本库

Gitolite维护的版本库默认位于安装用户主目录下的repositories目录中，即如果安

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



装用户为 git，则版本库都创建在 /home/git/repositories 目录之下。可以通过配置文件 .gitolite.rc 修改默认的版本库的根路径。

```
$REPO_BASE="repositories";
```

有多种创建版本库的方式。一种是在授权文件中用 repo 指令设置版本库（未使用正则表达式的版本库）的授权，当对 gitolite-admin 版本库执行 git push 操作时，自动在服务端创建新的版本库。另外一种方式是在授权文件中用正则表达式定义的通配符版本库，不会即时创建（也不可能被创建），而是被授权的用户在远程创建后推送到服务器上完成创建。

注意：在授权文件中出现的版本库名称不要带 .git 后缀，在创建版本库过程中会自动在版本库后面追加 .git 后缀。

### 30.5.1 在配置文件中出现的版本库，即时生成

尝试在授权文件 conf/gitolite.conf 中加入一段新的版本库授权指令，而这个版本库尚不存在。新添加到授权文件中的内容为：

```
repo testing2
    RW+                     = @all
```

然后将授权文件的修改提交并推送到服务器，会看到授权文件中添加新授权的版本库 testing2 被自动创建。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 375 bytes, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Already on 'master'
remote: creating testing2...
remote: Initialized empty Git repository in /home/git/repositories/testing2.git/
To gitadmin.bj:gitolite-admin.git
   278e54b..b6f05c1  master -> master
```

注意其中带 remote 标识的输出，可以看到版本库 testing2.git 被自动初始化了。

此外使用版本库组的语法（即用 @ 创建的组，用作版本库），也会被自动创建。例如下面的授权文件片段设定了一个包含两个版本库的组 @testing，当将新配置文件推送到服务器上时，会自动创建 testing3.git 和 testing4.git。

```
@testing = testing3 testing4

repo @testing
    RW+                     = @all
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


还有一种版本库语法，是用正则表达式定义的版本库，这类版本库因为所指的版本库并不确定，因此不可能自动创建。

### 30.5.2 通配符版本库，管理员通过推送创建

通配符版本库是用正则表达式语法定义的版本库，所指的并非某一个版本库而是和名称相符的一组版本库。要想使用通配符版本库，需要在服务器端 Gitolite 的安装用户（如 git）主目录下，修改配置文件 .gitolite.rc，使其包含如下配置：

```
$GL_WILDREPOS = 1;
```

使用通配符版本库，可以对一组版本库进行授权，非常有效。但是版本库的创建则不像前面介绍的那样，不会在授权文件推送到服务器时创建，而是由拥有版本库创建授权（C）的用户手工进行创建。

对于用通配符设置的版本库，用 C 指令指定能够创建此版本库的管理员（拥有创建版本库的授权）。例如：

```
repo ossxp/.+
    C                   = jiangxin
    RW                  = dev1 dev2
```

管理员 jiangxin 可以创建路径符合正则表达式“ossxp/.+”的版本库，用户 dev1 和 dev2 对版本库具有读写（但是没有强制更新）权限。

使用该方法创建版本库后，创建者的 uid 将被记录在版本库目录下的 gl-creator 文件中。该账号具有对该版本库最高的权限。该通配符版本库的授权指令中如果出现 CREATOR 将被创建者的 uid 替换。

- 本地建库。

```
$ mkdir somerepo
$ cd somerepo
$ git init
$ git commit --allow-empty
```

- 使用 git remote 指令设置远程版本库。

```
$ git remote add origin jiangxin-server① :ossxp/somerepo.git
```

- 运行 git push 完成在服务器端版本库的创建。

```
$ git push origin master
```

Gitolite 的原始实现是通配符版本库的管理员在对不存在的版本库执行 clone 操作时自动创建的。但是我认为这不是一个好的实践，经常会因为在克隆时把 URL 写错，从而导致在

① jiangxin-server 是设置的别名主机，是以 jiangxin 用户的公钥访问 server 服务器。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


服务器端创建垃圾版本库。因此我重新改造了 gitolite 通配符版本库创建的实现方法，改为在对版本库进行推送的时候进行创建，而 clone 一个不存在的版本库会报错退出。

### 30.5.3 直接在服务器端创建

当版本库的数量很多的时候，在服务器端直接通过 `git init` 命令创建，或者通过复制创建可能会更方便。但是要注意，在服务器端手工创建的版本库和 Gitolite 创建的版本库最大的不同在于钩子脚本。如果不能为手工创建的版本库正确设定版本库的钩子，会导致失去 Gitolite 特有的一些功能，例如失去分支授权的功能。

一个由 Gitolite 创建的版本库，hooks 目录下有三个钩子脚本实际上链接到 gitolite 安装目录下的相应的脚本文件中：

```
gitolite-hooked -> /home/git/.gitolite/hooks/common/gitolite-hooked
post-receive.mirrorpush -> /home/git/.gitolite/hooks/common/post-receive.mirrorpush
update -> /home/git/.gitolite/hooks/common/update
```

那么手工在服务器上创建的版本库，有没有自动更新钩子脚本的方法呢？

第一个方法是修改 Git 模板[①]，在创建版本库时自动创建初始的钩子脚本。再有就是重新执行一遍 Gitolite 的安装，会自动更新版本库的钩子脚本。安装过程一路按回车即可。

```
$ cd gitolite/src
$ ./gl-easy-install git server admin
```

除了要注意钩子脚本以外，还要确保服务器端版本库目录的权限和属主。

## 30.6 对 Gitolite 的改进

笔者对 Gitolite 进行扩展和改进，涉及的内容主要包括：

- 通配符版本库的创建方式和授权。

  原来的实现是克隆即创建（克隆者需要被授予 C 的权限）。同时还要通过另外的授权语句为用户设置 RW 权限，否则创建者没有读和写的权限。

  新的实现是通过推送创建版本库（推送者需要被授予 C 权限）。不必再为创建者赋予 RW 等权限，创建者自动具有对版本库最高的授权。
- 避免通配符版本库的误判。

  若将通配符版本库误判为普通版本库名称，会导致在服务器端创建错误的版本库。新的设计可以在通配符版本库的正则表达式之前添加 ^ 或之后添加 $ 字符避免误判。
- 改变默认配置。

  默认安装即支持通配符版本库。

① 参见第 8 篇第 41 章“41.2.2 Git 模板”的相关内容。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 版本库重定向。

  Gitosis的一个很重要的功能——版本库名称重定向，没有在Gitolite中实现。我为Gitolite增加了这个功能。

  在Git服务器架设的初期，版本库的命名可能非常随意，例如，redmine的版本库直接放在根下：redmine-0.9.x.git、redmine-1.0.x.git…随着redmine项目越来越复杂，可能就需要将其放在子目录下进行管理，例如放到ossxp/redmine/目录下。只需要在Gitolite的授权文件中添加下面一行map语句，就可以实现版本库名称的重定向。使用旧地址的用户不必重新检出，可以继续使用。

```
map  redmine.*  =  ossxp/redmine/$1
```

## 30.7 Gitolite功能拓展

### 30.7.1 版本库镜像

#### 1. 版本库镜像的用途和原理

Git版本库控制系统往往并不需要设计特别的容灾备份，因为每一个Git用户就是一个备份。但是下面的情况，就很有必要考虑容灾了。

- Git版本库的使用者很少（每个库可能只有一个用户）。
- 版本库克隆只限制在办公区并且服务器也在办公区内（所有鸡蛋都在一个篮子里）。
- Git版本库采用集中式的应用模型，需要建立双机热备（以便在故障出现时，实现快速的服务器切换）。

  Gitolite提供了服务器间版本库同步的设置。原理是：

- 主服务器通过配置文件~/.gitolite.rc中的变量$ENV{GL_SLAVES}设置镜像服务器的地址。
- 从服务器通过配置文件~/.gitolite.rc中的变量$GL_SLAVE_MODE设置为从服务器模式。
- 从主服务器端运行脚本gl-mirror-sync可以实现批量的版本库镜像。
- 主服务器的每一个版本库都配置post-receive钩子，一旦有提交，即时同步到镜像版本库。

#### 2. 版本库镜像的实现方法

在多个服务器之间设置Git库镜像的方法是：

（1）每个服务器都要安装Gitolite软件，而且要启用post-receive钩子。默认的钩子在源代码的hooks/common目录下，名称为post-receive.mirrorpush，要将其改名为 post-receive。否则版本库的post-receive脚本不能生效。

（2）主服务器配置到从服务器的公钥认证，配置使用特殊的shell：gl-mirror-shell。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


这是因为主服务器在向从服务器同步版本库的时候，如果没有创建从服务器上相应的版本库，则需要直接通过 SSH 登录到从服务器，执行创建命令创建版本库。因此需要通过一个特殊的 shell，能够同时支持 Gitolite 的授权访问及 shell 环境。这个特殊的 shell 就是 gl-mirror-shell。而且这个 shell 可以通过特殊的环境变量绕过服务器的权限检查，避免因为授权问题而导致同步失败。

实际应用中，不光主服务器，每个服务器都要进行类似设置，目的是主从服务器可能相互切换。注意：在 Gitolite 不同的安装模式下，gl-mirror-shell 的安装位置可能不同。

下面的命令用于更改服务器端 Gitolite 安装用户的 ~/.ssh/authorized_keys 配置文件，以便使用特定公钥的其他服务器在访问本服务器时使用这个特殊的 shell。假设在服务器 foo 上，配置服务器 bar 和 baz 使用特殊的 shell，而来自这两个服务器的连接分别使用 bar.pub 和 baz.pub 两个公钥文件。

- 对于以客户端安装方式部署的 Gitolite，可以通过下面的方法确定 gl-mirror-shell 的位置，然后修改 ~/.ssh/authorized_keys 文件。

```
# 在服务器 foo 上执行：
$ export GL_ADMINDIR=` cd $HOME;perl -e 'do ".gitolite.rc"; print $GL_ADMINDIR'`
$ cat bar.pub baz.pub |
    sed -e 's,^,command="'$GL_ADMINDIR'/src/gl-mirror-shell" ,' >>
    ~/.ssh/authorized_keys
```

- 对于以服务器端安装方式部署的 Gitolite，可以在路径中找到 gl-mirror-shell，进而设置 ~/.ssh/authorized_keys 文件。

```
# 在服务器 foo 上执行：
$ cat bar.pub baz.pub |
    sed -e 's,^,command="'$(which gl-mirror-shell)'" ,' >> ~/.ssh/authorized_keys
```

（3）在 foo 服务器上设置完毕后，可以从服务器 bar 或 baz 上远程执行下列命令进行测试：

- 执行命令后退出。

```
$ ssh git@foo pwd
```

- 进入 shell。

```
$ ssh git@foo bash -i
```

（4）在从服务器上设置配置文件 ~/.gitolite.rc。

进行如下设置后，将不允许用户直接推送到从服务器。但是主服务器仍然可以推送到从服务器，是因为主服务器版本库在推送到从服务器时，使用了特殊的环境变量，能够跳过从服务器版本库的 update 脚本。

```
$GL_SLAVE_MODE = 1
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（5）在主服务器上设置配置文件 ~/.gitolite.rc。

需要配置到从服务器的 SSH 连接，可以设置多个，用空格分隔。注意使用单引号，以避免 @ 字符被 Perl 当作数组解析。

```
$ENV{GL_SLAVES} = 'gitolite@bar gitolite@baz';
```

（6）在主服务器端执行 gl-mirror-sync 进行一次完整的数据同步。

需要以 Gitolite 安装用户的身份（如 git）执行。例如在服务器 foo 上执行到从服务器 bar 的同步。

```
$ gl-mirror-sync gitolite@bar
```

（7）之后，用户每次向主版本库同步，都会通过版本库的 post-receive 钩子即时同步到从版本库。

当主版本库出现故障时，就需要把从服务器切换为主服务器模式，这就需要进行主从版本库的切换设置。切换非常简单，就是修改 ~/.gitolite.rc 配置文件，修改 $GL_SLAVE_MODE 设置：主服务器设置为 0，从服务器设置为 1。注意在主服务器恢复之前，要修改主服务器的配置使之降级为从服务器，否则主服务器恢复工作后会造成同时存在多个主服务器，从而导致数据的相互覆盖。

## 30.7.2 Gitweb 和 Git daemon 支持

Gitolite 和 git-daemon 的整合很简单，就是由 Gitolite 创建的版本库会在版本库目录中创建一个空文件 git-daemon-export-ok。

Gitolite 和 Gitweb 的整合则提供了两个方面的内容。一个是可以设置版本库的描述信息，用于在 Gitweb 的项目列表页面中显示。另外一个是自动生成项目的列表文件供 Gitweb 参考，避免 Gitweb 使用低效率的目录递归搜索查找 Git 版本库列表。

可以在授权文件中设定版本库的描述信息，并在 gitolite-admin 管理库更新时创建到版本库的 description 文件中。

```
reponame = "one line of description"
reponame "owner name" = "one line of description"
```

- 第 1 行，为名为 reponame 的版本库设定描述。
- 第 2 行，同时设定版本库的属主名称，以及一行版本库描述。

  对于通配符版本库，使用这种方法则很不现实。Gitolite 提供了 SSH 子命令供版本库的创建者使用。

```
$ ssh git@server setdesc path/to/repos.git
$ ssh git@server getdesc path/to/repos.git
```

- 第一条指令用于设置版本库的描述信息。
- 第二条指令显示版本库的描述信息。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


至于生成 Gitweb 所用的项目列表文件，默认创建在用户主目录下的 projects.list 文件中。对于所有启用 Gitweb 的 [repo] 小节所设定的版本库，以及通过版本库描述隐式声明的版本库都会加入到版本库列表中。

### 30.7.3 其他功能拓展和参考

Gitolite 源码的 doc 目录包含用 markdown 标记语言编写的手册，可以直接在 Github 上查看。也可以使用 markdown 的文档编辑工具将 .mkd 文档转换为 html 文档。转换工具很多，有 rdiscount、Bluefeather、Maruku、BlueCloth2，等等。

在这些参考文档中，用户可以发现 Gitolite 包含的更多的小功能或秘籍，包括：

- 版本库设置。

  授权文件通过 git config 指令为版本库进行附加的设置。例如：

```
repo gitolite
    config hooks.mailinglist = gitolite-commits@example.tld
    config hooks.emailprefix = "[gitolite] "
    config foo.bar = ""
    config foo.baz =
```

- 多级管理员授权。

  可以为不同的版本库设定管理员，操作 gitolite-admin 库的部分授权文件。具体参考：doc/5-delegation.mkd。

- 自定义钩子脚本。

  因为 Gitolite 占用了几个钩子脚本，如果需要对同名钩子进行扩展，Gitolite 提供了级联的钩子脚本，将定制放在级联的钩子脚本里。

  例如：通过自定义 gitolite-admin 的 post-update.secondary 脚本，以实现无须登录服务器即可更改 .gitolite.rc 文件。具体参考：doc/shell-games.mkd。

  关于钩子脚本的创建和维护，具体参考：doc/hook-propagation.mkd。

- 管理员自定义命令。

  通过设置配置文件中的 $GL_ADC_PATH 变量，在远程执行该目录下的可执行脚本，如：rmrepo。

  具体参考：doc/admin-defined-commands.mkd。

- 创建匿名的 SSH 认证。

  允许匿名用户访问 Gitolite 提供的 Git 服务。即建立一个和 Gitolite 服务器端账号同 ID 同主目录的用户，设置其的特定 shell，并且允许口令为空。

  具体参考：doc/mob-branches.mkd。

- 可以通过名为 @all 的版本库进行全局的授权。

  但是不能在 @all 版本库中对 @all 用户组进行授权。

- 版本库或用户非常之多（几千个）的时候，需要使用大配置文件模式。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


因为 Gitolite 的授权文件要先编译才能生效，而编译文件的大小是和用户及版本库数量的乘积成正比的。选择大配置文件模式则不对用户组和版本库组进行扩展。

具体参考：doc/big-config.mkd。

- 授权文件支持包含语句，可以将授权文件分成多个独立的单元。
- 执行外部命令，如 rsync。
- Subversion 版本库支持。

  如果在同一个服务器上以 svn+ssh 方式运行 Subversion 服务器，可以使用同一套公钥，同时为用户提供 Git 和 Subversion 服务。
- HTTP 口令文件维护。通过名为 htpasswd 的 SSH 子命令实现。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 31 章 Gitosis 服务架设

Gitosis 是 Gitolite 的鼻祖，同样也是一款基于 SSH 公钥认证的 Git 服务管理工具，但是功能要比之前介绍的 Gitolite 弱一些。Gitosis 由 Python 语言开发，对于偏爱 Python 不喜欢 Perl 的开发者（我就是其中之一），可以对 Gitosis 加以关注。

Gitosis 的出现远早于 Gitolite，作者 Tommi Virtanen 从 2007 年 5 月就开始了 Gitosis 的开发，最后一次提交是在 2009 年 9 月，已经停止更新了。但是 Gitosis 依然有其生命力。

- 配置简洁，可以直接在服务器端编辑，可成为针对某些服务定制的、内置的、无须管理的 Git 服务。

  Gitosis 的配置文件非常简单，直接保存于服务安装用户（如 git）的主目录下的 `.gitosis.conf` 文件中，可以直接在服务器端创建和编辑。而与之相比，Gitolite 的授权文件需要复杂的编译，一般需要管理员克隆 gitolite-admin 库，远程编辑并推送至服务器。若要用 Gitolite 实现一个无须管理的 Git 服务，难度要大很多。
- 支持版本库重定向。

  版本库重定向一方面在版本库路径变更后保持旧的 URL 仍可工作，另一方面用在客户端，以简洁的地址屏蔽服务器端复杂的地址。例如我开发的一款备份工具（Gistore），版本库位于 `/etc/gistore/tasks/system/repo.git`（符号链接），客户端使用 `system.git` 即映射到复杂的服务器端地址。

  注：我在定制的 Gitolite，这个功能也已实现。
- Python 语言开发，对于喜欢 Python 不喜欢 Perl 的用户可以选择 Gitosis。
- 在 Github 上有很多 Gitosis 的克隆，我对 Gitosis 的改动放在了 github 上：*http://github.com/ossxp-com/gitosis*。

因为 Gitosis 是 Gitolite 的鼻祖，因此下面介绍的 Gitosis 实现机理会让您感到似曾相识：

- Gitosis 安装在服务器（如 server）的某个账号（如 git）之下。
- 管理员通过 git 命令检出名为 gitosis-admin 的版本库。

```
$ git clone git@server:gitosis-admin.git
```

- 管理员将 git 用户的公钥保存在 gitosis-admin 库的 keydir 目录下，并编辑 `gitosis.conf` 文件为用户授权。
- 当管理员提交对 gitosis-admin 库的修改并推送到服务器之后，服务器上 gitosis-admin 版本库的钩子脚本将执行相应的设置工作。
  - 新用户公钥自动追加到服务器端安装账号用户主目录中的 `.ssh/authorized_`

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


keys 文件中，并设置新用户的登录 shell 为 gitosis 的一条命令 gitosis-serve。

```
command="gitosis-serve
jiangxin",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no
-pty ssh-rsa <公钥内容来自于 jiangxin.pub ...>
```

- 更新服务器端的授权文件 ~/.gitosis.conf。

❑ 用户可以用 Git 命令访问授权的版本库。

❑ 当管理员授权后，用户可以远程在服务器上创建新版本库。

下面介绍 Gitosis 的部署和使用。在下面的示例中约定：服务器的名称为 server，Gitosis 的安装账号为 git。

## 31.1 安装 Gitosis

Gitosis 的部署和使用可以直接参考源代码中的 README.rst 。可以直接访问 Github 上我的 Gitosis 克隆，因为 Github 能够直接将 rst 文件显示为网页。

具体参考：

```
http://github.com/ossxp-com/gitosis/blob/master/README.rst
```

### 31.1.1 Gitosis 的安装

Gitosis 的官方 Git 库位于 git://eagain.net/gitosis.git。我在 Github 上创建了一个克隆[①]。Gitosis 的安装需要在服务器端执行。下面介绍直接从源代码进行安装，以便获得最新的改进。

❑ 使用 Git 下载 Gitosis 的源代码。

```
$ git clone git://github.com/ossxp-com/gitosis.git
```

❑ 进入 gitosis 目录，执行安装。

```
$ cd gitosis
$ sudo python setup.py install
```

❑ 可执行脚本安装在 /usr/local/bin 目录下。

```
$ ls /usr/local/bin/gitosis-*
/usr/local/bin/gitosis-init  /usr/local/bin/gitosis-run-hook
/usr/local/bin/gitosis-serve
```

### 31.1.2 服务器端创建专用账号

安装 Gitosis，还需要在服务器端创建专用账号，所有用户都通过此账号访问 Git 库。一

① *http://github.com/ossxp-com/gitosis*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


般为方便易记，选择 git 作为专用账号名称。

```
$ sudo adduser --system --shell /bin/bash --disabled-password --group git
```

创建用户 git，并设置用户的 shell 为可登录的 shell，如 /bin/bash，同时添加同名的用户组。

有的系统，只允许特定用户组（如 ssh 用户组）的用户才可以通过 SSH 协议登录，这就需要将新建的 git 用户添加到 ssh 用户组中。

```
$ sudo adduser git ssh
```

### 31.1.3 Gitosis 服务初始化

Gitosis 服务初始化，就是初始化一个 gitosis-admin 库，并为管理员分配权限，还要将 Gitosis 管理员的公钥添加到专用账号的 ~/.ssh/authorized_keys 文件中，具体操作过程如下。

（1）如果管理员在客户端没有公钥，使用下面的命令创建。

```
$ ssh-keygen
```

（2）管理员上传公钥到服务器。

```
$ scp ~/.ssh/id_rsa.pub server:/tmp
```

（3）服务器端进行 Gitosis 服务初始化。

以 git 用户身份执行 gitosis-init 命令，并向其提供管理员公钥。

```
$ sudo su - git
$ gitosis-init < /tmp/id_rsa.pub
```

（4）确保 gitosis-admin 版本库的钩子脚本可执行。

```
$ sudo chmod a+x ~git/repositories/gitosis-admin.git/hooks/post-update
```

## 31.2 管理 Gitosis

### 31.2.1 管理员克隆 gitolit-admin 管理库

当 Gitosis 安装完成后，在服务器端自动创建了一个用于 Gitosis 自身管理的 Git 库：gitosis-admin.git。

管理员在客户端克隆 gitosis-admin.git 库，注意要确保认证中使用的是正确的公钥：

```
$ git clone git@server:gitosis-admin.git
$ cd gitosis-admin/

$ ls -F
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
gitosis.conf  keydir/

$ ls keydir/
jiangxin.pub
```

可以看出 gitosis-admin 目录下有一个配置文件和一个目录 keydir。

- keydir/jiangxin.pub 文件

  keydir 目录下初始时只有一个用户公钥，即管理员的公钥。管理员的用户名来自公钥文件末尾的用户名。

- gitosis.conf 文件

  该文件为授权文件。初始内容为：

```
1  [gitosis]
2
3  [group gitosis-admin]
4  writable = gitosis-admin
5  members = jiangxin
```

可以看到授权文件的语法完全不同于之前介绍的 Gitolite 的授权文件。整个授权文件以用户组为核心，而非版本库为核心。

- 第 3 行开始定义了一个用户组 gitosis-admin。
- 第 5 行设定了该用户组包含的用户列表。

  初始时只有一个用户，即管理员公钥所属的用户。
- 第 4 行设定了该用户组对哪些版本库具有写操作。

在这里，配置对 gitosis-admin 版本库具有写操作。写操作自动包含了读操作。

### 31.2.2 增加新用户

增加新用户，就是允许新用户能够通过其公钥访问 Git 服务。只要将新用户的公钥添加到 gitosis-admin 版本库的 keydir 目录下，即完成新用户的添加，具体操作过程如下。

（1）管理员从用户获取公钥，并将公钥按照 username.pub 格式进行重命名。

用户可以通过邮件或其他方式将公钥传递给管理员，切记不要将私钥误传给管理员。如果发生私钥泄露，马上重新生成新的公钥 / 私钥对，并将新的公钥传递给管理员，并申请将旧的公钥作废。

关于公钥名称，我引入了类似 Gitolite 的实现：

- 用户从不同的客户端主机访问有着不同的公钥，如果希望使用同一个用户名进行授权，可以按照 username@host.pub 的方式命名公钥文件，和名为 username@pub 的公钥指向同一个用户 username。
- 也支持邮件地址格式的公钥，即形如 username@gmail.com.pub 的公钥。Gitosis 能够很智能地区分是以邮件地址命名的公钥还是相同用户在不同主机上的公钥。如果

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


是邮件地址命名的公钥，将以整个邮件地址作为用户名。

（2）管理员进入 gitosis-admin 本地克隆版本库中，复制新用户公钥到 keydir 目录。

```
$ cp /path/to/dev1.pub keydir/
$ cp /path/to/dev2.pub keydir/
```

（3）执行 git add 命令，将公钥添加到版本库。

```
$ git add keydir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   keydir/dev1.pub
#       new file:   keydir/dev2.pub
#
```

（4）执行 git commit，完成提交。

```
$ git commit -m "add user: dev1, dev2"
[master d7952a5] add user: dev1, dev2
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 keydir/dev1.pub
 create mode 100644 keydir/dev2.pub
```

（5）执行 git push，同步到服务器，才真正完成新用户的添加。

```
$ git push
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.03 KiB, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@server:gitosis-admin.git
   2482e1b..d7952a5  master -> master
```

如果这时查看服务器端 ~git/.ssh/authorized_keys 文件，会发现新增的用户公钥也附加在其中：

```
### autogenerated by gitosis, DO NOT EDIT
command="gitosis-serve
jiangxin",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
<用户 jiangxin 的公钥 ...>
command="gitosis-serve
dev1",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
<用户 dev1 的公钥 ...>
command="gitosis-serve
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
dev2",no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
<用户 dev1 的公钥 ...>
```

### 31.2.3 更改授权

新用户添加完毕后，可能需要重新进行授权。更改授权的方法也非常简单，即修改 gitosis.conf 配置文件，提交并推送，具体操作过程如下。

（1）首先，管理员进入 gitosis-admin 本地克隆版本库中，编辑 gitosis.conf。

```
$ vi gitosis.conf
```

（2）授权指令比较复杂，先通过建立一个新用户组并授权新版本库 testing 尝试一下更改授权文件。例如在 gitosis.conf 中添加如下的授权内容：

```
 1   [group testing-admin]
 2   members = jiangxin @gitosis-admin
 3   admin = testing
 4
 5   [group testing-devloper]
 6   members = dev1 dev2
 7   writable = testing
 8
 9   [group testing-reader]
10  members = @all
11  readonly = testing
```

- 上面的授权文件为版本库 testing 赋予了三个角色。分别是 @testing-admin 用户组、@testing-developer 用户组和 @testing-reader 用户组。
- 第 1 行开始的 testing-admin 小节，定义了用户组 @testing-admin 。
- 第 2 行设定该用户组包含的用户有 jiangxin，以及前面定义的 @gitosis-admin 用户组用户。
- 第 3 行用 admin 指令，设定该用户组用户可以创建版本库 testing 。admin 指令是笔者新增的授权指令，请确认安装的 Gitosis 包含笔者的改进。
- 第 7 行用 writable 授权指令，设定该 @testing-developer 用户组用户可以读写版本库 testing 。笔者改进后的 Gitosis 也可以使用 write 作为 writable 指令的同义词指令。
- 第 11 行用 readonly 授权指令，设定该 @testing-reader 用户组用户（所有用户）可以只读访问版本库 testing。笔者改进后的 Gitosis 也可以使用 read 作为 readonly 指令的同义词指令。

（3）编辑结束，提交改动。

```
$ git add gitosis.conf
$ git commit -q -m "auth for repo testing."
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（4）执行 git push，同步到服务器，才真正完成授权文件的编辑。

```
$ git push
```

## 31.3 Gitosis 授权详解

### 31.3.1 Gitosis 默认设置

在 [gitosis] 小节中定义 Gitosis 的默认设置如下：

```
1  [gitosis]
2  repositories = /gitroot
3  #loglevel=DEBUG
4  gitweb = yes
5  daemon = yes
6  generate-files-in = /home/git/gitosis
```

其中：

- 第 2 行，设置版本库默认的根目录是 /gitroot 目录。否则默认的路径是安装用户主目录下的 repositories 目录。
- 第 3 行，如果打开注释，则版本库操作时显示 Gitosis 的调试信息。
- 第 4 行，启用 Gitweb 的整合。可以通过 [repo name] 小节为版本库设置描述字段，将会显示在 Gitweb 中。
- 第 5 行，启用 git-daemon 的整合。即在新创建的版本库中，创建文件 git-daemon-export-ok。
- 第 6 行，设置创建的项目列表文件（供 Gitweb 使用）所在的目录。默认为安装用户主目录下的 gitosis 目录。

### 31.3.2 管理版本库 gitosis-admin

```
1  [group gitosis-admin]
2  write = gitosis-admin
3  members = jiangxin
4  repositories = /home/git
```

除了第 4 行，其他内容在前面都已经介绍过了，是 Gitosis 自身管理版本库的用户组设置。

第 4 行，重新设置了版本库的默认根路经，覆盖默认的 [gitosis] 小节中的默认根路径。实际的 gitosis-admin 版本库的路径为 /home/git/gitosis-admin.git。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 31.3.3 定义用户组和授权

下面的两个示例小节定义了两个用户组，并且用到了路径变换的指令。

```
1  [group ossxp-admin]
2  members = @gitosis-admin jiangxin
3  admin = ossxp/**
4  read = gistore/*
5  map admin redmine-* = ossxp/redmine/\1
6  map admin ossxp/redmine-* = ossxp/(redmine-.*):ossxp/redmine/\1
7  map admin ossxp/testlink-* = ossxp/(testlink-.*):ossxp/testlink/\1
8  map admin ossxp/docbones* = ossxp/(docbones.*):ossxp/docutils/\1
9
10 [group all]
11 read = ossxp/**
12 map read redmine-* = ossxp/redmine/\1
13 map read testlink-* = ossxp/testlink/\1
14 map read pysvnmanager-gitsvn = mirrors/pysvnmanager-gitsvn
15 map read ossxp/redmine-* = ossxp/(redmine-.*):ossxp/redmine/\1
16 map read ossxp/testlink-* = ossxp/(testlink-.*):ossxp/testlink/\1
17 map read ossxp/docbones* = ossxp/(docbones.*):ossxp/docutils/\1
18 repositories = /gitroot
```

在上面的示例中，演示了授权指令及 Gitosis 特色的 map 指令。

- 第 1 行，定义了用户组 @ossxp-admin 。
- 第 2 行，设定该用户组包含用户 jiangxin 及用户组 @gitosis-admin 的所有用户。
- 第 3 行，设定该用户组具有创建及读写与通配符 ossxp/** 匹配的版本库的权限。

  两个星号匹配任意字符，包括路径分隔符（/）。此功能属于笔者扩展的功能。
- 第 4 行，设定该用户组可以只读访问 gistore/* 匹配的版本库。

  一个星号匹配任意字符，路径分隔符（/）除外。此功能也属于笔者扩展的功能。
- 第 5 行，是 Gitosis 特有的版本库名称重定位功能。

  即对 redmine-* 匹配的版本库，经过名称重定位，在名称前面加上 ossxp/remdine。其中 \1 代表匹配的整个版本库名称。

  用户组 @ossxp-admin 的用户对于重定位后的版本库具有 admin（创建和读写）的权限。
- 第 6 行，是我扩展的版本库名称重定位功能，支持正则表达式。

  等号左边的名称进行通配符匹配，匹配后，再经过右侧的一对正则表达式进行转换（冒号前的用于匹配，冒号后的用于替换）。
- 第 10 行，使用了内置的 @all 用户组，因此不需要通过 members 设定用户，因为所有用户均属于该用户组。
- 第 11 行，设定所有用户均可以只读访问 ossxp/** 匹配的版本库。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 第 12 ～ 17 行，对特定路径进行映射，并分配只读权限。
- 第 18 行，设置版本库的根路径为 /gitroot，而非默认的版本库根路径。

### 31.3.4 Gitweb 整合

Gitosis 和 Gitweb 的整合提供了两个方面的内容。一个是可以设置版本库的描述信息，用于在 Gitweb 的项目列表页面中显示。另外一个是自动生成项目的列表文件供 Gitweb 参考，避免 Gitweb 使用低效率的目录递归搜索查找 Git 版本库列表。

例如在 gitosis.conf 中，下面的配置用于对 redmine-1.0.x 版本库的 Gitweb 整合进行设置。

```
1  [repo ossxp/redmine/redmine-1.0.x]
2  gitweb = yes
3  owner = Jiang Xin
4  description = Redmine 1.0.x 群英汇定制开发
```

- 第 1 行，repo 小节设定版本库的路径。

  版本库的实际路径是用版本库默认的根（即在 [gitosis] 小节中定义的或默认的）加上此小节中的版本库路径组合而成的。
- 第 2 行，启用 Gitweb 整合。如果省略，使用全局 [gitosis] 小节中 Gitweb 的设置。
- 第 3 行，用于设置版本库的属主。
- 第 4 行，用于设置版本库的描述信息，显示在 Gitweb 的版本库列表中。

每一个 repo 小节所指向的版本库，如果启用了 Gitweb 选项，则版本库名称汇总到一个项目列表文件中。该项目列表文件默认保存在 ~/gitosis/projects.list 中。

## 31.4 创建新版本库

Gitosis 维护的版本库位于安装用户主目录下的 repositories 目录中，即如果安装用户为 git，则版本库都创建在 /home/git/repositories 目录之下。可以通过配置文件 gitosis.conf 修改默认的版本库的根路径。

可以直接在服务器端创建，或者在客户端远程创建版本库。

在客户端远程创建版本库时，Gitosis 的原始实现是这样的：对版本库具有 writable（读写）权限的用户，当对一个不存在的版本库执行克隆操作时会自动创建版本库，克隆即创建。但是我认为这不是一个好的实践，会经常因为在克隆时把 URL 写错，从而导致在服务器端创建垃圾版本库。笔者改进的实现如下：

- 增加了名为 admin（或 init）的授权指令，只有具有此授权的用户，才能够创建版本库。
- 只具有 writable（或 write）权限的用户，不能在服务器上创建版本库。
- 不是通过克隆创建版本库，而是在对版本库进行推送的时候创建。当克隆一个不存在



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的版本库时会报错退出。

远程在服务器上创建版本库的方法如下：

（1）首先本地建库。

```
$ mkdir somerepo
$ cd somerepo
$ git init
$ git commit --allow-empty
```

（2）使用 git remote 指令添加远程版本库。

```
$ git remote add origin git@server:ossxp/somerepo.git
```

（3）运行 git push 完成在服务器端版本库的创建。

```
$ git push origin master
```

## 31.5 轻量级管理的 Git 服务

轻量级管理的含义是不采用默认的稍嫌复杂的管理模式（远程克隆 gitosis-admin 库，修改并推送的管理模式），而是直接在服务器端通过预先定制的配置文件提供 Git 服务。这种轻量级管理模式，对于为某些应用建立快速的 Git 库服务提供了便利。

例如在使用备份工具 Gistore 进行文件备份时，可以用 Gitosis 架设轻量级的 Git 服务，可以在远程使用 Git 命令进行双机甚至是异地备份。

首先创建一个专用账号，并设置该用户只能执行 gitosis-serve 命令。例如创建账号 gistore，通过修改 /etc/ssh/sshd_config 配置文件，实现限制该账号登录的可执行命令。

```
Match user gistore
    ForceCommand gitosis-serve gistore
    X11Forwarding no
    AllowTcpForwarding no
    AllowAgentForwarding no
    PubkeyAuthentication yes
    #PasswordAuthentication no
```

上述配置信息告诉 SSH 服务器，凡是以 gistore 用户登录的账号，都将强制执行 Gitosis 的命令：gitosis-serve gistore。

然后，在该用户的主目录下创建一个配置文件 .gitosis.conf（注意文件名前面的点号），如下：

```
[gitosis]
repositories = /etc/gistore/tasks
gitweb = yes
daemon = no
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
[group gistore]
members = gistore
map readonly * = (.*):\1/repo
```

上述配置的含义是：

- 只有用户 gistore 才能够访问 /etc/gistore/tasks 下的 Git 库。
- 版本库的名称需要变换，例如 system 库会变换为实际路径 /etc/gistore/tasks/system/repo.git。





更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 32 章 Gerrit 代码审核服务器

谷歌 Android 开源项目在 Git 的使用上有两个重要的创新，一个是为多版本库协同而引入的 repo，这在前面第 25 章已经详细讨论过。另外一个重要的创新就是 Gerrit——代码审核服务器。Gerrit 为 Git 引入的代码审核是强制性的，也就是说除非特别的授权设置，向 Git 版本库的推送必须要经过 Gerrit 服务器，修订必须经过代码审核的一套工作流之后，才可能经批准并纳入正式代码库中。

首先贡献者的代码通过 git 命令（或 repo 封装）推送到 Gerrit 管理下的 Git 版本库，推送的提交转化为一个一个的代码审核任务，审核任务可以通过 `refs/changes/` 下的引用访问到。代码审核者可以通过 Web 界面查看审核任务、代码变更，通过 Web 界面做出通过代码审核或打回等决定。测试者也可以通过 `refs/changes/` 之下的引用获取修订然后对其进行测试，如果测试通过就可以将该评审任务设置为校验通过（verified）。最后经过了审核和校验的修订可以通过 Gerrit 界面中的提交动作合并到版本库对应的分支中。

Android 项目网站上有一个代码贡献流程图[①]，详细地介绍了 Gerrit 代码审核服务器的工作流程。翻译后的工作流程图如图 32-1。

## 32.1 Gerrit 的实现原理

Gerrit 更准确地说应该称为 Gerrit2。因为 Android 项目最早使用的评审服务器 Gerrit 不是今天这个样子的。最早版本的 Gerrit 是用 Python 开发运行于 Google App Engine 上的，从 Python 之父 Guido van Rossum 开发的 Rietveld 分支而来。在这里要讨论的 Gerrit 实为 Gerrit2，是用 Java 语言实现的[②]。

### 1. SSH 协议的 Git 服务器

Gerrit 本身基于 SSH 协议实现了一套 Git 服务器，这样就可以对 Git 数据推送进行更为精确的控制，为强制审核的实现建立了基础。

Gerrit 提供的 Git 服务的端口并非标准的 22 端口，默认是 29418 端口。这个端口是可以被发现的，当访问 Gerrit 的 Web 界面时可以得到这个端口。对 Android 项目的代码审核服务器，访问 *https://review.source.android.com/ssh_info* 就可以查看到 Git 服务的服务器域名和开放的端口。下面用 curl 命令查看网页的输出。

```
$ curl -L -k http://review.source.android.com/ssh_info
```

① *http://source.android.com/source/life-of-a-patch.html*

② *http://code.google.com/p/gerrit/wiki/Background*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



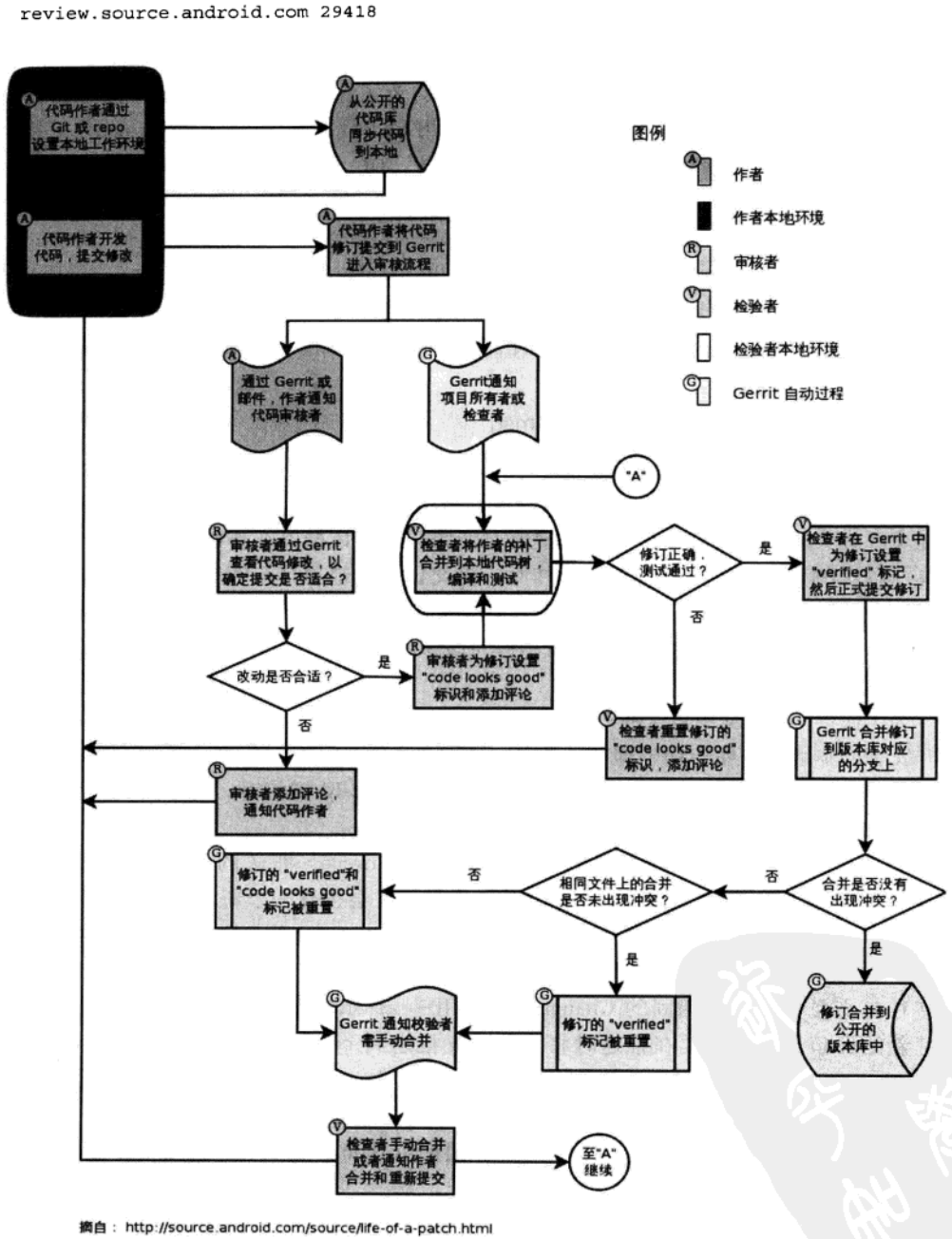



图 32-1 Gerrit 代码审核工作流



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


2. 特殊引用 refs/for 和 refs/changes

Gerrit 的 Git 服务器，禁止用户向 refs/heads 命名空间下的引用执行推送（除非特别的授权），即不允许用户直接向分支进行提交。为了让开发者能够向 Git 服务器提交修订，Gerrit 的 Git 服务器只允许用户向特殊的引用 refs/for/<branch-name> 下执行推送，其中 <branch-name> 即为开发者的工作分支。向 refs/for/<branch-name> 命名空间下推送并不会在其中创建引用，而是为新的提交分配一个 ID，称为 review-id，并为该 review-id 的访问建立如下格式的引用 refs/changes/nn/<review-id>/m，其中：

- review-id 是 Gerrit 为评审任务顺序而分配的全局唯一的号码。
- nn 为 review-id 的后两位数，位数不足用零补齐。即 nn 为 review-id 除以 100 的余数。
- m 为修订号，该 review-id 的首次提交修订号为 1，如果该修订被打回，重新提交修订号会自增。

3. Git 库的钩子脚本 hooks/commit-msg

为了保证已经提交审核的修订通过审核入库后，如果被别的分支拣选（cherry-pick）后再推送至服务器时不会产生新的重复的评审任务，Gerrit 设计了一套特殊的方法，即要求每个提交在提交说明中包含 Change-Id 键值对作为标签，该标签在首次生成时使用特殊的哈希算法以保障其唯一性。执行拣选操作时，提交说明会被保持，即来自原始提交说明中的 Change-Id 键值对也会保持不变，这样当新提交推送到 Gerrit 服务器时，Gerrit 会发现新的提交包含了已经处理过的 Change-Id，就不再为该修订创建新的评审任务和 review-id，而直接将提交入库。

为了使得 Git 提交中包含唯一的 Change-Id，Gerrit 提供了一个钩子脚本，将该脚本拷贝到开发者的本地 Git 库的钩子脚本目录中，即脚本文件 .git/hooks/commit-msg。这个钩子脚本在用户提交时，自动在提交说明中创建 Change-Id 键值对。至于如何实现 Change-Id 值的唯一性，可以参考该脚本。

