JAVA

多线程设计模式

结城 浩 著 博硕文化 译



- UML介绍说明
- 多线程的设计模式解说
- Java程序语言范例程序
- 练习问题自我检测学习效果
- API精华参考资料
- 范例程序代码到网站

www.tqbooks.net/download.asp

处下载

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

出 版 说 明

多线程与并发处理是程序设计好坏优劣的重要课题，本书通过浅显易懂的文字与实例来介绍与 Java 线程相关的设计模式理念，并且通过实际的 Java 程序范例和 UML 图示来一一解说，书中在程序代码的重要部分加上标注使读者更加容易解读。再配合众多的说明图解，无论对于初学者还是程序设计高手来说，这都是学习和认识设计模式的一本非常难得的参考书。

书中包含了 Java 语言线程的基础内容介绍、12 个重要的多线程设计模式和全书总结以及丰富的附录内容。每一章相关线程设计模式的介绍，都举一反三使学习更有效率。每章后附有习题，让读者可以温故知新，更加迅速地吸收书中精华，书中最后附上问题解答，方便读者学习验证。本书所有模式的解说中都配有相关的范例说明，相信通过本书的学习，读者一定会迅速学到自己想要掌握的知识。

本书相关范例可以从网站 http://www.tqbooks.net/download.asp 中下载，以模式名来区分子目录内容，书中示例放在 sample 目录中，而以 Q 和 A 开头的目录分别为问题与解答的相关程序文件，Others 目录中的文件为其他重要程序。

本书由 Softbank Publishing,Inc.通过 Japan UNI Agency,Inc.提供版权，并由博硕文化翻译，由中国铁道出版社计算机图书中心审选。陆正中、姚尚文、刘玉敏、陆正武、张勇、安冀苗等同志完成本书的整稿工作。

中国铁道出版

2005 年 3 月

目　录

漫谈 UML 1

UML 2
类图 2
类和层次结构的关系 2
接口与实现 3
聚合 4
访问控制 5
类间的关联性 5
顺序图 6
处理流程和对象间的协调 6
时序图 7

Introduction 1　Java 语言的线程 9

Java 语言的线程 10
何谓线程 10
明为追踪处理流程，实则追踪线程 10
单线程程序 11
多线程程序 12
Thread 类的 run 方法和 start 方法 13
线程的启动 17
线程的启动（1）——利用 Thread 类的子类 17
线程的启动（2）——利用 Runnable 接口 19
线程的暂时停止 21
线程的共享互斥 22
synchronized 方法 23
synchronized 阻挡 27
线程的协调 28
wait set——线程的休息室 29
wait 方法——把线程放入 wait set 29
notify 方法——从 wait set 拿出线程 31
notifyAll 方法——从 wait set 拿出所有线程 33
wait、notify、notifyAll 是 Object 类的方法 34

线程的状态移转.....35
跟线程有关的其他话题.....36
重点回顾.....36
练习问题.....37
Introduction 2　多线程程序的评量标准.....41
多线程程序的评量标准.....42
安全性——不损坏对象.....42
生存性——进行必要的处理.....42
复用性——可再利用类.....43
性能——能快速、大量进行处理.....43
评量标准的总结.....44
重点回顾.....44
练习问题.....44
第 1 章　Single Threaded Execution——能通过这座桥的，只有一个人.....47
Single Threaded Execution Pattern.....48
范例程序 1：不使用 Single Threaded Execution Pattern 的范例.....48
Main 类.....49
并非线程安全（thread-safe）的 Gate 类.....49
UserThread 类.....51
执行看看……果然出错了.....51
为什么会出错呢.....53
范例程序 2：使用 Single Threaded Execution Pattern 的范例.....54
线程安全的 Gate 类.....55
synchronized 所扮演的角色.....55
Single Threaded Execution Pattern 的所有参与者.....57
扩展思考方向的提示.....58
何时使用（适用性）.....58
生命性与死锁.....59
可重用性与继承异常.....60
临界区的大小与执行性能.....60
相关 Pattern.....61
Guarded Suspension Pattern（第 3 章）.....61
Read-Write Lock Pattern（第 6 章）.....61
Immutable Pattern（第 2 章）.....61
Thread-Specific Storage Pattern（第 11 章）.....62

进阶说明：关于 synchronized 62
synchronized 语法与 Before/After Pattern 62
这个 synchronized 在保护什么 64
该以什么单位来保护呢 64
获取谁的锁定来保护的呢 65
原子的操作 65
long 与 double 并不是原子的 65
重点回顾 67
练习问题 67
第 2 章　Immutable——想破坏它也没办法 75
Immutable Pattern 76
范例程序 76
使用 Immutable Pattern 的 Person 类 76
Main 类 77
PrintPersonThread 类 78
Immutable Pattern 的所有参与者 80
扩展思考方向的提示 81
何时使用（适用性） 81
思考成对的 mutable 类与 immutable 类 [执行性能] 81
为了保护类的不变性 [复用性] 82
标准类链接库里使用到的 Immutable Pattern 83
相关 Pattern 84
Single Threaded Execution Pattern（第 1 章） 84
Read-Write Lock Pattern（第 6 章） 84
Flyweight Pattern（参考附录 E[GoF][Yuki01]） 84
进阶说明：final 85
重点回顾 86
练习问题 86
第 3 章　Guarded Suspension——要等到我准备好喔 93
Guarded Suspension Pattern 94
范例程序 94
Request 类 95
RequestQueue 类 96
ClientThread 类 97
ServerThread 类 98

Main 类 99
java.util.LinkedList 类的操作 100
仔细分析 getRequest 方法 100
仔细分析 putRequest 方法 102
synchronized 的意义 102
wait 与锁定 103
Guarded Suspension Pattern 的所有参与者 103
扩展思考方向的提示 104
有条件的 synchronized 104
多线程版的 if 104
忘记更改状态与生命性 105
wait 与 notify/notifyAll 的责任 [复用性] 105
各种各样的称呼 105
相关 Pattern 107
Single Threaded Execution Pattern（第 1 章） 107
Balking Pattern（第 4 章） 107
Producer-Consumer Pattern（第 5 章） 107
Future Pattern（第 9 章） 107
重点回顾 107
练习问题 108
第 4 章 Balking——不需要的话，就算了吧 115
Balking Pattern 116
范例程序 116
Data 类 117
SaverThread 类 119
ChangerThread 类 119
Main 类 120
Balking Pattern 的所有参与者 122
扩展思考方向的提示 123
何时使用（适用性） 123
表达 balk 结果的方式 124
相关 Pattern 125
Guarded Suspension Pattern（第 3 章） 125
Observer Pattern（参考附录 E [GoF] [Yuki01]） 125
进阶说明：timeout 125

Balking Pattern 与 Guarded Suspension Pattern 的中间 125
wait 的结束是什么时候 125
guarded timed 的实现 126
synchronized 没有 timeout，也不能中断 129
重点回顾 129
练习问题 130
第 5 章 Producer-Consumer——我来做，你来用 135
Producer-Consumer Pattern 136
范例程序 136
Main 类 137
MakerThread 类 137
EaterThread 类 138
Table 类 139
分析 put 方法 141
分析 take 方法 142
Producer-Consumer Pattern 的所有参与者 144
扩展思考方向的提示 145
保护安全性的 Channel 参与者 [复用性] 145
不能直接传递吗 145
Channel 参与者负荷派生的问题 146
要以什么顺序传递 Data 参与者 146
[中间者的存在]隐含的意义 147
只有一个 Consumer 参与者时会如何 148
相关 Pattern 148
Mediator Pattern（参考附录 E [GoF] [Yuki01]） 148
Worker Thread Pattern（第 8 章） 149
Command Pattern（参考附录 E [GoF] [Yuki01]） 149
Strategy Pattern（参考附录 E [GoF] [Yuki01]） 149
进阶说明：InterruptedException 异常 149
可能会花一些时间，但是可以取消掉 149
后面接着 throws InterruptedException 的方法 149
sleep 方法与 interrupt 方法 150
wait 方法与 interrupt 方法 151
join 方法与 interrupt 方法 151
interrupt 方法只是改变中断状态而已 151

isInterrupted 方法——检查中断状态 152
Thread.interrupted 方法——检查并清除中断状态 153
Thread 类的 stop 方法不能使用 153
重点回顾 153
练习问题 154
第 6 章 Read-Write Lock——大家想看就看吧，不过看的时候不能写喔 159
Read-Write Lock Pattern 160
范例程序 160
Main 类 161
Data 类 161
WriterThread 类 164
ReaderThread 类 165
ReadWriteLock 类 166
执行结果 169
检验警戒条件 170
Read-Write Lock Pattern 的所有参与者 171
扩展思考方向的提示 172
利用同时“读取”不会冲突的特性，提高程序的性能 172
适合读取操作繁重时 172
适合读取比写入次数频繁时 173
锁定的意义 173
相关 Pattern 173
Immutable Pattern（第 2 章）。 173
Single Threaded Execution Pattern（第 1 章） 174
Guarded Suspension Pattern（第 3 章） 174
Before/After Pattern（参考附录 E [Lea]） 174
Strategized Locking Pattern（参考附录 E [POSA2]） 174
重点回顾 174
练习问题 175
第 7 章 Thread-Per-Message——这个工作交给你了 181
Thread-Per-Message Pattern 182
范例程序 182
Main 类 183
Host 类 183
Helper 类 184

Thread-Per-Message Pattern 的所有参与者 187
扩展思考方向的提示 188
提升响应性，降低延迟时间 188
适合在操作顺序无所谓时使用 189
不需要返回值的时候 189
应用在服务器的制作 189
调用方法＋启动线程→传送消息 189
相关 Pattern 190
Future Pattern（第 9 章） 190
Worker Thread Pattern（第 8 章） 190
进阶说明：进程与线程 190
重点回顾 191
练习问题 192
第 8 章　Worker Thread——等到工作来，来了就工作 199
Worker Thread Pattern 200
范例程序 200
Main 类 202
ClientThread 类 202
Request 类 203
Channel 类 204
WorkerThread 类 205
Worker Thread Pattern 的所有参与者 207
扩展思考方向的提示 209
启动线程是繁重的操作 209
控制承载量 209
invocation 与 execution 的分离 210
Runnable 接口的意义 211
多态的 Request 参与者 211
只有一条 Worker 参与者 212
相关 Pattern 213
Producer-Consumer Pattern（第 5 章） 213
Thread-Per-Message Pattern（第 7 章） 213
Command Pattern（参考附录 E [GoF][Yuki01]） 213
Future Pattern（第 9 章） 213
Flyweight Pattern（参考附录 E [GoF][Yuki01]） 213

Thread-Specific Storage Pattern（第 11 章）.....214
Active Object Pattern（第 12 章）.....214
进阶说明：Swing 的 Event-Dispatching Thread.....214
何谓 Event-Dispatching Thread.....214
Event-dispatching thread 只有一条.....215
Event-dispatching thread 会调用 Listener.....215
注册 Listener 的意义.....217
Event-dispatching thread 也处理画面的重绘.....217
javax.swing.SwingUtilities 类.....217
Swing 的单线程规则.....218
重点回顾.....219
练习问题.....219
第 9 章　Future——先给您这张提货单.....225
Future Pattern.....226
范例程序.....226
Main 类.....228
Host 类.....229
Data 接口.....231
FutureData 类.....231
RealData 类.....232
Future Pattern 的所有参与者.....234
扩展思考方向的提示.....236
能够提升 throughput 吗.....236
异步方法调用的“返回值”.....236
分离“准备返回值”与“使用返回值”.....237
变形——不让人等待的 Future 参与者.....237
变形——会改变的 Future 参与者.....237
谁需要考虑到多线程[复用性].....238
回调与 Future Pattern.....238
相关 Pattern.....239
Thread-Per-Message Pattern（第 7 章）.....239
Builder Pattern（参考附录 E [GoF][Yuki01]）.....239
Proxy Pattern（参考附录 E [GoF][Yuki01]）.....239
Guarded Suspension Pattern（第 3 章）.....239
重点回顾.....239