当 Gerrit 获取到用户向 refs/for/<branch-name> 推送的提交中包含“Change-Id: I...”的格式时，如果该 Change-Id 之前没有见过，会创建一个新的评审任务并分配新的 review-id，并在 Gerrit 的数据库中保存 Change-Id 和 review-id 的关联。

如果用户的提交因为某种原因被打回重做，开发者修改之后重新推送到 Gerrit 时就要注意在提交说明中使用相同的 Change-Id（使用 --amend 提交即可保持提交说明），以免创建新的评审任务。还要在推送时将当前分支推送到 refs/changes/<nn>/<review-id>/<m> 中，是为该评审任务的一个新的修订，其中 <nn> 和 <review-id> 和之前提交的评审任务的修订号相同，<m> 则要人工选择一个新的修订号。

以上说起来很复杂，但是在实际操作中只要使用 repo 这一工具，就相对容易多了。

4. 其余一切交给 Web

Gerrit 另外一个重要的组件就是 Web 服务器，通过 Web 服务器实现对整个评审工作流的控制。关于 Gerrit 工作流，请参见本章开头出现的 Gerrit 工作流程图。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


想要感受一下 Gerrit 的魅力？请直接访问 Android 项目的 Gerrit 网站：*https://review.source.android.com/*，您会看到如图 32-2 的界面。

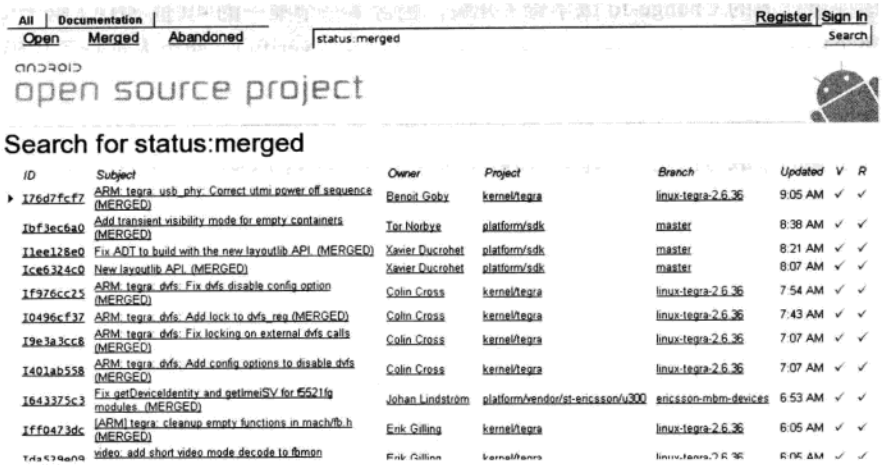



图 32-2 Android 项目代码审核网站

点击菜单中的“Merged”即可显示已经通过评审且合并到代码库中的审核任务。图 32-3 中显示的就是 Andorid 一个已经合并到代码库中的历史评审任务。

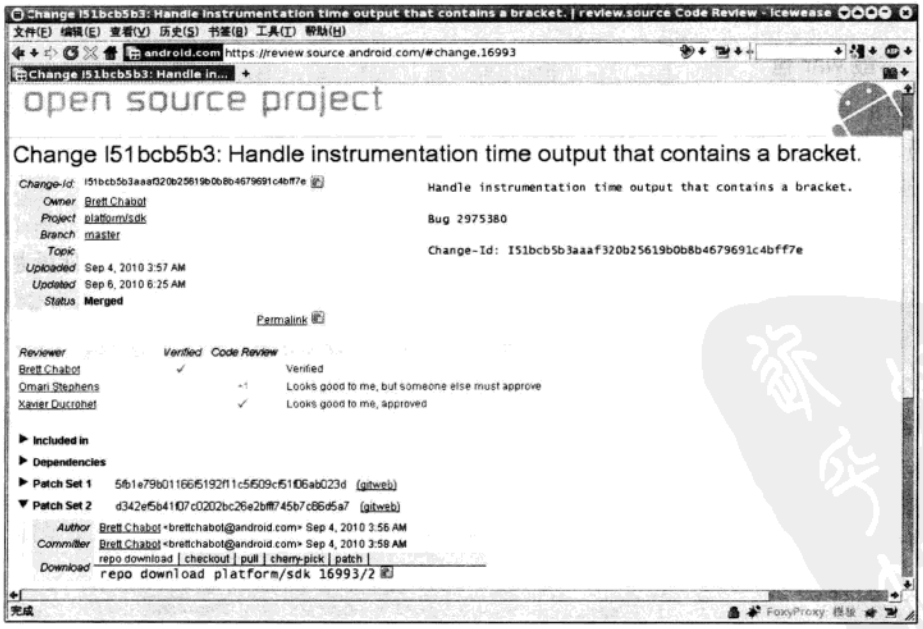



图 32-3 Android 项目的 16993 号评审

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


从图 32-3 中可以看出：

- URL 中显示的评审任务编号为 16993。
- 该评审任务的 Change-Id 以字母 I 开头，包含了一个唯一的 40 位 SHA1 哈希值。
- 整个评审任务有三个人参与，一个人进行了检查（verify），两个人进行了代码审核。
- 该评审任务的状态为已合并："merged"。
- 该评审任务总共包含两个补丁集：Patch set 1 和 Patch set 2。
- 补丁集的下载方法是：`repo download platform/sdk 16993/2`。

如果使用 repo 命令获取补丁集是非常方便的，因为封装后的 repo 屏蔽掉了 Gerrit 的一些实现细节，例如补丁集在 Git 库中的存在位置。如前所述，补丁集实际保存在 `refs/changes` 命名空间下。使用 `git ls-remote` 命令，从 Gerrit 维护的代码库中可以看到补丁集对应的引用名称。

```
$ git ls-remote \
      ssh://review.source.android.com:29418/platform/sdk \
      refs/changes/93/16993*
5fb1e79b01166f5192f11c5f509cf51f06ab023d        refs/changes/93/16993/1
d342ef5b41f07c0202bc26e2bfff745b7c86d5a7        refs/changes/93/16993/2
```

接下来就来介绍一下 Gerrit 服务器的部署和使用方法。

## 32.2 架设 Gerrit 的服务器

### 1. 下载 war 包

Gerrit 是由 Java 开发的，被封装为一个 war 包：gerrit.war，安装非常简洁。如果需要从源码编译出 war 包，可以参照相关文档[①]。不过最简单的就是从 Google Code 上直接下载编译好的 war 包。

从下面的地址下载 Gerrit 的 war 包：`http://code.google.com/p/gerrit/downloads/list`。在下载页面会有一个文件名类似 `Gerrit-x.x.x.war` 的 war 包，这个文件就是 Gerrit 的全部。示例中使用的是 2.1.5.1 版本，把下载的 `Gerrit-2.1.5.1.war` 包重命名为 `Gerrit.war`。下面的介绍就是基于这个版本。

### 2. 数据库选择

Gerrit 需要数据库来维护账户信息和跟踪评审任务等。目前支持的数据库类型有 PostgreSQL、MySQL 及嵌入式的 H2 数据库。

选择使用默认的 H2 内置数据库是最简单的，因为这样无须任何设置。如果想使用更为熟悉的 PostgreSQL 或 MySQL，则需要预先建立数据库。

① *http://gerrit.googlecode.com/svn/documentation/2.1.5/dev-readme.html*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


对于 PostgreSQL，在数据库中创建一个用户 gerrit，并创建一个数据库 reviewdb。

```
createuser -A -D -P -E gerrit
createdb -E UTF-8 -O gerrit reviewdb
```

对于 MySQL，在数据库中创建一个用户 gerrit 并为其设置口令（不要真如下面那样将口令设置为“secret”），并创建一个数据库 reviewdb。

```
$ mysql -u root -p

mysql> CREATE USER 'gerrit'@'localhost' IDENTIFIED BY 'secret';
mysql> CREATE DATABASE reviewdb;
mysql> ALTER DATABASE reviewdb charset=latin1;
mysql> GRANT ALL ON reviewdb.* TO 'gerrit'@'localhost';
mysql> FLUSH PRIVILEGES;
```

### 3. 以一个专用用户账号执行安装

在系统中创建一个专用的用户账号如：gerrit。以该用户身份执行安装，将 Gerrit 的配置文件、内置数据库、war 包等都自动安装在该用户主目录下的特定目录中。

```
$ sudo adduser gerrit
$ sudo su gerrit
$ cd ~gerrit
$ java -jar gerrit.war init -d review_site
```

在安装过程中会提出一系列问题。

（1）创建相关目录。

默认 Grerit 在安装用户主目录下创建 review_site，并把相关文件安装在这个目录之下。Git 版本库的根路径默认位于此目录之下的 git 目录中。

```
*** Gerrit Code Review 2.1.5.1
***

Create '/home/gerrit/review_site' [Y/n]?

*** Git Repositories
***

Location of Git repositories   [git]:
```

（2）选择数据库类型。

选择 H2 数据库是简单的选择，无须额外的配置。

```
*** SQL Database
***

Database server type           [H2/?]:
```

（3）设置 Gerrit Web 界面认证的类型。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


默认为 openid，即使用任何支持 OpenID 的认证源（如 Google、Yahoo！）进行身份认证。此模式支持用户自建账号，用户通过 OpenID 认证源的认证后，Gerrit 会自动从认证源获取相关属性如用户全名和邮件地址等信息创建账号。Android 项目的 Gerrit 服务器即采用此认证模式。

如果有可用的 LDAP 服务器，那么 ldap 或 ldap_bind 也是非常好的认证方式，可以直接使用 LDAP 中的已有账号进行认证，不过此认证方式下 Gerrit 的自建账号功能是关闭的。此安装示例选择的就是 LDAP 认证方式。

HTTP 认证也是可选的认证方式，此认证方式需要配置 Apache 的反向代理，并在 Apache 中配置 Web 站点的口令认证，通过口令认证后 Gerrit 在创建账号的过程中会询问用户的邮件地址并发送确认邮件。

```
*** User Authentication
***

Authentication method          [OPENID/?]: ?
       Supported options are:
         openid
         http
         http_ldap
         ldap
         ldap_bind
         development_become_any_account
Authentication method          [OPENID/?]: ldap
LDAP server                    [ldap://localhost]:
LDAP username                  :
Account BaseDN                 : dc=foo,dc=bar
Group BaseDN                   [dc=foo,dc=bar]:
```

（4）发送邮件设置。

默认使用本机的 SMTP 发送邮件。

```
*** Email Delivery
***

SMTP server hostname           [localhost]:
SMTP server port               [(default)]:
SMTP encryption                [NONE/?]:
SMTP username                  :
```

（5）Java 相关设置。

使用 OpenJava 和 Sun Java 均可。Gerrit 的 war 包要复制到 review_site/bin 目录中。

```
*** Container Process
***

Run as                         [gerrit]:
Java runtime                   [/usr/lib/jvm/java-6-sun-1.6.0.21/jre]:
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Copy gerrit.war to /home/gerrit/review_site/bin/gerrit.war [Y/n]?
Copying gerrit.war to /home/gerrit/review_site/bin/gerrit.war
```

（6）SSH 服务相关设置。

Gerrit 的基于 SSH 协议的 Git 服务非常重要，默认的端口为 29418。换成其他端口也无妨，因为 repo 可以自动探测到该端口。

```
*** SSH Daemon
***

Listen on address              [*]:
Listen on port                 [29418]:

Gerrit Code Review is not shipped with Bouncy Castle Crypto v144
  If available, Gerrit can take advantage of features
  in the library, but will also function without it.
Download and install it now [Y/n]?
Downloading http://www.bouncycastle.org/download/bcprov-jdk16-144.jar ...  OK
Checksum bcprov-jdk16-144.jar OK
Generating SSH host key ... rsa... dsa... done
```

（7）HTTP 服务相关设置。

默认启用内置的 HTTP 服务器，端口为 8080，如果该端口被占用（如 Tomcat），则需要更换为其他端口，否则服务启动失败。如下例就换成了 8081 端口。

```
*** HTTP Daemon
***

Behind reverse proxy           [y/N]? y
Proxy uses SSL (https://)      [y/N]? y
Subdirectory on proxy server   [/]: /gerrit
Listen on address              [*]:
Listen on port                 [8081]:
Canonical URL                  [https://localhost/gerrit]:

Initialized /home/gerrit/review_site
```

### 4. 启动 Gerrit 服务

Gerrit 服务正确安装后，运行 Gerrit 启动脚本来启动 Gerrit 服务。

```
$ /home/gerrit/review_site/bin/gerrit.sh start
Starting Gerrit Code Review: OK
```

服务正确启动之后，会看到 Gerrit 服务打开两个端口，这两个端口是在 Gerrit 安装时指定的。您的输出和下面的示例可能略有不同。

```
$ sudo netstat -ltnp | grep -i gerrit
tcp        0      0 0.0.0.0:8081            0.0.0.0:*               LISTEN
26383/GerritCodeRev
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
tcp        0      0 0.0.0.0:29418           0.0.0.0:*               LISTEN
26383/GerritCodeRev
```

### 5. 设置 Gerrit 服务开机自动启动

Gerrit 服务的启动脚本支持 start、stop、restart 参数，可以作为 init 脚本开机自动执行。

```
$ sudo ln -snf \
        /home/gerrit/review_site/bin/gerrit.sh \
        /etc/init.d/gerrit.sh
$ sudo ln -snf ../init.d/gerrit.sh /etc/rc2.d/S90gerrit
$ sudo ln -snf ../init.d/gerrit.sh /etc/rc3.d/S90gerrit
```

服务自动启动脚本 /etc/init.d/gerrit.sh 需要通过 /etc/default/gerritcodereview 提供一些默认的配置。以下面的内容来创建该文件。

```
GERRIT_SITE=/home/gerrit/review_site
NO_START=0
```

### 6. Gerrit 认证方式的选择

如果是开放的 Gerrit 服务，使用 OpenId 认证是最好的方法，就像谷歌 Android 项目的代码审核服务器配置的那样。任何人只要在可以作为 OpenId 提供者的网站上（如 Google、Yahoo！ 等）拥有账号，就可以直接通过 OpenId 注册，Gerrit 会在用户登录 OpenId 提供者网站成功后，自动获取（经过用户的确认）用户在 OpenId 提供者站点上的部分注册信息（如用户全名或邮件地址）在 Gerrit 上自动为用户创建账号。

如果架设有 LDAP 服务器，并且用户账号都在 LDAP 中进行管理，那么采用 LDAP 认证也是非常好的方法。登录时提供的用户名和口令通过 LDAP 服务器验证之后，Gerrit 会自动从 LDAP 服务器中获取相应的字段属性为用户创建账号。因为创建账号的用户全名和邮件地址来自于 LDAP，因此不能在 Gerrit 中更改，但是用户可以注册新的邮件地址。我在配置 LDAP 认证时遇到了一个问题就是创建账号的用户全名是空白的，这是因为没有正确设置 LDAP 的相关字段。如果 LDAP 服务器使用的是 OpenLDAP，Gerrit 会从 displayName 字段获取用户全名，如果使用 Active Directory 则用 givenName 和 sn 字段的值拼接形成用户全名。

Gerrit 还支持使用 HTTP 认证，这种认证方式需要架设 Apache 反向代理，在 Apache 中配置 HTTP 认证。用户若要访问 Gerrit 网站，首先需要通过 Apache 配置的 HTTP Basic Auth 认证，当 Gerrit 发现用户已经登录后，会要求用户确认邮件地址。当用户确认邮件地址后，再填写其他必须的字段完成账号注册。HTTP 认证方式的缺点除了在口令文件管理上需要管理员手工维护比较麻烦之外，还有一个缺点就是用户一旦登录成功后，想退出登录或更换其他用户账号登录会变得非常麻烦，除非关闭浏览器。关于用户切换有一个小窍门：例如 Gerrit 登录 URL 为 *https://server/gerrit/login/*，则用浏览器访问 *https://nobody:wrongpass@server/gerrit/login/*，即用错误的用户名和口令覆盖掉浏览器缓存的认证用户名和口令，这样就可以重新认证了。

在后面的 Gerrit 演示和介绍中，为了设置账号的方便，使用了 HTTP 认证，因此下面再

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


介绍一下 HTTP 认证的配置方法。

### 7. 配置 Apache 代理访问 Gerrit

默认 Gerrit 的 Web 服务端口为 8080 或 8081，通过 Apache 的反向代理就可以使用标准的 80（HTTP）或 443（HTTPS）来访问 Gerrit 的 Web 界面。

```
ProxyRequests Off
ProxyVia Off
ProxyPreserveHost On

<Proxy *>
      Order deny,allow
      Allow from all
</Proxy>

ProxyPass /gerrit/ http://127.0.0.1:8081/gerrit/
```

如果要配置 Gerrit 的 HTTP 认证，则还需要在上面的配置中插入 HTTP Basic Auth 认证的设置。

```
<Location /gerrit/login/>
  AuthType Basic
  AuthName "Gerrit Code Review"
  Require valid-user
  AuthUserFile /home/gerrit/review_site/etc/gerrit.passwd
</Location>
```

在上面的配置中，指定了口令文件的位置：/home/gerrit/review_site/etc/gerrit.passwd。可以用 htpasswd 命令维护该口令文件。

```
$ touch /home/gerrit/review_site/etc/gerrit.passwd
$ htpasswd -m /home/gerrit/review_site/etc/gerrit.passwd jiangxin
New password:
Re-type new password:
Adding password for user jiangxin
```

至此为止，Gerrit 服务安装完成。在正式使用 Gerrit 之前，先来研究一下 Gerrit 的配置文件，以免安装过程中遗漏或因错误的设置而影响使用。

## 32.3 Gerrit 的配置文件

Gerrit 的配置文件保存在部署目录下的 etc/gerrit.conf 文件中。如果对安装时的配置不满意，可以手工修改配置文件，重启 Gerrit 服务即可。

全部采用默认配置时的配置文件：

```
[gerrit]
        basePath = git
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        canonicalWebUrl = http://localhost:8080/
[database]
        type = H2
        database = db/ReviewDB
[auth]
        type = OPENID
[sendemail]
        smtpServer = localhost
[container]
        user = gerrit
        javaHome = /usr/lib/jvm/java-6-openjdk/jre
[sshd]
        listenAddress = *:29418
[httpd]
        listenUrl = http://*:8080/
[cache]
        directory = cache
```

如果采用LDAP认证，下面的配置文件片断配置了一个支持匿名绑定的LDAP服务器配置。

```
[auth]
        type = LDAP
[ldap]
        server = ldap://localhost
        accountBase = dc=foo,dc=bar
        groupBase = dc=foo,dc=bar
```

如果采用MySQL而非默认的H2数据库，下面的配置文件显示了相关配置。

```
[database]
        type = MYSQL
        hostname = localhost
        database = reviewdb
        username = gerrit
```

LDAP绑定或与数据库连接的用户口令保存在`etc/secure.config`文件中。

```
[database]
        password = secret
```

下面的配置将Web服务架设在Apache反向代理的后面。

```
[httpd]
        listenUrl = proxy-https://*:8081/gerrit
```

## 32.4 Gerrit的数据库访问

之所以要对数据库访问多说几句，是因为在Web界面往往无法配置对Gerrit的一些设置，需要直接修改数据库，而大部分用户在安装Gerrit时都会选用内置的H2数据库，可能大部分用户并不了解如何操作H2数据库。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


实际上无论选择何种数据库，Gerrit 都提供了两种数据库操作的命令行接口。第一种方法是在服务器端调用 gerrit.war 包中的命令入口，另外一种方法是远程 SSH 调用接口。

对于第一种方法，需要在服务器端执行，而且如果使用的是 H2 内置数据库还需要先将 Gerrit 服务停止。先以安装用户的身份进入 Gerrit 部署目录下，再执行命令调用 gerrit.war 包，如下：

```
$ java -jar bin/gerrit.war gsql
Welcome to Gerrit Code Review 2.1.5.1
(H2 1.2.134 (2010-04-23))

Type '\h' for help.  Type '\r' to clear the buffer.

gerrit>
```

当出现“gerrit>”提示符时，就可以输入 SQL 语句操作数据库了。

第一种方式需要登录到服务器上，而且操作 H2 数据库时还要预先停止服务，显然很不方便。但是这种方法也有存在的必要，就是不需要认证，尤其是在管理员账号尚未建立之前就可以查看和更改数据库。

当在 Gerrit 上注册了第一个账号时，即拥有了管理员账号，正确为该账号配置公钥之后，就可以访问 Gerrit 提供的 SSH 登录服务。Gerrit 的 SSH 协议提供访问数据库的第二种方法。下面的命令就是用管理员公钥登录 Gerrit 的 SSH 服务器，操作数据库。虽然演示用的是本机地址（localhost），但是操作远程服务器也是可以的，只要拥有管理员权限。

```
$ ssh -p 29418 localhost gerrit gsql
Welcome to Gerrit Code Review 2.1.5.1
(H2 1.2.134 (2010-04-23))

Type '\h' for help.  Type '\r' to clear the buffer.

gerrit>
```

运行命令 gerrit gsql 连接 Gerrit 的 SSH 服务。当连接上数据库管理接口后，便出现“gerrit>”提示符，在该提示符下可以输入 SQL 命令。下面的示例中，使用的数据库的后端为 H2 内置数据库。

可以输入 show tables 命令显示数据库列表。

```
gerrit> show tables;
 TABLE_NAME                  | TABLE_SCHEMA
 ----------------------------+-------------
 ACCOUNTS                    | PUBLIC
 ACCOUNT_AGREEMENTS          | PUBLIC
 ACCOUNT_DIFF_PREFERENCES    | PUBLIC
 ACCOUNT_EXTERNAL_IDS        | PUBLIC
 ACCOUNT_GROUPS              | PUBLIC
 ACCOUNT_GROUP_AGREEMENTS    | PUBLIC
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 ACCOUNT_GROUP_MEMBERS         | PUBLIC
 ACCOUNT_GROUP_MEMBERS_AUDIT   | PUBLIC
 ACCOUNT_GROUP_NAMES           | PUBLIC
 ACCOUNT_PATCH_REVIEWS         | PUBLIC
 ACCOUNT_PROJECT_WATCHES       | PUBLIC
 ACCOUNT_SSH_KEYS              | PUBLIC
 APPROVAL_CATEGORIES           | PUBLIC
 APPROVAL_CATEGORY_VALUES      | PUBLIC
 CHANGES                       | PUBLIC
 CHANGE_MESSAGES               | PUBLIC
 CONTRIBUTOR_AGREEMENTS        | PUBLIC
 PATCH_COMMENTS                | PUBLIC
 PATCH_SETS                    | PUBLIC
 PATCH_SET_ANCESTORS           | PUBLIC
 PATCH_SET_APPROVALS           | PUBLIC
 PROJECTS                      | PUBLIC
 REF_RIGHTS                    | PUBLIC
 SCHEMA_VERSION                | PUBLIC
 STARRED_CHANGES               | PUBLIC
 SYSTEM_CONFIG                 | PUBLIC
 TRACKING_IDS                  | PUBLIC
(27 rows; 65 ms)
```

输入 show columns 命令显示数据库的表结构。

```
gerrit> show columns from system_config;
 FIELD                       | TYPE          | NULL | KEY | DEFAULT
 ----------------------------+---------------+------+-----+--------
 REGISTER_EMAIL_PRIVATE_KEY  | VARCHAR(36)   | NO   |     | ''
 SITE_PATH                   | VARCHAR(255)  | YES  |     | NULL
 ADMIN_GROUP_ID              | INTEGER(10)   | NO   |     | 0
 ANONYMOUS_GROUP_ID          | INTEGER(10)   | NO   |     | 0
 REGISTERED_GROUP_ID         | INTEGER(10)   | NO   |     | 0
 WILD_PROJECT_NAME           | VARCHAR(255)  | NO   |     | ''
 BATCH_USERS_GROUP_ID        | INTEGER(10)   | NO   |     | 0
 SINGLETON                   | VARCHAR(1)    | NO   | PRI | ''
(8 rows; 52 ms)
```

关于 H2 数据库更多的 SQL 语法，请参考：*http://www.h2database.com/html/grammar.html*。下面开始介绍 Gerrit 的使用。

## 32.5 立即注册为 Gerrit 管理员

第一个 Gerrit 账户自动成为权限最高的管理员，因此 Gerrit 安装完毕后的第一件事情就是立即注册或登录，以便初始化管理员账号。下面的示例是在本机（localhost）以 HTTP 认证方式架设的 Gerrit 审核服务器。第一次访问的时候会弹出非常眼熟的 HTTP Basic Auth 认证界面，如图 32-4 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

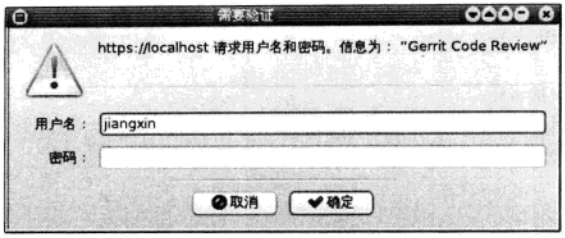




图 32-4 HTTP Basic Auth 认证界面

输入正确的用户名和口令登录后，系统自动创建 ID 为 1000000 的账号，该账号是第一个注册的账号，会被自动赋予管理员的身份。因为使用的是 HTTP 认证，用户的邮件地址等个人信息尚未确定，因此登录后首先进入到个人信息设置界面，如图 32-5。

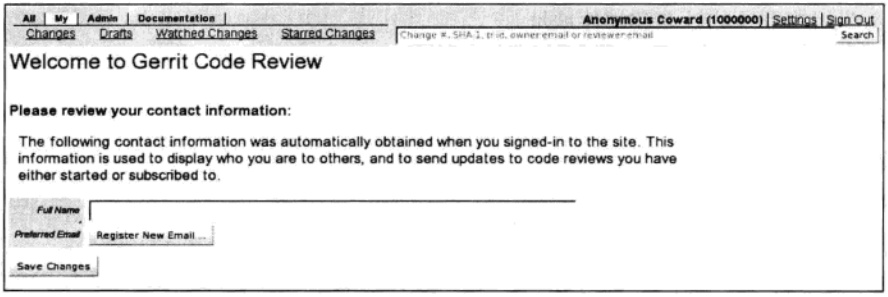



图 32-5 Gerrit 第一次登录后的个人信息设置界面

在图 32-5 中可以看到在菜单中有“Admin”菜单项，说明当前登录的用户被赋予了管理员权限。在图 32-5 的联系方式确认对话框中有一个注册新邮件地址的按钮，点击该按钮弹出邮件地址录入对话框，如图 32-6。

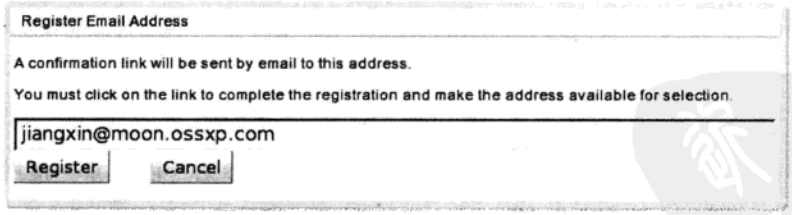



图 32-6 输入个人的邮件地址

必须输入一个有效的邮件地址以便能够收到确认邮件。这个邮件地址非常重要，因为 Git 代码提交时，在提交说明中出现的邮件地址需要和这个地址一致。填写了邮件地址后会收到一封确认邮件，点击邮件中的确认链接会重新进入到 Gerrit 账号设置界面，如图 32-7。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

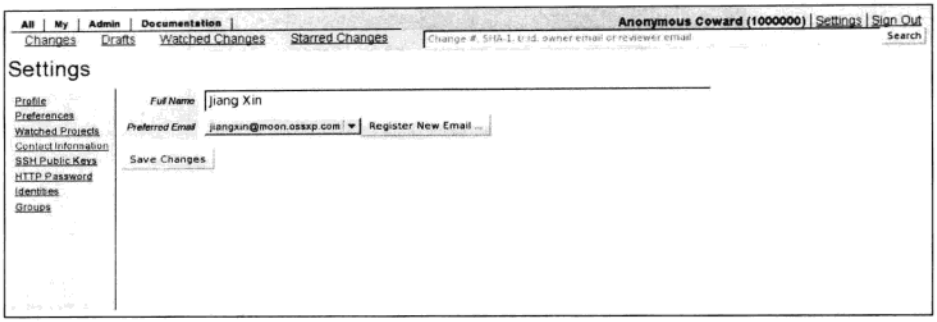




图 32-7 邮件地址确认后进入 Gerrit 界面

在 Full Name 字段输入用户名，点击保存更改后，右上角显示的“Anonymous Coward”就会显示为用户登录的姓名和邮件地址。

接下来需要做的最重要的一件事就是配置公钥（如图 32-8）。通过该公钥，注册用户可以通过 SSH 协议向 Gerrit 的 Git 服务器提交，如果具有管理员权限还能够远程管理 Gerrit 服务器。

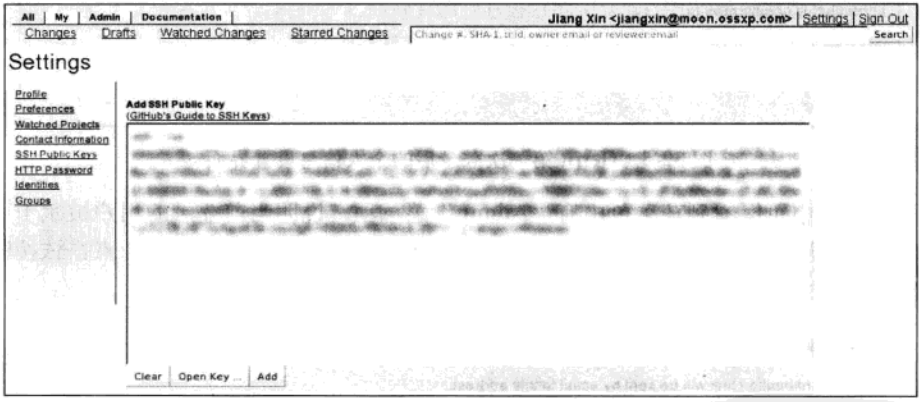



图 32-8 Gerrit 的 SSH 公钥设置界面

在文本框中粘贴公钥。关于如何生成和管理公钥，请参见“第 29 章 使用 SSH 协议”的相关内容。

点击“Add”按钮，完成公钥的添加。添加的公钥就会显示在列表中，如图 32-9。一个用户可以添加多个公钥。

点击左侧的“Groups”（用户组）菜单项，可以看到当前用户所属的分组，如图 32-10。

第一个注册的用户同时属于三个用户组，一个是管理员用户组（Administrators），另外两个分别是 Anonymous Users（任何用户）和 Registered Users（注册用户）。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

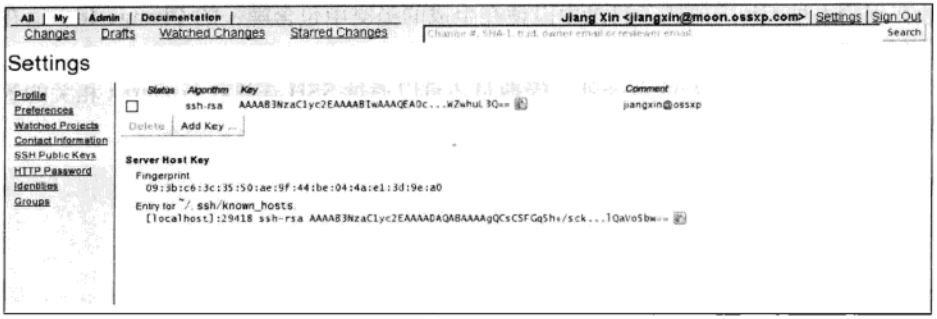




图 32-9 用户的公钥列表

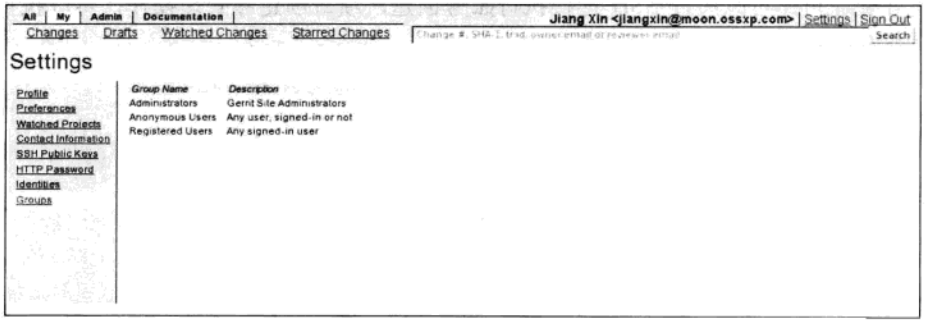



图 32-10 Gerrit 用户所属的用户组

## 32.6 管理员访问 SSH 的管理接口

在 Gerrit 个人配置界面中设置了公钥之后，就可以连接 Gerrit 的 SSH 服务器执行命令，示例使用的是本机 localhost，其实也可以使用远程 IP 地址。只是对于远程主机需要确认端口不要被防火墙拦截，Gerrit 的 SSH 服务器使用特殊的端口，默认是 29418。

任何用户都可以通过 SSH 连接执行 `gerrit ls-projects` 命令查看项目列表。下面的命令没有输出，是因为项目尚未建立。

```
$ ssh -p 29418 localhost gerrit ls-projects
```

可以执行 scp 命令从 Gerrit 的 SSH 服务器中拷贝文件。

```
$ scp -P 29418 -p -r localhost:/ gerrit-files
```

```
$ find gerrit-files -type f
gerrit-files/bin/gerrit-cherry-pick
gerrit-files/hooks/commit-msg
```

可以看出 Gerrit 服务器提供了两个文件可以通过 scp 下载，其中 commit-msg 脚本文件

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


应该放在用户本地Git库的钩子目录中以便在生成的提交中包含唯一的Change-Id。这在之前的Gerrit原理中介绍过。

除了普通用户可以执行的命令外，管理员还可以通过SSH连接执行Gerrit相关的管理命令。例如之前介绍的管理数据库：

```
$ ssh -p 29418 localhost gerrit gsql
Welcome to Gerrit Code Review 2.1.5.1
(H2 1.2.134 (2010-04-23))

Type '\h' for help.  Type '\r' to clear the buffer.

gerrit>
```

此外管理员还可以通过SSH连接执行账号创建，项目创建等管理操作，可以执行下面的命令查看帮助信息。

```
$ ssh -p 29418 localhost gerrit --help
gerrit COMMAND [ARG ...] [--] [--help (-h)]

 --           : end of options
 --help (-h)  : display this help text

Available commands of gerrit are:

   approve
   create-account
   create-group
   create-project
   flush-caches
   gsql
   ls-projects
   query
   receive-pack
   replicate
   review
   set-project-parent
   show-caches
   show-connections
   show-queue
   stream-events

See 'gerrit COMMAND --help' for more information.
```

更多的帮助信息，还可以参考Gerrit版本库中的帮助文件：Documentation/cmd-index.html。

## 32.7 创建新项目

一个Gerrit项目对应于一个同名的Git库，同时拥有一套可定制的评审流程。创建一个

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


新的 Gerrit 项目就会在对应的版本库根目录下创建 Git 库。管理员可以使用命令行创建新项目。

```
$ ssh -p 29418 localhost gerrit create-project --name new/project
```

执行 `gerrit ls-projects` 命令可以看到新项目已经成功创建。

```
$ ssh -p 29418 localhost gerrit ls-projects
new/project
```

在 Gerrit 的 Web 管理界面也可以看到新项目已经建立，如图 32-11。

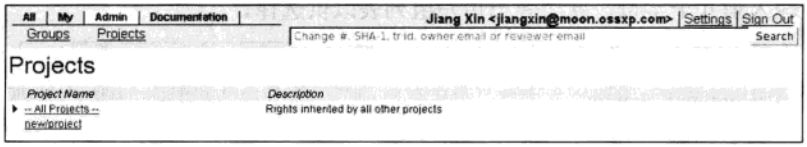



图 32-11 Gerrit 中项目列表

在项目列表中可以看到除了新建的 new/project 项目之外还有一个名为“-- All Projects --”的项目，其实它并非一个真实存在的项目，只是为了项目授权管理的方便——在“-- All Projects --”中建立的项目授权能够被其他项目共享。

在服务器端也可以看到在 Gerrit 部署中，版本库根目录下已经有同名的 Git 版本库被创建。

```
$ ls -d /home/gerrit/review_site/git/new/project.git
/home/gerrit/review_site/git/new/project.git
```

这个新的版本库刚刚初始化尚未包括任何数据。是否可以通过 `git push` 向该版本库推送一些初始数据呢？下面用 Gerrit 的 SSH 协议克隆该版本库，并尝试向其推送数据。

```
$ git clone ssh://localhost:29418/new/project.git myproject
Cloning into myproject...
warning: You appear to have cloned an empty repository.

$ cd myproject/

$ echo hello > readme.txt

$ git add readme.txt

$ git commit -m "initialized."
[master (root-commit) 15a549b] initialized.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 readme.txt
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 222 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/new/project.git
 ! [remote rejected] master -> master (prohibited by Gerrit)
error: failed to push some refs to 'ssh://localhost:29418/new/project.git'
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


向 Gerrit 的 Git 版本库推送失败，远程 Git 服务器返回错误信息："prohibited by Gerrit"。这是因为 Gerrit 默认不允许直接向分支推送，而是需要向 refs/for/<branch-name> 的特殊引用进行推送以便将提交转换为评审任务。

但是是否可以将版本库的历史提交不经审核，直接推送到 Gerrit 维护的 Git 版本库中呢？是的，只要通过 Gerrit 的管理界面为该项目授权：允许某个用户组（如 Administrators 组）的用户可以直接向分支推送。（注意该授权在推送完毕后尽快撤销，以免被滥用。）

Gerrit 的界面对用户非常友好（如图 32-12）。例如在添加授权的界面中，只要在用户组的输入框中输入前几个字母，就会弹出用户组列表以供选择。

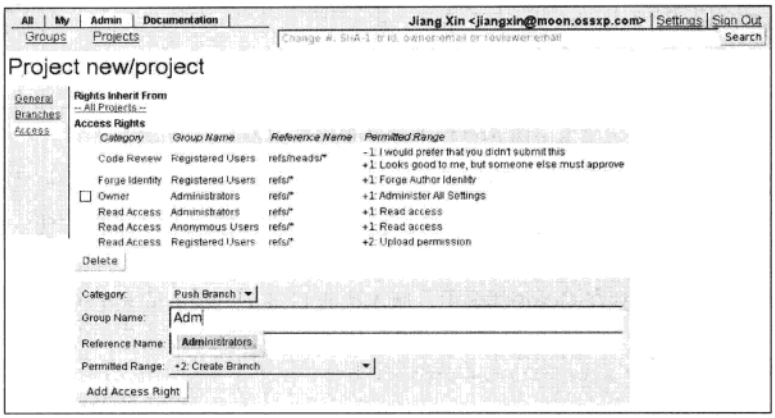



图 32-12 添加授权的界面

添加授权完毕后，项目"new/project"的授权列表就会出现新增的为 Administrators 管理员添加的"+2: Create Branch"授权，如图 32-13。

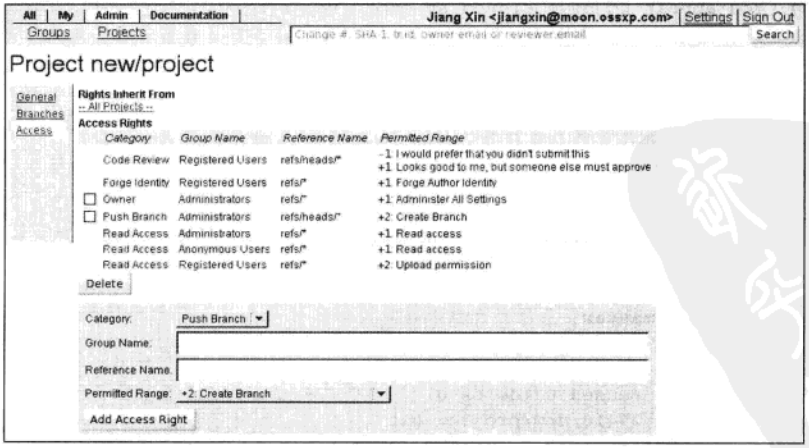



图 32-13 添加授权后的授权列表

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



因为已经为管理员分配了直接向 refs/heads/* 引用推送的授权，这样就能够向 Git 版本库推送数据了。再执行一次推送任务看看能否成功。

```
$ git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 222 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/new/project.git
 ! [remote rejected] master -> master (you are not committer jiangxin@ossxp.com)
error: failed to push some refs to 'ssh://localhost:29418/new/project.git'
```

推送又失败了，但是服务器端返回的错误信息却不同。上一次出错返回的是“prohibited by Gerrit”，而这一次返回的错误信息是“you are not committer”。

这是为什么呢？看看提交日志：

```
$ git log --pretty=full
commit 15a549bac6bd03ad36e643984fed554406480b2c
Author: Jiang Xin <jiangxin@ossxp.com>
Commit: Jiang Xin <jiangxin@ossxp.com>

    initialized.
```

提交者 Committer 为“*Jiang Xin <jiangxin@ossxp.com>*”，而 Gerrit 中注册的用户的邮件地址是“*jiangxin@moon.ossxp.com*”，两者之间不一致，导致 Gerrit 再一次拒绝了提交。如果再到 Gerrit 中看一下 new/project 的权限设置，会看到这样一条授权：

```
Category        Group Name        Reference Name  Permitted Range
========        ==========        ==============  ===============
Forge Identity  Registered Users  refs/*          +1: Forge Author Identity
```

这条授权的含义是：提交中的 Author 字段不进行邮件地址是否注册的检查，但是要对 Commit 字段进行邮件地址检查。如果增加一个更高级别的“Forge Identity”授权，也可以忽略对 Committer 的邮件地址的检查，但是尽量不要对授权进行非必须的改动，因为在提交的时候使用注册的邮件地址是一个非常好的实践。

下面就通过 git config 命令修改提交时所用的邮件地址，和 Gerrit 注册时用的地址保持一致。然后用 --amend 参数重新执行提交以便让修改后的提交者邮件地址在提交中生效。

```
$ git config user.email jiangxin@moon.ossxp.com

$ git commit --amend -m initialized
[master 82c8fc3] initialized
 Author: Jiang Xin <jiangxin@ossxp.com>
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 readme.txt

$ git push origin master
Counting objects: 3, done.
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Writing objects: 100% (3/3), 233 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/new/project.git
 * [new branch]      master -> master
```

看，这次提交成功了！之所以成功，是因为提交者的邮件地址更改了。看看重新提交的日志，可以发现 Author 和 Commit 的邮件地址不同，而 Commit 字段的邮件地址和注册时使用的邮件地址相同。

```
$ git log --pretty=full
commit 82c8fc3805d57cc0d17d58e1452e21428910fd2d
Author: Jiang Xin <jiangxin@ossxp.com>
Commit: Jiang Xin <jiangxin@moon.ossxp.com>

    initialized
```

注意，版本库初始化完成之后，应尽快删除为项目新增的“Push Branch”类型的授权，对新的提交强制使用 Gerrit 的评审流程。

## 32.8 从已有的 Git 库创建项目

如果已经拥有很多版本库，希望从这些版本库创建 Gerrit 项目，如果像上面介绍的那样一个一个地创建项目，再执行 `git push` 命令推送已经包含历史数据的版本库，将是十分麻烦的事情。那么有没有什么简单的办法呢？可以通过下面的步骤实现多项目的快速创建。

首先将已有版本库创建到 Gerrit 的版本库根目录下。注意版本库名称将会成为项目名（除去 .git 后缀），而且创建（或克隆）的版本库应为裸版本库，即使用 `--bare` 参数创建。

例如在 Gerrit 的 Git 版本库根目录下创建名为 hello.git 的版本库。下面的示例中我偷了一下懒，直接从 new/project 克隆到 hello.git。:)

```
$ git clone --mirror \
      /home/gerrit/review_site/git/new/project.git \
      /home/gerrit/review_site/git/hello.git
```

这时查看版本库列表，却看不到新建立的名为 hello.git 的 Git 库出现在项目列表中。

```
$ ssh -p 29418 localhost gerrit ls-projects
new/project
```

可以通过修改 Gerrit 数据库来注册新项目，即连接到 Gerrit 数据库，输入 SQL 插入语句。

```
$ ssh -p 29418 localhost gerrit gsql
Welcome to Gerrit Code Review 2.1.5.1
(H2 1.2.134 (2010-04-23))

Type '\h' for help.  Type '\r' to clear the buffer.

gerrit> INSERT INTO projects
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    -> (use_contributor_agreements ,submit_type ,name)
    -> VALUES
    -> ('N' ,'M' ,'hello');
UPDATE 1; 1 ms
gerrit>
```

注意 SQL 语句中的项目名称是版本库名称除去 .git 后缀的部分。在数据库插入数据后，再来查看项目列表就可以看到新注册的项目了。