练习问题 240
第 10 章 Two-Phase Termination——快把玩具收拾好，去睡觉吧 247
Two-Phase Termination Pattern 248
范例程序 249
CountupThread 类 250
Main 类 252
Two-Phase Termination Pattern 的所有参与者 254
扩展思考方向的提示 255
不可以使用 Thread 类的 stop 方法 255
只检查标识是不够周全的 256
只测试中断状态也是不够的 256
进行繁重的处理前，先检查终止请求 257
join 方法与 isAlive 方法 257
程序的结束与 addShutdownHook 方法 257
优雅终止的线程 259
相关 Pattern 259
Before/After Pattern（参考附录 E[Lea]） 259
Multiphase Cancellation Pattern（参考附录 E[Lea]） 259
Multi-Phase Startup Pattern 260
Balking Pattern（第 4 章） 260
进阶说明： 中断状态与 InterruptedException 异常的相互转换 260
中断状态→InterruptedException 异常的转换 260
InterruptedException 异常→转换为中断状态 261
InterruptedException 异常→转换为 InterruptedException 异常 262
重点回顾 262
练习问题 263
第 11 章 Thread-Specific Storage——每个线程的保管箱 273
Thread-Specific Storage Pattern 274
关于 java.lang.ThreadLocal 类 274
java.lang.ThreadLocal 是保管箱间 274
范例程序 1：没有使用 Thread-Specific Storage Pattern 的范例 275
Log 类 276
Main 类 277
范例程序 2：使用 Thread-Specific Storage Pattern 的范例 278
线程特有的 TSLog 类 278

Log 类 280
ClientThread 类 281
Main 类 282
Thread-Specific Storage Pattern 的所有参与者 283
扩展思考方向的提示 286
局部变量与 java.lang.ThreadLocal 类 286
放置线程特有信息的地方 287
不必担心被其他线程访问 288
throughput 的提升取决于实现 288
隐藏 context 的危险性 289
相关 Pattern 289
Singleton Pattern（参考附录 E [GoF][Yuki01]） 289
Worker Thread Pattern（第 8 章） 290
Single Threaded Execution Pattern（第 1 章） 290
Proxy Pattern（参考附录 E [GoF][Yuki01]） 290
进阶说明：Actor-based 与 Task-based 290
主体与客体 290
Actor-based 的思维 291
Task-based 的思维 291
实际上两个开发方式是混用的 292
重点回顾 292
练习问题 293

第 12 章　Active Object——接受异步消息的主动对象 295

Active Object Pattern 296
范例程序 296
使用端：Main 类 298
使用端：MakerClientThread 类 299
使用端：DisplayClientThread 类 300
“主动对象”端：ActiveObject 接口 300
“主动对象”端：ActiveObjectFactory 类 301
“主动对象”端：Proxy 类 302
“主动对象”端：SchedulerThread 类 303
“主动对象”端：ActivationQueue 类 304
“主动对象”端：MethodRequest 类 305
“主动对象”端：MakeStringRequest 类 306

“主动对象”端：DisplayStringRequest 类 307
“主动对象”端：Result 类 307
“主动对象”端：FutureResult 类 308
“主动对象”端：RealResult 类 309
“主动对象”端：Servant 类 309
执行范例程序 310
Active Object Pattern 的所有参与者 311
扩展思考方向的提示 314
到底做了什么事呢 314
Pattern 是否适合使用要考虑问题的规模 316
注意并发性 317
新增方法 317
Scheduler 参与者的角色 318
“主动对象”之间的对话 318
迈向分布处理——将线程的界限移到机器的界限 318
相关 Pattern 319
Producer-Consumer Pattern（第 5 章） 319
Future Pattern（第 9 章） 319
Worker Thread Pattern（第 8 章） 319
Thread-Specific Storage Pattern（第 11 章） 319
重点回顾 319
练习问题 320
总结　多线程程序设计的模式语言 325
多线程程序设计的模式语言 326
Pattern 与 Pattern Language 326
Single Threaded Execution Pattern（第 1 章）——能通过这座桥的，只有一个人 327
Immutable Pattern（第 2 章）——想破坏它也没办法 328
Guarded Suspension Pattern（第 3 章）——等到我准备好 329
Balking Pattern（第 4 章）——不需要的话，就算了吧 330
Producer-Consumer Pattern（第 5 章）——你来做，我来用 331
Read-Write Lock Pattern（第 6 章）——大家想看就看吧，不过看的时候不能写喔 332
Thread-Per-Message Pattern（第 7 章）——这个工作交给你了 334
Worker Thread Pattern（第 8 章）——等到工作来，来了就工作 334

Future Pattern（第 9 章）——先给您这张提货单 335
Two-Phase Termination Pattern（第 10 章）——快把玩具收拾好，
去睡觉吧 336
Thread-Specific Storage Pattern（第 11 章）——每个线程的保管箱 337
Active Object Pattern（第 12 章）——接受异步消息的主动对象 339
尾声 340
附录 A　练习问题的解答 341
附录 B　Java 的内存模型 461
附录 C　Java 线程的优先级 475
附录 D　线程相关的主要 API 479
附录 E　参考文献 487

漫谈 UML

UML

UML 是一种以可视化的观点来看程序系统，将规格和设计重点直接写下来的表现方式。UML 是 Unified Modeling Language 的缩写。中文译成“统一建模语言”。

本书利用到 UML 来表现 Design Patterns（设计模式）中类（class）和对象实例（Instance）的关系，所以在此先举出一些阅读本书时所必备的 UML 常识跟各位读者分享。不过，说明时的名词术语仍然承袭 Java 语言的习惯用法（比如，用 Java 的“field（字段）”取代 UML 的“attribute（属性）”，用 Java 的“method（方法）”取代 UML 的“operation（操作）”）。

UML 的规格相当庞大，本书中所引用的属性（也就是以下所解说的范围）仅仅为其中一小部分。如果各位读者有兴趣进一步了解 UML，可到下列网站获取 UML 的详细规格。

- UML Resource Page
  `http://www.omg.org/uml/`
- UML Resource Center
  `http://www.rational.com/uml/`

类图

UML 的类图（Class Diagram）是一种可表示一组类、对象实例和接口之间静态关系的图形。虽然称为类图，但使用到的不只是类而已。

类和层次结构的关系

Java 程序与对应类图的范例如图 0-1 所示。

```
abstract class parentClass {
   int field;
   static char field2;
   abstract void methodA();
   double methodB() {
        //...
   }
}
class ChildClass extend parentClass {
   void methodA() {
         //...
   }
   void methodC() {
        //...
   }
}
```



图 0-1　类层次体系关系的类图

图 0-1 所表示的是 ParentClass 和 ChildClass 这 2 个类之间的关系。白色箭头△方向就是类的层次结构关系。箭头是从子类（subclass）指向父类（superclass）（所以这是 extends 的箭头）。

ParentClass 是 ChildClass 的父类，ChildClass 则是 ParentClass 的子类。父类也称为基类（Base Class），而子类又称为派生类或扩充类。

类均以方框表示，方框内再以横线分割几个部分，分别是：

- 类名称
- 字段名称
- 方法名称

有时除了名称之外，还会有其他附属信息（如访问控制、方法的参数或类型等），或者是反其道而行省略图中不需要的项目（故不一定每次都能从类图还原成原始程序）。

abstract 类（抽象类）名称以斜体表示。如 ParentClass 为抽象类，故写成斜体。

static 字段（类字段）名称加下划线。如 field2 为类字段，故加上下划线。

abstract 方法（抽象方法）以斜体表示。如 ParentClass 的 methodA 是抽象方法，故写成斜体。

static 方法（类方法）名称加下划线。如 ChildClass 的 methodC 为类方法，故加上下划线。

▶▶ 深入说明——Java 和 C++的名词术语

Java 跟 C++的名词术语稍微有点不一样。Java 的字段（field）对应到 C++的成员变量（member variable），Java 的方法（method）则对应到 C++的成员函数（member function）。

▶▶ 深入说明——箭头方向

在 UML 当中，箭头是从子类指向父类。也许有人认为反过来应该比较容易理解，毕竟是先有父类才能生出子类。

其实，各位可以这么想：定义子类时，要用 extends 指定父类。所以子类一定会知道父类是谁；但是父类就不一定知道子类在哪里。必须知道对方在哪里，才能指出来，因此箭头方向是从子类指向父类。

接口与实现

图 0-2 也是类图范例之一。

图中表示一个 Printable 接口，该接口上实现了一个 PrintClass 类。接口名称为斜体。

虚线的白色箭头是表示接口与实现类的关系。箭头从实现类指向接口（所以这是 implements 的箭头）。

UML 中的 Java 接口则以<<interface>>表示。

```
interfald Printable {
    abstract void print();
    abstract void newpage();
}
class PrintClass implements Printable {
    void print(){
        //....
    }
    void newpage() {
        //.....
    }
}
```



图 0-2　接口和实现类的类图

聚合

图 0-3 也是类图范例之一。

图中所表示的是 Color（颜色）、Fruit（水果）、Basket（果篮）这 3 个类的关系。Basket 类中的 fruits 字段是 Fruit 类的数组，Basket 类的对象实例则有多个 Fruit 类的对象实例。而 Fruit 类的 color 字段是 Color 类类型，Fruit 类的对象实例则只有 1 个 Color 类的对象实例。从平面来看，它们的关系就好比是果篮里面有一些水果，各种水果有不同的颜色。

这种“结合”的关系就称为“聚合”（aggregation）。只要结合了对象实例，无论个数多少都有聚合关系。数组、java.util.Vector 类也好，如何实现也罢，只要有结合对象实例，就具有聚合关系。

白色菱形的直线代表聚合关系，请把它想成是把东西放在菱形的容器上。

```
class Color {
    //....
}

Class Fruit {
    color color;
    //....
}

class Basket {
   Fruit[] fruits;
```

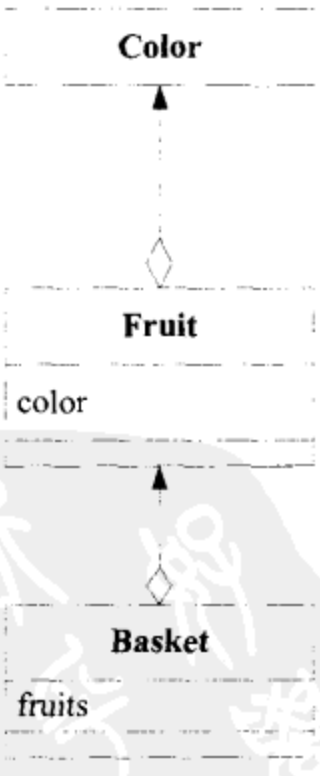


图 0-3　聚合关系的类图

```
        //...
    }
```

访问控制

图 0-4 还是类图的范例之一。

```
class someting {
    private    int privateFiels;
    protected  int protecteDield;
    public     int publicField;  {
    }
    private    void  privateMethod() {
    }
    public     void  publicMethod()  {
}
```

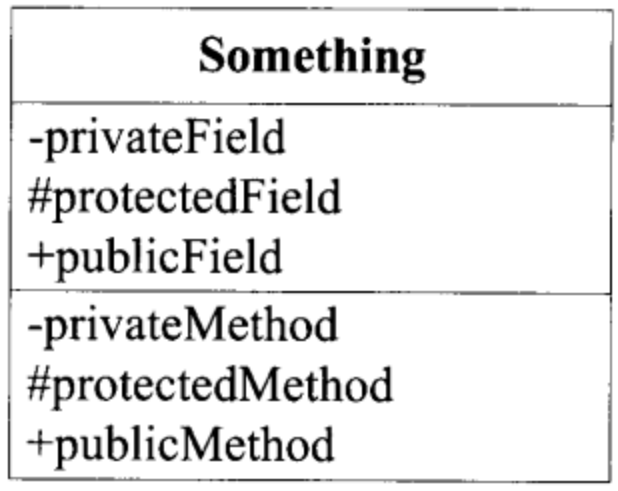


图 0-4　访问控制的类图

图中所表示的是方法和字段的访问控制。

UML 只要在方法或字段的名称前面加上符号，就可以表示访问控制。

如果是+号，则表示这是 public 的方法或字段，从任何位置都可以访问。

而如果是－号，则表示这是 private 的方法或字段，此时从类以外的位置就无法访问。

若是#，则表示这是 protected 的方法或字段，只有同一类、子类或同一 package 内的类才能访问。

类间的关联性

在相关的名称前面加上黑色三角形（◗），就可以表示类间的关联性。范例如图 0-5 所示。

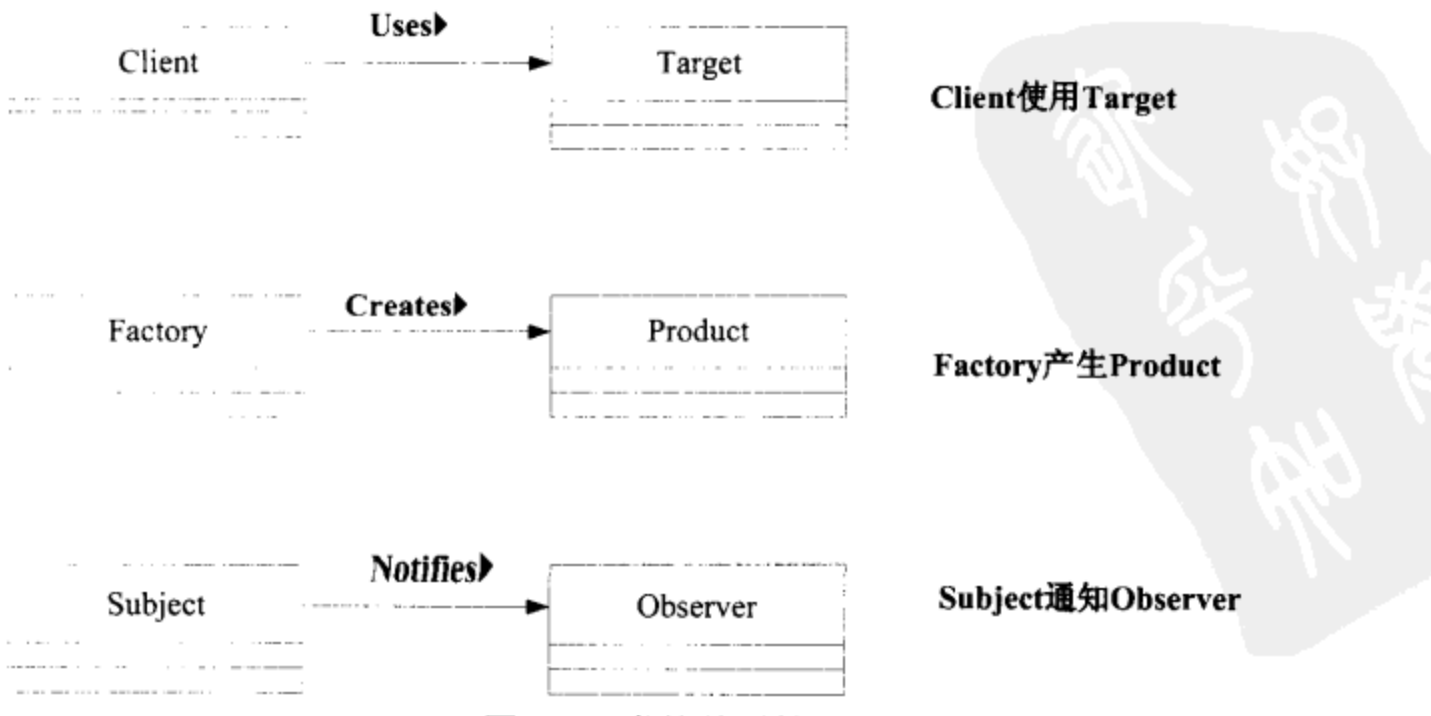


图 0-5　类的关联性

顺序图

UML 的顺序图（Sequence Diagram）是用来表示启动程序时，会按照怎样的顺序执行哪些方法、哪些现象会依什么顺序发生。

类图表示“不因时间而变化的部分（静态关系）”，而顺序图则表示“随着时间变化的部分（动态行为）”。

处理流程和对象间的协调

图 0-6 是一种顺序图的范例。

```
class Client {
    server  server;
    void    work() {
            server.open();
            server.point("Hello");
            server.close();
    }
    //...
}

class server {
    device  device;
    void open() {
        //....
    }
    //....
}

class Device {
   result  write(string s) {
     //....
     return result;
   }
}
```

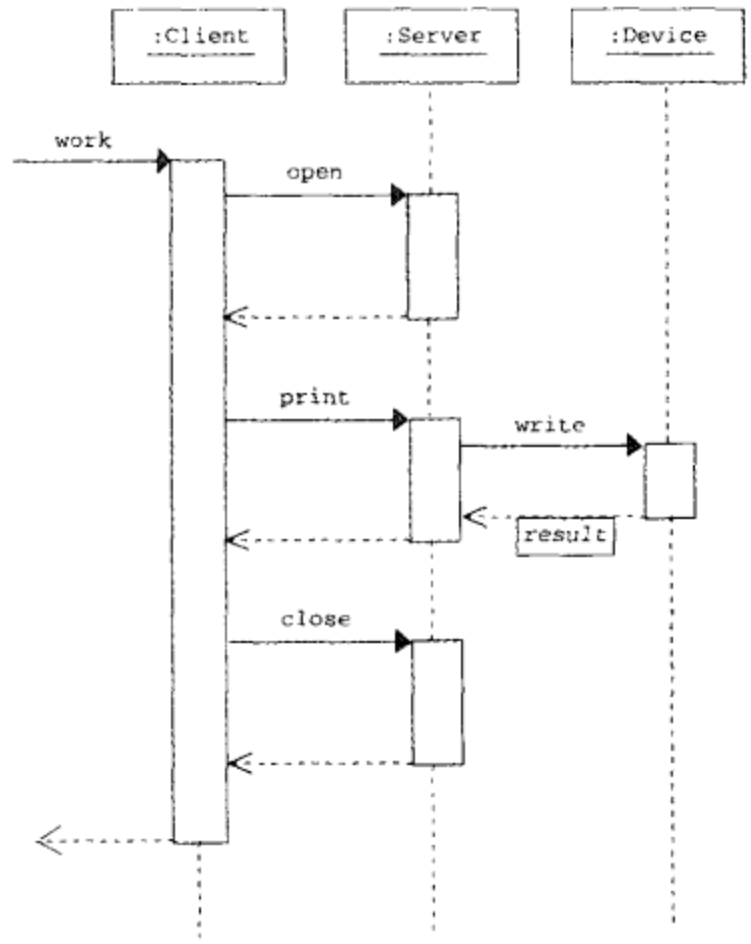


图 0-6　顺序图范例（调用方法）

图 0-6 的右边是顺序图，左边则是相对应的 Java 程序（一部分）。

在这个范例中出现了 3 个对象实例，分别对应到图例上方的 3 个方框。在方框中，类名称写在冒号（:）后面，而且还加上下划线，如:Client, :Server, :Device。意思是指 Client 类的对象实例、Server 类的对象实例以及 Device 类的对象实例。

如果每个对象实例均须命名，则将其名称写在冒号之前，如 server:Server。

所有对象实例都有一条往下延伸的虚线，称为 lifeline（生命线）。在此请将时间坐标看成由上而下，上面是过去、下面则是未来。只在有对象实例时，才会有 lifeline。

lifeline 中间有一个长方形的方框，表示该对象正在操作中。

图中有几个箭头指向左右两边。请先看到标识为 open 的箭头记号。若线段为实心且箭头全黑（→）则表示调用方法。此图是 client 调用 server 的 open 方法。因为已经成功调用出 open 方法，所以 server 对象实例是操作中的状态，故为长方形的方框。

open 箭头所指的 server 长方形方框下面，还有一条指向 client 的虚线箭头(⋯▶)。这是指 open 方法的 return（返回）。这个范例把所有方法的 return 都画出来，不过也有人宁愿省略以免麻烦。

因为控制又回到了 client，所以 server 对象实例本来操作中的长条方框就要结束。

继续以同样的方式调用 print 方法。这次是直接在 print 方法内就调用 device 对象实例的 write 方法。

类似这样两个以上的对象实例之间的操作都可以用图示表示出来。顺序图要跟着 lifeline、由上而下依次解读。如果出现箭头时，也要跟着去确认对象实例间的协调操作。

此外，在本书中如果返回值表示重大意义时，就像 Device 类的 write 方法的返回值 Result 一样，在箭头下面加上一个长方形方框来表示（这种方法并非由 UML1.4 所规范出来，而是根据附录 E 中的“POSA2”所使用的表示方法）。

时序图

在本书当中如果使用顺序图难以表示出线程的操作情况时，则使用时序图（Timethreads Diagram）来表示。时序图并不是由 UML 所规定出来的，而是附录 E 的++“Lea”中所使用的标示方法。本书以这种标示法为主，将对象实例名称配合使用于 UML 的标示法。时序图能够有效地将线程操作的重要点，以视觉方式表示出来。

图 0-7 是一个简单的时序图范例。这是相对于 Data 类的对象实例，两项线程将 setValue 方法调用出来的示意图。左边的线程获取对象实例的锁定后执行方法，右边的线程则是正要去获取该锁定，表示 Block 中的状态。

而圆角的长条形方框代表对象实例。:Data 中间的长条形方框代表对象实例所拥有的锁定，关于锁定以及线程在 Block 状态中的部分，将在 Introduction1 中做详细的解说。

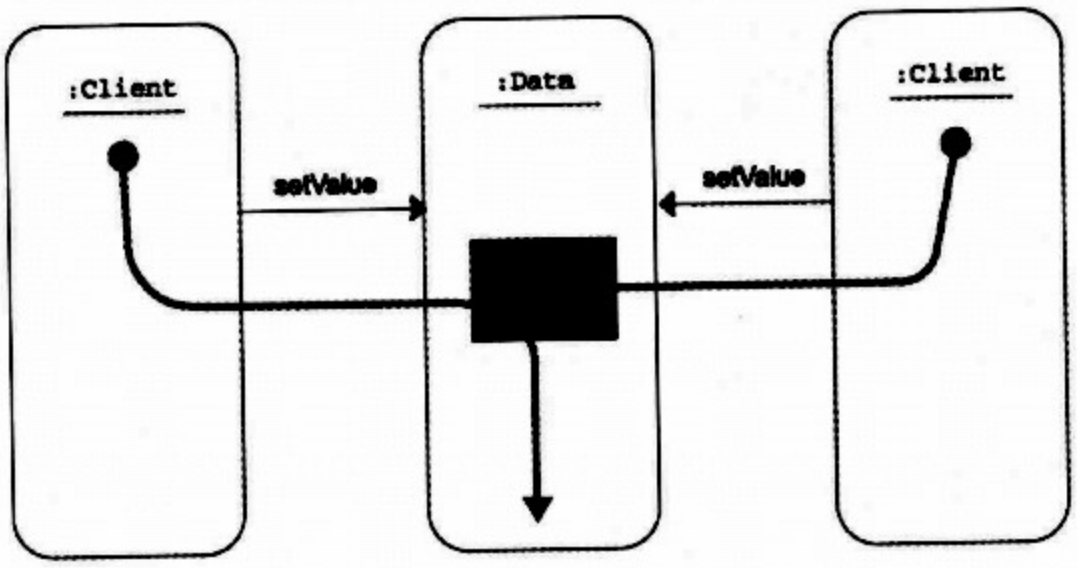
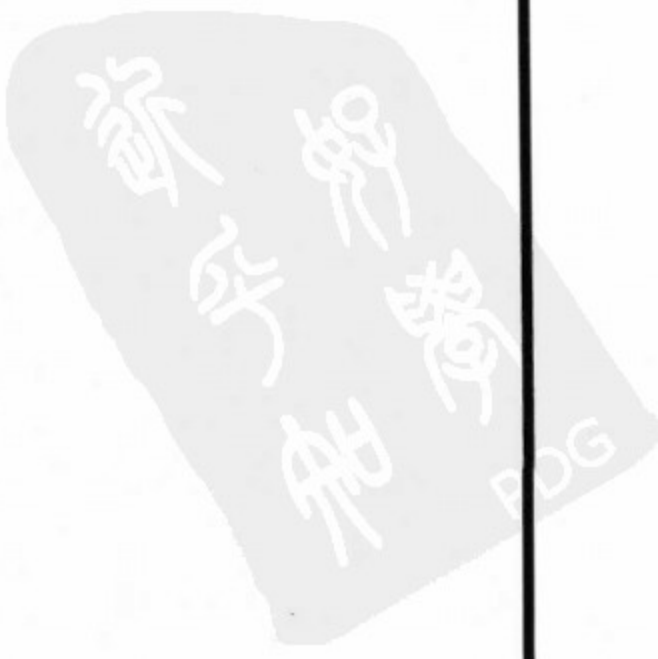