```
$ ssh -p 29418 localhost gerrit ls-projects
hello
new/project
```

可以登录到 Gerrit 项目对新建立的项目进行相关设置。例如修改项目的说明，项目的提交策略，是否要求提交说明中必须包含“Signed-off-by”信息等，如图 32-14。




图 32-14 项目基本设置

这种通过修改数据库从已有版本库创建项目的方法适合创建大批量的项目。下面就对新建立的 hello 进行一次完整的 Gerrit 评审流程。

## 32.9 定义评审工作流

刚刚安装好的 Gerrit 的评审工作流并不完整，还不能正常地开展评审工作，需要对项目授权进行设置以定制适合的评审工作流。

默认安装的 Gerrit 中只内置了四个用户组，如表 32-1 所示。

表 32-1 Gerrit 内置用户组

| 用户组 | 说明 |
| --- | --- |
| Administrators | Gerrit 管理员 |
| Anonymous Users | 任何用户，登录或未登录 |
| Non-Interactive Users | Gerrit 中执行批处理的用户 |
| Registered Users | 任何登录用户 |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


未登录的用户只属于 Anonymous Users，登录用户则同时拥有 Anonymous Users 和 Registered Users 的权限。对于管理员则还拥有 Administrators 用户组权限。

查看全局（伪项目“-- All Projects--”）的初始权限设置，会看到如表 32-2 一样的授权表格。

表 32-2 Gerrit 授权表格

| 编号 | 类别 | 用户组名称 | 引用名称 | 权限范围 |
|---|---|---|---|---|
| 1 | Code Review | Registered Users | refs/heads/* | -1: I would prefer that you didn't submit this<br>+1: Looks good to me, but someone else must approve |
| 2 | Forge Identity | Registered Users | refs/* | +1: Forge Author Identity |
| 3 | Read Access | Administrators | refs/* | +1: Read access |
| 4 | Read Access | Anonymous Users | refs/* | +1: Read access |
| 5 | Read Access | Registered Users | refs/* | +2: Upload permission |

对此表格中的授权解读如下：

- 对于匿名用户：根据第 4 条授权策略，匿名用户能够读取任意版本库。
- 对于注册用户：根据第 5 条授权策略，注册用户具有比第四条授权高一个等级的权限，即注册用户除了具有读取版本库权限外，还可以向版本库的 `refs/for/<branch-name>` 引用推送，产生评审任务的权限。

之所以这种可写的权限也放在“Read Access”类别中，是因为 Git 的写操作必须建立在拥有读权限之上，因此 Gerrit 将其与读取都放在“Read Access”归类之下，只不过高一个级别。

- 对于注册用户：根据第 2 条授权策略，在向服务器推送提交的时候，忽略对提交中 Author 字段的邮件地址检查。这个在之前已经讨论过。
- 对于注册用户：根据第 1 条授权策略，注册用户具有代码审核的一般权限，即能够将评审任务设置为“+1”级别（看起来不错，但需要通过他人认可），或者将评审任务标记为“-1”（评审任务没有通过不能提交）。
- 对于管理员：根据第 3 条策略，管理员能够读取任意版本库。

上面的授权策略仅仅对评审流程进行了部分设置。如：提交能够进入评审流程，因为登录用户（注册用户）可以将提交以评审任务的方式上传；注册用户可以将评审任务标记为“+1：看起来不错，但需其他人认可”。但是没有人有权限可以将评审任务提交逐一合并到正式的版本库中，即没人能够对评审任务做最终的确认及提交，因此评审流程是不完整的。

有两种方法可以实现对评审最终确认的授权，一种是赋予特定用户 Verified 类别中的“+1: Verified”的授权，另外一个方法是赋予特定用户 Code Review 类别中更高级别的授权：“+2: Looks good to me, approved”。要想实现对经过确认的评审任务的提交，还需要赋予特定用户 Submit 类别中的“+1: Submit”授权。

下面的示例中，创建两个新的用户组 Reviewer 和 Verifier，并为其赋予相应的授权。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


可以通过 Web 界面或命令行创建用户组。如果通过 Web 界面添加用户组，选择“Admin”菜单下的“Groups”子菜单，如图 32-15。

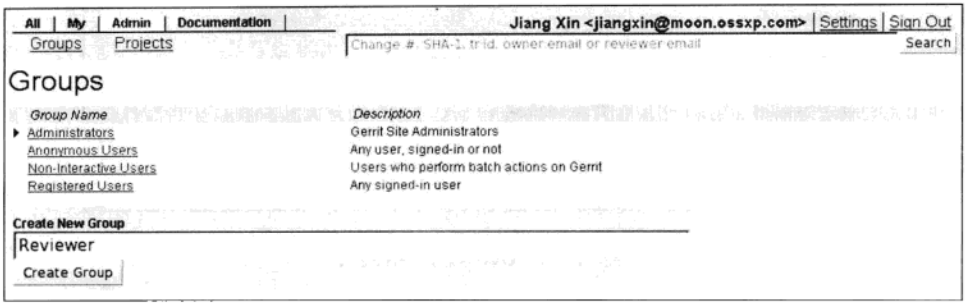



图 32-15 Gerrit 用户组创建

输入用户组名称后，点击“Create Group”按钮，进入创建用户组后的设置页，如图 32-16。

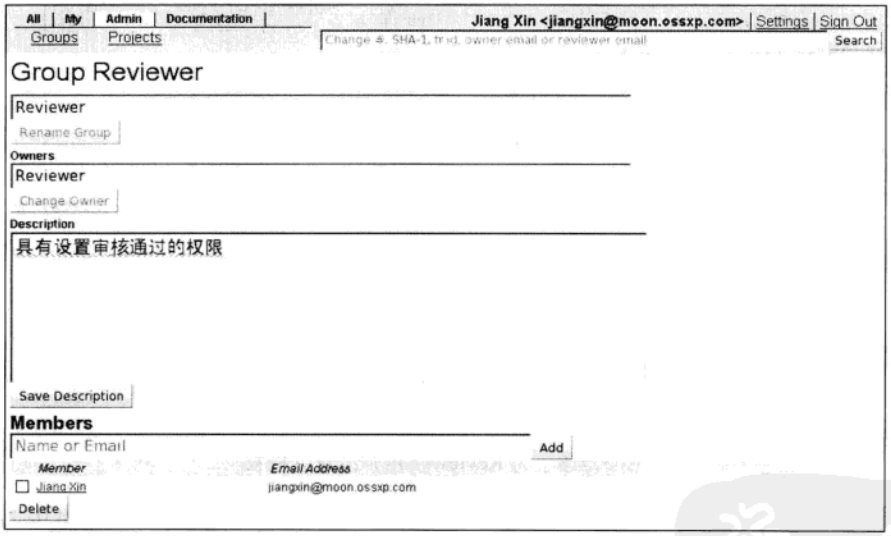



图 32-16 Gerrit 用户组设置页

注意到在用户设置页面中有一个 Owners 字段名称和用户组名称相同，实际上这是 Gerrit 关于用户组的一个特别的功能。一个用户组可以设置另外一个用户组为本用户组的 Owners，属于 Owners 用户组的用户实际上相当于本用户组的管理者，可以添加用户、修改用户组名称等。不过一般最常用的设置是使用同名的用户组作为 Owners。

在用户组设置页面的最下面，是用户组用户分配对话框，可以将用户分配到用户组中。

图 32-17 是添加了两个新用户组后的用户组列表：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





图 32-17 Gerrit 用户组列表

接下来要为新的用户组授权，需要访问“Admin”菜单下的“Projects”子菜单，点击对应的项目进入权限编辑界面。为了简便起见，选择“-- All Projects --”，对其授权的更改可以被其他的所有项目共享。图 32-18 是为 Reviewer 用户组建立授权过程的页面。

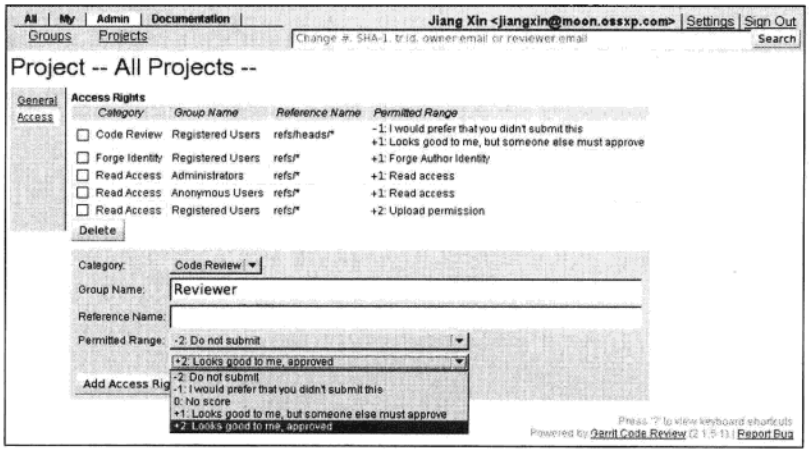



图 32-18 为 Reviewer 用户组建立授权

分别为两个新建立的用户组分配授权，如表 32-3 所示。编号从 6 开始，是因为这里补充的授权是建立在前面的默认授权列表的基础上的。

**表 32-3 新用户组权限分配表**

| 编号 | 类别 | 用户组名称 | 引用名称 | 权限范围 |
|---|---|---|---|---|
| 6 | Code Review | Reviewer | refs/* | -2: Do not submit<br>+2: Looks good to me, approved |
| 7 | Verified | Verifier | refs/* | -1: Fails<br>+1: Verified |
| 8 | Submit | Verifier | refs/* | +1: Submit |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


这样，就为 Gerrit 所有的项目设定了可用的评审工作流。

## 32.10 Gerrit 评审工作流实战

分别再注册两个用户账号 dev1@moon.ossxp.com 和 dev2@moon.ossxp.com，两个用户分别属于 Reviewer 用户组和 Verifier 用户组。这样 Gerrit 部署中就拥有了三个用户账号，用账号 jiangxin 进行代码提交，用 dev1 账号对任务进行代码审核，用 dev2 账号对审核任务进行最终的确认。

### 32.10.1 开发者在本地版本库中工作

repo 是 Gerrit 的最佳伴侣，凡是需要和 Gerrit 版本库交互的工作都封装在 repo 命令中。关于 repo 的用法在上一部分的 repo 多版本库协同的章节中已经详细介绍了。这里只介绍开发者如何只使用 git 命令来和 Gerrit 服务器交互。这样也可以更深入地理解 repo 和 Gerrit 整合的机制，具体操作过程如下。

（1）首先克隆 Gerrit 管理的版本库，使用 Gerrit 提供的运行于 29418 端口的 SSH 协议。

```
$ git clone ssh://localhost:29418/hello.git
Cloning into hello...
remote: Counting objects: 3, done
remote: Compressing objects: 100% (3/3)
Receiving objects: 100% (3/3), done.
```

（2）然后拷贝 Gerrit 服务器提供的 commit-msg 钩子脚本。

```
$ cd hello
$ scp -P 29418 -p localhost:/hooks/commit-msg .git/hooks/
```

（3）别忘了修改 Git 配置中提交者的邮件地址，以便和 Gerrit 中注册的地址保持一致。不使用 `--global` 参数调用 `git config` 可以只对本版本库的提交设定提交者邮件。

```
$ git config user.email jiangxin@moon.ossxp.com
```

（4）然后修改 readme.txt 文件并提交。注意提交的时候使用了“-s”参数，目的是在提交说明中加入“Signed-off-by:”标记，这在 Gerrit 提交中可能是必须的。

```
$ echo "gerrit review test" >> readme.txt
$ git commit -a -s -m "readme.txt hacked."
[master c65ab49] readme.txt hacked.
 1 files changed, 1 insertions(+), 0 deletions(-)
```

（5）查看一下提交日志，会看到其中有特殊的标签。

```
$ git log --pretty=full -1
commit c65ab490f6d3dc36429b8f1363b6191357202f2e
Author: Jiang Xin <jiangxin@moon.ossxp.com>
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Date:   Mon Nov 15 17:50:08 2010 +0800

    readme.txt hacked.

    Change-Id: Id7c9d88ebf5dac2d19a7e0896289de1ae6fb6a90
    Signed-off-by: Jiang Xin <jiangxin@moon.ossxp.com>
```

提交说明中出现了“Change-Id:”标签，这个标签是由钩子脚本“commit-msg”自动生成的。至于这个标签的含义，在前面 Gerrit 的实现原理中已经介绍过。

好了，准备把这个提交推送到服务器上吧。

## 32.10.2 开发者向审核服务器提交

由 Gerrit 控制的 Git 版本库不能直接提交，因为正确设置的 Gerrit 服务器，会拒绝用户直接向 refs/heads/* 推送。

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
nothing to commit (working directory clean)

$ git push
Counting objects: 5, done.
Writing objects: 100% (3/3), 332 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/hello.git
 ! [remote rejected] master -> master (prohibited by Gerrit)
error: failed to push some refs to 'ssh://localhost:29418/hello.git'
```

直接推送就会遇到“prohibited by Gerrit”的错误。

正确的做法是向特殊的引用推送，这样 Gerrit 会自动将新提交转换为评审任务。

```
$ git push origin HEAD:refs/for/master
Counting objects: 5, done.
Writing objects: 100% (3/3), 332 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/hello.git
 * [new branch]      HEAD -> refs/for/master
```

看到了吗，向 refs/for/master 推送成功。

## 32.10.3 审核评审任务

以 Dev1 用户登录 Gerrit 网站，点击“All”菜单下的“Open”标签，可以看到新提交到 Gerrit 的状态为 Open 的评审任务，如图 32-19。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

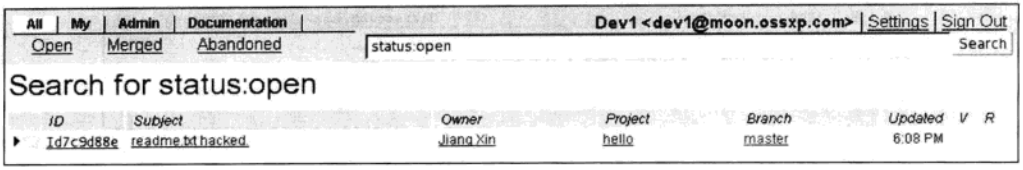




图 32-19 Gerrit 评审任务列表

点击该评审任务，显示关于此评审任务的详细信息，如图 32-20。

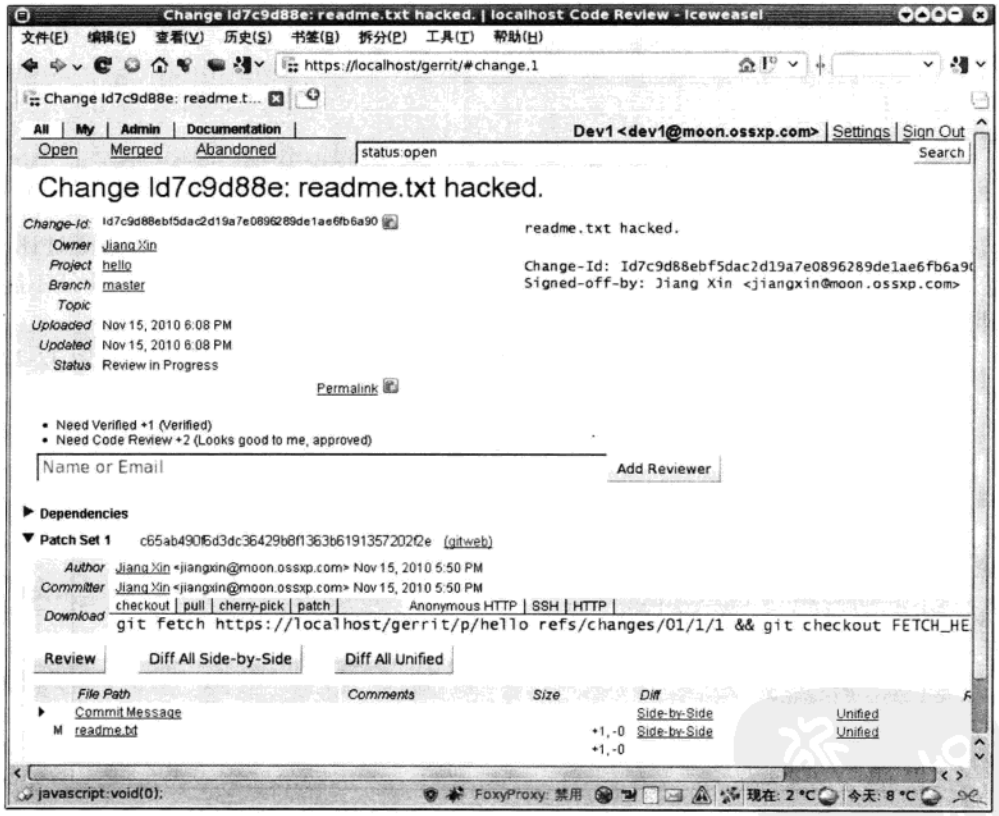



图 32-20 Gerrit 评审任务概述

从 URL 地址栏可以看到该评审任务的评审编号为 1。目前该评审任务有一个补丁集（Patch Set 1），可以点击“Diff All Side-by-Side”查看变更集，以决定该提交是否应该被接受。作为测试，先让此次提交通过代码审核，于是以 Dev1 用户身份点击“Review”按钮。点击“Review”按钮后，弹出代码评审对话框，如图 32-21。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

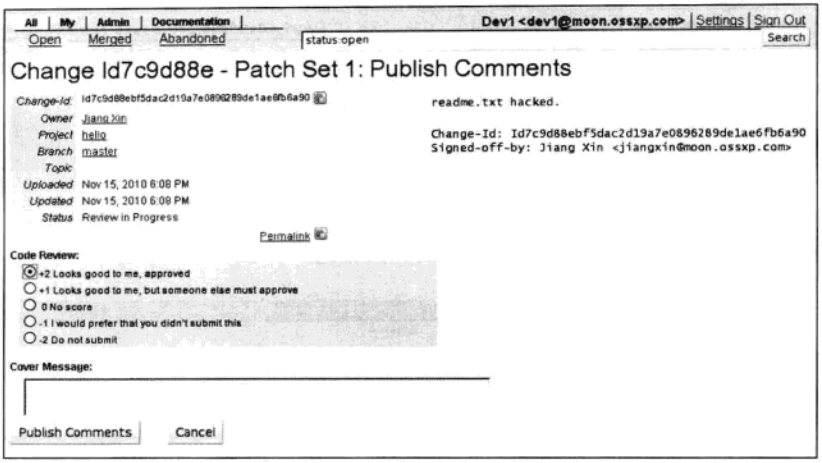




图 32-21 Gerrit 任务评审对话框

选择“+2: Looks good to me, approved.”，点击按钮“Publish Comments”以通过评审。注意因为没有给 Dev1 用户（Reviewer 用户组）授予 Submit 权限，因此此时 Dev1 还不能将此审核任务提交。

Dev1 用户做出通过评审的决定后，代码提交者 jiangxin 会收到一封邮件，如图 32-22。




图 32-22 Gerrit 通知邮件

## 32.10.4 评审任务没有通过测试

下面以 Dev2 账号登录 Gerrit，查看处于打开状态的评审任务，如图 32-23。会看到评审任务 1 的代码评审已经通过，但是尚未进行测试检查（Verify）。于是 Dev2 下载该补丁集，在本机进行测试。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

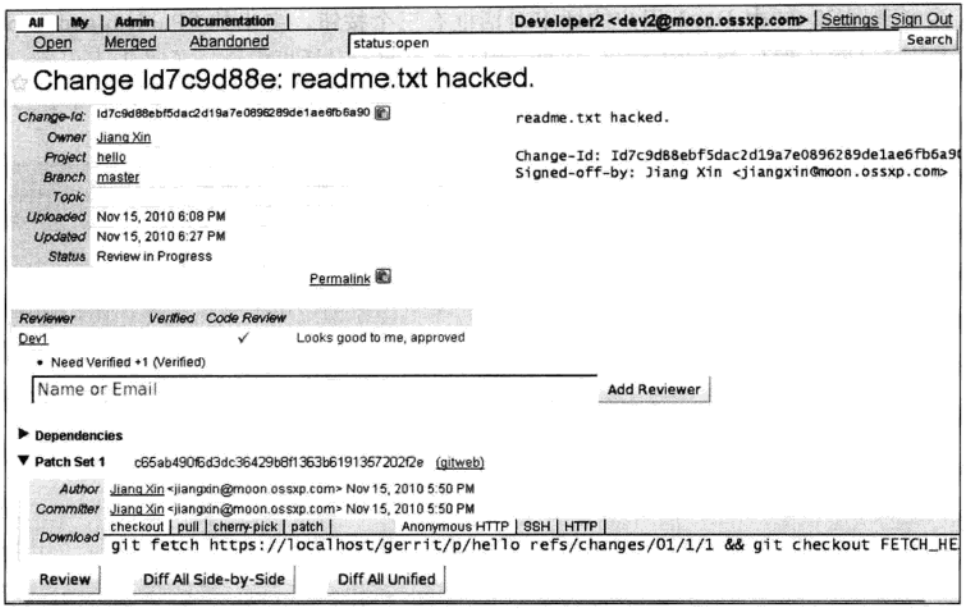




图 32-23 Gerrit 评审任务显示

假设测试没有通过，Dev2 用户点击该评审任务的“Review”按钮，重置该任务的评审状态，如图 32-24。

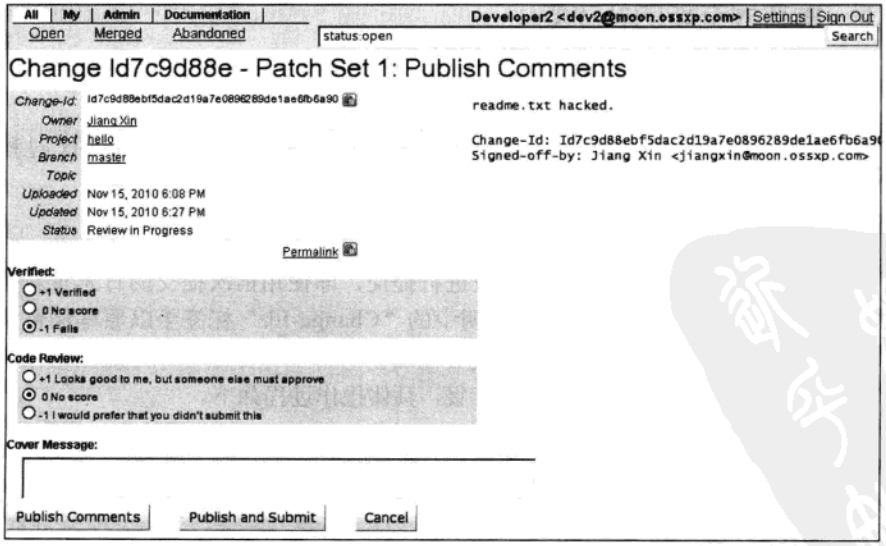



图 32-24 Gerrit 评审任务未通过



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


注意到图 32-24 中 Dev2 用户的评审对话框有三个按钮，多出的“Publish and Submit”按钮是因为 Dev2 拥有 Submit 授权。Dev2 用户在上面的对话框中选择了“-1: Fails”，点击“Publish Comments”按钮，该评审任务的评审记录被重置，同时提交者和其他评审参与者会收到通知邮件，如图 32-25。




图 32-25 Gerrit 通知邮件：评审未通过

## 32.10.5 重新提交新的补丁集

提交者收到代码被打回的邮件，一定很难过。不过这恰恰说明了这个软件过程已经相当的完善，现在发现问题总比在集成测试时甚至被客户发现要好得多吧。

根据评审者和检验者的提示，开发者对代码进行重新修改。下面的 Bugfix 过程仅仅是一个简单的示例，Bugfix 没有这么简单的，对吗？ ;-)

```
$ echo "fixed" >> readme.txt
```

重新修改后，需要使用 --amend 参数进行提交，即使用前次提交的日志重新提交，这一点非常重要。因为这样就会对原提交说明中的“Change-Id:”标签予以原样保留，当再将新提交推送到服务器时，Gerrit 不会为新提交生成新的评审任务编号，而是重用原有的任务编号，将新提交转化为老评审任务的新补丁集，具体操作过程如下。

（1）在执行 git commit --amend 时，可以修改提交说明，但是注意不要删除 Change-Id 标签，更不能修改它。

```
$ git add -u
$ git commit --amend

readme.txt hacked with bugfix.
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
Change-Id: Id7c9d88ebf5dac2d19a7e0896289de1ae6fb6a90
Signed-off-by: Jiang Xin <jiangxin@moon.ossxp.com>

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD^1 <file>..." to unstage)
#
# modified:   readme.txt
#
```

（2）提交成功后，执行 `git ls-remote` 命令会看到 Gerrit 维护的 Git 库中只有一个评审任务（编号 1），且该评审任务只有一个补丁集（Patch Set 1）。

```
$ git ls-remote origin
82c8fc3805d57cc0d17d58e1452e21428910fd2d        HEAD
c65ab490f6d3dc36429b8f1363b6191357202f2e        refs/changes/01/1/1
82c8fc3805d57cc0d17d58e1452e21428910fd2d        refs/heads/master
```

（3）把修改后的提交推送到 Gerrit 管理下的 Git 版本库中。注意依旧推送到 `refs/for/master` 引用中。

```
$ git push origin HEAD:refs/for/master
Counting objects: 5, done.
Writing objects: 100% (3/3), 353 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://localhost:29418/hello.git
 * [new branch]      HEAD -> refs/for/master
```

（4）推送成功后，再执行 `git ls-remote` 命令，会看到唯一的评审任务（编号 1）有了两个补丁集。

```
$ git ls-remote origin
82c8fc3805d57cc0d17d58e1452e21428910fd2d        HEAD
c65ab490f6d3dc36429b8f1363b6191357202f2e        refs/changes/01/1/1
1df9e8e05fcf97a46588488918a476abd1df8121        refs/changes/01/1/2
82c8fc3805d57cc0d17d58e1452e21428910fd2d        refs/heads/master
```

### 32.10.6 新修订集通过评审

当提交者重新针对评审任务进行提交时，原评审任务的审核者会收到通知邮件，提醒有新的补丁集等待评审，如图 32-26。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

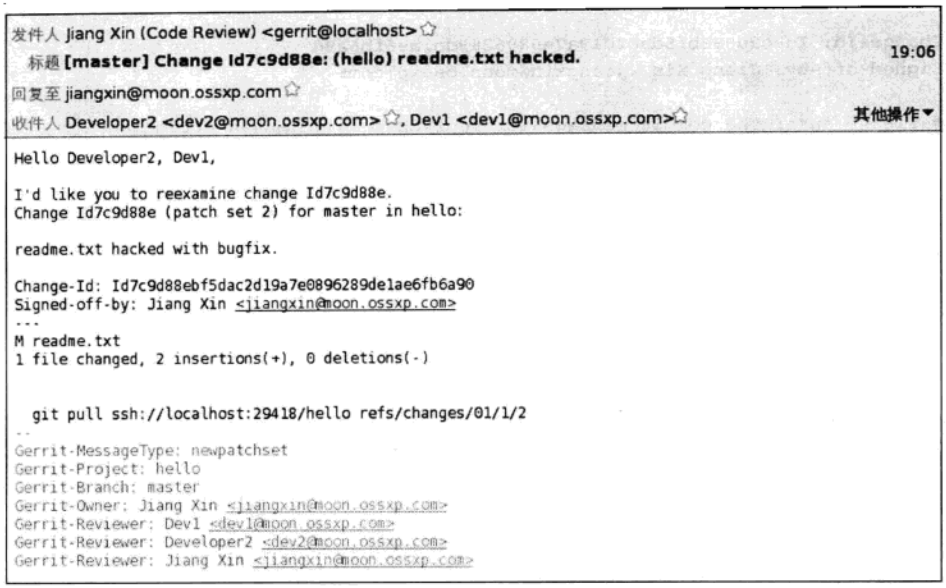




图 32-26 Gerrit 通知邮件：新补丁集

登录 Gerrit 的 Web 界面，可以看到评审任务 1 有了新的补丁集，如图 32-27 所示。

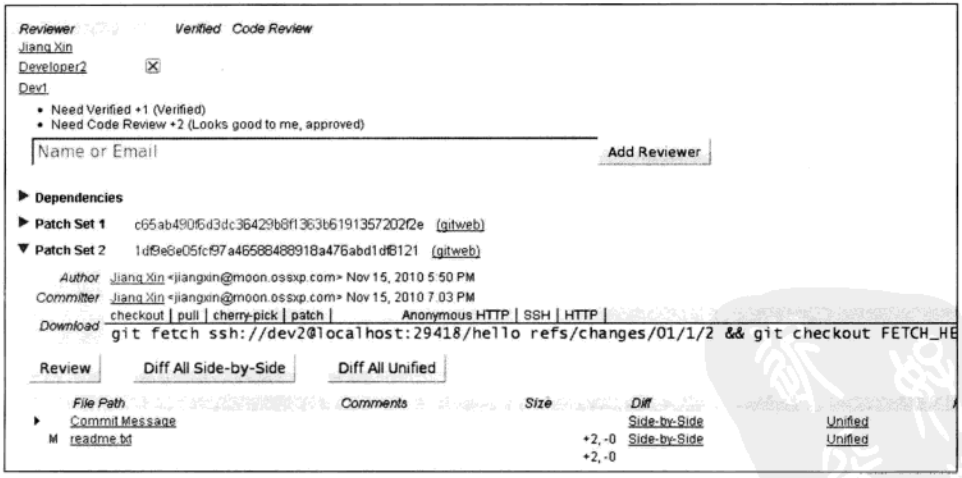



图 32-27 Gerrit 新补丁集显示

再经过代码审核和测试，这次 Dev2 用户决定让评审通过，点击了“Publish and Submit”按钮。Submit（提交）动作会将评审任务（refs/changes/01/1/2）合并到对应分支（master）中。图 32-28 显示的是通过评审完成合并的评审任务 1。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

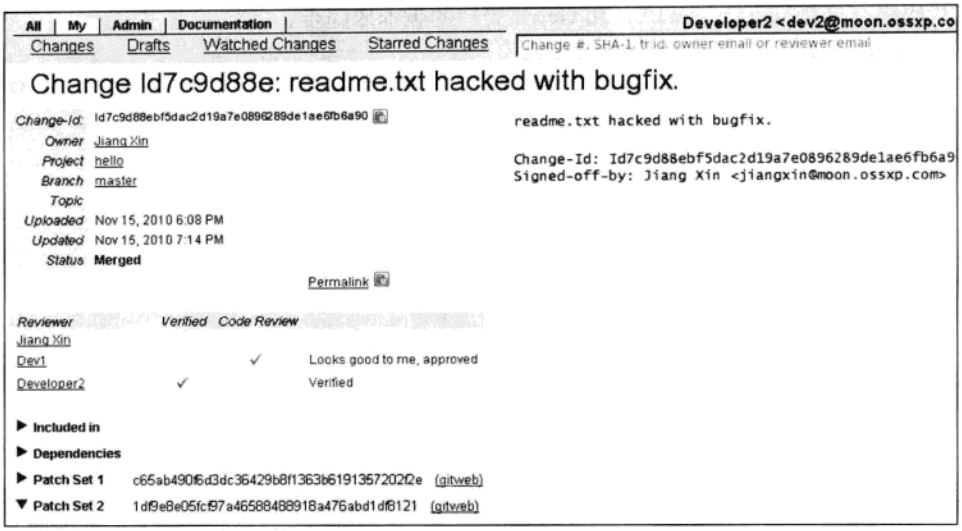




图 32-28 Gerrit 合并后的评审任务

## 32.10.7 从远程版本库更新

当 Dev1 和 Dev2 用户完成代码评审后，提交者会收到多封通知邮件。这其中最让人激动的就是代码被接受并合并到开发主线（master）中，如图 32-29 所示，这令开发者感到多么荣耀啊。




图 32-29 Gerrit 通知邮件：修订已合并



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


代码提交者执行 git pull，和 Gerrit 管理的版本库同步。

```
$ git ls-remote origin
1df9e8e05fcf97a46588488918a476abd1df8121        HEAD
c65ab490f6d3dc36429b8f1363b6191357202f2e        refs/changes/01/1/1
1df9e8e05fcf97a46588488918a476abd1df8121        refs/changes/01/1/2
1df9e8e05fcf97a46588488918a476abd1df8121        refs/heads/master

$ git pull
From ssh://localhost:29418/hello
   82c8fc3..1df9e8e  master     -> origin/master
Already up-to-date.
```

## 32.11 更多 Gerrit 参考

Gerrit 涉及的内容非常庞杂，还有诸如和 Gitweb、git-daemon 整合，Gerrit 界面定制等功能，恕不在此一一列举。可以直接参考 Gerrit 网站上的帮助[①]。

① *http://gerrit.googlecode.com/svn/documentation/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 33 章 Git 版本库托管

想不想在互联网上为自己的 Git 版本库建立一个克隆？这样就再也不必为数据的安全担忧（异地备份），还可以和他人共享数据、协同工作？但是这样做会不会很贵呢？比如要购买域名、虚拟主机、搭建 Git 服务器什么的？

实际上可以免费获得这种服务（Git 版本库托管服务）！ GitHub、Gitorious 等都可以免费提供这些服务。

## 33.1 Github

如果您是按部就班式地阅读本书，那么可能早就注意到本书的很多示例版本库都是放在 GitHub 上的。GitHub 提供了 Git 版本库托管服务，既包括收费的商业支持，也提供免费的服务，很多开源项目把版本库直接放在了 GitHub 上，如：jQuery、curl、Ruby on Rails 等。

注册一个 GitHub 账号非常简单，访问 GitHub 网站：*https://github.com/*，点击菜单中的“Pricing and Signup”就可以看到 GitHub 的服务列表（如图 33-1）。会看到其中有一个免费的服务：“Free for open source”，并且版本库托管的数量不受限制。当然免费的午餐是不管饱的，托管的空间只有 300MB，而且不能创建私有版本库。

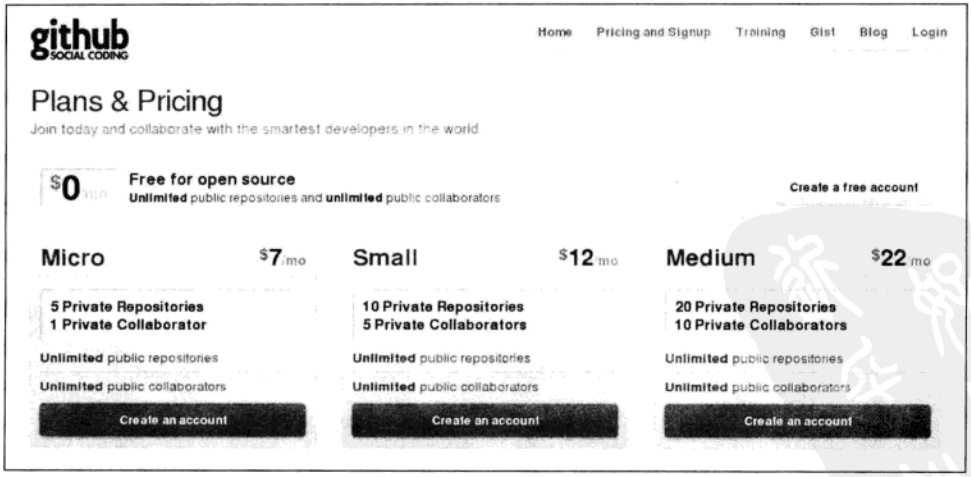



图 33-1 GitHub 服务价目表

点击按钮“Create a free account”，就可以创建一个免费的账号。GitHub 的用法和前面

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


介绍的 Gerrit Web 界面的用法很类似，一旦账号创建，应该马上为新建立的账号设置公钥，以便能够用 SSH 协议读写自己账号下创建的版本库，如图 33-2 所示。

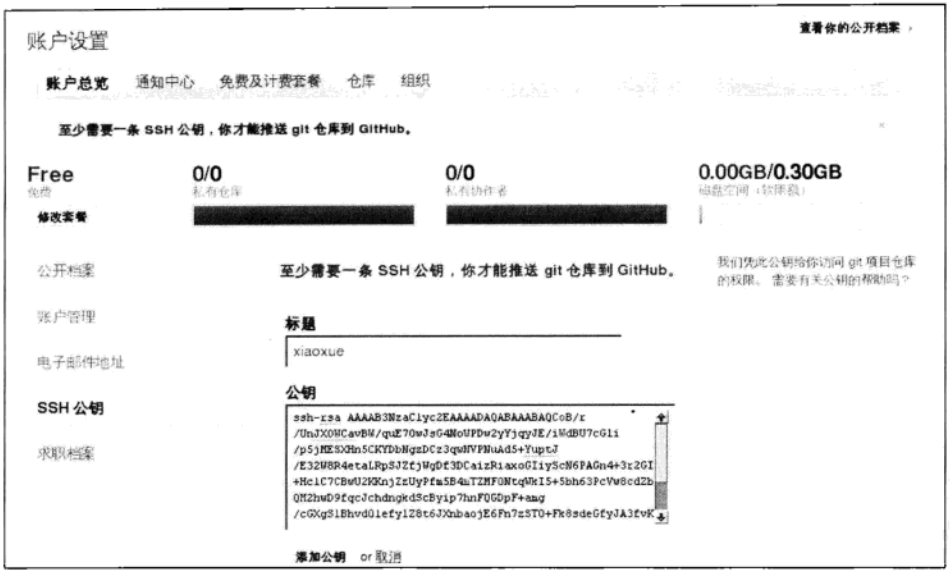



图 33-2 GitHub 上配置公钥

创建仓库的操作非常简单，首先点击菜单上的“主页”（Dashboard），再点击右侧边栏上的“新建仓库”按钮就可以创建新的版本库了，如图 33-3 所示。




图 33-3 GitHub 页面上的新建版本库按钮

新建版本库会浪费本来就不多的托管空间，从 GitHub 上已有的版本库派生（fork）一个克隆是一个好办法。首先通过 GitHub 搜索版本库名称，找到后点击“派生”按钮，就可以在自己的托管空间内建立相应版本库的克隆，如图 33-4 所示。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133





图 33-4 自 GitHub 上的版本库派生

版本库建立后就可以用 Git 命令访问 GitHub 上托管的版本库了。GitHub 提供三种协议可供访问，如图 33-5 所示。其中 SSH 协议和 HTTP 协议支持读写，Git-daemon 提供只读访问。对于自己的版本库当然选择支持读写的服务方式了，其中 SSH 协议是首选。

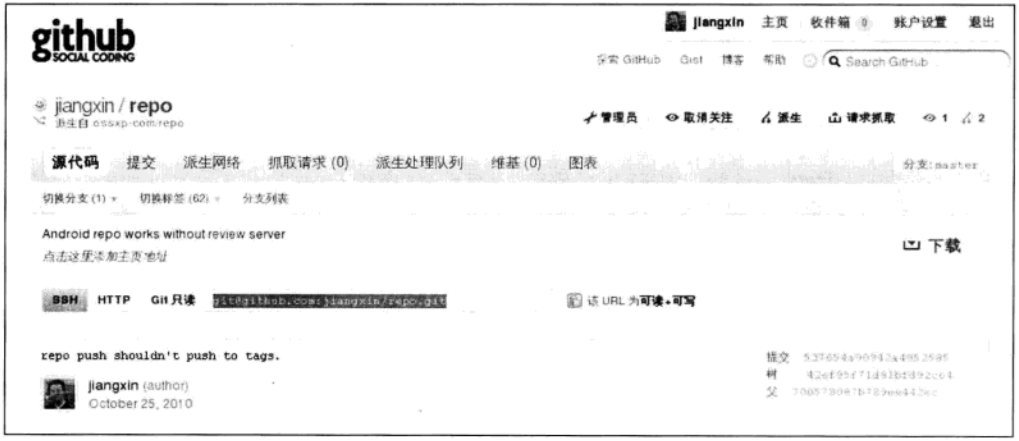



图 33-5 版本库访问 URL

## 33.2 Gitorious

Gitorious 是另外一个 Git 版本库托管提供商，网址为 http://gitorious.org/，如图 33-6 所示。最酷的是 Gitorious 本身的架站软件也是开源的，可以通过 Gitorious 上的 Gitorious 项目访问。如果你熟悉 Ruby on Rails，可以架设一个本地的 Gitorious 服务。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

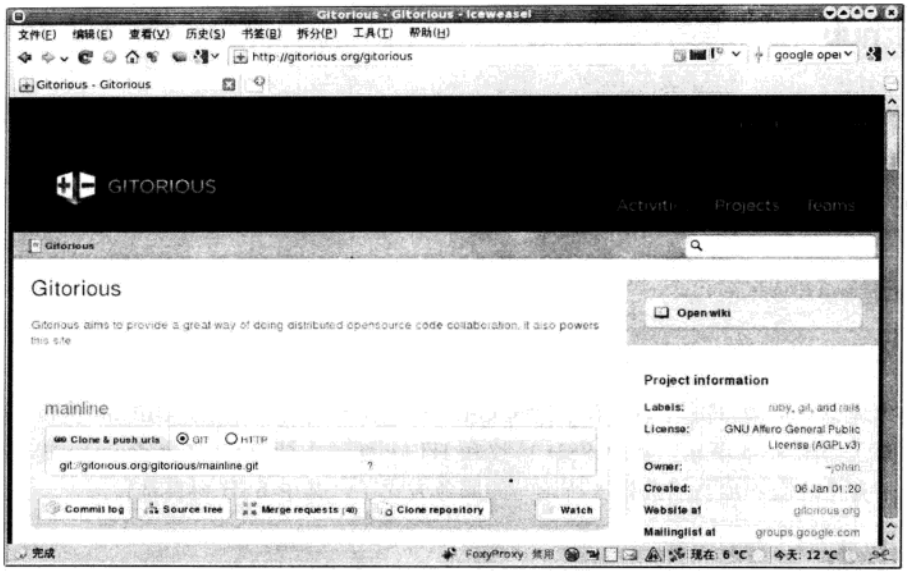




图 33-6 Gitorious 上的 Gitorious 项目

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第 6 篇
# 迁移到 Git

随着 Git 版本控制系统的成熟，越来越多的项目把版本控制系统迁移到了 Git 上。迁移大多是无损的，即迁移到 Git 后完美地保留了之前的变更历史、分支和里程碑。如果你正打算迁移版本控制系统，本篇介绍的版本库迁移方法和注意事项将会为你提供帮助。

本篇首先会介绍 CVS、Subversion、Mercurial 等几个著名的开源版本控制系统如何迁移到 Git 上。除这些之外的其他版本控制系统也许可以找到类似的迁移方案，或者可以针对 git-fast-import 通过编程的方式定制转换过程。在本篇的最后，还会介绍一个让 Git 版本库改头换面的更为强大的工具 git-filter-branch。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 34 章 CVS 版本库到 Git 的迁移

CVS 是最早被广泛使用的版本控制系统，因为其服务器端的存储结构非常简单，至今仍受到不少粉丝的钟爱。但是，它毕竟是几十年前的产物，因为设计上的原因，导致其缺乏现代版本控制系统的一些必备的功能，如：没有原子提交、分支管理不便（慢）、分支合并困难（因为合并过程缺乏跟踪）、不支持文件名 / 目录名的修改，等等。CVS 的很多用户都已经迁移到 Subversion 这一更好的集中式版本控制系统了。如果你还在使用 CVS，那么可以考虑直接迁移到 Git。

从 CVS 迁移到 Git 可以使用 cvs2svn 软件包中的 `cvs2git` 命令。为什么该项目的名称是 cvs2svn 而非 cvs2git 呢？这是因为该项目最早是为 CVS 版本库迁移到 Subversion 版本库服务的，只是最近才增加了 CVS 版本库转换为 Git 版本库的功能。cvs2svn 将 CVS 转换为 Subversion 版本库的过程一直以稳定著称，从 cvs2svn 2.1 版开始，增加了将 CVS 版本库转换为 Git 版本库的功能，这无疑让这个工具更具生命力，也减少了之前 CVS 到 Git 库的转换环节。在推出 cvs2git 功能之前，CVS 迁移到 Git 的路径通常是：先用 cvs2svn 将 CVS 版本库迁移到 Subversion 版本库，然后再用 git-svn 将 Subversion 版本库迁移到 Git。

关于 cvs2svn 及 cvs2git 的更多信息，可以参考下面的链接：

- *http://cvs2svn.tigris.org/cvs2svn.html*
- *http://cvs2svn.tigris.org/cvs2git.html*

## 34.1 安装 cvs2svn（含 cvs2git）

### 34.1.1 Linux 下 cvs2svn 的安装

大部分 Linux 发行版都提供 cvs2svn[①]的发布包，可以直接用平台自带的 cvs2svn 软件包。cvs2svn 从 2.1 版本开始引入了到 Git 库的转换，2.3.0 版本有了独立的 cvs2git 转换脚本，cvs2git 正在逐渐完善当中，因此请尽量选择最新版本的 cvs2svn 。