图 0-7　时序图的范例

Introduction 1

Java语言的线程



Java 语言的线程

本章先整理出 Java 语言里线程的基本事项，如果读者已经很熟悉这个部分，不妨就当作是暖身运动。本章的学习属性包括以下几个项目：

- 何谓线程
  单线程与多线程、Thread 类、run 方法和 start 方法
- 线程的启动
  Thread 类、Runnable 接口
- 线程的暂时停止
  sleep 方法
- 线程的共享互斥
  synchronized 方法、synchronized 语句和锁定
- 线程的协调
  Wait set、wait 方法、notify 方法和 notifyAll 方法

至于上述项目以外的部分，则视需要在各章另行解说。

线程相关的主要 API 则汇整于附录 D 内，请自行参阅。

何谓线程

明为追踪处理流程，实则追踪线程

我们在解读程序时，通常都是追着处理流程跑。

一开始，先执行这个语句……

↓

接着，执行这个语句……

↓

接着再执行这个语句……

一般的解读程序就是追踪这个流程。

假设现在有一份打印出来的程序内容，请拿起笔画出程序的执行顺序。则这些描绘出来的线段应该会连成一条弯弯曲曲的小河。

这条小河永远是一条，没有分叉，无论是方法调用、重复 for 语句、if 语句条件分支，或者更复杂的处理都没有影响。如果该程序的处理流程从头到尾就像只有一条线，则称该程序为单线程程序（single threaded program）。

在单线程程序里，“在某一时间点进行的处理”只有一个。如果有人提问：“现在正在执行程序的哪个部分？”，则可指出程序的某一点“就是这里”。这是因为在

单线程程序里，“正在执行程序的主体”只有一个。

线程（thread）的英文原意是“细丝”，Java 语言把上面所说“正在执行程序的主体”称为线程（thread）[注 1]。我们在解读程序时，看起来好像是顺着处理流程追下去，事实上是在追着线程的操作。

单线程程序

先执行一个简单的单线程程序看看吧，List I1-1 是一个连续输出 10000 次字符串“Good!”的程序。

List I1-1　单线程程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}
```

程序收录在下载文件夹 Introduction1/Others/SingleThread 内

如果你使用的是 Java Development Kit（JDK），则请在命令行输入：

```
javac Main.java
```

则 javac 命令会开始编译源文件 Main.java，即可产生类文件 Main.class。

然后，再回到命令行输入：

```
java Main
```

则 java 命令即开始执行程序，屏幕上也会出现 10000 个“Good!”，如图 I1-1 所示。

```
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
（下略）
```

图 I1-1　执行结果

开始执行 Java 程序后，至少会有 1 个线程开始操作。在 List I1-1 里，有操作的是被称为主线程的线程，执行的操作是输出字符串。

注 1：有时也称 thread of control。

若在命令行输入：

```
java 类名称
```

则可在 Java 执行环境下启动主线程。然后，再由主线程执行输入类（在命令行上）的 main 方法。当 main 方法里的所有处理均结束后，则主线程也同时结束。

List I1-1 里只有 1 个线程有操作，所以它是单线程程序（见图 I1-2）。

主线程

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}
```

图 I1-2　单线程程序的操作

▶▶ 补充说明：有些线程是在后台进行

为了方便说明，前面只简单地介绍了“只有 1 个线程有操作”。其实严格说来，有些线程是在非 Java 处理系统上跑，例如垃圾收集时使用的线程、GUI 相关线程等。

多线程程序

由 1 个以上的线程所构成的程序称为多线程程序（multithreaded program）。而 Java 程序语言当初一开始就把多线程处理列入规格了。

有不止 1 个线程在操作时，表示若要追踪各个线程的轨迹，就会派生出一团错综复杂的乱线团。

假设有人问到“请问现在正在执行程序的哪个部分？”时，都要指出“就在这里”。如果是多线程，你需要更多只手指头才能指出所有线程；也就是说，若正在执行的有 2 个线程，则应指出 2 个位置说“第 1 个线程正在执行这里，第 2 个线程则执行那里”。有 3 个线程，就得指出 3 个位置；有 100 个线程，就得指出 100 个位置。

当应用程序的规模大到一定程度，程序里的多线程会以某种形式存在。以下是几个常见的范例：

◈ *GUI 应用程序*

几乎所有 GUI 应用程序都一定有多线程。举例来说，假设现在有人用 Word 编

辑一个较大的文本文件，刚刚才做过单字“查找”的操作。当 Word 进行查找时，屏幕上会出现“停止查找”的按钮，用户随时均可停止查找。这个功能其实就用到了多线程。

（1）执行查找。

（2）显示按钮，若按下按钮时，则停止查找。

这 2 个操作分别交给不同的线程进行。这样一来，执行（1）的线程可以专心查找，执行（2）的线程也可专心在 GUI 操作上，程序就会变得比较简单。

◆ 比较花费时间的 I/O 处理

一般来说，文件和网络的 I/O 处理比较花费时间。如果在这段期间中，程序无法进行其他处理，程序性能便会大打折扣。遇到这种问题，有时也会使用多线程来解决。先把执行 I/O 处理和非 I/O 处理的线程分开，这样就能利用进行 I/O 处理时的时间，同时进行其他处理了。

◆ 多个客户端

基本上，网络上的服务器必须同时处理 1 个以上的客户端。不过，要是一定在服务器这边的程序设计加入 1 个以上客户端的概念的话，程序反而会变得更复杂。此时，不妨准备一个当有客户端连接到服务器时，会自动出来迎接这个客户端的线程。这样一来，服务器的程序就可以设计成好像只服务 1 个客户端，具体范例留到第 7 章的问题 7-6 再做介绍。

▶▶ 补充说明：兼具性能和扩充性的 I/O

从 Java 2 SDK，Standard Edition，v 1.4（J2SE 1.4）这个版本开始，已经加入了新的 I/O 相关类库（NIO）。有了这个类库，不必利用线程也能进行兼具性能和扩充性的 I/O 处理。详细属性请参见以下网址或 J2SE 1.4 以后版本的技术文件。

- Java 2 SDK, Standard Edition, v 1.4
  `http://java.sun.com/j2se/1.4/`
- New I/O APIs
  `http://java.sun.com/j2se/1.4/docs/guide/nio/index.html`
- Package java.nio
  `http://java.sun.com/j2se/1.4/docs/api/java/nio/package-summary.html`

Thread 类的 run 方法和 start 方法

那么，各位可以开始来学学看多线程程序了。Java 语言写成的程序一定是先从主线程开始操作，所以必须在程序的某个位置启动新的线程，才能算是真正的多线程程序。

线程的启动要利用下面这个类（一般称为 Thread 类）：

```
java.lang.Thread
```

请参见 List I1-2，这个 MyThread 类是 Thread 类扩充（extends）后的结果。

List I1-2 表示新线程的 MyThread 类（MyThread.java）

```
public class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.print("Nice!");
        }
    }
}
```

程序收录在下载文件夹 Introduction1/Others/TwoThreads 内

在 MyThread 类里声明了以下的 run 方法：

```
public vold run() {
        ...
}
```

新启动的线程的操作写在 run 方法里（run 是“跑”的意思）。启动了新的线程之后，这个线程就会调用 run 方法。当 run 方法结束时，线程也同样结束。这里 MyThread 类里的 run 方法没有写错，但光是这样，程序不会有任何操作。必须先重新启动线程后，再调用 run 方法。

真正执行的样子即如 List I1-3 所示。List I1-3 是先产生 1 个 MyThread 的实例，再利用这个实例启动新的线程。接着再进行自己的工作，也就是连续输出 10000 次“Good!”。主线程有 2 个工作：

- 启动输出“Nice!”的新线程；
- 输出“Good!”。

List I1-3 重新启动线程的程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}
```

程序收录在下载文件夹 Introduction1/Others/TwoThreads 内

请看 List I1-3，下面这行语句是主线程产生 MyThread 类的实例。这个产生出来的实例则被代入到变量 t 中。

```
MyThread t = new MyThread();
```

下面这行语句则是主线程启动新线程。

```
t.start();
```

start 方法是 Thread 类的方法。如果调用 start 方法，就会启动新线程。

请各位注意一下，被调用出来启动新线程的是 start 方法，而非 run 方法。当然你也可以调用 run 方法，只是这样不会启动新的线程。

调用出 start 方法之后，Java 执行处理系统就会在背后启动新的线程。再由这个新的线程调用 run 方法。调用 start 方法时，会有 2 个操作：

- 启动新的线程
- 调用 run 方法

start 方法跟 run 方法两者的关系可参见图 I1-3。图里确实有 2 条线（灰线）。

启动 List I1-2、List I1-3 程序的执行范例如图 I1-4 所示。

主线程

```
public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();                    新启动的线程

        for (int i = 0; i < 10000; i++) {
            System.out.print("Good!");
        }
    }
}

public class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.print("Nice!");
        }
    }
}
```

图 I1-3　启动新的线程（start 方法跟 run 方法的关系）