例如在 Debian 或 Ubuntu 下，可以通过下面的命令查看源里面的 cvs2svn 版本。

```
$ aptitude versions cvs2svn
p   2.1.1-1              stable                  990
p   2.3.0-2              testing,unstable        1001
```

可以看出 Debian 的 Testing 和 Sid 仓库中才有 2.3.0 版本的 cvs2svn。于是执行下面的命

① *http://cvs2svn.tigris.org/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


令安装在 Testing 版本才有的 2.3.0-2 版本的 cvs2svn：

```
$ sudo aptitude install cvs2svn/testing
```

如果对应的 Linux 发行版没有对应的版本也可以从源码开始安装。cvs2svn 的官方版本库在 http://cvs2svn.tigris.org/svn/cvs2svn/trunk 上，已经有人将 cvs2svn 项目转换为 Git 库。可以从 Git 库下载源码并安装 cvs2svn，具体操作步骤如下：

（1）下载 cvs2svn 源代码。

```
$ git clone git://repo.or.cz/cvs2svn.git
```

（2）进入 cvs2svn 源码目录，安装 cvs2svn。

```
$ cd cvs2svn
$ sudo make install
```

（3）安装用户手册。

```
$ sudo make man
```

cvs2svn 对其他软件包的依赖如下：

- Python 2.4 或以上版本，但是 Python 3.x 暂不支持。
- RCS：如果在转换中使用了 --use-rcs，就需要安装 RCS 软件包[①]。
- CVS：如果在转换中使用了 --use-cvs，就需要安装 CVS 软件包[②]。
- Git：1.5.4.4 或以上的版本。之前版本的 Git 的 git fast-import 命令有 Bug，加载 cvs2git 导出文件有问题。

### 34.1.2 Mac OS X 下 cvs2svn 的安装

Mac OS X 下可以使用 brew 安装 cvs2svn，具体步骤如下：

（1）Mac OS X 默认安装的 Python 缺少 cvs2svn 依赖的 gdbm 模组，先用 brew 来重新安装 Python。

```
$ brew install python
```

（2）安装 cvs2svn。

```
$ export PATH=/usr/local/bin:$PATH
$ brew install cvs2svn
```

① *http://www.cs.purdue.edu/homes/trinkle/RCS/*

② *http://www.nongnu.org/cvs/*



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 34.2 版本库转换的准备工作

### 34.2.1 版本库转换注意事项

转换CVS版本库时应该注意以下事项：

- 使用cvs2git转换CVS版本库必须在CVS的服务器端执行，即cvs2git必须能够通过文件系统直接访问CVS版本库中的“,v”文件。
- 在转换前，确保所有人的修改都已经提交到CVS版本库中。
- 在转换前，停止对CVS版本库的访问，以免在转换过程中有新提交写入。
- 在转换前，对原始版本库进行备份，以免误操作对版本库造成永久的破坏。
- 在转换完成后，永久停止CVS版本库的写入服务，可以仅开放只读服务。
  这是由于cvs2git是一次性操作，不能对CVS的后续提交执行增量式的到Git库的转换，因此当CVS版本库转换完毕后，须停止CVS服务。
- 先做小规模的试验性转换。
  转换CVS版本库切忌一上来就对整个版本库进行转换，等到发现日志乱码、文件名乱码、提交者ID不完全后重新转换会浪费大量的时间。
  应该先选择CVS版本库中的部分文件和目录作为样本，进行小规模的转换测试。
- 不要对包含CVSROOT目录的版本库的根进行操作，可以先对服务器目录的布局进行调整。如果转换直接针对包含CVSROOT目录的版本库根目录进行操作，会导致CVSROOT目录下的文件及更改历史也被纳入到Git版本库中，这是不需要的。

### 34.2.2 文件名乱码问题

CVS中保存的数据在服务器端直接和同名文件（文件多了一个“,v”后缀）相对应，如果转换的CVS版本库是从其他平台（如Windows）上拷贝过来的，就可能因为平台本身字符集不一致而导致中文文件名包含乱码，从而在CVS版本库的转换过程造成乱码。可以先对有问题的目录名和文件名进行重命名，将其转换为当前平台正确的编码。

### 34.2.3 提交说明乱码问题

CVS的提交说明中如果使用了中文，在转换后的版本库中可能显示为乱码，这是因为CVS的提交说明没有使用UTF-8字符集。前面提到过，最好先进行小规模的试验性转换，然后再对整个版本库进行正式的转换，就是为了防止在匆忙转换后发现提交说明中出现乱码。

下面来看一个使用了中文提交说明的CVS版本库。版本库是按如下方式部署的：CVSROOT为/cvshome/user，需要将之下的jiangxin/homepage/worldhello转换为一个Git版本库。先检查一下版本库中的数据，找出典型的目录用于转换。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


典型的数据是这样的：包含中文文件名，并且日志中包含中文。例如在版本库中，执行 CVS 查看日志命令，可以看到类似下面的输出。

```
RCS file: /cvshome/user/jiangxin/homepage/worldhello/archive/2003/.mhonarc.db,v
Working file: archive/2003/.mhonarc.db
head: 1.16
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 16;    selected revisions: 16
description:
----------------------------
revision 1.16
date: 2004-09-21 15:56:30 +0800;  author: jiangxin; state: Exp; lines:+3 -3; commitid: c2c414fdea20000;
<D0><U+07B8><C4><D3>?<FE><B5><D8><A3><BB>
<D0><U+07B8><C4><CB><D1><CB><F7><D2><FD><C7> 某
----------------------------
```

此版本库之前用 CVSNT 维护，默认字符集为 GBK，所以会在使用 UTF-8 字符集的操作系统上看到乱码。那么这里选取提交说明存在乱码的目录进行一次试验性的转换，具体操作过程如下。

（1）调用 cvs2git 执行转换，产生两个导出文件。这两个导出文件将作为 Git 版本库创建时的导入文件。命令行用了两个 --encoding 参数设置编码，会依次尝试将日志中的非 ASCII 字符转换为 UTF-8 字符。

```
$ cvs2git --blobfile git-blob.dat --dumpfile git-dump.dat \
  --encoding utf8 --encoding gbk --username cvs2git \
  /cvshome/user/jiangxin/homepage/worldhello/archive/2003/
```

（2）成功导出后，产生两个导出文件，一个保存各个文件的各个不同版本的数据内容，即在命令行指定的输出文件 git-blob.dat。另外一个文件是上面的命令行指定的 git-dump.dat，用于保存各个提交的相关信息（提交者、提交时间、提交日志等）。可以看出保存文件内容的导出文件（git-blob.dat）相对更大一些。

```
$ du -sh git*dat
9.8M    git-blob.dat
24K     git-dump.dat
```

（3）创建空的 Git 库，使用 Git 通用的数据迁移命令 git fast-import 将 cvs2git 的导出文件导入到版本库中。

```
$ mkdir test
$ cd test
$ git init
$ cat ../git-blob.dat ../git-dump.dat | git fast-import
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（4）检查导出结果。

```
$ git reset HEAD
$ git checkout.
$ git log -1
commit 8334587cb241076bcd2e710b321e8e16b5e46bba
Author: jiangxin <>
Date:   Tue Sep 21 07:56:31 2004 +0000

    修改邮件地址；
    修改搜索引擎；
```

很好，导出的Git库日志中，中文乱码问题已经解决。但是，提交日志中的Author对应的提交者不完整：缺乏邮件地址。这是因为CVS的提交者仅为用户登录ID，而Git的提交者信息还要包含邮件地址。cvs2git提供用户名映射的方法，不过不能使用命令行调用cvs2git，而是要通过加载配置文件来运行。因此正式进行的CVS到Git版本库转换要采用下面介绍的转换方法。

## 34.3 版本库转换

使用命令行参数调用cvs2git麻烦、可重用性差，而且可配置项有限。采用cvs2git配置文件模式运行不但能够简化cvs2git的命令行参数，而且还能够提供更多的、命令行无法提供的配置项，可以更精确地对CVS到Git版本库的转换进行定制。

### 34.3.1 配置文件解说

cvs2svn软件包提供了一个cvs2git的配置示例文件，见源码中的文件`cvs2git-example.options`[①]。

将该示例文件在本地复制一份对其进行更改。该文件是Python代码格式，以“#”（井号）开始的行是注释，不要随意更改文件缩进，因为缩进也是Python语法的一部分。可以考虑针对下列的选项进行定制。

（1）设置CVS版本库位置。

使用配置文件方式运行cvs2git，只能在配置文件中设置要转换的CVS版本库位置，而不能在命令行进行设置。具体来说，是在配置文件最后面的`run_options`的`set_project`方法中指定。

```
run_options.set_project(
    # CVS 版本库的位置（不是工作区，而是包含 ,v 文件的版本库）
    # 可以是版本库下的子目录。
    r'/cvshome/user/jiangxin/homepage/worldhello/archive/2003/',
```

① *http://repo.or.cz/w/cvs2svn.git/blob/HEAD:/cvs2git-example.options*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（2）设置导出文件的位置。

- 将 CVS 版本文件的内容导出至 blob 导出文件：cvs2svn-tmp/git-blob.dat。还设置了使用更稳定的“cvs”命令进行导出。

```
ctx.revision_collector = GitRevisionCollector(
    'cvs2svn-tmp/git-blob.dat',

    #RCSRevisionReader(co_executable=r'co'),
    CVSRevisionReader(cvs_executable=r'cvs'),
    )
```

- 另外一个导出文件的位置设定。默认位置：cvs2svn-tmp/git-dump.dat。

```
ctx.output_option = GitOutputOption(
    os.path.join(ctx.tmpdir, 'git-dump.dat'),

    # The blobs will be written via the revision recorder, so in
    # OutputPass we only have to emit references to the blob marks:
    GitRevisionMarkWriter(),

    # Optional map from CVS author names to git author names:
    author_transforms=author_transforms,
    )
```

（2）设置无提交用户信息时使用的用户名。这个用户名可以用接下来的用户映射转换为 Git 用户名。

```
ctx.username = 'cvs2svn'
```

（3）建立 CVS 用户和 Git 用户之间的映射。Git 用户名可以用 Python 的元组（tuple）语法 (name, email) 或字符串 u'name <email>'[①]来表示。

```
author_transforms={
    'jiangxin'  : ('Jiang Xin', 'jiangxin@ossxp.com'),
    'dev1'      : u'开发者 1 <dev1@ossxp.com>',

    'cvs2svn'   : 'cvs2svn <admin@example.com>',
    }
```

（4）字符集编码。即如何转换日志中的用户名、提交说明及文件名的编码。

对于可能在日志中出现的中文，必须做出与下面类似的设置。编码的顺序对输出会有影响，一般将 'utf8' 放在 'gbk' 之前，以保证当日志中同时出现两种编码时都能正常转换。（这是因为部分中文的 UTF8 编码在 GBK 中也存在着古怪的对应。）

```
ctx.cvs_author_decoder = CVSTextDecoder(
    [
```

① 字符串前面的字符 u 声明该字符串以 Unicode 格式保存。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        'utf8',
        'gbk',
        ],
    fallback_encoding='gbk'
    )

ctx.cvs_log_decoder = CVSTextDecoder(
    [
        'utf8',
        'gbk',
        ],
    fallback_encoding='gbk'
    )

ctx.cvs_filename_decoder = CVSTextDecoder(
    [
        'utf8',
        'gbk',
        ],
    #fallback_encoding='ascii'
    )
```

（5）是否忽略 .cvsignore 文件？默认保留 .cvsignore 文件。

无论选择保留或是不保留，最好在转换后手工进行 .cvsignore 到 .gitignore 的转换。因为 cvs2git 不能自动将 .cvsignore 文件转换为 .gitignore 文件。

```
ctx.keep_cvsignore = True
```

（6）对文件换行符等的处理。下面的配置原本是针对 CVS 到 Subversion 的属性转换，但是也会影响到 Git 转换时的换行符设置。维持默认值比较安全。

```
ctx.file_property_setters.extend([
    # 基于配置文件设置文件的 mime 类型
    #MimeMapper(r'/etc/mime.types', ignore_case=False),

    # 对于二进制文件（-kb 模式）不设置 svn:eol-style 属性（对于 Subversion 来说）
    CVSBinaryFileEOLStyleSetter(),

    # 如果文件是二进制，并且还没有设置 svn:mime-type，将其设置为
'application/octet-stream'。
    CVSBinaryFileDefaultMimeTypeSetter(),

    # 如果希望根据文件的 mime 类型来判断文件的换行符，打开下面的注释
    #EOLStyleFromMimeTypeSetter(),

    # 如果上面的规则没有为文件设置换行符类型，则为 svn:eol-style 设置默认类型
    #（二进制文件除外）
    # 默认把文件视为二进制，不为其设置换行符类型，这样最安全
    # 如果确认 CVS 的二进制文件都已经设置了 -kb 参数，或者使用上面的规则能够对
      文件类型做出正确判断，也可以使用下面的参数为非二进制文件设置默认换行符号
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    ## 'native': 服务器端文件的换行符保存为 LF，客户端根据需要自动转换
    ## 'CRLF':   服务器端文件的换行符保存为 CRLF，客户端亦为 CRLF
    ## 'CR':     服务器端文件的换行符保存为 CR，客户端亦为 CR
    ## 'LF':     服务器端文件的换行符保存为 LF，客户端亦为 LF
    DefaultEOLStyleSetter(None),

    # 如果文件没有设置 svn:eol-style ，也不为其设置 svn:keywords 属性
    SVNBinaryFileKeywordsPropertySetter(),

    # 如果没有设置 svn:keywords，基于文件的 CVS 模式进行设置。
    KeywordsPropertySetter(config.SVN_KEYWORDS_VALUE),

    # 设置文件的 svn:executable 属性，如果文件在 CVS 中标记为可执行文件。
    ExecutablePropertySetter(),
    ])
```

（7）是否只迁移主线，忽略分支和里程碑？

默认对所有分支和里程碑都进行转换。如果选择忽略分支和里程碑，则将 False 修改为 True。

```
ctx.trunk_only = False
```

（8）分支和里程碑的迁移及转换。

```
global_symbol_strategy_rules = [

    # 和正则表达式匹配的 CVS 标识，转换为 Git 的分支
    #ForceBranchRegexpStrategyRule(r'branch.*'),

    # 和正则表达式匹配的 CVS 标识，转换为 Git 的里程碑
    #ForceTagRegexpStrategyRule(r'tag.*'),

    # 忽略和正则表达式匹配的 CVS 标识，不进行（到 Git 分支 / 里程碑）转换
    #ExcludeRegexpStrategyRule(r'unknown-.*'),

    # 有歧义的 CVS 标识的处理选项
    # 默认根据使用频率自动确定转换为分支或里程碑
    HeuristicStrategyRule(),
    # 或者全部转换为分支
    #AllBranchRule(),
    # 或者全部转换为里程碑
    #AllTagRule(),

    ...

run_options.set_project(

    ...

    # A list of symbol transformations that can be used to rename
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    symbols in this project.
symbol_transforms=[
    #  是否需要重新命名里程碑？第一个参数用于匹配，第二个参数用于替换。
    #  RegexpSymbolTransform(r'release-(\d+)_(\d+)',r'release-\1.\2'),
    #  RegexpSymbolTransform(r'release-(\d+)_(\d+)_(\d+)',r'release-\1.\2.\3'),
```

### 34.3.2 运行 cvs2git 完成转换

参照上面的方法，从默认的 cvs2git 配置文件来进行定制，在本地创建一个文件（例如名为 cvs2git.options 的文件）。然后运行 cvs2git 完成版本库转换，具体操作步骤如下。

（1）使用 cvs2git 配置文件，命令行大大简化了。

```
$ cvs2git --options cvs2git.options
```

（2）成功导出后，产生两个导出文件，都保存在 cvs2git-tmp 目录中。

一个保存各个文件的各个不同版本的数据内容，即命令行指定的输出文件 git-blob.dat。另外一个文件是上面命令行指定的 git-dump.dat，用于保存各个提交的相关信息（提交者、提交时间、提交日志等）。

可以看出保存文件内容的导出文件相对更大一些。

```
$ du -sh cvs2svn-tmp/*
9.8M    cvs2svn-tmp/git-blob.dat
24K     cvs2svn-tmp/git-dump.dat
```

（3）创建空的 Git 库，使用 Git 通用的数据迁移命令 git-fast-import 将 cvs2git 的导出文件导入到版本库中。

```
$ mkdir test
$ cd test
$ git init
$ cat ../cvs2svn-tmp/git-blob.dat \
      ../cvs2svn-tmp/git-dump.dat | git fast-import
```

（4）检查导出结果。

```
$ git reset HEAD
$ git checkout.
$ git log -1
commit e3f12f57a77cbffcf62e19012507d041f1c9b03d
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Sep 21 07:56:31 2004 +0000

    修改邮件地址；
    修改搜索引擎；
```

可以看到，这一次的转换结果不但使得日志中的中文可以显示，而且提交者 ID 也转换成了 Git 的风格。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 34.4 迁移后的版本库检查

完成迁移还不能算是大功告成，还需要进行细致的检查。

### 1. 文件名和日志中的中文

如果转换过程参考了前面的步骤和注意事项，文件名和版本库提交日志中的中文就不应该出现乱码。

### 2. 图片文件被破坏

最典型的错误就是转换后部分图片被破坏从而无法显示。这是怎么造成的呢？

CVS 默认将提交的文件以文本方式添加，除非用户在添加文件时使用了 -kb 参数。用命令行提交的用户经常会忘记使用 -kb 参数，这就导致一些二进制文件（如图片文件）以文本文件的方式被添加到其中。文本文件在 CVS 检入和检出时会进行换行符转换，在服务器端换行符保存为 LF，在 Windows 上检出时为 CRLF。如果误以文本文件方式添加的图片中恰好出现了 CRLF，则在 Windows 上似乎没有问题（仍然是 CRLF），但是 CVS 库转换成 Git 库后，图片文件在 Windows 上再检出时，文件数据中原来的 CRLF 就被换成了 LF，导致文件被破坏。

出现这种情况是 CVS 版本库的使用和管理上出现了问题，应该在 CVS 版本库中对有问题的文件重新设置属性，标记为二进制文件。然后再进行 CVS 版本库到 Git 库的转换。

### 3. .cvsignore 文件的转换

CVS 版本库中可能存在 .cvsignore 文件用于设置文件忽略，相当于 Git 版本库中的 .gitignore。因为当前版本的 cvs2git 不能自动将 .cvsignore 转换为 .gitignore，这就需要在版本库迁移后手工完成。CVS 的 .cvsignore 文件只对目录内的文件有效，不会向下作用到子目录上，这一点和 Git 的 .gitignore 不同。还有 .cvsignore 文件每一行都用空格分割多个忽略，而 Git 的每个忽略单独为一行。

### 4. 迁移后的测试

一个简单的检查方法是，在同一台机器上分别用 CVS 和 Git 检出（或克隆），然后比较本地的差异。要在不同的系统上（Windows、Linux）分别进行测试。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 35 章　更多版本控制系统的迁移

## 35.1　SVN 版本库到 Git 的迁移

Subversion 版本库到 Git 版本库的转换，最好的方法就是 git-svn。而 git-svn 的使用方法在前面“第 26 章　Git 和 SVN 协同模型”一章中已经详细介绍过。本章的内容将不再对 git-svn 的用法做过多的重复，只在这里强调一下版本库迁移时的注意事项，相关 git-svn 的内容还请参照第 4 篇“第 26 章　Git 和 SVN 协同模型”。

在迁移之前要确认一个问题，Subversion 转换到 Git 库之后，Subversion 还能继续使用吗？意思是说还允许向 Subversion 提交吗？

如果回答是，那么直接查看“第 26 章 Git 和 SVN 协同模型”，用 Git 作为前端工具来操作 Subversion 版本库，而不要理会下面的内容。因为下面描述的迁移步骤针对的是 Subversion 到 Git 版本库的一次性的迁移。

### 1. 不在提交说明中出现 git-svn-id 标识

如果一次性、永久性地将 Subverison 迁移到 Git 库，可以选择“git-svn-id:”标识不在转换后的 Git 的提交日志中出现，这样根本看不出来转换后的 Git 库曾经用 Subversion 版本库维护过。

在 git-svn 的 clone 或 init 子命令行中使用参数：--no-metadata。Git 库的配置会自动将 svn-remote.noMetadata 配置为 1。之后执行 git svn fetch 时就不会在日志中产生“git-svn-id:”标识。

### 2. Subversion 用户名到 Git 用户名的映射

默认转换后 Git 库的提交者 ID 为如下格式：userid <userid@SVN-REPOS-UUID>，即在邮件地址的域名处以 SVN 版本库的 UUID 代替。可以在执行 git svn fetch 时，通过下面的参数提供一个映射文件完成 SVN 用户名到 Git 用户名的转换。

```
-A<filename>, --authors-file=<filename>
```

即用 -A 或 --authors-file 参数给出一个映射文件，这个文件帮助 git-svn 将 Subversion 用户名映射为 Git 用户名。此文件的每一行定义一个用户名映射，每一行的格式为：

```
loginname = User Name <user@example.com>
```

也可以通过下面的命令在 Git 库的 config 文件中设置，这样就不必在每次执行 git svn fetch 时都带上这个参数。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git config svn.authorsfile /path/to/authersfile
```

当设定了用户映射文件后，如果在执行 git svn fetch 时发现 SVN 的用户在该映射文件中没有定义，转换过程被中断。需要重新编辑用户映射文件，补充新的用户映射后，再重新执行 git-svn 命令。

### 3. 里程碑和分支的转换

使用默认的参数执行 SVN 到 Git 的转换时，SVN 的里程碑和分支转换到 Git 库的 refs/remotes 引用下。这会导致其他人从转换后的 Git 库克隆时，看不到 Subversion 原有的分支和里程碑。

当以默认的参数执行 git svn init 时，Git 的配置文件中会生成下面的配置：

```
[svn-remote "svn"]
        fetch = trunk:refs/remotes/trunk
        branches = branches/*:refs/remotes/*
        tags = tags/*:refs/remotes/tags/*
```

可以直接编辑 Git 配置文件，将其内容调整如下：

```
[svn-remote "svn"]
        fetch = trunk:refs/heads/master
        branches = branches/*:refs/heads/*
        tags = tags/*:refs/tags/*
```

之后，再执行 git svn fetch 后，就可以实现 SVN 的分支和里程碑正确地转换为 Git 库的里程碑。否则如果不对 Git 配置文件作出调整，就需要手动将 .git/refs/remotes/ 下的引用移动到 .git/refs/heads 及 .git/refs/tags 下。

### 4. 清除 git-svn 的中间文件

git-svn 的中间文件位于目录 .git/svn 下，删除此目录完成对 git-svn 转换数据库文件的清理。

## 35.2 Hg 版本库到 Git 的迁移

Mercurial（水银）是和 Git 同时代的、与之齐名的一款著名的分布式版本控制系统，也有相当多的使用者。就像水银又名汞一样，作为版本控制系统的 Mercurial 又称作 Hg（水银元素符号）。Hg 具有简单易用的优点，至少 Hg 按提交的顺序递增的数字编号让 Subversion 用户感到更为亲切。Hg 的开发语言除少部分因性能原因使用 C 语言外，大部分用 Python 语言开发完成，因而更易扩展，最终形成了 Hg 最具特色的插件系统。例如 MQ 就是 Hg 一个很有用的插件，通过 Quilt 式的补丁集实现对定制开发的特性分支的版本控制，当然 StGit 和 Topgit 也可以实现类似的功能。

但是 Hg 存在一些不足。例如服务器的存储效率不能和 Git 相比，服务器存储空间占用更大。Hg 还不支持真正的分支，所以不能向 git-svn 那样完整地对 Subversion 版本库进行转



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


换和互操作。Hg 的提交只能回退一次，要想多次回退和整理版本库需要用到 MQ 插件。作为定制开发的利器，Hg+MQ 不适合多人协作开发而 Git+Topgit 更为适合。

不论是何原因如果想从 Hg 迁移到 Git，用一个名为 fast-export 的转换工具就可以很方便地实现。fast-export 是一个用 Python 开发的命令行工具，可以将本地的 Hg 版本库迁移为 Git 版本库。其原理和 CVS 版本库迁移至 Git 时使用的 cvs2git 相仿，都是先从源版本库生成导出文件，再用 Git 的通用版本库转换工具 git-fast-import 导入到新建的 Git 版本库中。

安装 fast-export 非常简单，只要用 Git 克隆 fast-export 的版本库即可。

```
$ cd /path/to
$ git clone git://repo.or.cz/fast-export.git
```

完成克隆后，会看到 /path/to/fast-export 目录中有一个名为 hg-fast-import.sh 的脚本文件，该文件封装了对相应 Python 脚本的调用。使用该脚本可以实现 Hg 版本库到 Git 版本库的迁移。

下面就演示一下 Hg 版本库到 Git 版本库的转换，具体操作过程如下。

（1）要转换的 Hg 版本库位于路径 /path/to/hg/hello/.hg 下。

Hg 不支持真正的分支，而且版本库中可能存在尚未合并的多个头指针。检查一下不要存在具有相同分支名但尚未合并的多个头指针，否则转换会失败。下面显示的该 Hg 版本库中具有两个具名分支 r1.x 和 next，还有一个默认的未设置名称的头指针，因为分支名各不相同所以不会为转换过程造成麻烦。

```
$ hg heads
修改集：      7:afdd475caeee
分支：        r1.x
标签：        tip
父亲：        0:798a9568e10e
用户：        Jiang Xin <jiangxin@ossxp.com>
日期：        Fri Jan 14 17:01:47 2011 +0800
描述：
start new branch: r1.x


修改集：      6:7f5a46201dda
分支：        next
用户：        Jiang Xin <jiangxin@ossxp.com>
日期：        Fri Jan 14 17:01:04 2011 +0800
文件：        src/locale/zh_CN/LC_MESSAGES/helloworld.po
描述：
imported patch 100_locale_zh_cn.patch


修改集：      1:97f0a21021c6
用户：        Jiang Xin <worldhello.net AT gmail DOT com>
日期：        Sun Aug 23 23:53:05 2009 +0800
文件：        src/COPYRIGHT src/main.bak src/main.c
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
描述：
Fixed #6: import new upstream version hello-2.0.0
```

（2）初始化一个 Git 版本库，该版本库就是迁移的目标版本库。

```
$ mkdir -p /path/to/my/workspace/hello
$ cd /path/to/my/workspace/hello
$ git init
Initialized empty Git repository in /path/to/my/workspace/hello/.git/
```

（3）在刚刚完成初始化的 Git 工作区中调用 hg-fast-export.sh 脚本完成版本库转换。

```
$ /path/to/fast-export/hg-fast-export.sh -r /path/to/hg/hello
```

（4）转换完毕，执行 git branch 会看到 Hg 版本库中的具名分支都被转换为相应的分支，没有命名的默认头指针被转换为 master 分支。

```
$ git branch
* master
  next
  r1.x
```

在转换后的 Git 版本库目录中，保存了几个用于记录版本库转换进度的状态文件（.git/hg2git-*），当在 Git 工作区不带任何参数执行 hg-fast-export.sh 命令时，会继续进行增量式的转换，将 Hg 版本库中的新提交迁移到 Git 版本库中。

如果使用了多个不同的 Hg 克隆版本库进行分支管理，就需要对 Hg 版本库逐一进行转换，然后再对转换后的 Git 版本库进行合并。在合并 Git 版本库的时候可以参考下面的命令。

```
$ git remote add <name1> <path/to/repos/1>
$ git remote add <name2> <path/to/repos/2>
$ git remote update
$ git checkout -b <branch1> origin/<name1>/master
$ git checkout -b <branch2> origin/<name2>/master
```

## 35.3 通用版本库迁移

如果在前面的迁移方案中没有涉及您的版本控制工具，也不要紧，因为很可能通过搜索引擎就能找到一款合适的迁移工具。如果找不到相应的工具，可能是你使用的版本控制工具太冷门，或者是一款不提供迁移接口的商业版本控制工具[①]。这时你可以通过手工检入的方式，或者针对 Git 提供的版本库导入接口 git-fast-import 实现版本库导入。

手工检入的方式适合于只有少数几个提交，或者对大部分提交历史不关心而只需要对

① 实际上 Linux 版本控制系统迁移到 Git 时，最早也没有工具帮助迁移 Bitkeeper 中的历史代码。关于 Linux 如何利用嫁接（Grafts）实现新旧 Linux 版本控制系统的对接参见本书第 8 篇第 41 章“41.4 嫁接和替换”小节的相关内容。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


少数里程碑版本执行导入的版本库导入。这种版本库迁移方式非常简单，相当于在完成 Git 版本库初始化后，在工作区重复执行：工作区文件清理（rm -rf），文件复制，执行 git add -A 添加到暂存区，执行 git commit 提交。

但是如果需要将版本库完整的历史全部迁移到新的 Git 版本库中，手工检入的方法就不可取了，针对 git-fast-import 的编程将是一个好的方法。Git 提供了一个通用的版本库导入解决方案，即通过向命令 git fast-import 传递特定格式的字节流，就可以实现 Git 版本库的创建。工具 git-fast-import 的导入文件格式设计得相对比较简单，当理解了其格式约定后，可以相对容易地开发出针对特定版本库的迁移工具。

**1. 数据混杂在提交中的导入文件**

下面就是一个简单的导入文件，为说明方便在前面标注了行号，将这个文件保存为 /path/to/file/dump1.dat。

```
 1 commit refs/heads/master
 2 mark :1
 3 committer User1 <user1@ossxp.com> 1295312699 +0800
 4 data <<EOF
 5 My initial commit.
 6 EOF
 7 M 644 inline README
 8 data <<EOF
 9 Hello, world.
10 EOF
11 M 644 inline team/user1.txt
12 data <<EOF
13 I'm user1.
14 EOF
```

上面这段文字应该这样理解：

- 第 1 行以 commit 开头，标记一个提交的开始。该提交会创建（或更新）引用 refs/heads/master。
- 第 2 行以 mark 开头，是一个标记指令，将这个提交用“:1”标示以方便后面的提交参照。
- 第 3 行记录了这个提交的提交者是 User1，邮件地址为 <user1@ossxp.com>，提交时间则采用 Unix 时间格式。
- 第 4 ～ 6 行是该提交的提交说明，提交说明用 data 数据块的方式进行定义。
- 第 4 行在 data 语句后紧接着的 <<EOF 含义为 data 的内容到以 EOF 标记的行截止。这样的表示法称为“Here document”表示法[①]。
- 第 7 行以字母 M 开头，含义是修改（或新建）了一个文件，文件名为 README，而文

① *http://en.wikipedia.org/wiki/Here_document*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


件的内容以 inline 的方式提供。

- 第 8 ～ 10 行则是以内联（inline）数据块的方式提供 README 文件的内容。
- 第 11 行定义了该提交修改的第二个文件 team/user1.txt。该文件的内容也是以内联（inline）的方式给出。
- 第 12 ～ 14 行给出文件 team/user1.txt 的内容。

下面初始化一个新的版本库，并通过导入文件 /path/to/file/dump1.dat 的方式为版本库注入数据，操作步骤如下。

（1）初始化版本库。

```
$ mkdir -p /path/to/my/workspace/import
$ cd /path/to/my/workspace/import
$ git init
```

（2）调用 git fast-import 命令。

```
$ git fast-import < /path/to/file/dump1.dat
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:            5 (         0 duplicates                  )
      blobs  :            2 (         0 duplicates          0 deltas)
      trees  :            2 (         0 duplicates          0 deltas)
      commits:            1 (         0 duplicates          0 deltas)
      tags   :            0 (         0 duplicates          0 deltas)
Total branches:           1 (         1 loads     )
      marks:           1024 (         1 unique    )
      atoms:              3
Memory total:          2344 KiB
       pools:          2110 KiB
     objects:           234 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit      = 8589934592
pack_report: pack_used_ctr            =          1
pack_report: pack_mmap_calls          =          1
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =        323 /        323
---------------------------------------------------------------------
```

（3）看看提交日志。

```
$ git log --pretty=fuller --stat
commit 18f4310580ca915d7384b116fcb2e2ca0b833714
Author:     User1 <user1@ossxp.com>
AuthorDate: Tue Jan 18 09:04:59 2011 +0800
Commit:     User1 <user1@ossxp.com>
CommitDate: Tue Jan 18 09:04:59 2011 +0800
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    My initial commit.

 README          |    1 +
 team/user1.txt  |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
```

### 2. 在 blob 中定义数据并在提交中引用的导入文件

再来看一个导入文件。将下面的内容保存到文件 /path/to/file/dump2.dat 中。

```
 1 blob
 2 mark :2
 3 data 25
 4 Hello, world.
 5 Hi, user2.
 6 blob
 7 mark :3
 8 data <<EOF
 9 I'm user2.
10 EOF
11 commit refs/heads/master
12 mark :4
13 committer User2 <user2@ossxp.com> 1295312799 +0800
14 data <<EOF
15 User2's test commit.
16 EOF
17 from :1
18 M 644 :2 README
19 M 644 :3 team/user2.txt
```

上面的内容为了说明方便标注了行号，注意不要把行号也代入文件中。其中：

- 第 1 ～ 5 行定义了编号为“:2”的文件内容。该文件的内容共有 25 个字节，第 3 行开始的 data 文字块通过在后面跟上一个表示文件长度的十进制数字界定了内容的起止。
- 第 6 ～ 10 行定义了编号为“:3”的文件内容。第 8 行界定该文件内容使用了“here document”的语法，使用“here document”语法比较适合于文本内容，使用内容长度标示内容起止更适合于二进制文件。
- 第 11 行开始定义了一个新的提交。
- 第 12 行设定该提交的编号为“:4”。
- 第 17 行以 from 开头，定义了编号为“:1”的提交是该提交的父提交，即在 /path/to/file/dump1.dat 中定义的提交。
- 第 18 行和第 19 行设定了该提交更改的两个文件，这两个文件的内容不像之前的导出文件“dump1.dat”那样使用内联方式定义内容，而是采用引用方式引用前面定义的 blob 文字块作为文件的内容。

如果以增量方式导入 dump2.dat 会报错，因为在第 17 行引用的“:1”没有定义。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ git fast-import < /path/to/file/dump2.dat
fatal: mark :1 not declared
fast-import: dumping crash report to .git/fast_import_crash_21772
```

如果将文件 /path/to/file/dump2.dat 的第 17 行的引用修改为提交 ID，就可以增量导入了。不过，为了说明的方便，还是通过将两个导入文件一次性传递给 git fast-import 来创建一个新版本库，操作步骤如下。

（1）初始化版本库 import2。

```
$ mkdir -p /path/to/my/workspace/import2
$ cd /path/to/my/workspace/import2
$ git init
```

（2）调用 git fast-import 命令。

```
$ cat /path/to/file/dump1.dat \
      /path/to/file/dump2.dat | git fast-import
```

（3）导入之后的日志显示：

```
$ git log --graph --stat
* commit 73a6f2742f9da7c1b4bb8748e018a2becad39dd6
| Author: User2 <user2@ossxp.com>
| Date:   Tue Jan 18 09:06:39 2011 +0800
|
|     User2's test commit.
|
|  README          |    1 +
|  team/user2.txt  |    1 +
|  2 files changed, 2 insertions(+), 0 deletions(-)
|
* commit 18f4310580ca915d7384b116fcb2e2ca0b833714
  Author: User1 <user1@ossxp.com>
  Date:   Tue Jan 18 09:04:59 2011 +0800

      My initial commit.

   README          |    1 +
   team/user1.txt  |    1 +
   2 files changed, 2 insertions(+), 0 deletions(-)
```

### 3. 包含了合并提交及里程碑的导入文件

下面再来看一个导入文件，在这个导入文件中，包含了合并提交，以及创建里程碑。

```
1 blob
2 mark :5
3 data 25
4 Hello, world.
5 Hi, user1.
6 blob
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
 7 mark :6
 8 data 35
 9 Hello, world.
10 Hi, user1 and user2.
11 commit refs/heads/master
12 mark :7
13 committer User1 <user1@ossxp.com> 1295312899 +0800
14 data <<EOF
15 Say helo to user1.
16 EOF
17 from :1
18 M 644 :5 README
19 commit refs/heads/master
20 mark :8
21 committer User2 <user2@ossxp.com> 1295312900 +0800
22 data <<EOF
23 Say helo to both users.
24 EOF
25 from :4
26 merge :7
27 M 644 :6 README
28 tag refs/tags/v1.0
29 from :8
30 tagger Jiang Xin <jiangxin@ossxp.com> 1295312901 +0800
31 data <<EOF
32 Version v1.0
33 EOF
```

将这个文件保存到 /path/to/file/dump3.dat 中。下面就针对该文件内容进行简要的说明：

- 第 1 ～ 5 行和第 6 ～ 10 行定义了两个 blob 对象，代表了对 README 文件的两个不同的修改。
- 第 11 行开始定义了编号为“:7”的提交。从第 17 行可以看出该提交的父提交也是由 dump1.dat 导入的第一个提交。
- 第 19 行开始定义了编号为“:8”的提交。该提交为一个合并提交，除了在第 25 行设定了第一个父提交外，还由第 26 行给出了第二个父提交。
- 第 28 行开始定义了一个里程碑。里程碑的名字为 refs/tags/v1.0。第 29 行指定了该里程碑对应的提交。里程碑的说明由第 31 ～ 33 行的指令给出。

下面将之前的三个导入文件一次性传递给 git fast-import 来创建一个新版本库，操作步骤如下。

（1）初始化版本库 import3。

```
$ mkdir -p /path/to/my/workspace/import3
$ cd /path/to/my/workspace/import3
$ git init
```

（2）调用 git fast-import 命令。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cat /path/to/file/dump1.dat /path/to/file/dump2.dat \
      /path/to/file/dump3.dat | git fast-import
```

（3）查看创建的版本库的日志。

从日志中可以看出里程碑 v1.0 已经建立在最新的提交上了。

```
$ git log --oneline --graph --decorate
*   a47790e (HEAD, tag: refs/tags/v1.0, master) Say helo to both users.
|\
| * f486a44 Say helo to user1.
* | 73a6f27 User2's test commit.
|/
* 18f4310 My initial commit.
```

理解了 git-fast-import 的导入文件格式，针对特定的版本控制系统开发一个新的迁移工具就不是难事了。Hg 的迁移工具 fast-export 就是一个很好的参照。

## 35.4 Git 版本库整理

Git 提供了太多武器进行版本库的整理，可以将一个 Git 版本库改头换面成另外一个 Git 版本库。

- 使用交互式变基操作，将多个提交合并为一个（参见第 2 篇第 12 章“12.3.3 时间旅行三”小节）。
- 使用 StGit，合并提交及更改提交（参见第 3 篇第 20 章“20.3.1 StGit”小节）。
- 借助变基操作，抛弃部分历史提交（参见第 2 篇第 12 章“12.4 丢弃历史”小节）。
- 使用子树合并，将多个版本库整合在一起（参见第 4 篇“第 24 章　子树合并”）。
- 使用 git-subtree 插件，将版本库的一个目录拆分出来成为独立版本库的根目录（参见第 4 篇第 24 章“24.5.4 git subtree split”小节）。

但是有些版本库重整工作如果使用上面的工具会非常困难，而采用另外一个还没有被用到的 Git 命令 `git filter-branch` 却可以做到事半功倍。看看使用这个新工具来实现下面的这几个任务是多么的简单和优美。

（1）将版本库中某个文件彻底删除[①]。即凡是有该文件的提交都逐一做出修改，撤出该文件。

```
$ git filter-branch --tree-filter 'rm -f filename' -- --all
```

（2）更改历史提交中某一提交者的姓名及邮件地址。

```
$ git filter-branch --commit-filter '
      if [ "$GIT_AUTHOR_NAME" = "Xin Jiang" ]; then
          GIT_AUTHOR_NAME="Jiang Xin"
          GIT_AUTHOR_EMAIL="jiangxin@ossxp.com"
```

① 这里使用的命令并非最优实现，后面会介绍一个运行得更快的命令。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        GIT_COMMITTER_NAME="$GIT_AUTHOR_NAME"
        GIT_COMMITTER_EMAIL="$GIT_AUTHOR_EMAIL"
    fi
    git commit-tree "$@";
    ' HEAD
```

（3）为没有包含签名的历史提交添加签名。

```
$ git filter-branch -f --msg-filter '
    signed=false
    while read line; do
        if echo $line | grep -q Signed-off-by; then
            signed=true
        fi
        echo $line
    done
    if ! $signed; then
        echo ""
        echo "Signed-off-by: YourName <your@email.address>"
    fi
    ' HEAD
```

通过上面的例子，可以看出命令 git filter-branch 针对不同的过滤器提供可执行脚本，从不同的角度对 Git 版本库进行重构。该命令的用法如下：

```
git filter-branch [--env-filter <command>] [--tree-filter <command>]
        [--index-filter <command>] [--parent-filter <command>]
        [--msg-filter <command>] [--commit-filter <command>]
        [--tag-name-filter <command>] [--subdirectory-filter <directory>]
        [--prune-empty]
        [--original <namespace>] [-d <directory>] [-f | --force]
        [--] [<rev-list options>...]
```

这条命令异常复杂，但是大部分参数是用于提供不同接口的，因此还是比较好理解的。

- 该命令最后的 <rev-list> 参数提供要修改的版本范围，如果省略则相当于 HEAD 指向的当前分支。也可以使用 --all 来指代所有引用，但是要在 --all 和前面的参数间使用分隔符“--”。
- 运行 git filter-branch 命令改写分支之后，被改写的分支会在 refs/original 中对原始引用做备份。对于在 refs/original 中已有备份的，该命令拒绝执行，除非使用 -f 或 --force 参数。
- 其他的后面可带命令脚本 <command> 的参数（如 --env-filter <command>），为 git filter-branch 命令提供相应的编程接口，从不同的角度实现对 Git 版本库的过滤。下面针对各个过滤器分别进行介绍。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 35.4.1 环境变量过滤器

参数 --env-filter 用于设置一个环境变量过滤器。该过滤器用于修改环境变量，对特定的环境变量的修改会改变提交。下面的示例可用于修改作者 / 提交者的邮件地址[①]。

```
$ git filter-branch --env-filter '
    an="$GIT_AUTHOR_NAME"
    am="$GIT_AUTHOR_EMAIL"
    cn="$GIT_COMMITTER_NAME"
    cm="$GIT_COMMITTER_EMAIL"

    if [ "$cn" = "Kanwei Li" ]; then
      cm="kanwei@gmail.com"
    fi
    if [ "$an" = "Kanwei Li" ]; then
      am="kanwei@gmail.com"
    fi

    export GIT_AUTHOR_EMAIL=$am
    export GIT_COMMITTER_EMAIL=$cm
    '
```

这个示例和本节一开始介绍的更改作者 / 提交者信息的示例功能相同，但是使用了不同的过滤器，你可以根据喜好自由选择。

### 35.4.2 树过滤器

参数 --tree-filter 用于设置树过滤器。树过滤器会将每个提交检出到特定的目录中（.git-rewrite/ 目录，或者用 -d 参数指定的目录），针对检出目录中文件的修改、添加、删除会改变提交。注意此过滤器忽略 .gitignore，因此对检出目录的任何修改都会记录在新的提交中。之前介绍的文件删除就是一例，再比如对文件名的修改：

```
$ git filter-branch --tree-filter '
      [ -f oldfile ] && mv oldfile newfile || true
      ' -- --all
```

### 35.4.3 暂存区过滤器

树过滤器因为要将每个提交检出，因此非常费时，而参数 --index-filter 给出的暂存区过滤器则没有这个缺点。如果将之前使用树过滤器删除文件的操作换成用暂存区过滤器来实现，将会运行得更快。

```
$ git filter-branch --index-filter '
      git rm --cached --ignore-unmatch filename
      ' -- --all
```

① *http://kanwei.com/code/2009/03/29/fixing-git-email.html*



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


其中参数 --ignore-unmatch 让 git rm 命令不至于因为暂存区中不存在 filename 文件而失败。

### 35.4.4 父节点过滤器

参数 --parent-filter 用于设置父节点过滤器，该过滤器用于修改提交的父节点。提交原始的父节点通过标准输入传入脚本，而脚本的输出将作为提交的新的父节点。父节点参数的格式为：如果没有父节点（初始提交）则为空；如果有一个父节点，参数为“-p parent”；如果是合并提交，则有多个父节点，参数为“-p parent1 -p parent2 -p parent3 ...”。

下面的命令将当前分支的初始提交嫁接到 <graft-id> 所指向的提交上。

```
$ git filter-branch --parent-filter 'sed "s/^\$/-p <graft-id>/"' HEAD
```