```
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
Good! Good! Good! Good! Good! Good!Nice! Nice! Nice! Nice! Nice!
Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice!
Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice!
Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Good! Good!
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
Good! Good! Good! Good! Good! Good! Good! Good!Good! Nice! Nice!
Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice! Nice!
（下略）
```

图 I1-4　执行范例（交错输出 Good!和 Nice!）

看到图 I1-4，各位应该可以发现执行结果是交错输出字符串“Good!”和字符串“Nice!”。由于这 2 个线程是并发的操作，所以输出结果是交错混合出现。这 2 个线程的工作分担如下：

- 主线程输出字符串“Good!”
- 新启动的线程输出字符串“Nice!”

List I1-2、List I1-3 程序有 2 个线程在操作，所以这是一个多线程程序。

▶▶ 补充说明：并发与并行

我们来谈谈并发跟并行的差别。

当有 1 个以上的线程在操作时，若计算机只有 1 个中央处理器，根本不可能真正同时进行 1 个以上的处理。“输出字符串 Good!的线程”和“输出字符串 Nice !的线程”这两个线程的操作情况如下：

- 输出字符串 Good!的线程稍微操作一下后停止
  ↓
- 输出字符串 Nice!的线程稍微操作一下后停止
  ↓
- 输出字符串 Good!的线程稍微操作一下后停止
  ↓
- 输出字符串 Good!的线程……

事实上，程序的处理是像这样不断切换有操作的线程。这种操作就称为并发（concurrent）。

如果是在有 1 个以上中央处理器的计算机上跑的 Java 执行处理系统，则线程的操作可能是并行（parallel）而非并发。当 1 个以上的线程并行操作时，就可以真正同时进行 1 个以上的处理。

表现并发和并行两者间差异的模式如图 I1-5 所示。

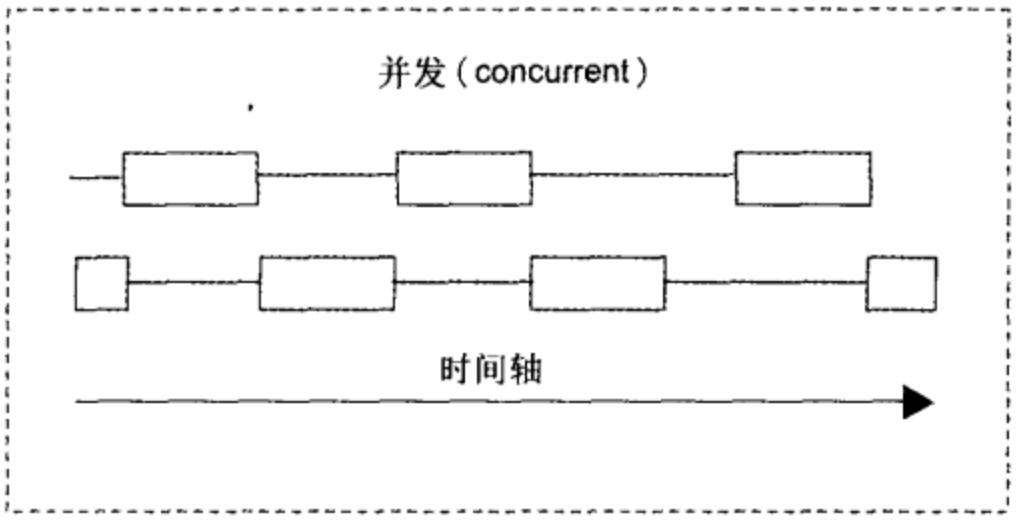


图 I1-5　并发与并行

线程的启动

再把话题拉回到线程启动的部分，如欲启动线程时，有下列 2 种方法：

（1）利用 Thread 类的子类的实例，启动线程。

（2）利用 Runnable 接口的实现类的实例，启动线程。

以下依次说明。

线程的启动(1)——利用 Thread 类的子类

先说明利用 Thread 类的子类的实例来启动线程的方法，这也是在前节使用过的方法。

PrintThread 类（List I1-4）是表示连续输出 10000 次指定字符串的线程。欲输出的字符串以构造器的参数指定，存储在 message 字段。PrintThread 则声明为 Thread 类的子类。

List I1-4 （Main.java）表示连续输出 10000 次指定字符串的线程的 PrintThread 类（PrintThread.java）

```
public class PrintThread extends Thread {
    private String message;
    public PrintThread(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.print(message);
        }
    }
}
```

程序收录在下载文件夹 Introduction1/Others/PrintThread 内

Main 类（List I1-5）这个程序则是先建立 2 个上面已声明的 PrintThread 类的实例，再利用它们来启动 2 个线程。

List I1-5 利用 PrintThread 类启动 2 个线程的程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new PrintThread("Good!").start();
        new PrintThread("Nice!").start();
    }

}
```

程序收录在下载文件夹 Introduction1/Others/PrintThread 内

在 main 方法里，则建立 PrintThread 类的实例后，立刻调用该实例的 start 方法（没有代入到变量的操作）。也就是：

```
new PrintThread("Good!").start();
```

这是先建立 PrintThread 类的实例，将该实例的 start 方法调用出来的语句。

调用出 start 方法后，就会启动新的线程。然后，这个线程再调用 PrintThread 类的实例的 run 方法。最后就变成是新启动的线程去进行连续输出 10000 次“Good!”的工作。

为了让程序更简洁，所以上面只用 1 个语句启动线程。不过，请注意：“建立 PrintThread 类的实例”和“启动该实例所对应的线程”是两个完全不同的处理。即

使已经建立了实例，仍然必须等到调用 start 方法才会启动线程。上面的语句也可以写成 2 个，即：

```
Thread t = new PrintThread("Good!");
t.start();
```

笔者再叮咛一次，：“PrintThread 的实例”和“线程本身”是两个不同的部分。即使建立了 PrintThread 的实例，也还没有启动线程，而且就算线程已经结束，PrintThread 的实例也不会就这样消失不见。

主线程在 Main 类的 main 方法里启动 2 个线程。因为 main 方法会立即结束，所以主线程也会立即结束，不过整个程序还没有结束喔。别忘了，在字符串输出结束之前，已经启动的 2 个线程都还活着。一直要等到所有线程都已经结束，程序才会结束。当所有线程都结束，这个程序才会正式结束。

▶▶ 补充说明：程序结束

当所有线程都结束时，程序也就结束了。不过，在判断是否结束时，并不包括 daemon thread。换句话说，即使 daemon thread 还在，但只要非 daemon thread 的线程都已经结束，程序即已结束。

daemon thread 是指在产生线程时，以 setDaemon 方法设置其为 daemon thread 的线程。

建立 Thread 类的子类、建立其实例、调用 start 方法……，这是利用 Thread 类的子类以启动线程的方法。

线程的启动(2)——利用 Runnable 接口

继续说明利用实现了 Runnable 接口的类的实例以启动线程的方法。Runnable 接口是 java.lang Package 里的接口，声明方法如下：

```
Public interface Runnable{
    Public abstract void run();
}
```

已实现 Runnable 接口的类必须实现 run 方法[注2]。

Printer 类（LIst I1-6）是表示一个连续输出 10000 次指定字符串的线程。欲输出的字符串以构造器的参数指定，存储在 message 字段。Print 类则声明为已实现（implements）了 Runnable 接口的类。此时，不必再将 Printer 类设为 Thread 的子类。

注 2：Runnable 接口虽然已经实现，但仍可声明未实现 run 方法的类，不过这个类会变成抽象类。

List I1-6 输出指定字符串的 Printer 类（Printer.java）

```
public class Printer implements Runnable {
    private String message;
    public Printer(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.print(message);
        }
    }
}
```

程序收录在下载文件夹 Introduction1/Others/Printer 内

Main 类（List I1-7）这个程序则是先建立 2 个 Printer 类的实例，再利用它们来启动 2 个线程。

List I1-7 利用 Runnable 接口启动 2 个线程的程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new Thread(new Printer("Good!")).start();
        new Thread(new Printer("Nice!")).start();
    }
}
```

程序收录在下载文件夹 Introduction1/Others/Printer 内

在 List I1-7 里，建立 Thread 类的实例时，就输出 Printer 类的实例作为构造器的参数。然后才利用 start 方法启动线程。也就是：

```
new Thread(new Printer("Good!")).start();
```

这里是先重新建立 Printer 类的实例，把它传出去作为参数后，再建立 Thread 类的实例，将该实例的 start 方法调用出来的语句。

调用出 start 方法后，就会启动新的线程。然后，这个线程再调用 Printer 类的实例的 run 方法。最后就变成是新启动的线程去进行连续输出 10000 次“Good!”的工作。上面的语句也可以写成 3 个，即：

```
Runnable r = new Printer("Good!");
Thread t = new Thread(r);
t.start();
```

建立一个实现了 Runnable 接口的类，将该实例传给 Thread 的构造器，调用 start 方法……，这就是利用 Runnable 接口来启动线程的方法。

无论是利用 Thread 类的子类的方法（1），还是利用实现了 Runnable 接口的类的方法（2）也好，用来启动新线程的方法永远是 Thread 类的 start 方法。

▶▶ 补充说明：Thread 类和 Runnable 接口

Thread 类本身也实现了 Runnable 接口，也有 run 方法。只不过 Thread 类的 run 方法主体里是空的，没有执行任何部分。Thread 类的 run 方法通常都是被子类覆盖（override）。

线程的暂时停止

讲到这里，好像也该休息一下了……，呵呵，要休息的是线程啦。本节将介绍暂时停止线程执行操作的方法。

利用 Thread 类的 sleep 方法即可暂时停止线程的执行操作。sleep 的英文原意是“睡着”的意思，sleep 是 Thread 类的类方法。

执行下面的语句时，即可让当前的线程（执行此语句的线程）暂时停止约 1000ms（约 1s）。

```
Thread.sleep(1000);
```

List I1-8 是一个连续输出 10 次“Good!”的程序，约每显示 1 次就停止线程 1000ms（约 1s）。也就是约每隔 1s 即输出“Good!”的意思。

List I1-8　约每隔 1s 即输出“Good!”的程序（Main.java）

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print("Good!");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

程序收录在下载文件夹 Introduction1/Others/Sleep 内

执行结果如图 I1-6 所示。

```
Good! Good! Good! Good! Good! Good! Good! Good! Good! Good!
←约每隔 1 秒输出 10 次 "Good!"
```

图 I1-6　执行结果

sleep 方法的调用被放在 try-catch 里面，这是因为 sleep 方法可能会抛出一个称为 InterruptedException 的异常。InterruptedException 是用在取消线程处理时的异常，详细解说请参见第 5 章的进阶说明。

在 List I1-8 里，让 catch 里面空无一物，这样即使抛出 InterruptedException 也不会做什么特别处理（即忽略）。

本书在模拟“较花费时间的处理”时常会用到 sleep 方法，不过实际写程序时的使用频率并没有这么高。勉强来说，大概只有在设计几分钟（或秒）后自动关闭的对话框或让用户只看到单击按钮时那一瞬间的样子时才会用到。

▶▶ 补充说明：设置 ns 单位

利用 sleep 方法所设置的暂时停止时间并不是很精密，所以不太适合用在实时性控制的方面。若利用以下的语法结构，则可以 ns（10^{-9}s）为单位设置停止时间。

```
Thread.sleep(ms, ns);
```

不过，一般的 Java 处理系统并不需要用到这么精密的控制方式。事实上，控制的精确度有多高要依 Java 处理系统而定。

▶▶ 补充说明：该怎么唤醒呢？

如果是要“半路唤醒”被 Thread.sleep 暂停的线程，则可利用 interrupt 方法。详细说明请参见第 5 章的“sleep 方法与 interrupt 方法”。类本身也实现了 Runnable 接口，也有 run 方法。只不过 Thread 类的 run 方法主体里是空的，没有执行任何部分。Thread 类的 run 方法通常都是被子类覆盖（override）。

线程的共享互斥

在多线程程序里，多个线程可自由操作，当然就可能同时操作到同一实例。这个情况有时会造成不必要的问题。假设现在是要从银行账户取钱，确认可用余额这部分的程序代码应为：

```
if (可用余额大于欲提领金额) {
  从可用余额减掉欲提领金额
}
```

先确认可用余额，再检查是否允许提领欲提领金额，如果可提领，则从可用余额减掉欲提领金额。这样才不会发生可用余额变负数的问题。

但是，若让 2 个线程同时执行这个程序代码，可用余额可能会变成负数。

举例来说，假设：

可用余额 = 1000 元

欲提领金额 = 1000 元

请见图 I1-7。

线程 A	线程 B
“可用余额大于欲提领金额？”	
→是。	
	<<<<在这里切换成线程 B>>>>
	“可用余额大于欲提领金额？”
	→是。
	“从可用余额减掉欲提领金额”
	→可用余额变 0 元。
<<<<在这里切换成线程 A>>>>	
“从可用余额减掉欲提领金额”	
→可用余额变成-1000 元。	
（时间轴由上而下）	

图 I1-7　线程 B 夹在线程 A 所执行的 2 个处理之间

当线程 A 和线程 B 同时操作时，由于时间差，可能会发生线程 B 夹在线程 A 的“确认可用余额”和“减去可用余额”这 2 个处理之间的情况。

这时候需要有一种“交通管制”来协助防止发生这类问题。例如，当某个线程正在执行这个部分时，则限制其他线程不可执行此部分。这种像是交通管制的操作通常称为共享互斥或互斥控制，英文写成 mutual exclusion。道理就跟十字路口红绿灯的处理一样，当直向行车是绿灯时，另一边的横向行车一定是红灯。

Java 语言在处理线程的共享互斥时，会使用到关键字 synchronized。

synchronized 方法

当一个方法加上关键字 synchronized 声明之后，就可以让 1 个线程操作这个方法。“让 1 个线程操作”并不是说只能让某一个特定的线程操作而已，而是指 1 次只能让 1 个线程执行。这种线程称为 synchronized 方法，又称为同步方法。

List I1-9 是利用到 synchronized 方法的类范例。在 Bank（银行）类里，deposit（存款）和 withdraw（取款）这 2 个方法都是 synchronized 方法。

List I1-9 具有 deposit 和 withdraw 两个 synchronized 方法的 Bank 类（Bank.java）