如果不是将初始提交（没有父提交）而是任意的一个提交嫁接到另外的提交上，可以通过 GIT_COMMIT 环境变量对提交进行判断，更改其父节点以实现嫁接。

```
$ git filter-branch --parent-filter \
      'test $GIT_COMMIT = <commit-id> && \
      echo "-p <graft-id>" || cat
      ' HEAD
```

关于嫁接，Git 可以通过配置文件 .git/info/grafts 来实现[①]，而 git filter-branch 命令可以基于该配置文件对版本库实现永久性的更改。

```
$ echo "$commit-id $graft-id" >> .git/info/grafts
$ git filter-branch $graft-id..HEAD
```

### 35.4.5 提交说明过滤器

参数 --msg-filter 用于设置提交说明过滤器。该过滤器用于改写提交说明。原始的提交说明作为标准输入传入脚本，而脚本的输出则作为新的提交说明。

例如，使用 git-svn 命令从 Subversion 迁移过来的 Git 版本库，默认情况下在提交说明中包含 git-svn-id: 字样的说明，如果需要将其清除，可以不必重新迁移，而是使用下面的命令重写提交说明。

```
$ git filter-branch --msg-filter 'sed -e "/^git-svn-id:/d"' -- --all
```

再如，为最新的 10 个提交添加“Acked-by:”格式的签名。

```
$ git filter-branch --msg-filter '
      cat &&
      echo "Acked-by: Bugs Bunny <bunny@bugzilla.org>"
```

① 参见第 8 篇第 41 章“41.4.1 提交嫁接”一节。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
' HEAD~10..HEAD
```

### 35.4.6 提交过滤器

参数 --commit-filter 用于设置提交过滤器。提交过滤器所给出的脚本，在版本库重整过程的每次提交时运行，取代要默认执行的 git commit-tree 命令。不过一般情况下会在脚本中调用 git commit-tree 命令。传递给脚本的参数格式为“<TREE_ID> [(-p <PARENT_COMMIT_ID>)...]”，提交日志以标准输入的方式传递给脚本。脚本的输出是新提交的提交 ID。作为扩展，如果脚本输出了多个提交 ID，则这些提交 ID 作为子提交的多个父节点。

使用下面的命令，可以过滤掉空提交（合并提交除外）。

```
$ git filter-branch --commit-filter '
      git_commit_non_empty_tree "$@"'
```

函数 git_commit_non_empty_tree 是在脚本 git-filter-branch 中已经定义过的函数。可以打开文件 $(git --exec-path)/git-filter-branch 查看。

```
# if you run 'git_commit_non_empty_tree "$@"' in a commit filter,
# it will skip commits that leave the tree untouched, commit the other.
git_commit_non_empty_tree()
{
  if test $# = 3 && test "$1" = $(git rev-parse "$3^{tree}"); then
    map "$3"
  else
    git commit-tree "$@"
  fi
}
```

如果只想跳过某个用户（如 badboy）的提交，而无论该提交是否为空，可以使用下面的命令：

```
$ git filter-branch --commit-filter '
      if [ "$GIT_AUTHOR_NAME" = "badboy" ];
      then
              skip_commit "$@";
      else
              git commit-tree "$@";
      fi' HEAD
```

其中，函数 skip_commit 也是在脚本 git-filter-branch 中已经定义好了的。该函数的作用就是处理传递给提交过滤器脚本的参数“<tree_id> -p parent1 -p parent2 ...”，形成“parent1 parent2”的输出。参见 Git 命令脚本 $(git --exec-path)/git-filter-branch 中相关的函数。

```
# if you run 'skip_commit "$@"' in a commit filter, it will print the (mapped)
    parents, effectively skipping the commit.
skip_commit()
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
{
  shift;
  while [ -n "$1" ];
  do
    shift;
    map "$1";
    shift;
  done;
}
```

### 35.4.7 里程碑名字过滤器

参数 --tag-name-filter 用于设置里程碑名字过滤器。该过滤器也是经常要用到的过滤器。上面介绍的各个过滤器都有可能改变提交 ID，如果在原有的提交 ID 上建有里程碑，可能会随之更新，但是会产生大量的警告日志，提示要使用里程碑过滤器。里程碑过滤器脚本以原始里程碑名称作为标准输入，并把新里程碑名称作为标准输出。如果不打算变更里程碑的名称，而只是希望里程碑随提交而更新，可以在脚本中使用 cat 命令。例如，下面的命令中同时使用里程碑名字过滤器和目录树过滤器。

```
$ git filter-branch --tree-filter '
      [ -f oldfile ] && mv oldfile newfile || true
      ' -- tag-name-filter 'cat' -- --all
```

在前面的里程碑一章中曾经提到过 git branch 命令没有提供里程碑重命名的功能，而使用里程碑名字过滤器可以实现里程碑的重命名。下面的的示例会修改里程碑的名字，将前缀为“old-prefix”的里程碑改名为前缀为“new-prefix”的里程碑。

```
$ git filter-branch --tag-name-filter '
      oldtag="cat"
      newtag=${oldtag#old-prefix}
      if [ "$oldtag" != "$newtag" ]; then
          newtag="new-prefix$newtag"
      fi
      echo $newtag
      '
```

注意签名里程碑重建后，因为签名不可能保持，所以新里程碑会丢弃签名，成为一个普通的包含说明的里程碑。

### 35.4.8 子目录过滤器

参数 --subdirectory-filter 用于设置子目录过滤器。子目录过滤器可以将版本库的一个子目录提取为一个新版本库，并将该子目录作为版本库的根目录。例如从 Subversion 转换到 Git 版本库会因为参数使用不当，将原 Subversion 的主线转换为 Git 版本库的一个目录 trunk。可以使用 git filter-branch 命令的子目录过滤器将 trunk 提取为版本库的根。

```
$ git filter-branch --subdirectory-filter trunk HEAD
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 7 篇
# Git 的其他应用

Git 的强大和别具一格源自于它在一开始就没有按照版本控制系统的思路进行设计。根据 Linus Torvalds 自己的说法："我真的是从一个文件系统开发者所要面对的问题的角度出发对 Git 进行设计的（嗨，内核是我开发的），并且我真的对于建立一个传统的 SCM 系统没有一点兴趣。[①]" Git 最初仅仅是一个可对内容进行追踪、可版本管理的另类的文件系统，在整个社区的努力下，Git 终于成为一个成功的现代的版本控制系统了，而基于 Git 的其他应用才刚刚开始。

维基是使用易于理解、"所见即所得"的文本来编辑网页，实现基于 Web 的协同著作工具，又称为"Web 的版本控制"。在名为 MZ Linux 的维基网站上[②]可以看到一份用 Git 作为后端实现的维基列表（大部分是技术上的试验）。

SpaghettiFS 项目[③]尝试用 Git 作为数据存储后端，提供了一个用户空间的文件系统（FUSE，Filesystem in Userspace）。而另外的一些项目如 gitfs[④]可以直接把 Git 版本库挂载为文件系统。

下面的章节通过几个典型的应用来介绍 Git 在版本控制领域之外的应用。让我们一起来领略 Git 的神奇吧。

① *http://kerneltrap.org/node/4982*
② *http://www.mzlinux.org/node/116*
③ *https://github.com/alex-morega/SpaghettiFS*
④ *http://code.google.com/p/gitfs/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 36 章 etckeeper

Linux / Unix 的用户对 /etc 目录是再熟悉不过了，这个最重要的目录中保存了大部分软件的配置信息，借以实现对软件的配置乃至对整个系统的启动过程进行控制。对于 Windows 用户来说，可以把 /etc 目录视为 Windows 中的注册表，只不过是文件化了，易管理了。

这么重要的 /etc 目录，如果其中的文件被错误编辑或删除，将会损失惨重。有一个名为 etckeeper[①]的项目借用分布式版本控制工具（如：Git、Mercurial、Bazaar、Darcs），可以帮助实现 /etc 目录的持续备份。

那么 etckeeper 是如何实现的呢？下面就以 Git 作为 etckeeper 的后端为例进行说明，其他的分布式版本控制系统大同小异。

- 将 /etc 目录 Git 化。于目录 /etc/.git 中创建 Git 库，将 /etc 目录作为工作区。
- 与系统的包管理器（如 Debian/Ubuntu 的 apt、Redhat 上的 yum 等）整合。一旦有软件包安装或删除，就对 /etc 目录下的改动执行提交操作。
- 除了能够记录 /etc 目录中的文件内容外，还可以记录文件属性等元信息。因为 /etc 目录下的文件的权限设置往往是非常重要和致命的。
- 因为 /etc 目录已经成了一个 Git 版本库，可以用 Git 命令对 /etc 下的文件进行操作：查看历史，回退到历史版本，等等。
- 也可以将 /etc 克隆到另外的主机中，实现双机备份。

## 36.1 安装 etckeeper

安装 etckeeper 非常简单，因为 etckeeper 在主流的 Linux 发行版中都有对应的安装包。使用相应 Linux 平台的包管理器（apt、yum）即可安装。

在 Debian / Ubuntu 上安装 etckeeper，如下：

```
$ sudo aptitude install etckeeper
```

安装 etckeeper 软件包，还会自动安装一个分布式版本控制系统工具，除非已经安装过了。这是因为 etckeeper 需要使用一个分布式版本控制系统作为存储管理后端。在 Debian / Ubuntu 上会依据下面的优先级进行安装：Git > Mercurial > Bazaar > Darcs。

在 Debian / Ubuntu 上，使用 dpkg -s 命令查看 etckeeper 的软件包依赖，就会看到这个优先级。

① *http://kitenet.net/~joey/code/etckeeper/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ dpkg -s etckeeper | grep "^Depends"
Depends: git-core (>= 1:1.5.4) | git (>= 1:1.7) | mercurial | bzr (>= 1.4~) |
darcs, debconf (>= 0.5) | debconf-2.0
```

## 36.2 配置 etckeeper

配置 etckeeper 首先要选择好一款分布式版本库控制工具，如 Git，然后用相应的版本控制工具初始化 /etc 目录，并做一次提交，具体操作过程如下。

（1）编辑配置文件 /etc/etckeeper/etckeeper.conf。

只要有下面一条配置就够了。告诉 etckeeper 使用 Git 作为数据管理后端。

```
VCS="git"
```

（2）初始化 /etc 目录。即将其 Git 化。执行下面的命令（需要以 root 用户的身份执行），会将 /etc 目录 Git 化。整个过程可能会比较慢，因为要对 /etc 下的文件执行 git add，文件又太多，所以会慢一些。

```
$ sudo etckeeper init
```

（3）执行第一次提交。注意使用 etckeeper 命令而非 Git 命令进行提交。

```
$ sudo etckeeper commit "this is the first etckeeper commit..."
```

这个过程也会比较慢，主要是因为 etckeeper 要扫描 /etc 下非 root 用户的文件及特殊权限的文件并进行记录。别忘了 Git 本身并不能记录文件属主及文件权限等信息。

## 36.3 使用 etckeeper

实际上由于 etckeeper 已经和系统的包管理工具（如 Debian/Ubuntu 的 apt，Redhat 上的 yum 等）进行了整合，所以 etckeeper 可以免维护运行。即一旦有软件包安装或删除，对 /etc 目录下的改动会自动执行提交操作。

当然也可以随时以 root 用户的身份执行下面的命令对 /etc 目录的改动进行手动提交。

```
$ sudo etckeeper commit
```

剩下的工作就交给 Git 了。可以在 /etc 目录中执行 git log、git show 等操作。但要注意以 root 用户的身份运行，因为 /etc/.git 目录的权限不允许普通用户操作。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 37 章 Gistore

当了解了 etckeeper 之后，您可能会有如我一样的疑问："有没有像 etckeeper 一样的工具，但是能备份任意的文件和目录呢？"

我在 Google 上搜索类似的工具无果，终于决定动手开发一个，因为无论是我还是我的客户，都需要一个更好用的备份工具。这就是 Gistore。

```
Gistore = Git + Store
```

2010 年 1 月，我在公司的博客上发表了 Gistore 0.1 版本的消息，参见：*http://blog.ossxp.com/2010/01/406/* 。并将 Gistore 的源代码托管在了 Github 上，参见：*http://github.com/ossxp-com/gistore*。

Gistore 的出现是受到了 etckeeper 的启发，通过 Gistore 用户可以把系统中任何目录的数据纳入到备份中，定制非常简单和方便。Gistore 的特点如下：

- 使用 Git 作为数据后端。数据回复和历史查看等均使用熟悉的 Git 命令。
- 每次备份即为一次 Git 提交，支持文件的添加 / 删除 / 修改 / 重命名等。
- 每次备份的日志自动生成，内容为此次修改的摘要信息。
- 支持备份回滚，可以设定保存备份历史的天数，让备份的空间占用维持在一个相对稳定的水平上。
- 支持跨卷备份。备份的数据源可以来自任何卷 / 目录或文件。
- 备份源如果已经 Git 化，也能够备份。例如 `/etc` 目录因为 etckeeper 被 Git 化，仍然可以对其用 Gistore 进行备份。
- 多机异地备份非常简单，使用 Git 克隆即可解决。可以采用 Git 协议、HTTP，或者更为安全的 SSH 协议。

说明：Gistore 目前只能运行在 Linux、Mac OS X 等类 Unix 操作系统上，因为在备份中使用了 mount、umount 命令和 / 或 FUSE 相关命令。

## 37.1 Gistore 的安装

### 37.1.1 软件依赖

Gistore 运行时需要将备份项逐一挂载到备份工作区中，因此需要安装相应的挂载工具。

- 如果在 Linux 上以普通用户的身份运行 Gistore，或者在 Mac OS X 上执行，则需要安

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


装 bindfs[①]，以便能够将备份目录挂载到 Gistore 工作区中。

- 如果在 Linux 上以管理员的身份运行，则可以不安装 bindfs，因为 Linux 下的 mount --rbind 命令可以实现备份目录到 Gistore 工作区的挂载。
- 如果以普通用户的身份执行，当运行挂载工具遇到授权问题时，如果安装并且正确配置了 sudo 命令，则会自动调用 sudo 命令执行。

### 37.1.2 从源码安装 Gistore

从源代码安装和运行 Gistore，可以确保安装的是最新的版本，具体操作步骤如下。

（1）先用 Git 从 Github 上克隆代码库。

```
$ git clone git://github.com/ossxp-com/gistore.git
```

（2）可以直接在克隆出的源码中运行 Gistore。

```
$ cd gistore
$ ./gistore --help
```

（3）也可以执行 setup.py 脚本，以 Python 软件包特有的方式安装。

```
$ sudo python setup.py install
$ which gistore
/usr/local/bin/gistore
```

### 37.1.3 用 easy_install 安装

Gistore 是用 Python 语言开发的，已经在 PYPI 上注册了：*http://pypi.python.org/pypi/gistore*。就像其他 Python 软件包一样，可以使用 easy_install 进行安装，具体操作过程如下。

（1）确保您的机器上已经安装了 setuptools[②]。

几乎每个 Linux 发行版都有 setuptools 软件包，可以直接用包管理器进行安装。在 Debian / Ubuntu 上可以使用下面的命令安装 setuptools：

```
$ sudo aptitude install python-setuptools
$ which easy_install
/usr/bin/easy_install
```

（2）使用 easy_install 命令安装 Gistore：

```
$ sudo easy_install -U gistore
```

① *http://code.google.com/p/bindfs/*

② *http://peak.telecommunity.com/DevCenter/setuptools*



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 37.2 Gistore的使用

先熟悉一下Gistore的术语。

- 备份库：通过gistore init命令创建用于数据备份的数据仓库。备份库包含的数据有：
  - Git版本库相关的目录和文件。如repo.git目录（相当于.git目录）、.gitignore文件等。
  - Gistore相关的配置。如.gistore/config文件。
- 备份项：可以为一个备份库指定任意多的备份项目。
  - 例如备份/etc目录，/var/log目录等。
  - 备份项在备份库的.gistore/config文件中指定，如上述备份项在配置文件中的写法为：

```
[store "/etc"]
        enabled = true
[store "/var/log"]
        enabled = true
```

- 备份任务：在执行Gistore命令时，可以指定一个任务或多个任务。
  任务就是一个备份库的路径，可以使用绝对路径，也可以使用相对路径。如果不提供备份任务，即不指定一个备份库路径，默认使用当前目录。除了使用路径外，还可以使用一个任务别名来标识备份任务。
- 任务别名。
  - 如果一个备份库在~/.gistore.d/tasks目录（非root用户），或者/etc/gistore/tasks目录（root用户）下建立了一个符号链接，则该符号链接的名称就是这个备份库的任务别名。
  - 通过任务别名的机制，将可能分散在磁盘各处的备份库汇总在一起，便于用户定位备份库。例如可以显示所有在~/.gistore.d/tasks目录或/etc/gistore/tasks目录备份的任务列表。

### 37.2.1 创建并初始化备份库

在使用Gistore开始备份之前，必须先初始化一个备份库。命令行格式如下：

```
用法：gistore init [备份任务]
```

初始化备份库的示例如下。

- 将当前目录作为备份库进行初始化：

```
$ mkdir backup
$ cd backup
$ gistore init
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 将指定的目录作为备份库进行初始化：

```
$ sudo gistore init /backup/database
```

当一个备份库初始化完毕后，包含下列文件和目录：

- 目录 repo.git：存储备份的 Git 版本库。
- 文件 .gistore/config：Gistore 的配置文件。
- 目录 logs：Gistore 运行的日志记录。
- 目录 locks：Gistore 运行的文件锁目录。

## 37.2.2 Gistore 的配置文件

在每一个备份库的 .gistore 目录下的 config 文件是该备份库的配置文件，用于记录 Gistore 的备份项内容，以及备份回滚设置等。

例如下面的配置内容（为描述方便添加了行号）：

```
 1  # Global config for all sections
 2  [main]
 3          backend = git
 4          backuphistory = 200
 5          backupcopies = 5
 6          rootonly = no
 7          version = 2
 8
 9  [default]
10          keepemptydir = no
11          keepperm = no
12
13  # Manage your backup list using: gistore add, gistore rm commands.
14  [store "/opt/mailman/archives"]
15          enabled = true
16  [store "/opt/mailman/conf"]
17          enabled = true
18  [store "/opt/moin/conf"]
19          enabled = true
```

如何理解这个配置文件呢？

- 第 2 行到第 7 行的 [main] 小节用于 Gistore 的全局设置。
- 第 3 行设置了 Gistore 使用的 SCM 后端为 Git，这是目前唯一可用的设置。
- 第 4 行设置了 Gistore 的每一个历史分支保存的最多的提交数目，默认为 200 个提交。当超过这个提交数目时，进行备份回滚。
- 第 5 行设置了 Gistore 保存的历史分支数量，默认为 5 个历史分支。每当备份回滚时，会将备份主线保存到名为 gistore/1 的历史分支中。
- 第 6 行设置非 root_only 模式。如果开启 root_only 模式，则只有 root 用户能够执行此备份库的备份。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 第7行设置了Gistore备份库的版本。
- 第9行开始的[default]小节设置后面的备份项小节的默认设置。在后面的[store ...]小节可以覆盖此默认设置。
- 第10行设置是否保留空目录。暂未实现。
- 第11行设置是否保持文件属主和权限。暂未实现。
- 第14行到第19行是备份项小节，小节名称以store开始，后面的部分即为备份项的路径。例如[store /opt/mailman/archives]的含义是：要对/opt/mailman/archives目录进行备份。

### 37.2.3 Gistore的备份项管理

请不要直接编辑.gistore/config文件，以免因为格式错误导致Gistore无法运行。可以通过git config命令对该文件进行操作，因为实际上这个文件就是用git config命令创建的。

```
$ git config -f .gistore/config store./some/dir.enabled false
$ git config -f .gistore/config -l
```

Gistore提供了几个子命令，对备份项进行管理。

**1. 添加备份项**

进入备份库目录，执行下面的命令，添加备份项/some/dir。注意备份项要使用全路径，即要以“/”开始。

```
$ gistore add /some/dir
```

**2. 删除备份项**

进入备份库目录，执行下面的命令，则删除备份项/some/dir。第一次执行该命令停用该备份项的备份，即将store./some/dir.enabled配置变量设置为false。当第二次执行该删除命令，则彻底删除该备份项。

```
$ gistore rm /some/dir
```

**3. 查看备份项**

进入备份库目录，执行gistore status命令，显示备份库的设置及备份项列表。

```
$ gistore status
         Task name : system
         Directory : /data/backup/gistore/system
           Backend : git
 Backup capability : 200 commits * 5 copies
       Backup list :
                     /backup/databases (--)
                     /backup/ldap (--)
                     /data/backup/gistore/system/.gistore (--)
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
/etc (AD)
/opt/cosign/conf (--)
/opt/cosign/factor (--)
/opt/cosign/lib (--)
/opt/gosa/conf (--)
/opt/ossxp/conf (--)
/opt/ossxp/ssl (--)
```

从备份库的状态输出可以看到：

- 备份库有一个任务别名为 system。
- 备份库的路径是 /data/backup/gistore/system。
- 备份的容量是 200*5，如果按每天备份一次来计算，总共可以保存 1000 天，差不多 3 年的数据备份。
- 在备份项列表，可以看到多达 10 个备份项。

  每个备份项后面的括号代表其备份选项，其中 /etc 的备份选项为 AD。A 代表记录并保持授权，D 的含义是保持空目录。

### 37.2.4 执行备份任务

执行备份任务非常简单：

- 进入到备份库根目录下，执行：

```
$ sudo gistore commit -m "The reason for backup"
```

- 或者在命令行上指定备份库的路径：

```
$ sudo gistore ci /backup/database
```

说明：ci 为 commit 命令的简称。

### 37.2.5 查看备份日志

备份库中的 repo.git 就是备份数据所在的 Git 库，这个 Git 库是一个不带工作区的裸库。可以对其执行 git log 命令来查看备份日志。

因为并非采用通常的 .git 作为版本库名称，而且不带工作区，需要通过 --git-dir 参数指定版本库的位置，如下：

```
$ git --git-dir=repo.git log
```

Gistore 提供了一个 log 子命令，能更方便地显示备份日志。该子命令实际是对上面的 Git 命令的封装，因此可以向其传递任何 git log 命令可以理解的参数。如：

```
$ gistore log --pretty=oneline
```

下面是我公司内的服务器每日备份的日志片断：



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
commit 9d16b5668c1a09f6fa0b0142c6d34f3cbb33072f
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Aug 5 04:00:23 2010 +0800

    Changes summary: total= 423, A: 407, D: 1, M: 15
    ------------------------------------------------
        A => etc/gistore/tasks/Makefile,
opt/cosign/lib/share/locale/cosign.pot,
opt/cosign/lib/templates-local.old/expired_error.html,
opt/cosign/lib/templates-local.old3/error.html,
opt/cosign/lib/templates/inc/en/0020_scm.html, ...402 more...
        D => etc/gistore/tasks/default
        M => .gistore/config, etc/gistore/tasks/gosa,
etc/gistore/tasks/testlink, etc/group, etc/gshadow-, ...10 more...

commit 01b6bce2e4ee2f8cda57ceb3c4db0db9eb90bbed
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Wed Aug 4 04:01:09 2010 +0800

    Changes summary: total= 8, A: 7, M: 1
    -------------------------------------
        A => backup/databases/blog_bj/blog_bj.sql,
backup/databases/ossxp/mysql.sql, backup/databases/redmine/redmine.sql,
backup/databases/testlink/testlink-1.8.sql,
backup/databases/testlink/testlink.sql, ...2 more...
        M => .gistore/config

commit 15ef2e88f33dfa7dfb04ecbcdb9e6b2a7c4e6b00
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Aug 3 16:59:12 2010 +0800

    Changes summary: total= 2665, A: 2665
    -------------------------------------
        A => .gistore/config, etc/apache2/sites-available/gems, etc/group-,
etc/pam.d/dovecot, etc/ssl/certs/0481cb65.0, ...2660 more...
```

从上面的日志可以看出：

- 备份发生在晚上 4 点钟左右。这是因为备份是在晚上自动执行的。
- ID 为“15ef2e8”的提交是一次手动提交。从提交说明中可以看到添加了 2665 个文件。
- 最新的备份 ID 为“9d16b56”，其中既有文件添加（A），又有文件删除（D），还有文件变更（M），会随机各选择 5 个文件出现在提交日志中。

如果想查看详细的文件变更列表，使用下面的命令：

```
$ gistore log -1 --stat 9d16b56

commit 9d16b5668c1a09f6fa0b0142c6d34f3cbb33072f
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Aug 5 04:00:23 2010 +0800
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
    Changes summary: total= 423, A: 407, D: 1, M: 15
    ------------------------------------------------
        A => etc/gistore/tasks/Makefile,
opt/cosign/lib/share/locale/cosign.pot,
opt/cosign/lib/templates-local.old/expired_error.html, opt/cosign/lib/templ
        D => etc/gistore/tasks/default
        M => .gistore/config, etc/gistore/tasks/gosa,
etc/gistore/tasks/testlink, etc/group, etc/gshadow-, ...10 more...

 .gistore/config                                    |    4 +
 backup/databases/redmine/redmine.sql               |   44 +-
 etc/apache2/include/redmine/redmine.conf           |   40 +-
 etc/gistore/tasks/Makefile                         |    1 +
 etc/gistore/tasks/default                          |    1 -
 etc/gistore/tasks/gosa                             |    2 +-

 ...

 opt/gosa/conf/sieve-spam.txt                       |    6 +
 opt/gosa/conf/sieve-vacation.txt                   |    4 +
 opt/ossxp/conf/cron.d/ossxp-backup                 |    8 +-
 423 files changed, 30045 insertions(+), 51 deletions(-)
```

在备份库的 logs 目录下，还有一个备份过程的日志文件 logs/gitstore.log。记录了每次备份的诊断信息，主要用于调试 Gistore。

### 37.2.6 查看及恢复备份数据

所有的备份数据，实际上都在 repo.git 目录指向的 Git 库中维护。如何获取这些备份数据呢？

**1. 克隆方式检出**

执行下面的命令，克隆裸版本库 repo.git：

```
$ git clone repo.git data
```

进入 data 目录，就可以以 Git 的方式查看历史数据，以及恢复历史数据。当然恢复出来的历史数据还要拷贝到原始位置才能真正实现数据的恢复。

**2. 分离的版本库和工作区方式检出**

还有一个稍微复杂点的方法，就是既然版本库已经在 repo.git 中了，可以直接利用它，避免克隆导致空间上的浪费，尤其是当备份库异常庞大的时候，具体操作过程如下。

（1）创建一个工作目录，如 export。

```
$ mkdir export
```

（2）设置环境变量，制定版本库和工作区的位置。注意使用绝对路径。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ export GIT_DIR=/path/to/repo.git
$ export GIT_WORK_TREE=/path/to/export
```

（3）然后就可以进入 export 目录，执行 Git 操作了。

```
$ cd /path/to/export
$ git status
$ git checkout .
```

### 37.2.7 备份回滚及设置

我在开发 Gistore 时，最麻烦的就是备份历史的管理。如果不对备份历史进行回滚，必然会导致提交越来越多，备份空间占用越来越大，直至磁盘空间占满。

最早的想法是使用 git rebase 命令，即将准备丢弃的早期备份历史压缩为一个提交，然后后面的提交再变基到压缩后的提交之上，这样就实现了对历史提交的丢弃。但是这样的操作既费时，又复杂。忽然有一天灵机一动，为什么不用分支来保留回滚的数据？至于备份主线（master 分支）则从一个新的提交开始重建。

回滚后 master 分支如何从一个新提交开始呢？较早的实现是直接重置到一个空提交（gistore/0）上，但是这样会导致接下来的备份非常耗时。最新的实现是使用 git hash-object 命令，直接对回滚前 master 分支的最新提交进行改造，创造出一个没有父提交的新提交。

备份回滚的具体实现过程是：

（1）每次备份，都提交在 Git 库的分支 master 上。

（2）当 Git 库的 master 分支的提交数量达到规定的阈值（默认 200）时，则从 master 分支建立新的分支：gistore/1。如果已有分支 gistore/1 则建立分支 gistore/2，依次类推。

（3）然后 master 分支重置到一个用 git hash-object 命令基于当前最新提交建立的一个新提交（没有父提交）。

（4）如果保存备份历史的分支数量达到预先设定的阈值（默认 5 个分支），分支依次回滚。

- 用分支 gistore/2 重置分支 gistore/1，用分支 gistore/3 重置分支 gistore/2，依次类推。即保存最早备份历史的分支 gistore/1 被丢弃。
- 基于分支 master 建立分支 gistore/5。

（5）当分支重置及回滚发生后，对备份库的远程数据同步不会有什么影响，传输的数据量也仅是新增备份和上一次备份的差异。

虽然备份历史由于 master 分支的重置被分割为多个独立的片段，但是因为使用了 Git 提交嫁接[①]的功能，执行 gistore log 可以看到 master 分支及其他形如 gistore/N 分支的所有提交日志。奥秘就在 repo.git/info/grafts 文件。

① 第8篇第41章“41.4.h1 提交嫁接”小节。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 37.2.8 注册备份任务别名

因为 Gistore 可以在任何目录下创建备份任务，用户很难确定当前到底存在多少个备份库，因此需要提供一个机制，让用户能够对备份库进行统一管理。还有一个原因，就是在使用 Gistore 时若使用长长的备份库路径作参数会显得非常笨拙。任务别名就是用来解决这些问题的。

任务别名实际上就是备份库在用户主目录下的 ~/.gistore.d/tasks 目录（非管理员）或 /etc/gistore/tasks 目录（管理员）下创建的符号连接。例如：管理员在 /etc/gistore/tasks 目录下创建备份库的符号链接：

```
$ sudo ln -s /home/jiangxin/Desktop/mybackup /etc/gistore/tasks/jx
$ sudo ln -s /backup/database /etc/gistore/tasks/db
```

然后就可以用别名来访问对应的备份库，简化备份命令：

```
$ sudo gistore commit jx
$ sudo gistore commit db
```

查看一份完整的备份列表也非常简单，执行 gistore list 命令即可。

```
$ sudo gistore list
db          : /backup/database
jx          : /home/jiangxin/Desktop/mybackup
```

当 gistore list 命令后面指定某个任务列表时，相当于执行 gistore status 命令，查看备份状态信息：

```
$ sudo gistore list db
```

可以用一条命令对所有的任务别名执行备份：

```
$ sudo gistore commit-all
```

### 37.2.9 自动备份：crontab

在 /etc/cron.d/ 目录下创建一个文件，如 /etc/cron.d/gistore，包含如下内容：

```
## gistore backup
0   4   *   *   *    root   /usr/bin/gistore commit-all -vvvv
```

这样每天凌晨 4 点，就会以 root 用户的身份执行 gistore commit-all 命令。参数 -vvvv 的含义是提供更多的诊断输出。

将 Gistore 备份库在 /etc/gistore/tasks 目录下创建一个符号链接，就可以每天自动启动相应 Gistore 备份库的备份。

## 37.3 Gistore 双机备份

Gistore 备份库的主体就是 repo.git，即一个 Git 库。可以架设一个 Git 服务器，远程

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


主机通过克隆该 Git 服务器的备份库实现双机备份甚至是异地备份。而且最酷的是，整个数据同步的过程是可视的、快速的和无痛的，感谢伟大而又神奇的 Git。

最好使用公钥认证的、基于 SSH 的 Git 服务器架设，一来可以实现无口令的数据同步，二来增加了安全性，因为备份数据中可能包含敏感数据。

还有可以直接利用现成的 /etc/gistore/tasks 目录作为版本库的根。当然还需要在架设的 Git 服务器上，使用一个地址变换的小窍门。Gitosis 服务器软件的地址变换魔法正好可以帮助实现，请参见第 5 篇第 31 章“31.5 轻量级管理的 Git 服务”。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 38 章　补丁中的二进制文件

有的时候，需要将代码的改动以补丁文件的方式进行传递，最终合并至版本库。例如直接在软件部署目录内进行改动，再将改动传送到开发平台。或者是因为在某个开源软件的官方版本库中没有提交权限，需要将自己的改动以补丁文件的方式提供给官方。

关于补丁文件的格式，补丁的生成和应用在第 3 篇的“第 20 章 补丁文件交互”当中已经进行了详细介绍，使用的是 `git format-patch` 和 `git am` 命令，但这两个命令仅对 Git 库有效。如果没有使用 Git 对改动进行版本控制，而仅仅是两个目录：一个改动前的目录和一个改动后的目录，大部分人会选择使用 GNU 的 `diff` 命令及 `patch` 命令实现补丁文件的生成和补丁的应用。

但是 GNU 的 `diff` 命令（包括很多版本控制系统，如 SVN 的 `svn diff` 命令）生成的差异输出有一个非常大的不足或者说漏洞，就是差异输出不支持二进制文件。如果生成了新的二进制文件（如图片），或者二进制文件发生了变化，在差异输出中无法体现，当这样的差异文件被导出，应用到代码树中，会发现二进制文件或二进制文件的改动丢失了！

Git 突破了传统差异格式的限制，通过引入新的差异格式，实现了对二进制文件的支持。并且更为神奇的是，不必使用 Git 版本库对数据进行维护，可以直接对两个普通目录进行 Git 方式的差异比较和输出。

## 38.1　Git 版本库中二进制文件变更的支持

### 1. 创建支持二进制文件的补丁

对 Git 工作区的修改进行差异比较（`git diff --binary`），可以输出二进制的补丁文件。包含二进制文件差异的补丁文件可以通过 `git apply` 命令应用到版本库中。可以通过下面的示例看看 Git 的补丁文件是如何对二进制文件提供支持的，具体操作过程如下。

（1）首先建立一个空的 Git 版本库。

```
$ mkdir /tmp/test
$ cd /tmp/test
$ git init
initialized empty Git repository in /tmp/test/.git/

$ git commit --allow-empty -m initialized
[master (root-commit) 2ca650c] initialized
```

（2）然后在工作区创建一个文本文件 `readme.txt`，以及一个二进制文件 `binary.data`。二进制的数据是从系统中的二进制文件 `/bin/ls` 读取而来的，当然可以用任何其他

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的二进制文件代替。

```
$ echo hello > readme.txt
$ dd if=/bin/ls of=binary.data count=1 bs=32
记录了 1+0 的读入
记录了 1+0 的写出
32 字节 (32 B) 已复制，0.0001062 秒，301 kB/秒
```

注：拷贝 /bin/ls 可执行文件（二进制）的前 32 个字节作为 binary.data 文件。

（3）如果执行 git diff --cached 则看到的是未扩展的差异格式。

```
$ git add .
$ git diff --cached
diff --git a/binary.data b/binary.data
new file mode 100644
index 0000000..dc2e37f
Binary files /dev/null and b/binary.data differ
diff --git a/readme.txt b/readme.txt
new file mode 100644
index 0000000..ce01362
--- /dev/null
+++ b/readme.txt
@@ -0,0 +1 @@
+hello
```

可以看到对于 binary.data，此差异文件没有给出差异内容，而只是一行“Binary files ... and ... differ”。

（4）再执行 git diff --cached --binary 试试看，即增加了参数 --binary。

```
$ git diff --cached --binary
diff --git a/binary.data b/binary.data
new file mode 100644
index
0000000000000000000000000000000000000000..dc2e37f81e0fa88308bec48cd5195b6
542e61a20
GIT binary patch
literal 32
bcmb<-^>JfjWMqH=CI&kO5HCR00W1UnGBE;C

literal 0
HcmV?d00001

diff --git a/readme.txt b/readme.txt
new file mode 100644
index 0000000..ce01362
--- /dev/null
+++ b/readme.txt
@@ -0,0 +1 @@
+hello
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


看到了吗，此差异文件给出了二进制文件 binary.data 差异的内容，并且差异内容经过 base85[①]文本化了。

（5）提交后，用新的内容覆盖 binary.data 文件。

```
$ git commit -m "new text file and binary file"
[master 7ab2d01] new text file and binary file
 2 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 binary.data
 create mode 100644 readme.txt

$ dd if=/bin/ls of=binary.data count=1 bs=64
记录了 1+0 的读入
记录了 1+0 的写出
64 字节 (64 B) 已复制，0.00011264 秒，568 kB/秒

$ git commit -a -m "change binary.data."
[master a79bcbe] change binary.data.
 1 files changed, 0 insertions(+), 0 deletions(-)
```

（6）再来看看更改后的二进制文件的新差异格式。

```
$ git show HEAD --binary
commit a79bcbe50c1d278db9c9db8e42d9bc5bc72bf031
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Sun Oct 10 19:22:30 2010 +0800

    change binary.data.

diff --git a/binary.data b/binary.data
index
dc2e37f81e0fa88308bec48cd5195b6542e61a20..bf948689934caf2d874ff8168cb716f
bc2a127c3 100644
GIT binary patch
delta 37
hcmY#zn4qBGzyJX+<}pH93=9qo77QFfQiegA0RUZd1MdI;

delta 4
LcmZ=zn4kav0;B;E
```

（7）更简单的，使用 git format-patch 命令，直接将最近的两次提交导出为补丁文件。

```
$ git format-patch HEAD^^
0001-new-text-file-and-binary-file.patch
0002-change-binary.data.patch
```

毫无疑问，这两个补丁文件都包含了对二进制文件的支持。

① Git 源代码 diff.c 的 emit_binary_diff_body 函数。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
$ cat 0002-change-binary.data.patch
From a79bcbe50c1d278db9c9db8e42d9bc5bc72bf031 Mon Sep 17 00:00:00 2001
From: Jiang Xin <jiangxin@ossxp.com>
Date: Sun, 10 Oct 2010 19:22:30 +0800
Subject: [PATCH 2/2] change binary.data.

---
 binary.data |  Bin 32 -> 64 bytes
 1 files changed, 0 insertions(+), 0 deletions(-)

diff --git a/binary.data b/binary.data
index
dc2e37f81e0fa88308bec48cd5195b6542e61a20..bf948689934caf2d874ff8168cb716f
bc2a127c3 100644
GIT binary patch
delta 37
hcmY#zn4qBGzyJX+<}pH93=9qo77QFfQiegA0RUZd1MdI;

delta 4
LcmZ=zn4kav0;B;E

--
1.7.1
```

### 2. 应用包含二进制文件差异的补丁

应用包含二进制文件差异的补丁，不能使用 GNU patch 命令，因为前面曾经说过 GNU 的 diff 和 patch 不支持二进制文件的补丁，当然也不支持 Git 的新的补丁格式。将 Git 格式的补丁应用到代码树，只能使用 Git 命令，即 git apply 命令。

接着前面的例子。首先将版本库重置到最近两次提交之前的状态，即丢弃最近的两次提交，然后将两个补丁都合并到代码树中，具体操作步骤如下。

（1）重置版本库到两次提交之前的状态。

```
$ git reset --hard HEAD^^
HEAD is now at 2ca650c initialized

$ ls
0001-new-text-file-and-binary-file.patch  0002-change-binary.data.patch
```

（2）使用 git apply 应用补丁。

```
$ git apply 0001-new-text-file-and-binary-file.patch
0002-change-binary.data.patch
```

（3）可以看到 64 字节长度的 binary.data 又回来了。

```
$ ls -l
总用量 16
-rw-r--r-- 1 jiangxin jiangxin 754 10月 10 19:28
0001-new-text-file-and-binary-file.patch
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
-rw-r--r-- 1 jiangxin jiangxin 524 10 月 10 19:28
0002-change-binary.data.patch
-rw-r--r-- 1 jiangxin jiangxin  64 10 月 10 19:34 binary.data
-rw-r--r-- 1 jiangxin jiangxin   6 10 月 10 19:34 readme.txt
```

（4）最后不要忘了提交。

```
$ git add readme.txt binary.data
$ git commit -m "new text file and binary file from patch files."
[master 7c1389f] new text file and binary file from patch files.
 2 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 binary.data
 create mode 100644 readme.txt
```

Git 对补丁文件的扩展，实际上不只是增加了二进制文件的支持，还提供了对文件重命名（rename from 和 rename to 指令）、文件拷贝（copy from 和 copy to 指令）、文件删除（deleted file 指令）及文件权限（new file mode 和 new mode 指令）的支持。

## 38.2 对非 Git 版本库中二进制文件变更的支持

不在 Git 版本库中的文件和目录可以比较生成 Git 格式的补丁文件吗，以及可以执行应用补丁的操作吗？

是的，Git 的 `diff` 命令和 `apply` 命令支持对非 Git 版本库 / 工作区进行的操作。但是 1.7.3 以前版本的 Git 的 `git apply` 命令有一个 Bug，这个 Bug 导致目前的 `git apply` 命令只能应用 patch level（补丁文件前缀级别）为 1 的补丁。我已经将改正这个 Bug 的补丁文件提交到了 Git 开发列表中[①]，但有其他人先于我修正了这个 Bug。不管最终的修正方法如何，在新版本的 Git 中，这个问题应该已经解决。

下面的示例将演示如何对非 Git 版本库使用 `git diff` 和 `git patch` 命令。首先准备两个目录，一个为 `hello-1.0` 目录，在其中创建一个文本文件及一个二进制文件。

```
$ mkdir hello-1.0
$ echo hello > hello-1.0/readme.txt
$ dd if=/bin/ls of=hello-1.0/binary.dat count=1 bs=32
记录了 1+0 的读入
记录了 1+0 的写出
32 字节 (32 B) 已复制，0.0001026 秒，312 kB/ 秒
```

另外一个 `hello-2.0` 目录，其中的文本文件和二进制文件都有所更改。

```
$ mkdir hello-2.0
$ printf "hello\nworld\n" > hello-2.0/readme.txt
$ dd if=/bin/ls of=hello-2.0/binary.dat count=1 bs=64
记录了 1+0 的读入
```

① *http://marc.info/?l=git&m=129058163119515&w=2*



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
记录了 1+0 的写出
64 字节 (64 B) 已复制，0.0001022 秒，626 kB/ 秒
```

然后执行 git diff 命令。命令中的 --no-index 参数对于不在版本库中的目录 / 文件进行比较时可以省略。其中还用了 --no-prefix 参数，这样就可以生成前缀级别（patch level）为 1 的补丁文件。

```
$ git diff --no-index --binary --no-prefix \
           hello-1.0 hello-2.0 > patch.txt
$ cat patch.txt
diff --git hello-1.0/binary.dat hello-2.0/binary.dat
index
dc2e37f81e0fa88308bec48cd5195b6542e61a20..bf948689934caf2d874ff8168cb716fbc2a
127c3 100644
GIT binary patch
delta 37
hcmY#zn4qBGzyJX+<}pH93=9qo77QFfQiegA0RUZd1MdI;

delta 4
LcmZ=zn4kav0;B;E

diff --git hello-1.0/readme.txt hello-2.0/readme.txt
index ce01362..94954ab 100644
--- hello-1.0/readme.txt
+++ hello-2.0/readme.txt
@@ -1 +1,2 @@
 hello
+world
```

进入到 hello-1.0 目录，执行 git apply 应用补丁，即使 hello-1.0 不是一个 Git 库。

```
$ cd hello-1.0
$ git apply ../patch.txt
```

会惊喜地发现 hello-1.0 应用补丁后，已经变得和 hello-2.0 一样了。

```
$ git diff --stat . ../hello-2.0
```

命令 git apply 也支持反向应用补丁。反向应用补丁后，hello-1.0 中的文件被还原，和 hello-2.0 比较又可以看到差异了。

```
$ git apply -R ../patch.txt
$ git diff --stat . ../hello-2.0
 {. => ../hello-2.0}/binary.dat |  Bin 32 -> 64 bytes
 {. => ../hello-2.0}/readme.txt |    1 +
 2 files changed, 1 insertions(+), 0 deletions(-)
```

## 38.3 其他工具对 Git 扩展补丁文件的支持

Git 对二进制提供支持的扩展的补丁文件格式，已经成为补丁文件格式的新标准，被其

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


他一些应用软件所接受。例如 Mercual/Hg 就提供了对 Git 扩展补丁格式的支持。

为 hg diff 命令增加 --git 参数，实现 Git 扩展 diff 格式的输出。

```
$ hg diff --git
```

Hg 的 MQ 插件提供了对 Git 补丁的支持。

```
$ cat .hg/patches/1.diff
# HG changeset patch
# User Jiang Xin <worldhello.net AT gmail DOT com>
# Date 1286711219 -28800
# Node ID ba66b7bca4baec41a7d29c5cae6bea6d868e2c4b
# Parent  0b44094c755e181446c65c16a8b602034e65efd7
new data

diff --git a/binary.data b/binary.data
new file mode 100644
index
0000000000000000000000000000000000000000..dc2e37f81e0fa88308bec48cd5195b6542e
61a20
GIT binary patch
literal 32
bc${}+u^>JfjWMqH=CI&kO5HCR00n7&gGBE;C
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第39章 云 存 储

通过云存储，将个人数据备份在网络上是非常吸引人的服务，比较著名的公司或产品有Dropbox[①]、Surgarsync[②]、Live Mesh[③]、Syncplicity[④]等。这些产品的特点是能够和操作系统的shell整合，例如和Windows的资源管理器或Linux上的nautilus整合，当本地数据有改动时会自动同步到远程的“云存储”上。用户可以在多个计算机或手持设备上配置和同一个“云端”的账号同步，从而实现在多个计算机或多个手持设备上的数据同步。

## 39.1 现有云存储的问题

遗憾的是我并未使用过上述云存储服务，主要是支持Linux操作系统的云存储客户端比较少，或者即使有也因为网络的局限而无法访问，但是可以通过相关文档了解到其实现的机理。

- 仅支持对部分历史数据的备份。

  Dropbox支持30天数据备份，Surgarsync每个文件仅保留5个备份（对于付费用户），对于免费用户仅保留2个备份。
- 数据同步对网络带宽的依赖比较高。
- “云端”被多个设备共享，冲突解决比较困难。

  Surgarsync会将冲突的文件自动保存为多份，造成磁盘空间超出配额。其他有的产品在遇到冲突时停止同步，让用户决定选择哪个版本。

在介绍Git的书里介绍云存储，是因为上述云存储实现和Git有关吗？不是。实际上通过上面各个云存储软件特性的介绍，有经验的Linux用户会感觉这些产品在数据同步时和Linux下的rsync、unison等数据同步工具非常类似，也许只是在服务器端增加了历史备份而已。

已经有用户尝试将云存储和Git结合使用，就是将Git库本身放在本机的云存储同步区（如Dropbox目录下），Git库被同步至云端。即用云存储作为二传手，实际上还是基于本地协议操作Git。这样实际上是会有问题的。

- 如果两台机器各自进行了提交，云存储同步一定会引发冲突，这种冲突是难以解决的。
- 云端对Git的每个文件都进行备份，包括执行`git gc`命令打包后丢弃掉的松散对象。这实际对于Git是不需要的，会浪费本来就有限的空间配额。

---

① *https://www.dropbox.com/*
② *https://www.sugarsync.com/*
③ *https://www.mesh.com/*
④ *http://www.syncplicity.com/*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 因为版本库操作触发的 `git gc --auto` 命令会周期性地整理版本库，从而导致即使 Git 版本库的一个小提交也可能会触发大量的云存储数据传输。

## 39.2 Git 式云存储畅想

GitHub 是 Git 风格的云存储，但缺乏像之前提到的云存储提供的傻瓜式服务，只有 Git 用户才能真正利用好，这大大限制了 Git 在云存储领域的推广。下面是我的一个预言：一个结合了 Git 和傻瓜式云存储的网络存储服务终将诞生。新的傻瓜式云存储将有下列特征：

### 1. 差异同步传输

用户体验最为关键的是网络传输，如果用 Git 可以在同步时实现仅对文件差异进行数据传输，会大大提高同步效率。之所以现有的在线备份系统实现不了“差异同步传输”，是因为没有在本地对上一次同步时的数据做备份，只能通过时间戳或文件的哈希值判断文件是否改变，而无法得出文件修改前后的差异。

可以很容易地测试云存储软件的网络传输性能。准备一个大的压缩包（使同步时的压缩传输可以忽略），测试一下同步时间。再在该文件后面追加几个字节，然后检查同步时间。比较前后两个时间，就可以看出同步是否实现了仅针对差异的同步传输。

### 2. 可预测的本地及云端存储空间的占用

要想实现前面提到的差异同步传输，就必须在本地保存上一次同步时文件的备份。Subversion 是用一份冗余的本地拷贝实现的，这样本地存储大小是实际文件的两倍。Git 在本地是完全版本库，占用空间的逐渐增加会变得不可预测。

使用 Git 实现云存储，就要解决在本地及在服务器端空间占用不可预测的问题。对于服务器端，可以采用前面介绍的 Gistore 软件重整版本库的方法，或者通过基于历史版本重建提交然后变基来实现提交数量的删减。对于客户端来说，只保留一个提交就够了，类似 Subversion 的文件的原始拷贝，这就需要在客户端基于 Git 原理重新实现。

### 3. 更高效的云端存储效率

现有的云存储效率不高，很有可能因为冗余备份而导致存储超过配额，即使服务提供商的配额计算是以最后一个版本计算的，实际的磁盘占用还是很可观的。

Git 底层实现了一个对内容跟踪的文件系统，相同内容的文件即使文件名和目录不同，在 Git 看来都是一个对象并用一个文件存储（文件名是内容相关的 SHA1 哈希值）。因此 Git 方式实现的云存储在空间的节省上有先天的优势。

### 4. 自动进行冲突解决

冲突解决是和文件同步相关的，只有通过“差异同步传输”解决了同步的性能瓶颈，才能为冲突解决打下基础。先将冲突的各个版本都同步到本地，然后进行自动冲突解决，如果冲突无法自动解决，再提示用户手工解决冲突。还有，如果在手工冲突解决时引入类似



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


kdiff3 一样的工具，对用户会更有吸引力。

### 5. Git 提交中引入特殊标识

如果使用变基或其他技术实现备份提交数量的删减，就会在云端的提交与本地数据的合并上产生问题。可以通过为提交引入特殊的唯一性标识，不随着 Git 变基而改变，就像在 Gerrit 中的 Change-Id 标签一样。

我相信，基于 Git 的文件系统及传输机理可以实现一个更好用的云存储服务[①]。

① 在本书基本完稿时，听说了一个名为 SparkleShare 的项目，似乎就是一个基于 Git 的云存储方案。网址：*http://sparkleshare.org/*。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第 8 篇

# Git 杂谈

Git 有着非常庞杂的命令集和功能，到目前为止尚有一些命令及要点还没有介绍。在构思本书的过程中，我尝试用 FreeMind 软件将准备讲述的 Git 的各个命令和要点在各个章节之间拖动，以期在内容上更加充实，组织上更加合理，讲述上更加方便，但最终还是剩下了一些 Git 命令和要点没有安排在前面的章节中。于是这些不常用的 Git 命令和要点（缺乏它们会影响一本被冠以“权威指南”的书的完备性）放在本书的最后“Git 杂谈”中予以讲述。

本篇首先用一章的内容来介绍跨平台项目在使用 Git 时应注意的事项，包括字符集问题、文件名大小写问题、文本文件换行符问题。然后，接下来的一章会概要介绍本书到目前为止尚未涉猎到的 Git 话题和相关命令，如：属性、钩子、模板、稀疏检出、浅克隆及嫁接，还会介绍 `git replace` 和 `git notes` 等命令。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 40 章　跨平台操作 Git

您是在什么平台（操作系统）中使用 Git 呢？图 40-1 是网上发布的一个 Git 使用平台调查结果的截图[①]，从中可以看出排在前三位的是：Linux、Mac OS X 和 Windows。而 Windows 用户中又以使用 msysGit 的用户居多。

**On which operating system(s) do you use Git?** (Choice - Multiple answers)

| | | |
|---|---|---|
| GNU/Linux | 85% | 7243 |
| *BSD (FreeBSD, OpenBSD, NetBSD, etc.) | 7% | 639 |
| MacOS X (Darwin) | 46% | 3932 |
| MS Windows/Cygwin | 10% | 823 |
| MS Windows/msysGit (MINGW) | 25% | 2151 |
| other Unix | 2% | 211 |
| Other, please specify | 1% | 117 |
| | Total respondents | 8542 |
| | Respondents who skipped this question | 299 |

图 40-1　Git 用户操作系统使用分布图

在如今手持设备争夺激烈的年代，在什么操作系统上进行软件开发工作已经变得不那么重要了，很多手持设备都提供可以运行在各种主流操作系统上的虚拟器，因此一个项目团队的成员根据各自的使用习惯，可能使用不同的操作系统。当一个团队中不同的成员在不同的平台中使用 Git 进行交互时，可能会遇到平台兼容性的问题。

即使团队成员都在同一种操作系统上工作（如 Windows），但 Git 服务器可能架设在另外的平台上（如 Linux），或者产品的源代码被分发到另外的平台上进行编译、部署，同样都会遇到平台兼容性的问题。

## 40.1　字符集问题

本书第 1 篇“第 3 章 安装 Git”就已经详细介绍了不同平台对本地字符集（如中文）的支持情况，本章将再做一次简单的概述。

Linux、Mac OS X 及 Windows 下的 Cygwin 默认使用 UTF-8 字符集。Git 运行在这些平台下，能够使用本地语言（如中文）写提交说明、命名文件，甚至使用本地语言命名分支和

① *http://www.survs.com/results/33Q0OZZE/MV653KSPI2*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


里程碑。在这些平台上唯一要做的就是对 Git 进行如下设置，以便使用了本地语言（如中文）命名文件后，能够在状态查看、差异比较时正确地显示文件名。

```
$ git config --global core.quotepath false
```

但是如果在 Windows 平台使用 msysGit，或者在其他平台使用非 UTF-8 字符集，要想使用本地语言撰写提交说明、命名文件名和目录名就非常具有挑战性了。例如对于使用 GBK 字符集的中文 Windows，需要为 Git 进行如下设置，才能够在提交说明中正确地使用中文。

```
$ git config --system core.quotepath false
$ git config --system i18n.commitEncoding gbk
$ git config --system i18n.logOutputEncoding gbk
```

当像上面那样设置 i18n.commitEncoding 后，如果执行提交，就会在提交对象中嵌入编码设置的指令。例如在 Windows 中使用 msysGit 执行一次提交，在 Linux 上使用 git cat-file 命令查看提交时会出现乱码，需要使用 iconv 命令对输出进行字符集转换，才能正确查看该提交对象。从下面输出的倒数第三行可以看到 encoding gbk 这条对字符集进行设置的指令。

```
$ git cat-file -p HEAD | iconv -f gbk -t utf-8
tree 00e814cda96ac016bcacabcf4c8a84156e304ac6
parent 52e6454db3d99c85b1b5a00ef987f8fc6d28c020
author Jiang Xin <jiangxin@ossxp.com> 1297241081 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1297241081 +0800
encoding gbk

添加中文说明。
```

因为在提交对象中声明了正确的字符集，因此在 Linux 下可以用 git log 命令正确显示 msysGit 生成的包含中文的提交说明。

但是对于非 UTF-8 字符集平台（如 msysGit），Git 当前版本（1.7.4）对使用本地字符（如中文）命名文件或目录的支持尚不完善。文件名和目录名实际上是写在树对象中的，Git 在创建树对象时，以本地字符集而非 UTF-8 字符集进行保存，因而在跨平台时会造成文件名乱码。例如下面的示例显示的是在 Linux 平台（UTF-8 字符集）下查看由 msysGit 提交的包含中文文件的树对象。注意要在 git cat-file 命令的后面通过管道符号调用 iconv 命令进行字符转换，否则不能正确地显示中文。如果直接在 Linux 平台检出，检出的文件名显示为乱码。

```
$ git cat-file -p HEAD^{tree} | iconv -f gbk -t utf-8
100644 blob 8c0b112f56b3b9897007031ea38c130b0b161d5a    说明.txt
```

## 40.2 文件名大小写问题

Linux、Solaris、BSD 及其他类 Unix 操作系统使用的是大小写敏感的文件系统，而



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


Windows 和 Mac OS X（默认安装）的文件系统则是大小写不敏感的文件系统。即用文件名 README、readme 及 Readme（混合大小写）进行访问，在 Linux 等操作系统上访问的是不同的文件，而在 Windows 和 Mac OS X 上则指向同一个文件。换句话说，不同的文件 README、readme 及 Readme 在 Linux 等操作系统上可以共存，而在 Windows 和 Mac OS X 上，这些文件只能同时存在一个，另外的会被覆盖，因为在大小写不敏感的操作系统看来，这些文件是同一个文件。

如果在 Linux 上为 Git 版本库添加了两个文件名仅大小写不同的文件（文件内容各不相同），如 README 和 readme。当推送到服务器的共享版本库上，并在文件名大小写不敏感的操作系统中克隆或同步时，就会出现问题。例如在 Windows 和 Mac OS X 平台上执行 git clone 后，在本地工作区中出现两个同名文件中的一个，而另外一个文件会被覆盖，这就会导致 Git 对工作区中的文件造成误判，在提交时会导致文件内容被破坏。

当一个项目存在跨平台开发的情况时，为了避免这类问题的发生，在一个文件名大小写敏感的操作系统（如 Linux）中克隆版本库后，应立即对版本库进行如下设置，让版本库的行为好似对文件名大小写不敏感。

```
$ git config core.ignorecase true
```

Windows 和 Mac OS X 在初始化一个版本库或克隆一个版本库时，会自动在版本库中包含配置变量 core.ignorecase 为 true 的设置，除非版本库不是通过克隆而是直接从 Linux 上拷贝而来的。

当版本库包含了 core.ignorecase 为 true 的配置后，文件名在文件添加时便被唯一确定。如果之后修改文件内容的同时又修改了文件名中字母的大小写，只会被视为对文件内容的修改，执行提交不会改变文件名。如果对添加文件时设置的文件名的大小写不满意，确实需要对文件重命名，对于 Linux 来说很简单，直接运行 git mv 命令显式地对文件进行重命名操作。例如执行下面的命令就可以将 changelog 文件的文件名修改为 ChangeLog。

```
$ git mv changelog ChangeLog
$ git commit
```

但是对于 Windows 和 Mac OS X 却不能这么操作，因为 Git 会拒绝这样的重命名操作：

```
$ git mv changelog ChangeLog
fatal: destination exists, source=changelog, destination=ChangeLog
```

而需要像下面这样，先将文件重命名为另外的一个名称，再执行一次重命名改回为正确的文件名，如下：

```
$ git mv changelog non-exist-filename
$ git mv non-exist-filename ChangeLog
$ git commit
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## 40.3 换行符问题

每一个通用的版本控制系统，无论是 CVS、Subversion、Git，还是其他，都要面对换行符转换的问题。这是因为作为通用的版本控制系统，一定会面对来自不同操作系统的文件，而不同的操作系统在处理文本文件时，可能使用不同的换行符。

**1. 不同的操作系统可能使用不同的换行符**

文本文件的每一行结尾都用一个或两个特殊的 ASCII 字符进行标识，这个标识就是换行符。主要的换行符有三种：LF（line feed 即换行，C 语言等用 '\n' 表示，相当于十六进制的 0x0A）、CR（Carriage return 即回车，C 语言等用 '\r' 表示，相当于十六进制的 0x0D）和 CRLF（即由两个字符 CR + LF 组成，即 "\r\n"，相当于十六进制的 0x0D 0x0A），分别用在不同的操作系统中[①]。

- LF 换行符：用于 Multics、Unix、类 Unix（如 GNU/Linux、AIX、Xenix、Mac OS X、FreeBSD 等）、BeOS、Amiga、RISC OS 等操作系统中。
- CRLF 换行符：用于 DEC TOPS-10、RT-11 和其他早期的非 Unix，以及 CP/M、MP/M、DOS（MS-DOS、PC-DOS 等）、Atari TOS、OS/2、Microsoft Windows、Symbian OS、Palm OS 等系统中。
- CR 换行符：用于 Commodore 8 位机、TRS-80、苹果 II 家族、Mac OS 9 及更早版本。

实际上，自从苹果的 Mac OS 从第 10 版转向 Unix 内核开始，依据不同的文本文件换行符，主流的操作系统可以划分为两大阵营：一个是微软 Windows 作为一方，使用 CRLF 作为换行符；另外一方包括 Unix、类 Unix（如 Linux 和 Mac OS X 等）使用 LF 作为换行符。分属不同阵营的操作系统之间交换文本文件会因为换行符的不同而造成障碍，而使用版本控制系统，也同样会遇到换行符的麻烦：

- 编辑器不能识别换行符，可能会显示为特殊字符，如 Linux 上的编辑器显示的 ^M 特殊字符，就是拜 Windows 的 CRLF 换行符所赐。或者丢弃换行符，如来自 Linux 的文本文件，在 Windows 上打开可能会因为识别不了换行符，导致所有的行合并在一起。
- 版本库中的文件被来自不同操作系统的用户改来改去，在某一次提交中换行符为 LF，在下一次提交中被替换为 CRLF，这不但会在查看文件版本间的差异时造成困惑（所有的行都显示为变更），还会给版本库的存储带来不必要的冗余。
- 可能会在一个文件中引入混杂的换行符，即有的行是 LF，而有的行是 CRLF。无论在哪个操作系统中用编辑器打开这样的文件，或多或少都会感到困惑。
- 如果版本控制系统提供文本文件换行符的自动转换，在 Windows 平台将版本库文件导出为源码包并发布，当该源码包被 Linux 用户下载后，编译、运行可能会有问题，反之亦然。

① *http://en.wikipedia.org/wiki/Newline*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 2. 文件类型判别，是换行符转换的基础

几乎所有的版本库控制系统都采用这样的解决方案：对于文本文件，在版本库中保存时换行符使用 LF，当从版本库检出到工作区时，则根据平台的不同或用户设置的不同，来对文本文件的换行符进行转换（转换为 LF、CR 或 CRLF）。

为什么换行符转换要特意强调文本文件呢？这是因为如果对二进制文件（程序或数据）当中出现的换行符进行上述转换，会导致二进制文件被破坏。因此判别文件类型是文本文件还是二进制文件，是进行文件换行符正确转换的基础。

有的版本控制系统，如 CVS，必须在添加文件时人为地设定文件类型（用 -kb 参数设定二进制文件），一旦用户忘记对二进制文件进行标记，就会造成二进制文件被破坏。这种破坏有时会藏得比较深，例如在 Linux 上检出文件一切正常，因为版本库中被误判为文本文件的图形文件中所包含的字符 0x0A 在 Linux 上检出时没有改变，但是在 Windows 上检出时会导致图形文件中的 0x0A 字符被转换为 0x0D 0x0A 两个字符，从而造成图片被破坏。

有的版本控制系统可以自动识别文本文件和二进制文件，但是识别算法存在问题。例如 Subversion 检查文件的前 1024 字节的内容，如果其中包含 NULL 字符（0x00），或者超过 15% 的字符是非 ASCII 字符，则 Subversion 认定此文件为二进制文件[①]。这种算法会将包含大量中文的文本文件当作二进制文件，不进行换行符转换，也不进行版本间的比较（除非强制执行）。

Git 显然比 Subversion 更了解这个世界上文字的多样性，因此在判别二进制文件上没有多余的判别步骤，只对 blob 对象的前 8000 个字符进行检查，如果其中出现 NULL 字符（0x00）则当作二进制文件，否则为文本文件[②]。Git 还允许用户通过属性文件对文件类型进行设置，属性文件设置优先。

Git 默认并不开启文本文件的换行符转换，因为毕竟 Git 对文件是否是二进制文件所做的猜测存在误判的可能。如果用户通过属性文件或其他方式显式地对文件类型进行了设置，则 Git 就会对文本文件开启换行符转换。

下面是一个属性文件的示例，为方便描述标以行号。

```
1  *.txt          text
2  *.vcproj       eol=crlf
3  *.sh           eol=lf
4  *.jpg          -text
5  *.jpeg         binary
```

包含了上面属性文件的版本库，会将以 .txt、.vcproj、.sh 为扩展名的文件视为文本文件，在处理过程中会进行换行符转换，而将以 .jpg、.jpeg 为扩展名的文件视为二进制文件，不进行换行符转换。

---

① 参见 Subversion 源代码 subversion/libsvn_subr/io.c 中的 svn_io_detect_mimetype2 函数。

② 参见 Git 源代码 xdiff-interface.c 中的 buffer_is_binary 函数。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 3. 依据属性文件进行换行符转换

关于属性文件，会在后面的章节详细介绍，现在可以将其理解为工作区目录下的 .gitattributes 文件，其文件匹配方法及该文件的作用范围和 .gitignore 文件非常类似。

像上面的属性文件示例中，第 1 行设置了扩展名为 .txt 的文件具有 text 属性，则所有扩展名为 .txt 的文件添加到版本库时，在版本库中创建的 blob 文件的换行符一律被转换为 LF。而当扩展名为 .txt 的文件检出到工作区时，则根据平台的不同而使用不同的换行符，如在 Linux 上检出则使用 LF 换行符，如在 Windows 上检出则使用 CRLF 换行符。

示例中的第 2 行设置扩展名为 .vcproj 的文件的属性 eol 的值为 crlf，隐含着该文件属于文本文件的含义，当向版本库添加扩展名为 .vcproj 的文件时，在版本库中创建的 blob 文件的换行符一律转换为 LF。而当该类型的文件检出到工作区时，则一律使用 CRLF 作为换行符，不管是在 Windows 上检出，还是在 Linux 上检出。

同理示例中的第 3 行设置的扩展名为 .sh 的文件也会进行类似的换行符转换，区别在于该类型文件无论在哪个平台检出，都使用 LF 作为换行符。

像上面那样逐一为不同类型的文件设置换行符格式显得很麻烦，可以在属性文件中添加下面的设置，为所有文件开启文件类型自动判断。

```
*  text=auto
```

当为所有文件设置了 text=auto 的属性后，Git 就会在文件检入和检出时对文件是否是二进制进行判断，采用前面提到的方法：如果文件头部的 8000 个字符中出现 NULL 字符则为二进制文件，否则为文本文件。如果判断文件是文本文件就会启用换行符转换。至于本地检出文件采用什么换行符格式，实际上是由 core.eol 配置变量进行设置的，不过因为 core.eol 未被设置时会采用默认值 native，才使得工作区文本文件的检出采用操作系统默认的换行符格式。配置变量 core.eol 除了默认的 native 外，还可以使用 lf 和 crlf，不过一般较少用到。

### 4. 使用 Git 配置变量控制换行符转换

在 Git 1.7.4 之前，用属性文件的方式来设置文件的换行符转换，只能逐一为版本库进行设置，如果要为本地所有的版本库设定文件换行符转换就非常麻烦。Git 1.7.4 提供了全局可用的属性文件，实现了对换行符转换设定的全局控制，我们会在后面的章节予以介绍。现在介绍另外一个方法，即通过配置变量 core.autocrlf 来开启文本文件换行符转换的功能。例如执行下面的命令，对配置变量 core.autocrlf 进行设置：

```
$ git config --global core.autocrlf true
```

默认 Git 不对配置变量 core.autocrlf 进行设置，同时如果没有通过属性文件指定文件的类型，Git 不对文件进行换行符转换。但是将配置变量 core.autocrlf 设置为下列值时，会开启 Git 对文件类型的智能判别并对文本文件执行换行符转换。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 设置配置变量 core.autocrlf 为 true。
  效果就相当于为版本库中的所有文件设置了 text=auto 的属性。即通过 Git 对文件类型的自动判定，对文本文件进行换行符转换。在版本库的 blob 文件中使用 LF 作为换行符，而检出到工作区时无论是什么操作系统都使用 CRLF 为换行符。注意当设置了 core.autocrlf 为 true 时，会忽略 core.eol 的设置，工作区文件始终使用 CRLF 作为换行符，这对于 Windows 下的 Git 非常适合，但不适用于 Linux 等操作系统。
- 设置配置变量 core.autocrlf 为 input。
  同样开启文本文件的换行符转换，但只是在文件提交到版本库时，将新增入库的 blob 文件的换行符转换为 LF。如果将文件从版本库检出到工作区，则不进行文件转换，即版本库中的文件若是采用 LF 换行符，检出仍旧是 LF 作为换行符。这个设置对 Linux 等操作系统下的 Git 非常适合，但不适合于 Windows。

### 5. 换行符转换的异常捕获

无论用户是通过属性文件来设定文件的类型，还是通过 Git 智能判别，都可能错误地将二进制文件识别为文本文件，在转换过程中造成文件的破坏。有一种情况下的破坏最为严重，就是误判的文件中包含不一致的换行符（既有 CRLF，又有 LF），这将会导致保存到版本库中的 blob 对象无论通过何种转换方式都不能还原回原有的文件。

Git 提供了名为 core.safecrlf 的配置变量，可以用于捕获这种不可逆的换行符转换，以提醒用户注意。将配置变量 core.safecrlf 设置为 true 时，如果发现存在不可逆换行符转换，会报错退出，拒绝执行不可逆的换行符转换。如果将配置变量 core.safecrlf 设置为 warn 则允许不可逆的转换，但发现不可逆转换时会发出警告。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 第 41 章 Git 的其他特性

## 41.1 属性

Git 通过属性文件为版本库中的文件或目录添加属性。设置了属性的文件或目录，在执行 Git 相关操作时会做特殊处理，正如之前介绍换行符转换时设置了文本属性（text）的文件那样。

### 41.1.1 属性定义

属性文件是一个普通的文本文件，每一行对一个路径（可使用通配符）设置相应的属性。语法格式如下：

```
<pattern> <attr1> <attr2> ...
```

其中路径由可以使用通配符的<pattern>来定义，在<pattern>后面可以设置一个或多个属性，不同的属性之间用空格分开。路径中通配符的用法和文件忽略（.gitignore）的语法格式相同，参见本书第 2 篇第 10 章的“10.8 文件忽略”的相关内容。下面以 text 属性为例来介绍属性的不同写法：

- text

  直接通过属性名进行设置，相当于将 text 的属性值设置为 true。

  对于设置了 text 属性的文件，不再需要 Git 来猜测文件的类型，而可以直接判定为文本文件并进行相应的换行符转换。

- -text

  在属性名前用减号标识，相当于将 text 的属性值设置为 false。

  对于设置了取反 text 属性的文件，直接判定为二进制文件，在文件检入和检出时不进行换行符转换。

- !text

  在属性名前面添加感叹号，相当于没有设置该属性，既不等于 true，也不等于 false。

  对于未定义 text 属性的文件，根据 Git 是否配置了 core.autocrlf 配置变量来决定是否进行换行符转换。因此对于 !text（没有定义 text 属性）和 -text（text 属性设置取反），两者存在差异。

- text=auto

  属性除了上述 true、false 和未设置三个状态外，还可以对属性用相关的枚举值（预定义的字符串）进行设置。不同的属性值可能有不同的枚举值，对于 text 属性可以设

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


置为 auto。

对于 text 属性设置为 auto 的文件，文件类型实际上尚未确定，需要 Git 读取文件内容进行智能判别，判别为文本文件则进行换行符转换。显然当设置 text 属性为 auto 时，并不等同于设置为 true。

### 41.1.2 属性文件及优先级

属性文件可以以 .gitattributes 文件名保存在工作区目录中，提交到版本库后就可以和其他用户共享项目文件的属性设置。属性文件也可以保存在工作区之外，例如保存在文件 .git/info/attributes 中，则仅对本版本库生效，若保存在 /etc/gitattributes[①]文件中则对全局生效。在查询某个工作区某一文件的属性时，不同位置的属性文件具有不同的优先级，Git 依据下列顺序依次访问属性文件：

（1）文件 .git/info/attributes 具有最高的优先级。

（2）接下来检查工作区同一目录下的 .gitattributes，并依次向上递归查找 .gitattributes 文件，直到工作区的根目录。

（3）然后查询由 Git 的配置变量 core.attributesfile 指定的全局属性文件。

（4）最后是系统属性文件，即文件 $(prefix)/etc/gitattributes。不同的 Git 安装方式下这个文件的位置可能不同，但是该文件始终和 Git 的系统配置文件（可以通过 git config --system -e 命令打开系统配置文件从而知道其位置）位于同一目录中。

注意：只有在 1.7.4 或更新版本的 Git 中才提供后两种（全局的和系统级的）属性文件。可以通过下面的例子来理解属性文件的优先级和属性设置方法。

首先来看看某个版本库及系统中所包含的属性文件：

- 其一是位于版本库中的文件 .git/info/attributes，内容如下：

```
a*      foo !bar -baz
```

- 其二是位于工作区子目录 t 下的属性文件，即 t/.gitattributes，内容如下：

```
ab*     merge=filfre
abc     -foo -bar
*.c     frotz
```

- 再一个就是位于工作区根目录下的属性文件 .gitattributes，内容如下：

```
abc     foo bar baz
```

- 如果系统文件 /etc/gitconfig 中包含如下配置，则每个用户主目录下的 .gitattributes 文件都被作为全局属性文件。

```
[core]
  attributesfile = ~/.gitattributes
```

① 随着 Git 安装方式的不同，这个文件的位置也可能不同。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


❑ 位于用户主目录下的属性文件，即文件 ~/.gitattributes 的内容如下：

```
* text=auto
```

当查询工作区文件 t/abc 的属性时，根据属性文件的优先级，按照下列顺序进行检索：

（1）先检查属性文件 .git/info/attributes。显然该文件中唯一的一行就和文件 t/abc 匹配，因此文件 t/abc 的属性如下：

```
foo   : true
bar   : 未设置
baz   : false
```

（2）再检查和文件 t/abc 同目录的属性文件 t/.gitattributes。该属性文件的前两行和路径 t/abc 相匹配，但是因为属性文件 .git/info/attributes 已经提供了 foo 和 bar 的属性，因此第二行对 foo 和 bar 属性的设置不起作用。经过这一步，文件 t/abc 获得的属性为：

```
foo   : true
bar   : 未设置
baz   : false
merge : filfre
```

（3）然后沿工作区的当前目录向上遍历属性文件，找到工作区根目录下的属性文件 .gitattributes 进行检查。因为前面的属性文件已经提供了 foo、bar 和 baz 属性设置，所以文件 t/abc 的属性和上面第 2 步的结果一样。

（4）因为将 core.attributesfile 设置为 ~/.gitattributes 文件，因此接下来查找用户主目录下的文件即 .gitattributes。该文件唯一的一行匹配所有的文件，因此 t/abc 又被附加了新的属性值 text=auto。最终，文件 t/abc 的属性如下：

```
foo   : true
bar   : 未设置
baz   : false
merge : filfre
text  : auto
```

Git 提供了一个查看文件属性设置的命令：git check-attr。针对本例用下面的命令可以查看到文件 t/abc 各个属性的设置情况。

```
$ git check-attr foo bar baz merge text -- t/abc
t/abc: foo: set
t/abc: bar: unspecified
t/abc: baz: unset
t/abc: merge: filfre
t/abc: text: auto
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 41.1.3 常用属性介绍

1. text

属性text用于显式地指定文件的类型：二进制（-text）、文本文件（text）或是开启文件类型的智能判别（text=auto）。对于文本文件，Git会对其进行换行符转换。本篇第40章“40.3 换行符问题”中已经详细介绍了属性text的用法，并且在本章“41.1.1 属性定义”的示例中对属性text的取值做了总结，在此不再赘述。

在“40.3 换行符问题”一节中，我们还知道可以通过在Git配置文件中设置core.autocrlf配置变量，来开启Git对文件类型的智能判别，并对文本文件开启换行符转换。那么Git的配置变量core.autocrlf和属性text有什么异同呢？

将Git的配置变量core.autocrlf设置为true或input，相当于设置了属性text=auto。但是Git配置文件中的配置变量只能在本地进行设置并且只对本地版本库有效，不能通过共享版本库传递到其他用户的本地版本库中，因而core.autocrlf开启的换行符转换不能跟其他用户共享，或者说不能将换行符转换策略设置为整个项目（版本库）的强制规范。属性文件则不同，可以被检入到版本库中并通过共享版本库传递给其他用户，因此可以通过在检入的.gitattributes文件中设置text属性，或者干脆设置text=auto属性，强制同一项目的所有用户在提交文本文件时都要规范换行符。

建议所有可能要进行跨平台开发的项目都在项目根目录中检入一个.gitattributes文件，根据文件扩展名设置文件的text属性，或者设置即将介绍的eol属性。

2. eol

属性eol用于设定文本文件的换行符格式。对于设置了eol属性的文件，如果没有设定text属性时，默认会设置text属性为true。属性eol的取值如下：

- eol=crlf

  当文件检入版本库时，blob对象使用LF作为换行符。当检出到工作区时，使用CRLF作为换行符。

- eol=lf

  当文件检入版本库时，blob对象使用LF作为换行符，检出的时候工作区的文件也使用LF作为换行符。

除了通过属性设定换行符格式外，还可以在Git的配置文件中通过core.eol配置变量来设定。两者的区别在于配置文件中的core.eol配置变量设置的换行符是一个默认值，没有通过eol属性指定换行符格式的文本文件会采用core.eol的设置。变量core.eol的值可以设定为lf、crlf和native。默认core.eol的取值为native，即采用操作系统标准的换行符格式。

下面的示例通过属性文件设置文件的换行符格式。

```
*.vcproj        eol=crlf
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
*.sh            eol=lf
```

扩展名为 .vcproj 的文件使用 CRLF 作为换行符，而扩展名为 .sh 的文件则使用 LF 作为换行符。在版本库中检入类似的属性文件，会使得 Git 客户端无论在什么操作系统中都能够在工作区检出一致的换行符格式，这样无论是在 Windows 上还是在 Linux 上使用 git archive 命令将工作区文件打包，导出的文件都会保持正确的换行符格式。

### 3. ident

属性 ident 开启文本文件中的关键字扩展，即关键字 $Id$ 的自动扩展。当检出到工作区时，$Id$ 自动扩展为 $Id:，后面紧接着 40 位 SHA1 哈希值（相应 blob 对象的哈希值），然后以一个 $ 字符结尾。当文件检入时，要把内容中出现的以 $Id: 开始，以 $ 结束的内容替换为 $Id$ 再保存到 blob 对象中。

这个功能可以说是对 CVS 相应功能的致敬。自动扩展的内容使用的是 blob 的哈希值而非提交本身的哈希值，因此并无太大的实际意义，不建议使用。如果希望在文本文件中扩展出提交者姓名、提交 ID 等更有实际意义的内容，可以参照后面介绍的属性 export-subst。

### 4. filter

属性 filter 为文件设置一个自定义转换过滤器，以便文件在检入版本库及检出到工作区时进行相应的转换。定义转换过滤器通过 Git 配置文件来完成，因此这个属性应该只在本地进行设置，而不要也不能通过检入到版本库中的 .gitattributes 文件来传递。

例如下面的属性文件设置了所有的 C 语言源文件在检入和检出的时候使用名为 indent 的代码格式化过滤器。

```
*.c     filter=indent
```

然后还要通过 Git 配置文件设定 indent 过滤器，示例如下：

```
[filter "indent"]
        clean = indent
        smudge = cat
```

定义过滤器只要设置两条命令，一条是名为 clean 的配置设定的的命令，用于在文件检入时执行，另外一条是名为 smudge 的配置设定的命令，用于将文件检出到工作区。对于本例，在代码检入时执行 indent 命令对代码格式化后，再保存到版本库中。当检出到工作区时，执行 cat 命令实际上相当于直接将 blob 对象复制到工作区。

### 5. diff

和前面介绍的属性不同，属性 diff 不会对文件检入检出造成影响，而只是在查看文件历史变更时起作用。属性 diff 可以取值如下：

- diff

  进行版本间的差异比较时，以文本方式进行比较，即使文件看起来像是二进制文件（包含 NULL 字符），或者被设置为二进制文件（-text）。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- -diff

  不以文本方式进行差异比较，而以二进制方式进行比较。默认二进制文件不进行差异比较，因此包含 -diff 属性设置的文件在差异比较时不显示内容上的差异。对于有些文本文件（如 postscript 文件）进行差异比较没有意义，可以对其设置 -diff 属性，避免在显示提交版本间的差异时造成干扰。

- !diff

  不设置 diff 属性，相当于在执行差异比较时要对文件内容进行智能判别，如果文件看起来像是文本文件，则显示文本格式的差异比较。

- diff=<driver>

  设定一个外部的驱动用于文件的差异比较。例如对于 Word 文档的差异比较就可以通过这种方式进行配置。

Word 文档属于二进制文件，默认不显示差异比较。在 Linux 上有一个名为 antiword 的应用软件可以将 Word 文档转换为文本文件显示，借助该软件就可以实现在 Linux（包括 Mac OS X）上显示 Word 文件版本间的差异。

下面的 Git 配置就定义了一个名为 antiword 的适用于 Word 差异比较的驱动：

```
[diff "antiword"]
  textconv=antiword
```

其中 textconv 属性用于设定一个文件转换命令行，这里设置为 antiword，用于将 Word 文档转换为纯文本。

然后还需要设置属性，修改版本库下的 .git/info/attributes 文件就可以了，新增的属性设置如下：

```
*.doc  diff=antiword
```

关于更多的差异比较的外部驱动的设置，可以执行 git help --web attributes 来参见相关的帮助。

### 6. merge

属性 merge 用于为文件设置指定的合并策略，受影响的 Git 命令有：git merge、git revert 和 git cherry-pick 等。属性 merge 可以取值如下：

- merge

  使用内置的三向合并策略。

- -merge

  将当前分支的文件版本设置为暂时的合并结果，并且声明合并发生了冲突，这实际上是二进制文件默认的合并方式。可以对文本文件设置该属性，使得在合并时的行为类似于二进制文件。

- !merge

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


和定义了 merge 属性的效果类似，使用内置的三向合并策略。然而当通过 Git 配置文件的 merge.default 配置变量设置了合并策略后，如果没有为文件设置 merge 属性，则使用 merge.default 设定的策略。

- merge=<driver>

  使用指定的合并驱动执行三向文件合并。驱动可以是内置的三个驱动，也可以是用户通过 Git 配置文件自定义的驱动。

下面重点说一说通过枚举值来指定在合并时使用的内置驱动和自定义驱动。先来看看 Git 提供的三个内置驱动：

- merge=text

  默认文本文件在进行三向合并时使用的驱动。会在合并后的文本文件中用特殊的标识 <<<<<<< 、======= 和 >>>>>>> 来标记冲突的内容。

- merge=binary

  默认二进制文件在进行三向合并时使用的驱动。会在工作区中保持当前分支中的版本不变，但是会通过在三个暂存区中进行冲突标识，使得文件处于冲突状态。

- merge=union

  在文本文件进行三向合并的过程中，不使用冲突标识符来标记冲突，而是将冲突双方的内容简单地罗列在文件中。用户应该对合并后的文件进行检查。请慎用此合并驱动。

用户还可以自定义驱动。例如 Topgit 就使用自定义合并驱动的方式来控制两个 Topgit 管理文件 .topmsg 和 .topdeps 的合并行为。

Topgit 会在版本库的配置文件 .git/info/config 中添加下面的设置来定义一个名为 ours 的合并驱动。注意不要将此 ours 驱动和本书第 3 篇第 16 章“16.6 合并策略”一节中介绍的 ours 合并策略弄混淆。

```
[merge "ours"]
  name = \"always keep ours\" merge driver
  driver = touch %A
```

定义的合并驱动的名称由 merge.*.name 给出，合并时执行的命令则由配置 merge.*.driver 给出。本例中使用了命令 touch %A，含义为对当前分支中的文件进行简单的触碰（更新文件时间戳），亦即合并冲突时采用本地版本，丢弃其他版本。

Topgit 还会在版本库 .git/info/attributes 属性文件中包含下面的属性设置：

```
.topmsg merge=ours
.topdeps  merge=ours
```

上述设置的含义为在遇到合并冲突时，对这两个 Topgit 管理文件采用在 Git 配置文件中设定的 ours 合并驱动。Topgit 之所以要这么实现是因为不同特性分支的管理文件之间并无关联，也不需要合并，在遇到冲突时只使用自己的版本即可。这对于要经常执行变基和分支合并的 Topgit 来说，设置这个策略可以简化管理，但是这个合并设置在特定情况下也存在不合理之处。例如两个用户工作在同一分支上，同时更改了 .topmsg 文件以修改特性分支的描述，



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


在合并时会覆盖对方的修改，这显然是不好的行为。但是权衡利弊，还是如此实现最好。

7. whitespace

Git 可以对文本文件中空白字符的使用是否规范做出检查，在进行文件差异比较时，将使用不当的空白字符用红色进行标记（开启 color.diff.whitespace[①]）。也可以在执行 git apply 时通过参数 --whitespace=error 防止错误的空白字符应用到提交中。

Git 默认开启对下面三类错误空白字符的检查。

- blank-at-eol
  在行尾出现的空白字符（换行符之前）被视为误用。
- space-before-tab
  在行首缩进中出现在 TAB 字符前面的空白字符视为误用。
- blank-at-eof
  在文件末尾的空白行视为误用。

Git 还支持对更多空白字符的误用做出检测，包括：

- indent-with-non-tab
  用 8 个或更多的空格进行缩进视为误用。
- tab-in-indent
  在行首的缩进中使用 TAB 字符视为误用。显然这个设置和上面的 indent-with-non-tab 互斥。
- trailing-space
  相当于同时启用 blank-at-eol 和 blank-at-eof。
- cr-at-eol
  将行尾的 CR（回车）字符视为换行符的一部分。也就是说，在行尾前出现的 CR 字符不会引起 trailing-space 报错。
- tabwidth=<n>
  设置一个 TAB 字符相当于几个空格，默认为 8 个。

可以通过 Git 配置文件中的 core.whitespace 配置变量，设置开启更多的空白字符检查，将要开启的空白字符检查项用逗号分开即可。

如果希望对特定路径进行空白字符检查，则可以通过属性 whitespace 进行设置。属性 whitespace 可以有如下设置：

- whitespace
  开启所有的空白字符误用检查。
- -whitespace
  不对空白字符进行误用检查。

① 如果设置 color.ui 配置变量为 true，则针对所有 Git 命令开启颜色输出。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- !whitespace

  使用 core.whitespace 配置变量的设置进行空白字符误用检查。

- whitespace=...