```
public class Bank {
    private int money;
    private String name;

    public Bank(String name, int money) {
        this.name = name;
        this.money = money;
    }

    // 存款
    public synchronized void deposit(int m) {
        money += m;
    }

    // 取款

    public synchronized boolean withdraw(int m) {
        if (money >= m) {
            money -= m;
            return true;      //已取款
        } else {
            return false;     //余额不足
        }
    }

    public String getName() {
        return name;
    }
}
```

程序收录在下载文件夹 Introduction1/Others/Sync 内

当 1 个线程正在执行 Bank 实例的 deposit 方法时，其他线程就不能执行同一实例的 deposit 方法或 withdraw 方法。欲执行的线程必须排队等候。

Bank 类里还有一个 getName 方法。它不是 synchronized 方法，所以无论其他线程是否正在执行 deposit 或 withdraw，getName 方法随时都能执行[注3]。

一个实例的 synchronized 方法只能允许 1 次 1 个线程执行。但是，非 synchronized 的方法就没有这个限制，同时可供 2 个以上的线程执行，如图 I1-8 所示。这个图是有 2 个线程同时执行 getName 方法的情况。

注 3：getName 没有成为 synchronized 方法的原因，是因为此方法未使用可同时读写的字段。

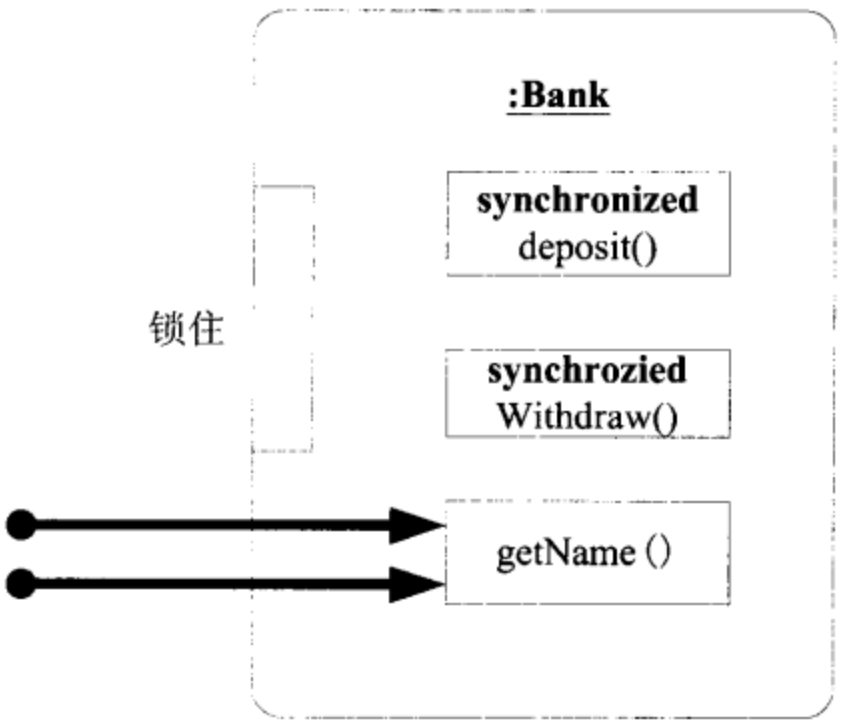


图 I1-8　多个线程可同时执行非 synchronized 方法的 getName 方法

另一方面，synchronized 方法不允许同时有 1 个以上的线程执行。图 I1-8 以在 synchronized 方法左方放一个“锁定”的长方形来表示。一旦线程获取这样的锁定（acquire）时，这里就会像筑起防护罩一样防止其他线程进入。

图 I1-9 则是表示 1 个线程执行 deposit 方法的情形。由于这个线程已经获取锁定，因此其他线程就不能再执行同一实例的 synchronized 方法。在图 I1-9 里，表示锁定的长方形被涂灰是表示已经获取锁定。

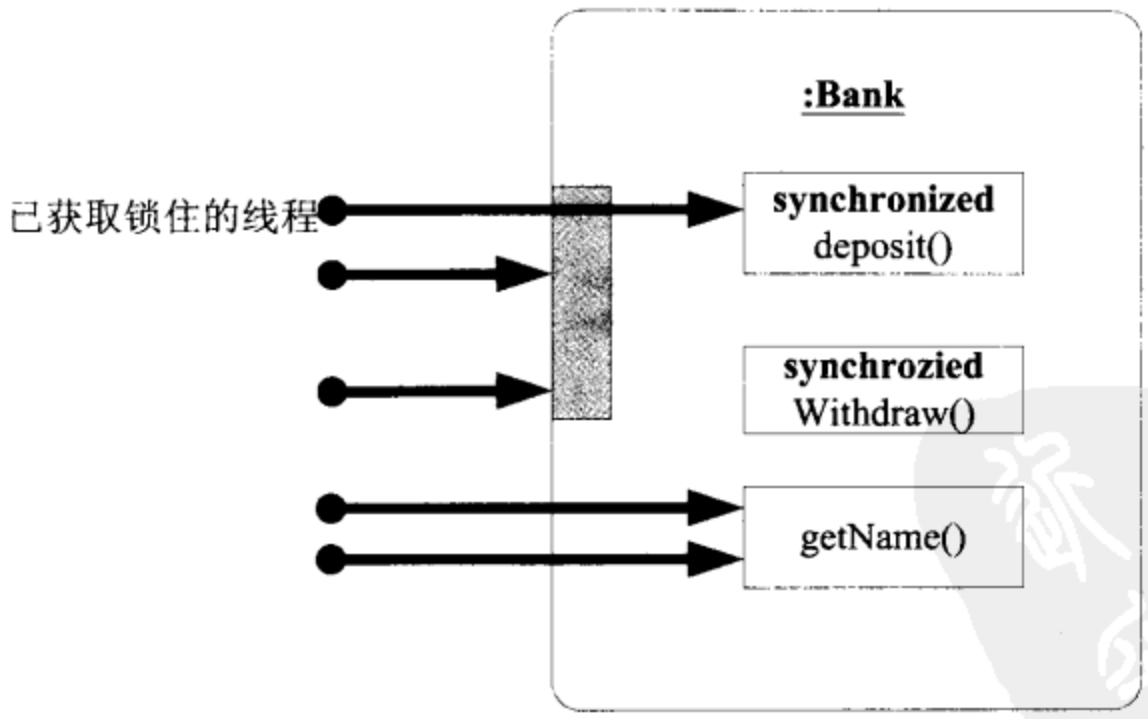


图 I1-9　synchronized 方法只允许 1 次有 1 个线程执行

请注意：在图 I1-9 当中，有无锁定对非 synchronized 的 getName 方法完全没有影响。即使线程没有获取锁定，多个线程仍可自由进入非 synchronized 的方法。

当执行 synchronized 方法的线程结束执行该方法后，锁定即被释放（release）。

图 I1-10 用锁定的长方形变白来表示释放锁定。

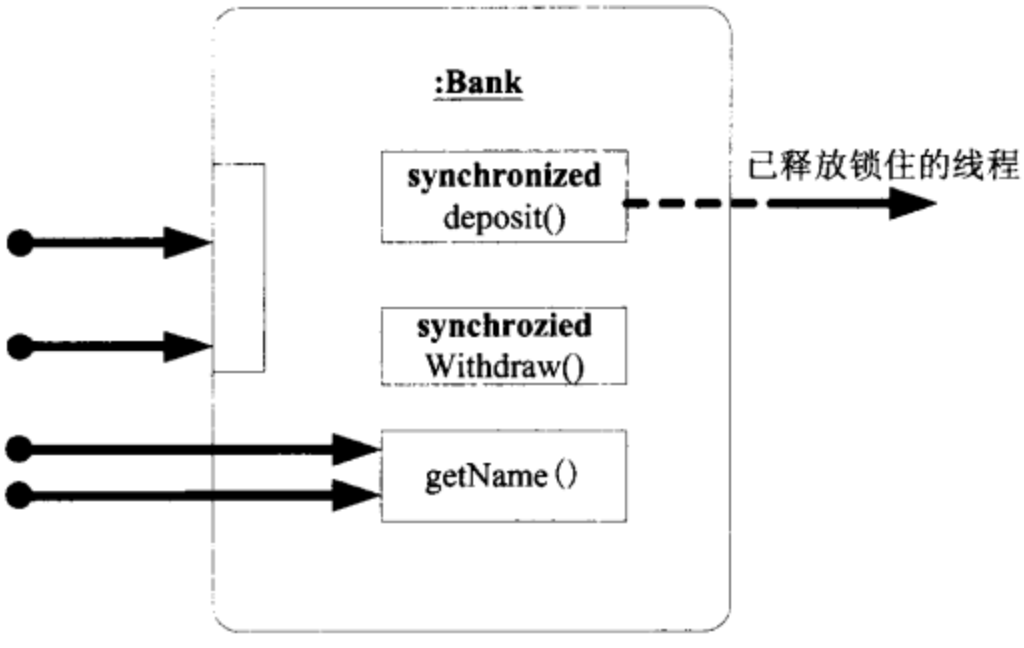


图 I1-10　线程结束执行 synchronized 方法 deposit，释放锁定

当锁定被释放后，刚才因为锁定而不得其门而入的多个线程便开始抢夺锁定，一定会有 1 个线程获取锁定。在同一时间点能获取锁定的也只有 1 个线程，若等待的线程不只 1 个，没抢到的线程就得继续等候。图 I1-11 是表示重新获取锁定的另一个线程开始执行 synchronized 方法的情况。

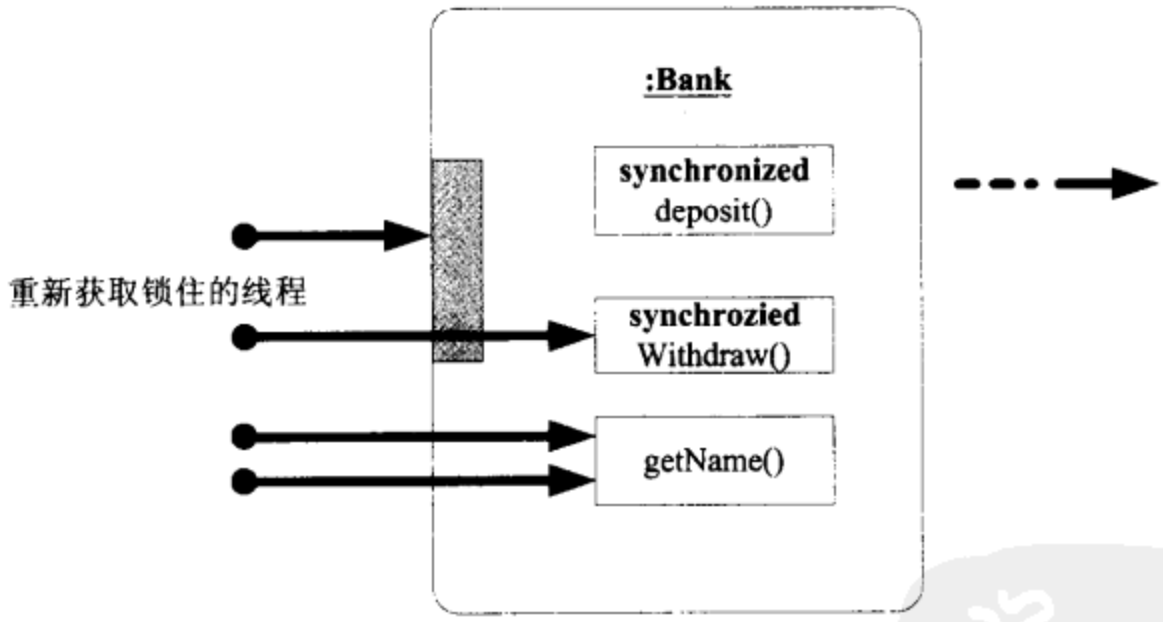


图 I1-11　另一个线程获取锁定开始执行 synchronized 方法

只要有实例就会相对有 1 个锁定。因此不是说因为某个实例的 synchronized 方法正在执行中，导致无法执行其他实例的 synchronized 方法。图 I1-12 是假设有 bank1 和 bank2 两个实例，而由其他线程同时执行 synchronized 方法的情况。

▶▶ 补充说明：锁定和监视

线程共享互斥的架构称为监视(monitor)，而获取锁定有时也称为“持有(own)监视”。

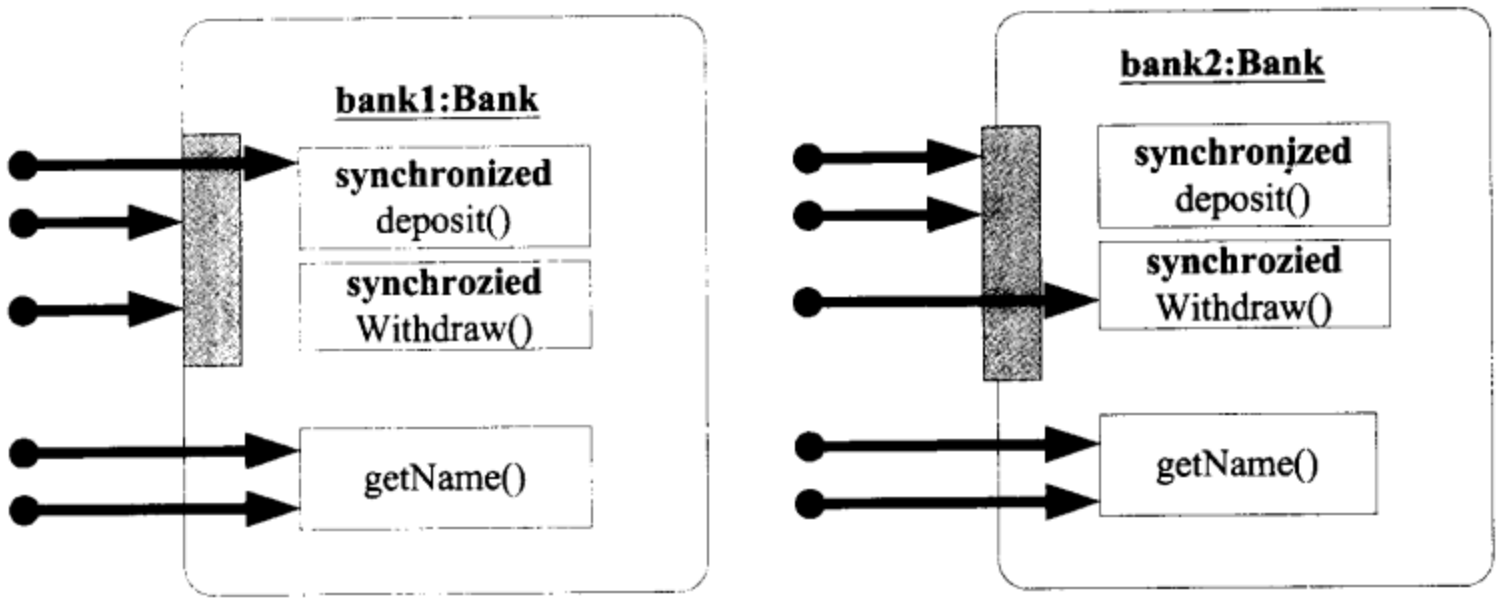


图 I1-12　只要有实例就会相对有 1 个锁定

synchronized 阻挡

如果只想启动方法里一部分的线程，而非启动整个方法时，则可使用 synchronized 阻挡，其格式如下：