  和 core.whitespace 的语法一样，用逗号分隔各个空白字符检查项。

8. export-ignore

设置了该属性的文件和目录在执行 git archive 时不予导出。

9. export-subst

如果为文件设置了属性 export-subst，则在使用 git archive 导出项目文件时，会展开相应文件内容中的占位符，然后再添加到归档中。注意如果在使用 git archive 导出时使用树 ID，而没有使用提交或里程碑，则不会展开占位符。

占位符的格式为 $Format:PLACEHOLDERS$，其中 PLACEHOLDERS 使用 git log --pretty=format: 相同的参数（具体请参见 git help log 显示的帮助页面）。例如：$Format:%H$ 将展开为提交的哈希值，$Format:%an$ 将展开为提交者姓名。

10. delta

如果设置属性 delta 为 false，则不对该路径指向的 blob 文件执行 Delta 压缩。

11. encoding

设置文件所使用的字符集，以便使用 GUI 工具（如 gitk 和 git-gui）时能够正确显示文件内容。基于性能上的考虑，gitk 默认不检查该属性，除非通过 gitk 的偏好设置启用“Support per-file encodings”。

如果没有为文件设置 encoding 属性，则使用 git.encoding 配置变量。

12. binary

属性 binary 严格来说是一个宏，相当于 -text -diff。即禁止换行符转换，以及禁止以文本方式显示文件差异。

用户也可以自定义宏。自定义宏只能在工作区根目录中的 .gitattributes 文件中添加，以内置的 binary 宏为例，相当于在属性文件中进行了如下的设置：

```
[attr]binary -diff -text
```

## 41.2 钩子和模板

### 41.2.1 Git 钩子

Git 的钩子脚本位于版本库的 .git/hooks 目录下，当 Git 执行特定操作时会调用特定的钩子脚本。当版本库通过 git init 或 git clone 创建时，会在 .git/hooks 目录下创建示例脚本，用户可以参照示例脚本的写法开发适合的钩子脚本。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


钩子脚本要设置为可运行，并使用特定的名称。Git 提供的示例脚本都带有 .sample 扩展名，是为了防止被意外运行。如果需要启用相应的钩子脚本，需要对其重命名（去掉 .sample 扩展名）。下面分别对可用的钩子脚本逐一进行介绍。

### 1. applypatch-msg

该钩子脚本由 git am 命令调用。在调用时向该脚本传递一个参数，即保存有提交说明的文件的文件名。如果该脚本运行失败（返回非零值），则 git am 命令在应用该补丁之前终止。

这个钩子脚本可以修改文件中保存的提交说明，以便规范提交说明以符合项目的标准（如果有的话）。如果提交说明不符合项目标准，脚本直接以非零值退出，则拒绝提交。

Git 提供的示例脚本 applypatch-msg.sample 只是简单地调用 commit-msg 钩子脚本（如果存在的话）。这样通过 git am 命令应用补丁和执行 git commit 一样，都会执行 commit-msg 脚本，因此如需定制，请更改 commit-msg 脚本。

### 2. pre-applypatch

该钩子脚本由 git am 命令调用。该脚本没有参数，在补丁应用后但尚未提交前运行。如果该脚本运行失败（返回非零值），则不会提交已经应用了补丁的工作区文件。

这个脚本可以用于对应用补丁后的工作区进行测试，如果测试没有通过则拒绝提交。

Git 提供的示例脚本 pre-applypatch.sample 只是简单地调用 pre-commit 钩子脚本（如果存在的话）。这样通过 git am 命令应用补丁和执行 git commit 一样都会执行 pre-commit 脚本，因此如需定制，请更改 pre-commit 脚本。

### 3. post-applypatch

该钩子脚本由 git am 命令调用。该脚本没有参数，在补丁应用并且提交之后运行，因此该钩子脚本不会影响 git am 的运行结果，可以用于发送通知。

### 4. pre-commit

该钩子脚本由 git commit 命令调用。可以通过传递 --no-verify 参数而禁用。该脚本在获取提交说明之前运行。如果该脚本运行失败（返回非零值），Git 提交就被终止。

该脚本主要用于对提交数据的检查，例如对文件名进行检查（是否使用了中文文件名），或者对文件内容进行检查（是否使用了不规范的空白字符）。

Git 提供的示例脚本 pre-commit.sample 禁止提交在路径中使用非 ASCII 字符（如中文字符）的文件。如果确有使用的必要，可以在 Git 配置文件中设置配置变量 hooks.allownonascii 为 true 以允许在文件名中使用非 ASCII 字符。Git 提供的该示例脚本也对不规范的空白字符进行检查，如果发现则终止提交。

Topgit 为所管理的版本库设置了自己的 pre-commit 脚本，检查工作的 Topgit 特性分支是否正确地设置了 Topgit 的两个管理文件 .topdeps 和 .topmsg，以及定义的分支依赖是否存在着重复依赖和循环依赖等。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


### 5. prepare-commit-msg

该钩子脚本由 git commit 命令调用，在默认的提交信息准备完成后但编辑器尚未启动之前运行。

该脚本有 1 ～ 3 个参数。第一个参数是包含提交说明的文件的文件名。第二个参数是提交说明的来源，可以是 message（由 -m 或 -F 参数提供），可以是 template（如果使用了 -t 参数或由 commit.template 配置变量提供），或者是 merge（如果提交是一个合并或存在 .git/MERGE_MSG 文件），或者是 squash（如果存在 .git/SQUASH_MSG 文件），或者是 commit 并跟着一个提交 SHA1 哈希值（如果使用 -c、-C 或 --amend 参数）。

如果该脚本运行失败（返回非零值），Git 提交就被终止。

该脚本用于对提交说明进行编辑，并且该脚本不会因为 --no-verify 参数被禁用。

Git 提供的示例脚本 prepare-commit-msg.sample 可以用于向提交说明中嵌入提交者的签名，或者将来自 merge 的提交说明中含有“Conflicts:”的行去掉。

### 6. commit-msg

该钩子脚本由 git commit 命令调用，可以通过传递 --no-verify 参数而禁用。该脚本有一个参数，即包含有提交说明的文件的文件名。如果该脚本运行失败（返回非零值），Git 提交被终止。

该脚本可以直接修改提交说明，可以用于规范提交说明以符合项目的标准（如果有的话）。如果提交说明不符合标准，可以拒绝提交。

Git 提供的示例脚本 commit-msg.sample 检查提交说明中出现的相同的含“Signed-off-by”的行，如果发现重复签名即报错并终止提交。

Gerrit 服务器需要每一个向其进行推送的 Git 版本库在本地使用 Gerrit 提供的 commit-msg 钩子脚本，以便在创建的提交中包含形如“Change-Id: I...”的变更集标签。

### 7. post-commit

该钩子脚本由 git commit 命令调用，不带参数运行，在提交完成之后被触发执行。

该钩子脚本不会影响 git commit 的运行结果，可以用于发送通知。

### 8. pre-rebase

该钩子脚本由 git rebase 命令调用，用于防止某个分支参与变基。

Git 提供的示例脚本 pre-rebase.sample 是针对 Git 项目自身的情况而开发的，当一个功能分支已经合并到 next 分支后，禁止该分支进行变基操作。

### 9. post-checkout

该钩子脚本由 git checkout 命令调用，在完成工作区更新之后触发执行。该钩子脚本有三个参数，分别是：之前的 HEAD 所指向的引用、新的 HEAD 所指向的引用（可能和前一个一样也可能不一样），以及一个用于表示此次检出是否是分支检出的标识（分支检出为 1，文件检出是 0）。该钩子脚本不会影响 git checkout 命令的运行结果。



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


除了由 git checkout 命令调用外，该钩子脚本也在 git clone 命令执行后被触发执行，除非在克隆时使用了禁止检出的 --no-checkout (-n) 参数。在由 git clone 调用时，第一个参数给出的引用是空引用，则第二个和第三个参数都为 1。

这个钩子一般用于版本库的有效性检查，自动显示和前一个 HEAD 的差异，或者设置工作区属性。

10. post-merge

该钩子脚本由 git merge 命令调用，当在本地版本库完成 git pull 操作后触发执行。该钩子脚本有一个参数，标识合并是否是一个压缩合并。该钩子脚本不会影响 git merge 命令的运行结果。如果合并因为冲突而失败，则该脚本不会执行。

该钩子脚本可以与 pre-commit 钩子脚本一起实现对工作区目录树属性（如权限 / 属主 /ACL 等）的保存和恢复。参见 Git 源码文件 contrib/hooks/setgitperms.perl 中的示例。

11. pre-receive

该钩子脚本由远程版本库的 git receive-pack 命令调用。当从本地版本库完成一个推送之后，在远程服务器上开始批量更新引用之前，该钩子脚本被触发执行。该钩子脚本的退出状态决定了更新引用的成功与否。

该钩子脚本在接收（receive）操作中只执行一次。传递参数不通过命令行，而是通过标准输入进行传递。通过标准输入传递的每一行的语法格式为：

```
<old-value> <new-value> <ref-name>
```

<old-value> 是引用更新前的老的对象 ID，<new-value> 是引用即将更新到的新的对象 ID，<ref-name> 是引用的全名。当创建一个新引用时，<old-value> 是 40 个 0。

如果该钩子脚本以非零值退出，一个引用也不会更新。如果该脚本正常退出，每一个单独的引用的更新仍有可能被 update 钩子所阻止。

标准输出和标准错误都重定向到在另外一端执行的 git send-pack 上，所以可以直接通过 echo 命令向用户传递信息。

12. update

该钩子脚本由远程版本库的 git receive-pack 命令调用。当从本地版本库完成一个推送之后，在远程服务器上更新引用时，该钩子脚本被触发执行。该钩子脚本的退出状态决定了更新引用的成功与否。

该钩子脚本在每一个引用更新的时候都会执行一次。该脚本有三个参数。

- 参数 1：要更新的引用的名称。
- 参数 2：引用中保存的旧对象名称。
- 参数 3：将要保存到引用中的新对象名称。

正常退出（返回 0）则允许更新该引用，而以非零值退出则禁止 git-receive-pack

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


更新该引用。

该钩子脚本可以用于防止某些引用被强制更新，因为该脚本可以通过检查新旧引用对象是否存在继承关系，从而提供更为细致的“非快进式推送”的授权。

该钩子脚本也可以用于记录（如用邮件）引用变更历史 old..new。然而因为该脚本不知道完整的引用更新，所以可能会导致每一个引用发送一封邮件。因此如果要发送通知邮件，可能 post-receive 钩子脚本更为适合。

另外，该脚本可以实现基于路径的授权。

标准输出和标准错误都重定向到在另外一端执行的 git send-pack 上，所以可以直接通过 echo 命令向用户传递信息。

Git 提供的示例脚本 update.sample 展示了对多种危险的 Git 操作行为进行控制的可行性。

- 只有将配置变量 hooks.allowunannotated 设置为 true 才允许推送轻量级里程碑（不带说明的里程碑）。
- 只有将配置变量 hooks.allowdeletebranch 设置为 true 才允许删除分支。
- 如果将配置变量 hooks.denycreatebranch 设置为 true 则不允许创建新分支。
- 只有将配置变量 hooks.allowdeletetag 设置为 true 才允许删除里程碑。
- 只有将配置变量 hooks.allowmodifytag 设置为 true 才允许修改里程碑。

相比 Git 的示例脚本，Gitolite 服务器为其管理的版本库设置的 update 钩子脚本更实用也更强大。Gitolite 实现了用户认证，并通过检查授权文件，实现基于分支和路径的写操作授权，等等。具体内容请参见本书第 5 篇“第 30 章 Gitolite 服务架设”的相关内容。

### 13. post-receive

该钩子脚本由远程版本库的 git receive-pack 命令调用。当从本地版本库完成一个推送，并且在远程服务器上的所有引用都更新完毕后，该钩子脚本被触发执行。

该钩子脚本在接收（receive）操作中只执行一次。该脚本不通过命令行传递参数，但是像 pre-receive 钩子脚本那样，通过标准输入以相同格式获取信息。

该钩子脚本不会影响 git-receive-pack 的结果，因为调用该脚本时工作已经完成。

该钩子脚本胜过 post-update 脚本之处在于：它可以获得所有引用的老的和新的值，以及引用的名称。

标准输出和标准错误都重定向到在另外一端执行的 git send-pack 上，所以可以直接通过 echo 命令向用户传递信息。

Git 提供的示例脚本 post-receive.sample 引入了 contrib/hooks 目录下的名为 post-receive-email 的示例脚本（默认被注释），以实现发送通知邮件的功能。

Gitolite 服务器要对其管理的 Git 版本库设置 post-receive 钩子脚本，以实现当版本库有变更后将数据传输到各个镜像版本库。

### 14. post-update

该钩子脚本由远程版本库的 git receive-pack 命令调用。当从本地版本库完成一个



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


推送之后，即当所有引用都更新完毕后，在远程服务器上该钩子脚本被触发执行。

该脚本接收不定长的参数，每个参数实际上就是已经成功更新的引用名。

该钩子脚本不会影响 git-receive-pack 的结果，因此主要用于通知。

钩子脚本 post-update 虽然能够提供哪些引用被更新了，但是该脚本不知道引用更新前后的对象 SHA1 哈希值，所以在这个脚本中不能记录形如 old..new 的引用变更范围。而钩子脚本 post-receive 知道引用更新前后的对象 ID，因此更适合于此种场合。

标准输出和标准错误都重定向到在另外一端执行的 git send-pack 上，所以可以直接通过 echo 命令向用户传递信息。

Git 提供的示例脚本 post-update.sample 会运行 git update-server-info 命令，以更新哑协议需要的索引文件。如果通过哑协议共享版本库，应该启用该钩子脚本。

#### 15. pre-auto-gc

该钩子脚本由 git gc --auto 命令调用，不带参数运行，如果以非零值退出会导致 git gc --auto 被中断。

#### 16. post-rewrite

该钩子脚本由一些重写提交的命令调用，如 git commit --amend、git rebase，而 git-filter-branch 当前尚未调用该钩子脚本。

该脚本的第一个参数用于判断调用来自哪个命令，当前有 amend 和 rebase 两个取值，也可能将来会传递其他的更多命令相关的参数。

该脚本通过标准输入接收一个重写提交列表，每一行输入的格式如下：

```
<old-sha1> <new-sha1> [<extra-info>]
```

前两个是旧的和新的对象 SHA1 哈希值。而 <extra-info> 参数是和调用命令相关的，当前该参数为空。

### 41.2.2 Git 模板

当执行 git init 或 git clone 创建版本库时，会自动在版本库中创建钩子脚本（.git/hooks/*）、忽略文件（.git/info/exclude）及其他文件，实际上这些文件均拷贝自模板目录。如果需要本地版本库使用定制的钩子脚本等文件，直接在模板目录内创建（文件或符号链接）会事半功倍。

Git 按照下列顺序第一个确认的路径即为模板目录。

（1）如果执行 git init 或 git clone 命令时，提供 --template=<DIR> 参数，则使用指定的目录作为模板目录。

（2）由环境变量 $GIT_TEMPLATE_DIR 指定的模板目录。

（3）由 Git 配置变量 init.templatedir 指定的模板目录。

（4）默认的模板目录，根据 Git 安装路径的不同可能位于不同的目录下。可以通过下面

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


命令确认其实际位置：

```
$ echo $(dirname $(dirname $(git --html-path)))/git-core/templates
/usr/share/git-core/templates
```

如果在执行版本库初始化时传递了空的模板路径，则不会在版本库中创建钩子脚本等文件。

```
$ git init --template=    simplegit
Initialized empty Git repository in /path/to/my/workspace/simplegit/.git/
```

执行下面的命令，查看新创建的版本库 .git 目录下的文件。

```
$ ls -F simplegit/.git/
HEAD     config   objects/ refs/
```

可以看到不使用模板目录创建的版本库下面的文件少的可怜。而通过对模板目录下的文件的定制，可以使得在建立的版本库中包含预先设置好的钩子脚本、忽略文件、属性文件甚至 config 配置文件等。这给对服务器或对版本库操作有特殊要求的项目带来了方便。

## 41.3 稀疏检出和浅克隆

### 41.3.1 稀疏检出

从 1.7.0 版本开始 Git 提供稀疏检出的功能。所谓稀疏检出就是，本地版本库检出时不检出全部，只将指定的文件从本地版本库中检出到工作区，而其他未指定的文件则不予检出（即使这些文件存在于工作区，其修改也会被忽略）。

要想实现稀疏检出的功能，必须同时设置 core.sparseCheckout 配置变量，并存在文件 .git/info/sparse-checkout。即首先要设置 Git 配置变量 core.sparseCheckout 为 true，然后编辑 .git/info/sparse-checkout 文件，将要检出的目录或文件的路径写入其中。其中文件 .git/info/sparse-checkout 的格式就和 .gitignore 文件格式一样，路径可以使用通配符。

稀疏检出是如何实现的呢？实际上 Git 在 index（即暂存区）中为每个文件提供一个名为 skip-worktree 的标志位，默认这个标志位处于关闭状态。如果开启该标志位，则无论工作区对应的文件存在与否，或者是否被修改，Git 都认为工作区该文件的版本是最新的、无变化的。Git 通过配置文件 .git/info/sparse-checkout 定义一个要检出的目录和/或文件列表，当前 Git 的 git read-tree 命令及其他基于合并的命令（git merge、git checkout 等）能够根据该配置文件更新的 index 中文件的 skip-worktree 标志位，实现版本库文件的稀疏检出。

先在工作区 /path/to/my/workspace 中创建一个示例版本库 sparse1，创建后的 sparse1 版本库中包含如下内容：

```
$ ls -F
```

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
doc1/ doc2/ doc3/
$ git ls-files -s -v
H 100644 ce013625030ba8dba906f756967f9e9ca394464a 0      doc1/readme.txt
H 100644 ce013625030ba8dba906f756967f9e9ca394464a 0      doc2/readme.txt
H 100644 ce013625030ba8dba906f756967f9e9ca394464a 0      doc3/readme.txt
```

即版本库 sparse1 中包含三个目录 doc1、doc2 和 doc3。命令 git ls-files 的 -s 参数用于显示对象的 SHA1 哈希值及所处的暂存区的编号。而 -v 参数则显示工作区文件的状态，每一行命令输出的第一个字符即是文件状态：字母 H 表示文件已被暂存，如果是字母 S 则表示该文件 skip-worktree 的标志位已开启。

下面我们就来体验一下稀疏检出的功能。

（1）修改版本库的 Git 配置变量 core.sparseCheckout，将其设置为 true。

```
$ git config core.sparseCheckout true
```

（2）设置 .git/info/sparse-checkout 的内容，如下：

```
$ printf "doc1\ndoc3\n" > .git/info/sparse-checkout
$ cat .git/info/sparse-checkout
doc1
doc3
```

（3）执行 git checkout 命令后，会发现工作区中的 doc2 目录不见了。

```
$ git checkout
$ ls -F
doc1/ doc3/
```

（4）这时如果用 git ls-files 命令查看，会发现 doc2 目录下的文件被设置了 skip-worktree 标志。

```
$ git ls-files -v
H doc1/readme.txt
S doc2/readme.txt
H doc3/readme.txt
```

（5）修改 .git/info/sparse-checkout 的内容，如下：

```
$ printf "doc3\n" > .git/info/sparse-checkout
$ cat .git/info/sparse-checkout
doc3
```

（6）执行 git checkout 命令后，会发现工作区中的 doc1 目录也不见了。

```
$ git checkout
$ ls -F
doc3/
```

（7）这时如果用 git ls-files 命令查看，会发现 doc1 和 doc2 目录下的文件都被设

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


置了 skip-worktree 标志。

```
$ git ls-files -v
S doc1/readme.txt
S doc2/readme.txt
H doc3/readme.txt
```

（8）修改 .git/info/sparse-checkout 的内容，使之包含一个星号，即在工作区检出所有的内容。

```
$ printf "*\n" > .git/info/sparse-checkout
$ cat .git/info/sparse-checkout
*
```

（9）执行 git checkout，会发现所有目录又都回来了。

```
$ git checkout
$ ls -F
doc1/ doc2/ doc3/
```

文件 .git/info/sparse-checkout 的格式类似于 .gitignore 的格式，也支持用感叹号实现反向操作。例如不检出目录 doc2 下的文件，而检出其他文件，可以使用下面的语法（注意顺序不能写反）：

```
*
!doc2/
```

注意如果使用命令 git checkout -- <file>...，即不是切换分支而是用分支中的文件替换暂存区和工作区，则忽略 skip-worktree 标志。例如下面的操作中，虽然 doc2 被设置为不检出，但是执行 git checkout . 命令后，所有的目录还是都被检出了。

```
$ git checkout .
$ ls -F
doc1/ doc2/ doc3/
$ git ls-files -v
H doc1/readme.txt
S doc2/readme.txt
H doc3/readme.txt
```

如果修改 doc2 目录下的文件，或者在 doc2 目录下添加新文件，Git 会视而不见。

```
$ echo hello >> doc2/readme.txt
$ git status
# On branch master
nothing to commit (working directory clean)
```

若此时通过取消 core.sparseCheckout 配置变量的设置而关闭稀疏检出，也不会改变目录 doc2 下的文件的 skip-worktree 标志。这种情况或者通过 git update-index --no-skip-worktree -- <file>... 来更改 index 中对应文件的 skip-worktree 标



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


志，或者重新启用稀疏检出更改相应文件的检出状态。

如果在克隆一个版本库时只希望检出部分文件或目录，可以在执行克隆操作的时候使用 --no-checkout 或 -n 参数，不进行工作区文件的检出。例如下面的操作从前面示例的 sparse1 版本库克隆到 sparse2 中，不进行工作区文件的检出。

```
$ git clone -n sparse1 sparse2
Cloning into sparse2...
done.
```

检出完成后可以发现 sparse2 的工作区是空的，而且版本库中也不存在 index 文件。如果执行 git status 命令会看到所有文件都被标识为删除。

```
$ cd sparse2
$ git status -s
D  doc1/readme.txt
D  doc2/readme.txt
D  doc3/readme.txt
```

如果希望通过稀疏检出的功能，只检出其中一个目录如 doc2，可以用如下方法实现：

```
$ git config core.sparseCheckout true
$ printf "doc2\n" > .git/info/sparse-checkout
$ git checkout
```

之后看到工作区中检出了 doc2 目录，而其他文件被设置了 skip-worktree 标志。

```
$ ls -F
doc2/
$ git ls-files -v
S doc1/readme.txt
H doc2/readme.txt
S doc3/readme.txt
```

### 41.3.2 浅克隆

上一节介绍的稀疏检出，可以部分检出版本库中的文件，但是版本库本身仍然包含所有的文件和历史。如果只对一个大的版本库的最近的部分历史提交感兴趣，而不想克隆整个版本库，稀疏检出是解决不了这个问题的，应采用本节介绍的浅克隆。

实现版本库的浅克隆非常简单，只需要在执行 git clone 或 git fetch 操作时用 --depth <depth> 参数设定要获取的历史提交的深度（<depth> 大于 0），就会把源版本库分支上最近的 <depth> + 1 个历史提交作为新版本库的全部历史提交。

通过浅克隆方式克隆出来的版本库，每一个提交的 SHA1 哈希值和源版本库都相同，包括提交的根节点也是如此，但是 Git 通过特殊的实现，使得浅克隆的根节点提交看起来没有父提交。正因为浅克隆的提交对象的 SHA1 哈希值和源版本库一致，所以浅克隆版本库可以执行 git fetch 或 git pull 从源版本库获取新的提交。但是浅克隆版本库也存在着很多限制，如：

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


- 不能从浅克隆版本库克隆出新的版本库。
- 其他版本库不能从浅克隆版本库中获取提交。
- 其他版本库不能推送提交到浅克隆版本库。
- 不要从浅克隆版本库推送提交至其他版本库，除非确认推送的目标版本库包含浅克隆版本库中缺失的全部历史提交，否则会因为目标版本库包含不完整的提交历史而导致版本库无法操作。
- 在浅克隆版本库中执行合并操作时，如果所合并的提交出现在浅克隆版本库的历史中，则可以顺利合并，否则会出现大量的冲突，就好像和无关的历史进行合并一样。

由于浅克隆包含上述限制，因此浅克隆一般用于对远程版本库的查看和研究，如果在浅克隆版本库中进行了提交，最好通过 `git format-patch` 命令导出为补丁文件再应用到远程版本库中。

下面的操作使用 `git clone` 命令创建一个浅克隆。注意：源版本库如果是本地版本库，就要使用 `file://` 协议，若直接使用本地路径则不会实现浅克隆。

```
$ git clone --depth 2 file:///path/to/repos/hello-world.git shallow1
```

然后进入到本地克隆目录中，会看到当前分支上只有 3 个提交。

```
$ git log  --oneline
c4acab2 Translate for Chinese.
683448a Add I18N support.
d81896e Fix typo: -help to --help.
```

查看提交的根节点 `d81896e`，会看到该提交实际上也包含父提交。

```
$ git cat-file -p HEAD^^
tree f9d7f6b0af6f3fffa74eb995f1d781d3c4876b25
parent 10765a7ef46981a73d578466669f6e17b73ac7e3
author user1 <user1@sun.ossxp.com> 1294069736 +0800
committer user2 <user2@moon.ossxp.com> 1294591238 +0800

Fix typo: -help to --help.
```

而查看该提交的父提交，Git 会报错。

```
$ git log 10765a7ef46981a73d578466669f6e17b73ac7e3
fatal: bad object 10765a7ef46981a73d578466669f6e17b73ac7e3
```

对于正常的 Git 版本库来说，如果对象库中丢失一个提交绝对是大问题，版本库不可能被正常使用。而浅克隆之所以看起来一切正常，是因为 Git 使用了类似嫁接（下一节即将介绍）的技术。

在浅克隆版本库中存在一个文件 `.git/shallow`，这个文件中罗列了应该被视为提交根节点的提交 SHA1 哈希值。查看这个文件会看到提交 `d81896e` 正在其中：

```
$ cat .git/shallow
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
b56bb510a947651e4717b356587945151ac32166
d81896e60673771ef1873b27a33f52df75f70515
e64f3a216d346669b85807ffcfb23a21f9c5c187
```

列在 .git/shallow 文件中的提交会构建出对应的嫁接提交，使用类似嫁接文件 .git/info/grafts（下节讨论）的机制，当 Git 访问这些对象时就好像这些对象是没有父提交的根节点一样。

## 41.4 嫁接和替换

### 41.4.1 提交嫁接

提交嫁接可以实现在本地版本库上将两条完全不同的提交线（分支）嫁接（连接）到一起。对于一些在迁移版本控制系统时遇到困难的项目，该技术会非常有帮助。例如 Linux kernel 本身的版本控制系统在迁移到 Git 上时，尚没有任何工具可以将 Linux 的提交历史从旧的 Bitkeeper 版本控制系统中导出，直到后来通过 bkcvs 将 Bitkeeper 上的 Linux 历史提交导入到 Git 中。如何将新旧两条开发线连接到一起呢？于是就发明了提交嫁接，以实现新旧两条开发线的合并，这样 Linux 开发者就可以在一个开发分支中由最新的提交追踪到原来位于 Bitkeeper 中的提交[①]。

提交嫁接是通过在版本库中创建 .git/info/grafts 文件来实现的。该文件每一行的格式为：

```
<commit sha1> <parent sha1> [<parent sha1>]*
```

用空格分开各个字段，其中第一个字段是一个提交的 SHA1 哈希值，而后面用空格分开的其他 SHA1 哈希值则作为该提交的父提交。把一个提交线的根节点作为第一个字段，第二个提交线的顶节点作为第二个字段，就实现了两个提交线的嫁接，看起来就像是一条提交线了。

在本书第 6 篇第 35 章的“35.4 Git 版本库整理”中介绍的 git filter-branch 命令在整理版本库时，如果发现存在 .git/info/grafts 则会在物理上完成提交的嫁接，实现嫁接的永久生效。

### 41.4.2 提交替换

提交替换是在 1.6.5 或更新版本的 Git 中提供的功能，和提交嫁接类似，不过提交替换不是用一个提交来伪装成另外一个提交的父提交，而是直接替换另外的提交，在不影响其他提交的基础上实现对历史提交的修改。

提交替换是通过在特殊命名空间 .git/refs/replace/ 下定义引用来实现的。引用

① *https://git.wiki.kernel.org/index.php/GraftPoint*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的名称是要被替换掉的提交 SHA1 哈希值，而引用文件的内容（引用所指向的提交）就是用于替换的（正确的）提交 SHA1 哈希值。由于提交替换通过引用进行定义，因此可以在不同的版本库之间传递，而不像提交嫁接只能在本地版本库中使用。

Git 提供 git replace 命令来管理提交替换，用法如下：

```
用法 1：git replace [-f] <object> <replacement>
用法 2：git replace -d <object>...
用法 3：git replace -l [<pattern>]
```

其中：

- 用法 1 用于创建提交替换，即在 .git/refs/replace 目录下创建名为 <object> 的引用，其内容为 <replacement>。如果使用 -f 参数，还允许级联替换，即用于替换的提交可以是另外一个已经在 .git/refs/replace 中定义的替换。
- 用法 2 用于删除已经定义的替换。
- 用法 3 显示已经存在的提交替换。

提交替换可以被大部分 Git 命令理解，除了一些针对被替换的提交使用 --no-replace-objects 参数的命令。例如：

- 当提交 foo 被提交 bar 替换后，显示未被替换前的 foo 提交：

```
$ git --no-replace-objects cat-file commit foo
...foo 的内容...
```

- 不使用 --no-replace-objects 参数，则访问 foo 会显示替换后的 bar 提交：

```
$ git cat-file commit foo
...bar 的内容...
```

提交替换使用引用进行定义，因此可以通过 git fetch 和 git push 在版本库之间传递。但因为默认 Git 只同步里程碑和分支，因此需要在命令中显式地给出提交替换的引用表达式，如：

```
$ git fetch origin refs/replace/*
$ git push  origin refs/replace/*
```

提交替换也可以实现两个分支的嫁接。例如要将分支 A 嫁接到 B 上，就相当于将分支 A 的根提交 <BRANCH_A_ROOT> 的父提交设置为分支 B 的最新提交 <BRANCH_B_CURRENT>。可以先创建一个新提交 <BRANCH_A_NEW_ROOT>，其父提交设置为 <BRANCH_B_CURRENT> 而提交的其他字段和 <BRANCH_A_ROOT> 完全一致。然后设置提交替换，用 <BRANCH_A_NEW_ROOT> 替换 <BRANCH_A_ROOT> 即可。

可以使用下面的命令创建 <BRANCH_A_NEW_ROOT>，注意要用实际值替换下面命令中的 <BRANCH_A_ROOT> 和 <BRANCH_B_CURRENT>。

```
$ git cat-file commit <BRANCH_A_ROOT> |
  sed -e "/^tree / a \
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


```
        parent $(git rev-parse <BRANCH_B_CURRENT>)" |
git hash-object -t commit -w --stdin
```

其中 git cat-file commit 命令用于显示提交的原始信息，sed 命令用于向原始提交中插入一条 parent SHA1... 的语句，而命令 git hash-object 是一个 Git 底层命令，可以将来自标准输入的内容创建为一个新的提交对象。

上面命令的输出即是 <BRANCH_A_NEW_ROOT> 的值。执行下面的替换命令，完成两个分支的嫁接。

```
$ git replace <BRANCH_A_ROOT> <BRANCH_A_NEW_ROOT>
```

## 41.5 Git 评注

从 1.6.6 版本开始，Git 提供了一个 git notes 命令可以为提交添加评注，在不改变提交对象的情况下，实现在提交说明的后面附加评注。图 41-1 展示了 Github[①]利用 git notes 实现的显示评注（如果存在的话）和添加评注的界面。

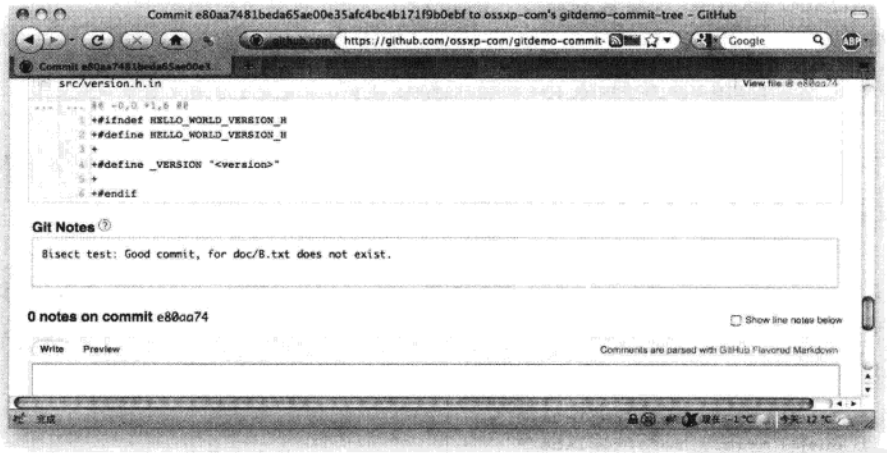



图 41-1 Github 上显示和添加评注

### 41.5.1 评注的奥秘

实际上 Git 的评注可以针对任何对象，而且评注的内容也不限于文字，因为评注的内容保存在 Git 对象库中的一个 blob 对象中。不过评注目前最主要的应用还是在提交说明后添加

① *https://github.com/ossxp-com/gitdemo-commit-tree/commit/6652a0dce6a5067732c00ef0a220810a7230655e*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


文字评注。

在第 2 篇第 11 章的“11.4.6 二分查找”中用到的 gitdemo-commit-tree 版本库实际上就包含了提交评注，只不过之前尚未将评注获取到本地版本库而已。如果工作区中的 gitdemo-commit-tree 版本库已经不存在了，可以使用下面的命令从 Github 上再克隆一个：

```
$ git clone -q git://github.com/ossxp-com/gitdemo-commit-tree.git
$ cd gitdemo-commit-tree
```

执行下面的命令，查看最后一次提交的提交说明：

```
$ git log -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 9 16:07:11 2010 +0800

    Add Images for git treeview.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
```

下面为默认的 origin 远程版本库再添加一个引用获取表达式，以便在执行 git fetch 命令时能够同步评注相关的引用。命令如下：

```
$ git config --add remote.origin.fetch refs/notes/*:refs/notes/*
```

执行 git fetch 会获取到一个新的引用 refs/notes/commits，如下：

```
$ git fetch
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From git://github.com/ossxp-com/gitdemo-commit-tree
 * [new branch]      refs/notes/commits -> refs/notes/commits
```

当获取新的评注相关的引用之后，再来查看最后一次提交的提交说明。下面的命令输出中，提交说明的最后两行就是附加的提交评注。

```
$ git log -1
commit 6652a0dce6a5067732c00ef0a220810a7230655e
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Thu Dec 9 16:07:11 2010 +0800

    Add Images for git treeview.

    Signed-off-by: Jiang Xin <jiangxin@ossxp.com>

Notes:
    Bisect test: Bad commit, for doc/B.txt exists.
```



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


附加的提交评注来自于哪里呢？显然应该和刚刚获取到的引用相关。查看一下获取到的最新引用，会发现引用 refs/notes/commits 指向的是一个提交对象。

```
$ git show-ref refs/notes/commits
6f01cdc590048927411193l8ceb2330d6dc0cef1 refs/notes/commits
$ git cat-file -t refs/notes/commits
commit
```

既然新获取的评注引用是一个提交对象，那么就应该能够查看评注引用的提交日志：

```
$ git log --stat refs/notes/commits
commit 6f01cdc590048927411193l8ceb2330d6dc0cef1
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Feb 22 09:32:10 2011 +0800

    Notes added by 'git notes add'

 6652a0dce6a5067732c00ef0a220810a7230655e |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

commit 9771e1076d2218922acc9800f23d5e78d5894a9f
Author: Jiang Xin <jiangxin@ossxp.com>
Date:   Tue Feb 22 09:31:54 2011 +0800

    Notes added by 'git notes add'

 e80aa7481beda65ae00e35afc4bc4b171f9b0ebf |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

从上面的评注引用的提交日志可以看出，存在两次提交，并且从提交说明可以看出是使用 git notes add 命令添加的。至于每次提交添加的文件却很让人困惑，所添加文件的文件名居然是 40 位的哈希值。

您当然可以通过 git checkout -b 命令检出该引用来研究其中所包含的文件，不过也可以运用我们已经学习到的 Git 命令直接对其进行研究。

- 用 git show 命令显示目录树。

```
$ git show -p refs/notes/commits^{tree}
tree refs/notes/commits^{tree}

6652a0dce6a5067732c00ef0a220810a7230655e
e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
```

- 用 git ls-tree 命令查看文件大小及对应的 blob 对象的 SHA1 哈希值。

```
$ git ls-tree -l refs/notes/commits
100644 blob 80b1d2……      47    6652a0……
100644 blob e894f2……      56    e80aa7……
```

- 文件名既然是一个 40 位的 SHA1 哈希值，那么文件名一定有意义，通过下面的命令

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


可以看到文件名包含的40位哈希值实际对应于一个提交。

```
$ git cat-file -p 6652a0dce6a5067732c00ef0a220810a7230655e
tree e33be9e8e7ca5f887c7d5601054f2f510e6744b8
parent 81993234fc12a325d303eccea20f6fd629412712
author Jiang Xin <jiangxin@ossxp.com> 1291882031 +0800
committer Jiang Xin <jiangxin@ossxp.com> 1291882892 +0800

Add Images for git treeview.

Signed-off-by: Jiang Xin <jiangxin@ossxp.com>
```

- 用 git cat-file 命令查看该文件的内容，可以看到其内容就是附加在相应提交上的评注。

```
$ git cat-file -p
refs/notes/commits:6652a0dce6a5067732c00ef0a220810a7230655e
Bisect test: Bad commit, for doc/B.txt exists.
```

综上所述，评注记录在一个 blob 对象中，并且以所评注对象的 SHA1 哈希值命名。因为对象 SHA1 哈希值的唯一性，所以可以将评注都放在同一个文件系统下而不会相互覆盖。针对这个包含所有评注的、特殊的文件系统的更改被提交到一个特殊的引用 refs/notes/commits 当中。

## 41.5.2 评注相关命令

Git 提供了 git notes 命令对评注进行管理。如果执行 git notes list，或者像下面这样不带任何参数进行调用，会显示和上面 git ls-tree 类似的输出：

```
$ git notes
80b1d249069959ce5d83d52ef7bd0507f774c2b0 6652a0dce6a5067732c00ef0a220810a7230655e
e894f2164e77abf08d95d9bdad4cd51d00b47845 e80aa7481beda65ae00e35afc4bc4b171f9b0ebf
```

右边的一列是要评注的提交对象，而左边一列是附加在对应提交上的包含评注内容的 blob 对象。显示附加在某个提交上的评注可以使用 git notes show 命令。如下：

```
$ git notes show G^0
Bisect test: Good commit, for doc/B.txt does not exist.
```

注意上面的命令中使用 G^0 而非 G，是因为 G 是一个里程碑对象，而评注是建立在由里程碑对象所指向的一个提交对象上。

添加评注可以使用下面的 git notes add 和 git notes append 子命令：

```
用法 1：git notes add [-f] [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]
用法 2：git notes append [-F <file> | -m <msg> | (-c | -C) <object>] [<object>]
```

用法 1 是添加评注，而用法 2 是在已有评注后面追加（修改已有评注）。两者的命令行格式和 git commit 非常类似，可以用类似写提交说明的方法写提交评注。如果省略最后一个是 <object> 参数，则意味着向头指针 HEAD 添加评注。子命令 git notes add 中



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


的参数 -f 意味着强制添加，会覆盖对象中已有的评注。

使用 git notes copy 子命令可以将一个对象的评注拷贝到另外一个对象上。

```
用法：git notes copy [-f] ( --stdin | <from-object> <to-object> )
```

修改评注可以使用下面的 git notes edit 子命令：

```
用法：git notes edit [<object>]
```

删除评注可以使用的 git notes remote 子命令，而 git notes prune 则可以清除已经不存在的对象上的评注。用法如下：

```
用法 1：git notes remove [<object>]
用法 2：git notes prune [-n | -v]
```

评注以文件形式保存在特殊的引用中，如果该引用被共享并且同时有多人撰写评注，则可能出现该引用的合并冲突。可以用 git notes merge 命令来解决合并冲突。评注引用也可以使用其他的引用名称，合并其他的评注引用也可以使用本命令。下面是 git notes merge 命令的语法格式，具体操作请参见 git help notes 帮助。

```
用法 1：git notes merge [-v | -q] [-s <strategy> ] <notes_ref>
用法 2：git notes merge --commit [-v | -q]
用法 3：git notes merge --abort [-v | -q]
```

### 41.5.3 评注相关配置

默认提交评注保存在引用 refs/notes/commits 中，这个默认的设置可以通过 core.notesRef 配置变量来修改。如需更改，要在 core.notesRef 配置变量中使用引用的全称而不能使用缩写。

在执行 git log 命令显示提交评注的时候，如果配置了 notes.displayRef 配置变量（可以使用通配符，并且可以配置多个），则在显示提交评注时，除了会参考 core.notesRef 设定的引用（或默认的 refs/notes/commits 引用）外，还会参考 notes.displayRef 指向的引用（一个或多个）来显示评注。

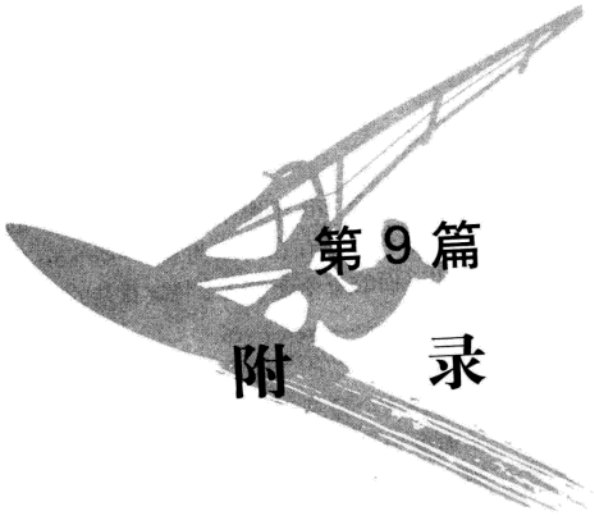
配置变量 notes.rewriteRef 用于配置哪个/哪些引用中的提交评注会随着提交的修改而复制到新的提交之上。这个配置变量可以使用多次，或者使用通配符，但该配置变量没有默认值，因此为了使得提交评注能够随着提交的修改（修补提交、变基等）而继续保持，必须对该配置变量进行设定。如：

```
$ git config --global notes.rewriteRef refs/notes/*
```

还有 notes.rewrite.amend 和 notes.rewrite.rebase 配置变量可以分别对两种提交修改模式（amend 和 rebase）是否启用评注复制进行设置，默认启用。配置变量 notes.rewriteMode 默认设置为 concatenate，即提交评注复制到修改后的提交时，如果已有评注则对评注进行追加操作。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



# 第9篇 附 录

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 附录 A Git 命令索引

每一个 Git 子命令都和特定目录下的一个名为 git-<cmd> 的文件相对应，也就是在这个特定目录下存在的名为 git-<cmd> 的可执行文件[①]可以用命令 git <cmd> 来执行。可以用下面的命令查看这个特定目录的位置：

```
$ git --exec-path
/usr/lib/git-core/
```

在这个目录下有 150 多个可执行文件，也就是说 Git 有非常多的子命令。在如此众多的子命令中，常用的实际上只有不到 1/3，其余的命令或者作为底层命令供其他命令及脚本调用，或者用于某些生僻场合，或者已经过时但出于兼容性的考虑而仍然保留。下面的表格分门别类地对所有的 Git 命令进行概要性的介绍，凡是在本书出现过的命令将标出其所在的章节号和页码。

## A.1 常用的 Git 命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git add | 4.1；10.2.3；10.4 | 59；117；118 | 添加至暂存区 |
| git add--interactive | 10.6 | 122 | 交互式添加 |
| git apply | 38.1 | 534 | 应用补丁 |
| git am | 20.2 | 298 | 应用邮件格式补丁 |
| git annotate | — | — | 同义词，等同于 git blame |
| git archive | 10.9 | 129 | 文件归档打包 |
| git bisect | 11.4.6 | 152 | 二分查找 |
| git blame | 11.4.5 | 151 | 文件逐行追溯 |
| git branch | 18.2 | 258 | 分支管理 |
| git cat-file | 6.1 | 83 | 版本库对象研究工具 |
| git checkout | 8.1；18.4.2 | 99；262 | 检出到工作区、切换或创建分支 |
| git cherry-pick | 12.3.1 | 162 | 提交拣选 |
| git citool | 11.1 | 131 | 图形化提交，相当于 git gui 命令 |
| git clean | 5.3 | 78 | 清除工作区未跟踪文件 |
| git clone | 13.1；41.3.2 | 180；567 | 克隆版本库 |

① 其中有几个脚本不能单独运行，而是被其他脚本包含，用于提供相应的函数库，如 git-sh-setup。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git commit | 4.1；4.4；5.4 | 60；65；81 | 提交 |
| git config | 4.3 | 63 | 查询和修改配置 |
| git describe | 17.1 | 235 | 通过里程碑直观地显示提交 ID |
| git diff | 5.3 | 80 | 差异比较 |
| git difftool | — | — | 调用图形化差异比较工具 |
| git fetch | 19.1 | 286 | 获取远程版本库的提交 |
| git format-patch | 20.1 | 296 | 创建邮件格式的补丁文件。参见 git am 命令 |
| git grep | 4.2 | 61 | 文件内容搜索定位工具 |
| git gui | 11.1 | 131 | 基于 Tcl/Tk 的图形化工具，侧重提交等操作 |
| git help | 3.1.2 | 23 | 帮助 |
| git init | 4.1；13.4 | 59；185 | 版本库初始化 |
| git init-db | — | — | 同义词，等同于 git init |
| git log | 11.4.3 | 147 | 显示提交日志 |
| git merge | 16.1 | 211 | 分支合并 |
| git mergetool | 16.4.2 | 222 | 图形化冲突解决 |
| git mv | 10.4 | 119 | 重命名 |
| git pull | 13.1 | 180 | 拉回远程版本库的提交 |
| git push | 13.1 | 180 | 推送至远程版本库 |
| git rebase | 12.3.2 | 167 | 分支变基 |
| git rebase--interactive | 12.3.3 | 171 | 交互式分支变基 |
| git reflog | 7.2 | 95 | 分支等引用变更记录管理 |
| git remote | 19.3 | 290 | 远程版本库管理 |
| git repo-config | — | — | 同义词，等同于 git config |
| git reset | 7.1 | 94 | 重置改变分支“游标”指向 |
| git rev-parse | 4.2；11.4.1 | 62；141 | 将各种引用表示法转换为哈希值等 |
| git revert | 12.5 | 177 | 反转提交 |
| git rm | 10.2.2 | 116 | 删除文件 |
| git show | 11.4.3 | 149 | 显示各种类型的对象 |
| git stage | — | — | 同义词，等同于 git add |
| git stash | 9.2 | 108 | 保存和恢复进度 |
| git status | 5.1 | 71 | 显示工作区文件状态 |
| git tag | 17.1 | 234 | 里程碑管理 |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## A.2 对象库操作相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git commit-tree | 12.4 | 175 | 从树对象创建提交 |
| git hash-object | 12.4 | 176 | 从标准输入或文件计算哈希值或创建对象 |
| git ls-files | 5.3；41.3.1 | 79；564 | 显示工作区和暂存区文件 |
| git ls-tree | 5.3 | 79 | 显示树对象包含的文件 |
| git mktag | — | — | 读取标准输入创建一个里程碑对象 |
| git mktree | — | — | 读取标准输入创建一个树对象 |
| git read-tree | 24.2 | 349 | 读取树对象到暂存区 |
| git update-index | 41.3.1 | 565 | 工作区内容注册到暂存区及暂存区管理 |
| git unpack-file | — | — | 创建临时文件包含指定 blob 的内容 |
| git write-tree | 5.3 | 79 | 从暂存区创建一个树对象 |

## A.3 引用操作相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git check-ref-format | 17.7 | 248 | 检查引用名称是否符合规范 |
| git for-each-ref | — | — | 引用迭代器，用于 shell 编程 |
| git ls-remote | 13.4 | 186 | 显示远程版本库的引用 |
| git name-rev | 17.1 | 236 | 将提交 ID 显示为友好名称 |
| git peek-remote | — | — | 过时命令，请使用 git ls-remote |
| git rev-list | 11.4.2 | 144 | 显示版本范围 |
| git show-branch | — | — | 显示分支列表及拓扑关系 |
| git show-ref | 14.1 | 187 | 显示本地引用 |
| git symbolic-ref | — | — | 显示或设置符号引用 |
| git update-ref | — | — | 更新引用的指向 |
| git verify-tag | — | — | 校验 GPG 签名的 Tag |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## A.4 版本库管理相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git count-objects | — | — | 显示松散对象的数量和磁盘占用 |
| git filter-branch | 35.4 | 511 | 版本库重构 |
| git fsck | 14.2 | 190 | 对象库完整性检查 |
| git fsck-objects | — | — | 同义词，等同于 git fsck |
| git gc | 14.4 | 193 | 版本库存储优化 |
| git index-pack | — | — | 从打包文件创建对应的索引文件 |
| git lost-found | — | — | 过时，请使用 git fsck --lost-found 命令 |
| git pack-objects | — | — | 从标准输入读入对象 ID，打包到文件 |
| git pack-redundant | — | — | 查找多余的 pack 文件 |
| git pack-refs | 14.1 | 188 | 将引用打包到 .git/packed-refs 文件中 |
| git prune | 14.2 | 190 | 从对象库删除过期对象 |
| git prune-packed | — | — | 将已经打包的松散对象删除 |
| git relink | — | — | 为本地版本库中相同的对象建立硬连接 |
| git repack | 14.4 | 193 | 将版本库未打包的松散对象打包 |
| git show-index | 14.1 | 188 | 读取包的索引文件，显示打包文件中的内容 |
| git unpack-objects | — | — | 从打包文件释放文件 |
| git verify-pack | — | — | 校验对象库打包文件 |

## A.5 数据传输相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git fetch-pack | 15.1 | 201 | 执行 git fetch 或 git pull 命令时在本地执行此命令，用于从其他版本库获取缺失的对象 |
| git receive-pack | 15.1 | 201 | 执行 git push 命令时在远程执行的命令，用于接受推送的数据 |
| git send-pack | 15.1 | 201 | 执行 git push 命令时在本地执行的命令，用于向其他版本库推送数据 |
| git upload-archive | — | — | 执行 git archive --remote 命令基于远程版本库创建归档时，远程版本库执行此命令传送归档 |
| git upload-pack | 15.1 | 201 | 执行 git fetch 或 git pull 命令时在远程执行此命令，将对象打包、上传 |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## A.6 邮件相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git imap-send | — | — | 将补丁通过 IMAP 发送 |
| git mailinfo | — | — | 从邮件导出提交说明和补丁 |
| git mailsplit | — | — | 将 mbox 或 Maildir 格式邮箱中的邮件逐一提取为文件 |
| git request-pull | 21.2.1 | 313 | 创建包含提交间差异和执行 PULL 操作地址的信息 |
| git send-email | 20.1 | 297 | 发送邮件 |

## A.7 协议相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git daemon | 28.2 | 406 | 实现 Git 协议 |
| git http-backend | 27.2 | 400 | 实现 HTTP 协议的 CGI 程序，支持智能 HTTP 协议 |
| git instaweb | 27.3.4 | 405 | 即时启动浏览器通过 gitweb 浏览当前版本库 |
| git shell | — | — | 受限制的 shell，提供仅执行 Git 命令的 SSH 访问 |
| git update-server-info | 15.1 | 201 | 更新哑协议需要的辅助文件 |
| git http-fetch | — | — | 通过 HTTP 协议获取版本库 |
| git http-push | — | — | 通过 HTTP/DAV 协议推送 |
| git remote-ext | — | — | 由 Git 命令调用，通过外部命令提供扩展协议支持 |
| git remote-fd | — | — | 由 Git 命令调用，使用文件描述符作为协议接口 |
| git remote-ftp | — | — | 由 Git 命令调用，提供对 FTP 协议的支持 |
| git remote-ftps | — | — | 由 Git 命令调用，提供对 FTPS 协议的支持 |
| git remote-http | — | — | 由 Git 命令调用，提供对 HTTP 协议的支持 |
| git remote-https | — | — | 由 Git 命令调用，提供对 HTTPS 协议的支持 |
| git remote-testgit | — | — | 协议扩展示例脚本 |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## A.8 版本库转换和交互相关命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git archimport | — | — | 导入 Arch 版本库到 Git |
| git bundle | — | — | 提交打包和解包，以便在不同版本库间传递 |
| git cvsexportcommit | — | — | 将 Git 的一个提交作为一个 CVS 检出 |
| git cvsimport | — | — | 导入 CVS 版本库到 Git，或者使用 cvs2git |
| git cvsserver | — | — | Git 的 CVS 协议模拟器，可供 CVS 命令访问 Git 版本库 |
| git fast-export | — | — | 将提交导出为 git-fast-import 格式 |
| git fast-import | 35.3 | 506 | 其他版本库迁移至 Git 的通用工具 |
| git svn | 26.1 | 380 | Git 作为前端操作 Subversion |

## A.9 合并相关的辅助命令

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git merge-base | 11.4.2 | 146 | 供其他脚本调用，找到两个或多个提交最近的共同祖先 |
| git merge-file | — | — | 针对文件的两个不同版本执行三向文件合并 |
| git merge-index | — | — | 对 index 中的冲突文件调用指定的冲突解决工具 |
| git merge-octopus | — | — | 合并两个以上的分支。参见 git merge 的 octopus 合并策略 |
| git merge-one-file | — | — | 由 git merge-index 调用的标准辅助程序 |
| git merge-ours | — | — | 合并使用本地版本，抛弃他人版本。参见 git merge 的 ours 合并策略 |
| git merge-recursive | — | — | 针对两个分支的三向合并。参见 git merge 的 recursive 合并策略 |
| git merge-resolve | — | — | 针对两个分支的三向合并。参见 git merge 的 resolve 合并策略 |
| git merge-subtree | — | — | 子树合并。参见 git merge 的 subtree 合并策略 |
| git merge-tree | — | — | 显式三向合并结果，不改变暂存区 |
| git fmt-merge-msg | — | — | 供执行合并操作的脚本调用，用于创建一个合并提交说明 |
| git rerere | — | — | 重用所记录的冲突解决方案 |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## A.10 杂项

| 命令 | 相关章节 | 页数 | 简要说明 |
|---|---|---|---|
| git bisect--helper | — | — | 由 git bisect 命令调用，确认二分查找进度 |
| git check-attr | 41.1.2 | 551 | 显示某个文件是否设置了某个属性 |
| git checkout-index | — | — | 从暂存区拷贝文件至工作区 |
| git cherry | 18.4.4 | 265 | 查找没有合并到上游的提交 |
| git diff-files | — | — | 比较暂存区和工作区，相当于 git diff --raw |
| git diff-index | — | — | 比较暂存区和版本库，相当于 git diff --cached --raw |
| git diff-tree | — | — | 比较两个树对象，相当于 git diff --raw A B |
| git difftool--helper | — | — | 由 git difftool 命令调用，默认要使用的差异比较工具 |
| git get-tar-commit-id | 10.9 | 129 | 从 git archive 创建的 tar 包中提取提交 ID |
| git gui--askpass | — | — | 命令 git gui 的获取用户口令输入界面 |
| git notes | 41.5.2 | 570 | 提交评论管理 |
| git patch-id | — | — | 补丁过滤行号和空白字符后生成补丁唯一 ID |
| git quiltimport | 20.3.2 | 305 | 将 Quilt 补丁列表应用到当前分支 |
| git replace | 41.4.2 | 569 | 提交替换 |
| git shortlog | — | — | 对 git log 的汇总输出，适合于产品发布说明 |
| git stripspace | — | — | 删除空行，供其他脚本调用 |
| git submodule | 23.1 | 337 | 子模组管理 |
| git tar-tree | — | — | 过时命令，请使用 git archive |
| git var | — | — | 显示 Git 环境变量 |
| git web--browse | — | — | 启动浏览器以查看目录或文件 |
| git whatchanged | — | — | 显示提交历史及每次提交的改动 |
| git-mergetool--lib | — | — | 包含于其他脚本中，提供合并 / 差异比较工具的选择和执行 |
| git-parse-remote | — | — | 包含于其他脚本中，提供操作远程版本库的函数 |
| git-sh-setup | — | — | 包含于其他脚本中，提供 shell 编程的函数库 |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 附录 B Git 与 CVS 面对面

## B.1 面对面访谈录

Git：我的提交是原子提交。每次提交都对应于一个目录树（树对象）。因为我的提交 ID 是对目录树及相关的提交信息建立的一个 SHA1 哈希值，所以可以保证数据的完整性。

CVS：我承认这是我的软肋，一次错误或冲突的提交会导致部分数据被提交，而部分数据没有提交，版本库的完整性会被破坏，所以人们才设计出来 Subversion（SVN）来取代我。

Git：我的分支和里程碑管理非常快捷。因为我的分支和里程碑就是一个记录提交 ID 的 40 字节的文件，你的呢？

CVS：你怎么又提到别人的痛处了！我的分支和里程碑创建速度还是很快的，……嗯……，如果在版本库中只有几个文件的话。当然如果版本库中的文件很多，创建分支和里程碑就需要花费很长的时间。有些人对此忍无可忍，于是设计出 SVN 来取代我。

Git：其实我不用里程碑都没有关系，因为每个提交 ID 就对应于唯一的一个提交状态。

CVS：这也是我做不到的。我没有全局版本号的概念，每个文件都通过单独的版本号记录其变更历史，所以人们在使用我的时候必须经常用里程碑（tag）对我的状态进行标识。还需要提醒一句的是，如果版本库中的文件太多，创建里程碑是很耗时的，因为要逐一打开每个版本库中的文件，并在其中记录里程碑和文件版本的关联。

Git：我的工作区很干净。只在工作区的根目录下有一个 `.git` 目录，此外再无其他辅助目录或文件。

CVS：我要在工作区的每一个目录下都放置一个 CVS 目录，这个目录下有个 Entries 文件很重要，记录了对应工作区文件的检出版本及时间戳等信息。这样做的好处是可以将工作区移动到任何其他磁盘和目录，而毫不影响使用，甚至我可以将工作区的一个子目录拿出来，作为独立的工作区。

Git：我也可以将工作区移动到其他磁盘，但是要保证工作区下的 `.git` 目录和工作区一同移动。不可以只移动工作区下的一个目录到其他磁盘或目录，那样的话移出的目录就不能工作了。

Git：我的网络传输效率很高。在和其他版本库交互时，对方会告诉我他有什么，我也知道我有什么，因为只传输缺失对象的打包文件，所以效率很高而且能够显示传输进度。

CVS：这一点我不行。因为我本地没有文件做对照，所以我在传输的时候不可能做到增量传输。

Git：我甚至可以不需要网络，因为我在本地拥有完整的版本库，几乎所有的操作都在本

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


地完成。

CVS：我的操作处处需要网络，如果版本库在网络中的其他服务器上，而且网速又比较慢，那么查看日志或历史版本都需要等很长时间。

CVS：你怎么没有更新（update）命令？还有你为什么老是要执行检出命令（checkout）？对我而言，检出命令只在工作区创建时一次性完成。

Git：你的检出命令（checkout）是从远程版本库服务器获取数据来完成本地工作区的创建，版本库仍然位于远程服务器上。你的更新（update）命令执行得很慢，对吗？之所以你需要执行更新命令是因为你的版本库在远程啊。别忘了我的版本库是在本地，本地版本库会随着我在本地工作区中的操作（如提交）而更新。我的检出（checkout）操作是将本地版本库的数据检出到本地工作区，用于恢复本地丢失或错误改动的文件，也用于切换不同的分支。我也有一个和你的更新（update）操作类似的、比较耗时的网络操作命令：git fetch 或 git pull，这两个操作是从别人的版本库获取他人的改动。一般使用我（Git）做团队协作的时候，会部署一个集中共享的版本库，我就用这两个命令（git fetch 或 git pull）从共享的版本库执行拉回操作。也许你（CVS）会觉得 git fetch 或 git pull 和你的 cvs update 命令更像吧。至于你的检出命令（cvs checkout），实际上和我的克隆命令（git clone）很相似，只不过我的克隆命令不但创建了本地工作区，而且在本地还复制了和远程版本库一样的本地版本库。

CVS：为什么你的检入（commit）命令执行得那么快？

Git：是的，我的检入命令飞一般就执行完了，这也是因为版本库就在本地。也许你（CVS）会觉得我的推送命令（git push）和你的检入命令（cvs commit）更相像，其实这是一个误会。如果我不做本地提交，是没有东西可以推送（git push）的。你（CVS）每一次提交都要和版本库进行网络通信，而我可以在本地版本库进行多次提交，直到我的主人想喝咖啡了才执行一次 git push，将我本地版本库中的新提交推送给远程版本库。

CVS：我每个文件都一个独立的版本号，你有吗？

Git：每个文件一个版本号？这有什么值得夸耀的？我听说你最早是用脚本对 RCS 系统进行封装实现的，所以你的每个文件都有一个独立的版本控制，这让你变得很零碎。我听说某些商业版本控制系统也是这样，真糟糕。我的每次提交都有一个全球唯一的版本号，不但在本地版本库中是唯一的，和其他人的版本库也不会有冲突。

CVS：我能一次检出一个目录，你好像不能吧？

Git：所以我有子模组，以及 repo 等第三方工具，可以帮助我把一个大的版本库拆成多个版本库组合来使用啊。而且我还有稀疏检出的功能，只不过很少有人用到罢了。

CVS：我能添加空目录，你好像不能吧？

Git：是的，我现在还不能记录空目录。但是用户可以在空目录下创建一个隐含文件，并将该隐含文件添加到版本库中，这就实现了空目录的添加功能。你，CVS，目录管理是你的软肋，你很难实现目录的重命名，而目录重命名对我来说却是小菜一碟。

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


## B.2 Git 和 CVS 命令对照

| 比较项目 | CVS 命令 | GIT 命令 |
|---|---|---|
| URL | :pserver:user@host:/path/to/cvsroot | git://host/path/to/repos.git |
| | /path/to/cvsroot | ssh://user@host/path/to/repos.git |
| | | user@host:path/to/repos.git |
| | | file:///path/to/repos.git |
| | | /path/to/repos.git |
| 版本库初始化 | cvs -d <path> init | git init [--bare] <path> |
| 导入数据 | cvs -d <url> import -m ... | git clone; git add .; git commit |
| 版本库检出 | cvs -d <url> checkout [-d <path>] <module> | git clone <url> <path> |
| 版本库分支检出 | cvs -d <url> checkout -r <branch> <module> | git clone -b <branch> <url> |
| 工作区更新 | cvs update | git pull |
| 更新至历史版本 | cvs update -r <rev> | git checkout <commit> |
| 更新到指定日期 | cvs update -D <date> | git checkout HEAD@'{<date>}' |
| 更新至最新提交 | cvs update -A | git checkout master |
| 切换至里程碑 | cvs update -r <tag> | git checkout <tag> |
| 切换至分支 | cvs update -r <branch> | git checkout <branch> |
| 还原文件 / 强制覆盖 | cvs up -C <path> | git checkout -- <path> |
| 添加文件 | cvs add <TextFile> | git add <TextFile> |
| 添加文件（二进制） | cvs add -kb <BinaryFile> | git add <BinaryFile> |
| 删除文件 | cvs remove -f <path> | git rm <path> |
| 移动文件 | mv <old> <new>; cvs rm <old>; cvs add <new> | git mv <old> <new> |
| 反删除文件 | cvs add <path> | git add <path> |
| 工作区差异比较 | cvs diff -u | git diff |
| | | git diff --cached |
| | | git diff HEAD |
| 版本间差异比较 | cvs diff -u -r <rev1> -r <rev2> <path> | git diff <commit1> <commit2> -- <path> |
| 查看工作区状态 | cvs -n up | git status |
| 提交 | cvs commit -m "<msg>" | git commit -a -m "<msg>" ; git push |
| 显示提交日志 | cvs log <path> \| less | git log |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 比较项目 | CVS 命令 | GIT 命令 |
| --- | --- | --- |
| 逐行追溯 | cvs annotate | git blame |
| 显示里程碑 / 分支 | cvs status -v | git tag<br>git branch |
| | | git show-ref |
| 创建里程碑 | cvs tag [-r <rev>] <tagname> . | git tag [-m "<msg>"] <tagname> [<commit>] |
| 删除里程碑 | cvs rtag -d <tagname> | git tag -d <tagname> |
| 创建分支 | cvs rtag -b -r <rev> -b <branch> <module> | git branch <branch> <commit> |
| | | git checkout -b <branch> <commit> |
| 删除分支 | cvs rtag -d <branch> | git branch -d <branch> |
| 导出项目文件 | cvs -d <url> export -r <tag> <module> | git archive -o <output.tar> <tag> <path> |
| | | git archive -o <output.tar> --remote=<url> <tag> <path> |
| 分支合并 | cvs update [-j <start>] -j <end>; cvs commit | git merge <branch> |
| 显示文件列表 | cvs ls | git ls-files |
| | cvs -d <url> rls -r <rev> | git ls-tree <commit> |
| 更改提交说明 | cvs admin -m <rev>:<msg> <path> | git commit --amend |
| 撤销提交 | cvs admin -o <range> <path> | git reset [ --soft \| --hard ] HEAD^ |
| 杂项 | .cvsignore 文件 | .gitignore 文件 |
| | 参数 -kb 设置二进制模式 | -text 属性 |
| | 参数 -kv 开启关键字扩展 | export-subst 属性 |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 附录 C Git 与 SVN 面对面

## C.1 面对面访谈录

Git：我的提交历史本身就是一幅美丽的图画——DAG（Directed Acylic Graph，有向无环图），可以看到各个分支之间的合并关系。而你 SVN，你的提交历史怎么是一条直线呢？要是在重症监护室看到你，还以为你挂掉了呢？

SVN：我觉得挺好，至少我每次提交会有一个全局的版本号，而且我的版本号是递增的。你的版本号不是递增的吧？

Git：你说的对，我的版本号不是一个简单递增的数字，而是一个长达 40 位的十六进制数字（哈希值），但是可以使用短格式，只要不冲突。虽然我的提交编号看起来似乎是无序的，但实际上我的每一个提交都记录了父提交甚至是双亲或多亲提交，因此可以很容易地从任意一个提交开始建立一条指向历史提交的跟踪链。

SVN：是啊，我的一个提交和前一个提交有时根本没有关系，例如一个提交是发生在主线 /trunk 中的，下一个提交可能就发生在 /branches/1.3.x 分支中。你要知道，要想画出一个像你那样的分支图，我要做多少工作吗？我不容易呀。

Git：我一直很奇怪，你的分支和里程碑怎么看起来和目录一样？我的分支和里程碑名字虽然看起来像是目录，但实际上和工作区的目录完全没有关系，只是对提交 ID 的一个记号而已。

SVN：我一开始觉得我用轻量级拷贝的方式实现分支和里程碑会很酷，也很快。但是我发现很多人在使用我的时候，直接在版本库的根目录下创建文件而不是把文件创建在 /trunk 目录下，这就导致这些人无法再创建分支和里程碑了，因为无法将根目录拷贝到子目录呀！

Git：那么你是如何对分支合并进行跟踪的呢？因为我有 DAG 的提交关系图，很容易就可以看出分支之间的合并历史，但是你是怎么做到的呢？

SVN：我用了一点小技巧，通过属性（svn:mergeinfo）记录了合并的分支名和版本范围，这样再合并的时候，我会根据相关属性确定是否要合并。但是如果经常在子目录下合并，会有太多的 svn:mergeinfo 属性等待我检查，我会很困扰。还有我的这个功能是在 1.5 以后的版本才提供的，因此老版本会破坏这个机制。

SVN：对了，我的属性能干很多事哦，我甚至可以把我的照片作为属性附加在文件上。

Git：这点我承认，你的属性非常强大。其实我也支持属性，只不过实现方式不同罢了。而且我可以通过评注的方式为任意对象（提交、文件、里程碑等）添加评注，也可以实现把照片作为评注附加在文件上，可是这个功能有什么实际用处呢？

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


SVN：我有轻量级拷贝，而我的分支和里程碑就是通过拷贝实现的，很强大哦。

Git：我根本就不需要轻量级拷贝，因为我对文件的保存和文件的路径无关，我只关心内容。所以相同内容的文件无论它们的文件名相差有多大，在我这里只保存一份。而你 SVN，如果用户忘了用轻量级拷贝，版本库是不是负担很重啊。

SVN：听说你不能针对目录授权，这可是我的强项，所以公司无论大小都在用我作为版本控制系统。

Git：不要说你的授权了，简直是一团糟。虽然这本书的作者为你写了一个图形化的授权管理工具[①]，对你的授权会有所改善，但是你糟糕的分支和里程碑的实现，会导致授权在新的分支和里程碑中又要逐一进行设置，工作量其大无比。虽然泛路径授权是一个解决方案，但是官方并没有提供啊。

Git：说说我的授权吧。如果你认真地读过本书服务器架设的相关章节，你会为我能够提供按照分支，以及按照路径进行写操作授权的功能而击掌叫好的。当然我的读操作授权还不能做到很精细，但是可以将版本库拆分成若干个小的版本库啊，再参照本书介绍的各种多版本库协同模式，也会找到一个适合的解决方案的啊。

Git：我的工作区很干净。只在工作区的根目录下有一个 .git 目录，此外再无其他。

SVN：我要在工作区的每一个目录下都放置一个 .svn 目录，这个目录在 Linux 下可是隐藏的哦。这个目录下不但有跟踪工作区文件状态的跟踪文件，而且还有每一个文件的原始拷贝呢。这样有的操作就可以脱离网络执行了，例如：差异比较、工作区文件的回滚。

Git：嗯，你要是像我一样能再多保存一点内容（整个版本库）就更好了。像你这样在每个工作区子目录下都有一个 .svn 目录，而且每个 .svn 目录下都有文件的原始拷贝，在进行内容搜索的时候会搜索出两份吧，太干扰了。而且你这么做和 CVS 一样有安全风险，造成本地文件名的信息泄漏，千万不要在 Web 服务器上用 SVN 检出哦。

Git：我的操作可以不需要网络。因为我在本地拥有完整的版本库，几乎所有操作都是在本地完成的。

SVN：正如前面说到的，我有部分命令可以不需要网络，但是其他绝大多数命令还是要依赖网络的。

SVN：你怎么没有更新（update）命令啊？还有你为什么老是要执行检出命令（checkout）？对我而言，检出命令只在工作区创建时一次性完成。

Git：你的这个问题怎么和 CVS 问的一样。你的更新（update）命令执行的很慢，对吧？首先你要用检出命令（checkout）建立工作区，然后你要经常执行更新（update）命令进行更新，否则很容易造成你的更改和他人的更改发生冲突。

Git：之所以你需要更新是因为你的版本库在远程啊。别忘了我的版本库是在本地，本地的版本库会随着我在本地工作区中的操作（如提交）而更新。我的检出（checkout）操作

① *http://www.ossxp.com/doc/pysvnmanager/user-guide/user-guide.html*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


一般用于用户切换分支，或者从本地版本库检出丢失的文件或覆盖本地错误改动的文件。如果我没记错的话，你切换分支用的是 svn switch 命令对么？

Git：实际上我也有一个比较耗时的网络操作命令：git fetch 或 git pull，这两个操作是从远程版本库获取他人的改动。一般使用我（Git）做团队协作的时候，会部署一个集中共享的版本库，我就从这个共享的版本库执行拉回操作。也许你（SVN）会觉得 git fetch 或 git pull 和你的 svn update 命令更像吧。至于你的检出命令（svn checkout），实际上和我的克隆命令（git clone）很相似，只不过我的克隆命令不但创建了本地工作区，而且在本地还复制了和远程版本库一样的本地版本库。

SVN：为什么你的检入（commit）命令执行得那么快？

Git：是的，我的检入命令飞一般就执行完了，这也是因为我的版本库就在本地。也许你（SVN）会觉得我的推送命令（git push）和你的检入命令（svn commit）更相像，其实这是一个误会。如果我不做本地提交，是不能通过推送命令（git push）将我的本地提交共享给（推送给）其他版本库的。你（SVN）每一次的提交都要和版本库进行网络通信，而我可以在本地版本库进行多次提交，直到我的主人想喝咖啡了才执行一次 git push，将我本地版本库中的新提交推送给远程版本库。

SVN：我能一次检出一个目录，你好像不能吧？

Git：所以我有子模组，以及 repo 等第三方工具，可以帮助我把一个大的版本库拆成多个版本库组合来使用啊。而且我还有稀疏检出的功能，只不过很少有人用到罢了。

SVN：我能添加空目录，你好像不能吧？

Git：是的，我现在还不能记录空目录，但是用户往往在空目录下创建一个隐含文件，并将该隐含文件添加到版本库中，这就实现了空目录添加的功能。

## C.2 Git 和 SVN 命令对照

| 比较项目 | SVN 命令 | GIT 命令 |
|---|---|---|
| URL | svn://host/path/to/repos | git://host/path/to/repos.git |
| | https://host/path/to/repos | ssh://user@host/path/to/repos.git |
| | file:///path/to/repos | user@host:path/to/repos.git |
| | | file:///path/to/repos.git |
| | | /path/to/repos.git |
| 版本库初始化 | svnadmin create <path> | git init [--bare] <path> |
| 导入数据 | svn import <path> <url> -m ... | git clone; git add .; git commit |
| 版本库检出 | svn checkout <url/of/trunk> <path> | git clone <url> <path> |
| 版本库分支检出 | svn checkout <url/of/branches/name> <path> | git clone -b <branch> <url> <path> |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 比较项目 | SVN 命令 | GIT 命令 |
| --- | --- | --- |
| 工作区更新 | svn update | git pull |
| 更新至历史版本 | svn update -r <rev> | git checkout <commit> |
| 更新到指定日期 | svn update -r {<date>} | git checkout HEAD@'{<date>}' |
| 更新至最新提交 | svn update -r HEAD | git checkout master |
| 切换至里程碑 | svn switch <url/of/tags/name> | git checkout <tag> |
| 切换至分支 | svn switch <url/of/branches/name> | git checkout <branch> |
| 还原文件 / 强制覆盖 | svn revert <path> | git checkout -- <path> |
| 添加文件 | svn add <path> | git add <path> |
| 删除文件 | svn rm <path> | git rm <path> |
| 移动文件 | svn mv <old> <new> | git mv <old> <new> |
| 清除未跟踪文件 | svn status \| sed -e "s/^?//" \| xargs rm | git clean |
| 清除工作锁定 | svn clean | - |
| 获取文件历史版本 | svn cat -r<rev> <url/of/file>@<rev> > <output> | git show <commit>:<path> > <output> |
| 反删除文件 | svn cp -r<rev> <url/of/file>@<rev> <path> | git add <path> |
| 工作区差异比较 | svn diff | git diff |
| | | git diff --cached |
| | | git diff HEAD |
| 版本间差异比较 | svn diff -r <rev1>:<rev2> <path> | git diff <commit1> <commit2> -- <path> |
| 查看工作区状态 | svn status | git status -s |
| 提交 | svn commit -m "<msg>" | git commit -a -m "<msg>" ; git push |
| 显示提交日志 | svn log \| less | git log |
| 逐行追溯 | svn blame | git blame |
| 显示里程碑 / 分支 | svn ls <url/of/tags/> | git tag |
| | svn ls <url/of/branches/> | git branch |
| | | git show-ref |
| 创建里程碑 | svn cp <url/of/trunk/> <url/of/tags/name> | git tag [-m "<msg>"] <tagname> [<commit>] |
| 删除里程碑 | svn rm <url/of/tags/name> | git tag -d <tagname> |
| 创建分支 | svn cp <url/of/trunk/> <url/of/branches/name> | git branch <branch> <commit><br>git checkout -b <branch> <commit> |
| 删除分支 | svn rm <url/of/branches/name> | git branch -d <branch> |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 比较项目 | SVN 命令 | GIT 命令 |
| --- | --- | --- |
| 导出项目文件 | svn export -r <rev> <path> <output/path> | git archive -o <output.tar> <commit> |
| | svn export -r <rev> <url> <output/path> | git archive -o <output.tar> --remote=<url> <commit> |
| 反转提交 | svn merge -c -<rev> | git revert <commit> |
| 提交拣选 | svn merge -c <rev> | git cherry-pick <commit> |
| 分支合并 | svn merge <url/of/branch> | git merge <branch> |
| 冲突解决 | svn resolve --accept=<ARG> <path> | git mergetool |
| | svn resolved <path> | git add <path> |
| 显示文件列表 | svn ls | git ls-files |
| | svn ls <url> -r <rev> | git ls-tree <commit> |
| 更改提交说明 | svn ps --revprop -r<rev> svn:log "<msg>" | git commit --amend |
| 撤销提交 | svnadmin dump、svnadmin load 及 svndumpfilter | git reset [ --soft \| --hard ] HEAD^ |
| 属性 | svn:ignore | .gitignore 文件 |
| | svn:mime-type | text 属性 |
| | svn:eol-style | eol 属性 |
| | svn:externals | git submodule 命令 |
| | svn:keywords | export-subst 属性 |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 附录 D Git 与 Hg 面对面

## D.1 面对面访谈录

Git：你好 Hg，我发现我们真的很像。

Hg：是啊，人们把我们都归类为分布式版本控制工具，所以我们之间的相似度，要比和 CVS、SVN 的相似度高得多了。

Hg：我是用 Python 和少部分的 C 语言实现的，你呢？

Git：我的核心当然是使用 C 语言了，因为 Linus Torvalds 最爱用 C 语言了。我的很多命令还使用了 Shell 脚本和 Perl 语言开发，Python 用的很少。

Hg：大量使用 C 语言，是你的性能比我高的原因吗？

Git：当然不是了，你不也在核心模块使用 C 语言了么？问题的关键在于我的对象库设计得非常优秀。你不要忘了我是谁发明的，那可是大名鼎鼎的 Linux 之父 Linus Torvalds 啊，他对 Linux 文件系统可是再熟悉不过的了，所以他能够以文件系统开发者的视角来实现我的核心。

Git：还有我的网络传输过程非常直观，可以显示实时的进度，好像我从你那里没有看到。之所以我能够有这样的实现，是因为我使用了“智能协议”。在网络传输的两端都启用了相应的辅助程序，实现差异传输及传输进度的计算和显示。

Hg：实际上我也支持进度显示[①]，不过是通过 Progress 插件[②]实现的，需要通过修改配置文件启用该插件。

Hg：我有一个特点是 SVN 用户非常喜欢的，就是我的顺序数字版本号。

Git：你的顺序数字版本号只在本地版本库中有效。也就是说，你不能像 SVN 那样将顺序数字版本号作为项目本身的版本号，因为换成另外一个版本库的克隆，那个数字版本号就会不一样了。

Hg：我觉得你的暂存区（stage）的概念太古怪了。我提交的时候，改动的文件会直接提交而不需要进行什么注册到暂存区的操作。

Git：让读者来做评判吧。如果读者读过本书的第 2 篇，一定会说 Git 的暂存区帅呆了。

Hg：我只允许用户对最近的一次提交进行回滚撤销，而你（Git）怎么能允许用户撤销

① 感谢来自中国台湾的 Willie Wu 对我博客的评论。

② *http://mercurial.selenic.com/wiki/ProgressExtension*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133



任意多次历史提交呢？那样安全吗？

Git：这就是我的对象库和引用设计的强大之处，我可以使用 git reset 命令将工作分支进行任意的重置，丢弃任意多的历史。至于安全性，我的重置命令有一个保险：reflog，我随时可以参照 reflog 的记录来弥补错误的重置。

Hg：我们的 revert 命令好像不同？

Git：你 Hg 的 hg revert 命令和 SVN 的 svn revert 命令相似，是取消本地修改，用原始拷贝覆盖。你的这个操作在我这里是用 git checkout 命令来实现的。我也有一个 git revert 命令，但是这个命令是针对某个历史提交进行的反向操作，以取消该历史提交的改动。

Hg：我执行日志查看能够看到文本显示的分支图，你呢？

Git：我需要在日志显示时添加参数，即使用命令 git log --graph。我支持通过建立别名来实现简洁的调用，例如建立一个名为 glog 的别名。

Git：我听说你 Hg 不支持分支？

Hg：你说的是昨天的我，现在有了 Bookmarks 插件[①]，我也拥有和你类似的分支实现。不过传统来讲我还是以克隆来实现分支的。

Git：实际上我的每一个克隆的版本库也相当于独立的分支，但是因为我有强大的分支功能，因此很多用户还没有意识到。使用 Topgit 的用户就应该使用版本库克隆作为 Topgit 本身的分支管理。

Git：还有，因为我对分支的完整支持，使得我可以和 SVN 很好地协同工作。我可以将整个 SVN 转换为本地的 Git 库，但是你 Hg，显然只能每次转换一个分支。

Hg：是的，我要向你多学习。

## D.2 Git 和 Hg 命令对照

| 比较项目 | HG 命令 | GIT 命令 |
|---|---|---|
| URL | http://host/path/to/repos | git://host/path/to/repos.git |
| | ssh://user@host/path/to/repos | ssh://user@host/path/to/repos.git |
| | file:///path/to/repos | user@host:path/to/repos.git |
| | /path/to/repos | file:///path/to/repos.git |
| | | /path/to/repos.git |
| 配置 | [ui]<br>username = Firstname Lastname <mail@addr> | [user]<br>name = Firstname Lastname<br>email = mail@addr |

① *http://mercurial.selenic.com/wiki/BookmarksExtension*

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 比较项目 | HG 命令 | GIT 命令 |
| --- | --- | --- |
| 版本库初始化 | hg init <path> | git init [--bare] <path> |
| 版本库克隆 | hg clone <url> <path> | git clone <url> <path> |
| 获取版本库更新 | hg pull --update | git pull |
| 更新至历史版本 | hg update -r <rev> | git checkout <commit> |
| 更新到指定日期 | hg update -d <date> | git checkout HEAD@'{<date>}' |
| 更新至最新提交 | hg update | git checkout master |
| 切换至里程碑 | hg update -r <tag> | git checkout <tag> |
| 切换至分支 | hg update -r <branch> | git checkout <branch> |
| 还原文件 / 强制覆盖 | hg update -C <path> | git checkout -- <path> |
| 添加文件 | hg add <path> | git add <path> |
| 删除文件 | hg rm <path> | git rm <path> |
| 添加及删除文件 | hg addremove | git add -A |
| 移动文件 | hg mv <old> <new> | git mv <old> <new> |
| 撤销添加、删除等操作 | hg revert <path> | git reset -- <path> |
| 清除未跟踪文件 | hg clean | git clean -fd |
| 获取文件历史版本 | hg cat -r<rev> <path> > <output> | git show <commit>:<path> > <output> |
| 反删除文件 | hg add <path> | git add <path> |
| 工作区差异比较 | hg diff | git diff |
| | | git diff --cached |
| | | git diff HEAD |
| 版本间差异比较 | hg diff -r <rev1> -r <rev2> <path> | git diff <commit1> <commit2> -- <path> |
| 查看工作区状态 | hg status | git status -s |
| 提交 | hg commit -m "<msg>" | git commit -a -m "<msg>" |
| 推送提交 | hg push | git push |
| 显示提交日志 | hg log \| less | git log |
| | hg glog \| less | git log --graph |
| 逐行追溯 | hg annotate | git annotate, git blame |
| 显示里程碑 / 分支 | hg tags | git tag |
| | hg branches<br>hg bookmarks | git branch |
| | hg heads | git show-ref |

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133


（续）

| 比较项目 | HG 命令 | GIT 命令 |
| --- | --- | --- |
| 创建里程碑 | hg tag [-m "<msg>"] [-r <rev>] <tagname> | git tag [-m "<msg>"] <tagname> [<commit>] |
| 删除里程碑 | hg tag --remove <tagname> | git tag -d <tagname> |
| 创建分支 | hg branch <branch><br>hg bookmark <branch> | git branch <branch> <commit> |
| | | git checkout -b <branch> <commit> |
| 删除分支 | hg commit --close-branch<br>hg bookmark -d <branch> | git branch -d <branch> |
| 导出项目文件 | hg archive -r <rev> <output.tar.gz> | git archive -o <output.tar> <commit> |
| | | git archive -o <output.tar> --remote=<url> <commit> |
| 反转提交 | hg backout <rev> | git revert <commit> |
| 提交拣选 | - | git cherry-pick <commit> |
| 分支合并 | hg merge <rev> | git merge <commit> |
| 变基 | hg rebase | git rebase |
| 冲突解决 | hg resolve --tool=<tool> | git mergetool |
| | hg resolve -m <path> | git add <path> |
| 更改提交说明 | Hg + MQ | git commit --amend |
| 撤销最后一次提交 | hg rollback | git reset [ --soft \| --hard ] HEAD^ |
| 撤销多次提交 | Hg + MQ | git reset [ --soft \| --hard ] HEAD~<n> |
| 撤销历史提交 | Hg + MQ | git rebase -i <commit>^ |
| 启动 Web 浏览 | hg serve | git instaweb |
| 二分查找 | hg bisect | git bisect |
| 内容搜索 | hg grep | git grep |
| 提交导出补丁文件 | hg export | git format-patch |
| 工作区根目录 | hg root | git rev-parse --show-toplevel |
| 杂项 | .hgignore 文件 | .gitignore 文件 |
| | pager 扩展 | 内置分页器 |
| | color 扩展 | color.* 配置变量 |
| | mq 扩展 | StGit, Topgit |
| | graphlog 扩展 | git log --graph |
| | hgk 扩展 | gitk |



更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

# 国内首本Maven著作，内容全面，实战性强

## 公认的Maven专家亲自执笔，中外技术专家联袂推荐，权威性毋庸置疑

作者：许晓斌 著 ISBN：978-7-111-32154-5 定价：65.00 元




你是否早已厌倦了日复一日的手工构建工作？你是否对各种项目风格迥异的构建系统感到恐惧？Maven——这一Java社区事实标准的项目管理工具，能帮你从琐碎的手工劳动中解脱出来，帮你规范整个组织的构建系统。不仅如此，它还有依赖管理、自动生成项目站点等超酷的特性，已经有无数的开源项目使用它来构建并促进团队交流，每天都有数以万计的开发者访问中央仓库以获取他们需要的依赖。

本书内容全面而系统，Maven的原理、使用方法和高级应用尽含其中；注重实战是本书的另一个特点，不仅各个知识点都辅有小案例，而且还有一个综合性的案例贯穿全书。如果你想使用Maven，或正在使用Maven，本书将给你绝佳的指导。

随着近两年Maven在国内的普及，越来越多的公司和项目开始接受并使用其作为项目构建与依赖管理工具，Java开发人员对Maven相关的资料的需求也越来越迫切。Juven Xu作为Sonatype的员工和《Maven权威指南》的译者，对Maven有着非常深刻的理解，他为Maven中文社区所做的工作也为推动Maven的发展做出了重要的贡献。这本书是Juven牺牲了将近一年的业余时间创作而成的，内容全面、实战性强、深度和广度兼备，是Maven中文社区不可多得的优秀参考资料。

——Maven中文用户组

本国语言的Maven参考资料永远是受欢迎的，而现在Juven Xu——一位活跃在开源社区的知名Maven专家正好有条件编写一本关于Maven的中文图书。本书将带领你一步步从认识Maven开始，逐渐走向更高级的现实世界中的真实项目应用。这本书的主要内容不仅包括Maven在Web领域的应用、使用Maven管理版本发布，以及如何编写自己的Maven插件，而且还涵盖了许多如何在企业环境中应用Maven的技术细节，例如Eclipse集成、Nexus仓库管理器以及用Hudson进行持续集成等。如果你是一位正在使用Maven的中国程序员，本书是必备的！

——John Smart
Wakaleo Consuting首席咨询顾问
《Java Power Tools》（O'Reilly出版）作者

终于看到Maven的中文图书面世啦！虽然Maven是Java项目构建工具的事实标准，但是相关的资料还比较少，更别说完整和系统的中文资料了。非常感谢Juven牺牲业余时间完成了本书，为Java程序员快速了解和使用Maven提供了参考。虽然很多Java项目都在使用Maven进行管理，但是仍然有不少开发人员还在使用传统的方法管理项目，相信这本书会给你带来不一样的项目构建和管理体验。

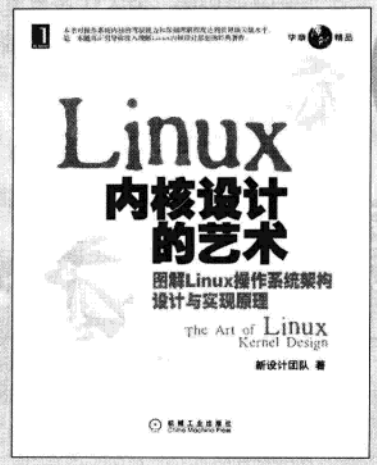
——雷卷
淘宝技术专家
mvnsearch.org作者

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

**本书对操作系统内核的驾驭能力和深刻理解程度达到世界顶尖级水平，是一本能真正引导你深入理解Linux内核设计思想的经典著作。**

作者：新设计团队 著　ISBN：978-7-111-34744-6　定价：79.00 元




**结合真实的源码、349幅内核运行时序图和详细的文字描述，以一种开创性的方式对Linux内核进行了极为直观和透彻的阐述，读者可通过阅读本书提出自己的设计思想。**

对于操作系统而言，稳定且可靠地运行是最重要的。现行技术方案是将用户进程与用户进程之间、用户进程与操作系统之间进行分离，操作系统可以管理用户进程，但是用户进程之间不能互相干预，更不能“侵入”操作系统内核。然而这一切是如何实现的呢？用户程序和操作系统都是程序，使用的是同一个CPU、同一套指令集、同一个内存条，为什么会有这么大的区别呢？

**绝妙之处在于操作系统的设计指导思想：主奴机制！**

操作系统的设计者依托硬件，在主奴机制设计思想的指导下，把操作系统内核与用户进程间的关系设计为“主子”与“奴才”的关系。该指导思想贯穿整个操作系统设计的始终，涉及操作系统的方方面面。以内存管理为例，操作系统内核可以访问所有的内存空间，包括用户进程的内存空间，而用户进程则不能访问其他用户进程的内存空间，更不可能访问操作系统内核的内存空间。更有甚者，用户进程甚至不知道自身的代码以及所访问的内存究竟在物理内存中的什么位置……可以说，操作系统就是围绕着“主奴机制”这个设计指导思想实现的。

本书不仅在主奴机制等宏观的设计指导思想方面有独到的见解，而且对操作系统源代码的实现原理和编程技巧等微观内容的讲解与分析也细致入微。例如：

- 为什么操作系统的内核代码不是加载在0x00000的位置？
- 为什么 jmpi 0, 8中的“8”不能简单地当作阿拉伯数字“8”看待？
- 为什么用C语言写的操作系统却不是从main函数开始执行？
- 为什么调用main函数时用return而不用call？

……

**所有这些在同类书中可能都不曾提及的问题在本书中都有精彩的论述。**

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133




www.hzbook.com

**填写读者调查表 加入华章书友会**
**获赠精彩技术书 参与活动和抽奖**

**尊敬的读者：**

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名： 书号：7-111-( )

| 姓名： | 性别：☐男 ☐女 | 年龄： | 职业： |
|---|---|---|---|
| 通信地址： | | E-mail： | |
| 电话： | 手机： | 邮编： | |

**1. 您是如何获知本书的：**

☐朋友推荐 ☐书店 ☐图书目录 ☐杂志、报纸、网络等 ☐其他

**2. 您从哪里购买本书：**

☐新华书店 ☐计算机专业书店 ☐网上书店 ☐其他

**3. 您对本书的评价是：**

| | | | | |
|---|---|---|---|---|
| 技术内容 | ☐很好 | ☐一般 | ☐较差 | ☐理由_____ |
| 文字质量 | ☐很好 | ☐一般 | ☐较差 | ☐理由_____ |
| 版式封面 | ☐很好 | ☐一般 | ☐较差 | ☐理由_____ |
| 印装质量 | ☐很好 | ☐一般 | ☐较差 | ☐理由_____ |
| 图书定价 | ☐太高 | ☐合适 | ☐较低 | ☐理由_____ |

**4. 您希望我们的图书在哪些方面进行改进？**

_____

_____

**5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。**

_____

_____

**6. 您有没有写作或翻译技术图书的想法？**

☐是，我的计划是_____ ☐否

**7. 您希望获取图书信息的形式：**

☐邮件 ☐信函 ☐短信 ☐其他_____

**请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收**

**邮编：100037 电话：（010）88379512 传真：（010）68311602 E-mail: hzjsj@hzbook.com**

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133

定做电子书，海量电子书，

各科电子书，代寻电子书。

ＱＱ：1759560190

更多Java开发、Selenium自动化测试、性能测试视频、代码、PDF：http://javaone.taobao.com QQ:785347133