```
synchronized (表达式){
  ...
}
```

则可把获取锁定的实例传给“表达式”这部分。如欲更精密控制共享互斥的范围，则需使用 synchronized 阻挡，具体范例请见第 6 章的说明。

◆ *synchronized 实例方法和 synchronized 阻挡*

假设现在有一个类型如下的 synchronized 实例方法：

```
synchronized void method(){
  ...
}
```

在功能上，它跟下面这个以 synchronized 阻挡为主的方法有异曲同工之妙。

```
void method() {
    synchronized (this) {
        ...
    }
}
```

换句话说，synchronized 实例方法是使用 this 锁定去做线程的共享互斥。

◆ *synchronized 类方法和 synchronized 阻挡*

假设现在有一个类型如下的 synchronized 类方法，synchronized 类方法有限制同

时只能让 1 个线程执行。这个部分跟 synchronized 实例方法一样。但是，利用 synchronized 类方法的锁定跟利用 synchronized 实例方法并不相同。

```
class Something{
    static synchronized void methos() {
        ...
    }
}
```

在功能上，它跟下面这个以 synchronized 阻挡为主的方法有异曲同工之妙。

```
class Something{
    static void method() {
    synchronized (Something.class) {
        ...
        }
    }
}
```

换句话说，synchronized 类方法是使用该类的类对象的锁定去做线程的共享互斥。Something.class 是对应 Something 类的 java.lang.Class 类的实例。

▶▶ 补充说明： synchronized 语句

“Java 规格（The Java Language Specification）”（参见附录 E[JIS]）则把 synchronized 阻挡叫做 synchronized 语句（synchronized statement）。

有关线程共享互斥的部分，在 Single Threaded Execution（第 1 章）也有详细说明。

线程的协调

如前节所述，当有 1 个线程正在执行 synchronized 方法时，其他线程无法执行该方法。这是简单型的共享互斥。

假设我们现在想做更进一步的处理，不只是有线程再执行就得乖乖等候的简单型而已，例如：

- 若该空间有空闲则写入数据，若非空闲则等候直到空出为止（等候）；
- 当该空间已空出，则通知正在等候的线程（通知）。

这是根据“空间是否有空闲”为条件所进行的线程处理。Java 里有 wait、notify 和 notifyAll 三个方法可进行这个处理。wait 是让线程乖乖等候的方法，而 notify 和 notifyAll 则是启动等候中线程的方法。

wait set——线程的休息室

在开始学习 wait、notify 和 notifyAll 之前，先稍微讲讲 wait set。所有实例都有一个 wait set，wait set 是一个在执行该实例的 wait 方法时、操作停止的线程的集合。它有点类似线程的休息室，而且每个实例都会有。

一执行 wait 方法时，线程便会暂时停止操作，进入 wait set 这个休息室。除非发生下列其中一种情况，否则线程会永远被留在这个 wait set 里。当发生下列任一情况时，线程便会退出 wait set。

- 有其他线程以 notify 方法唤醒该线程；
- 有其他线程以 notifyAll 方法唤醒该线程；
- 有其他线程以 interrupt 方法唤醒该线程；
- wait 方法已经到期。

以下依次用图示说明 wait、notify 和 notifyAll 三种方法。至于 interrupt 方法的部分，则请参见第 5 章“wait 方法与 interrupt 方法”的解说；wait 方法的到期，则请参见第 4 章的进阶说明。

wait 方法——把线程放入 wait set

使用 wait（等候）方法时，线程便进入 wait set。假设现在已经执行一个如下的语句：

```
obj.wait();
```

则目前的线程会暂时停止执行，进入实例 obj 的 wait set。这个操作称为：

线程在 obj 上 wait

如果实例方法里还有如下的语句时：

```
wait();
```

则其意义同：

```
this.wait();
```

故执行 wait()的线程就会进入 this 的 wait set。此时，就变成了线程在 this 上 wait。

如欲执行 wait()方法，线程需获取锁定（这是规则）。但是当线程进入 wait set 时，已经释放了该实例的锁定。整个过程的图示如图 I1-13～I1-15 所示。

▶▶ 补充说明：wait set

wait set 是一个虚拟的概念。它既不是实例的字段，也不是可以获取在实例上 wait 中线程的列表的方法。

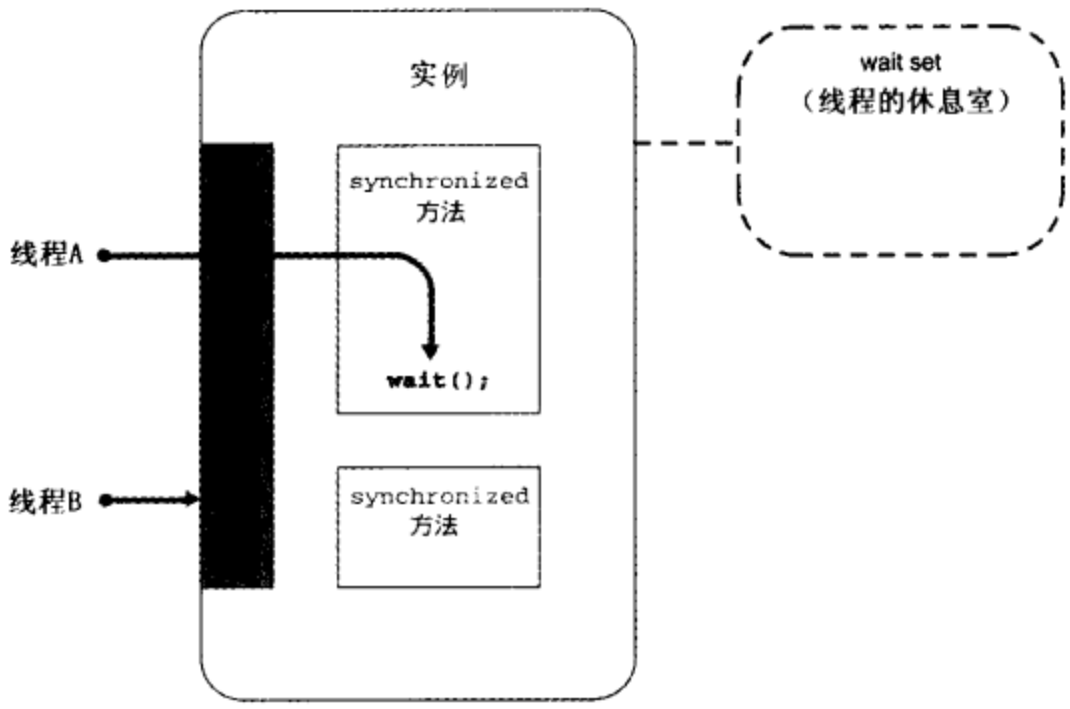


图 I1-13　获取锁定的线程 A 开始执行 wait 方法

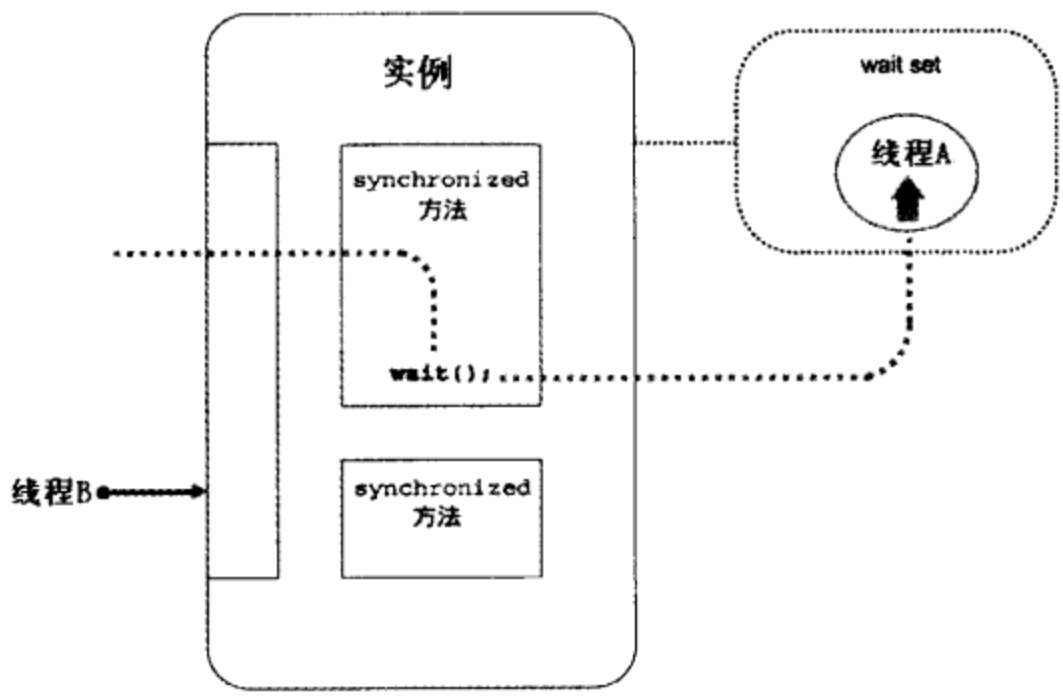


图 I1-14　线程 A 进入 wait set，并释放锁定

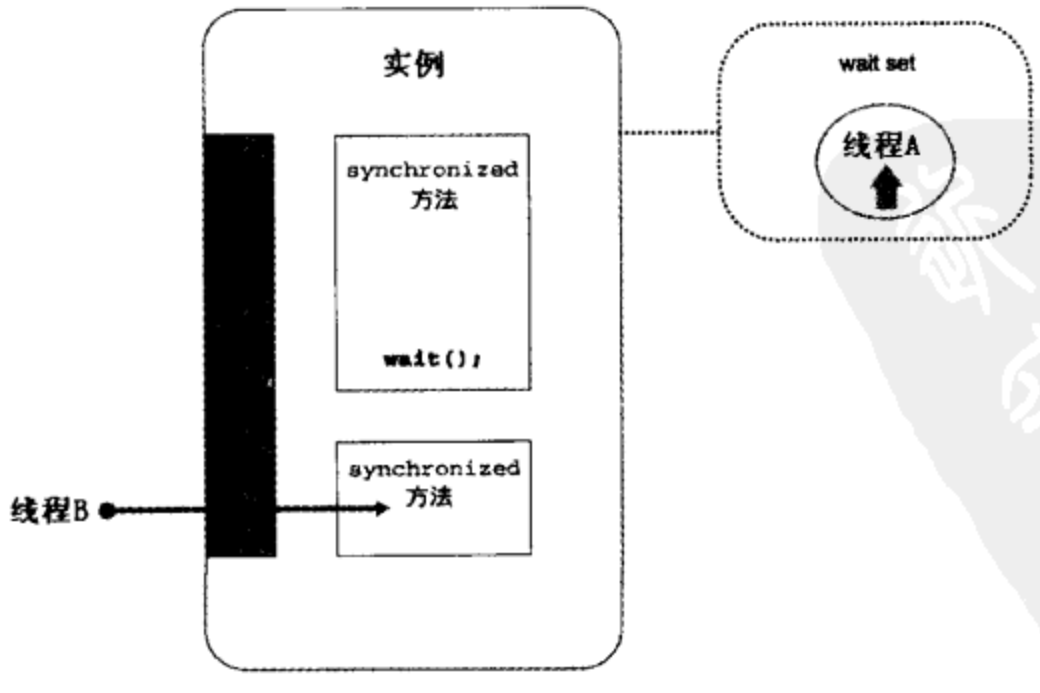


图 I1-15　于是线程 B 便可获取锁定

notify 方法——从 wait set 拿出线程

使用 notify（通知）方法时，可从 wait set 里抓出 1 个线程。假设现在已经执行一个如下的语句：

```
obj.notify();
```

则从 obj 的 wait set 里的线程中挑出 1 个，唤醒这个线程。被唤醒的线程便退出 wait set。

这个过程的图示如图 I1-16～I1-19 所示。

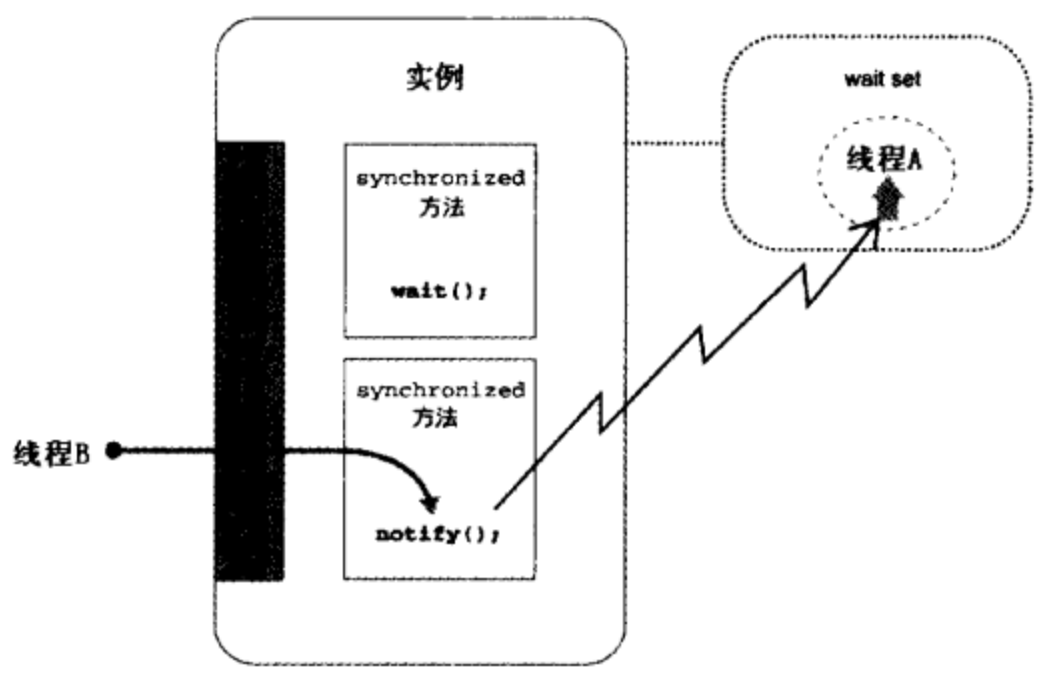


图 I1-16　获取锁定的线程 B 开始执行 notify 方法

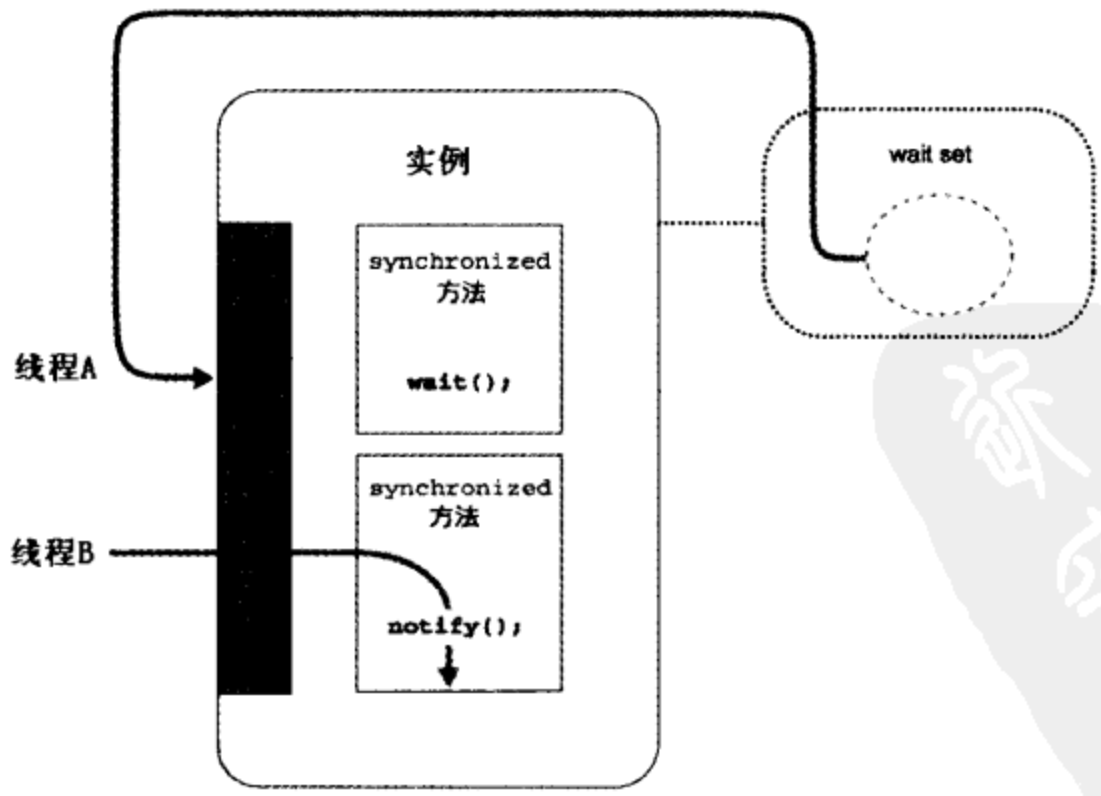


图 I1-17　线程 A 退出 wait set，打算继续进行 wait 的下一个操作；但刚才执行 notify 的线程 B 还握着锁定不放

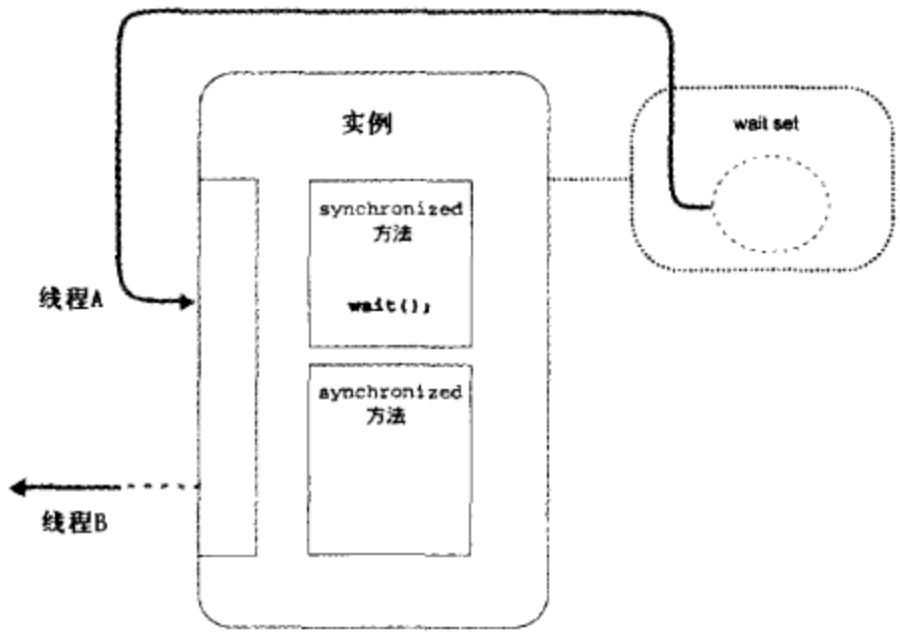


图 I1-18　刚才执行过 notify 的线程 B 开始释放锁定

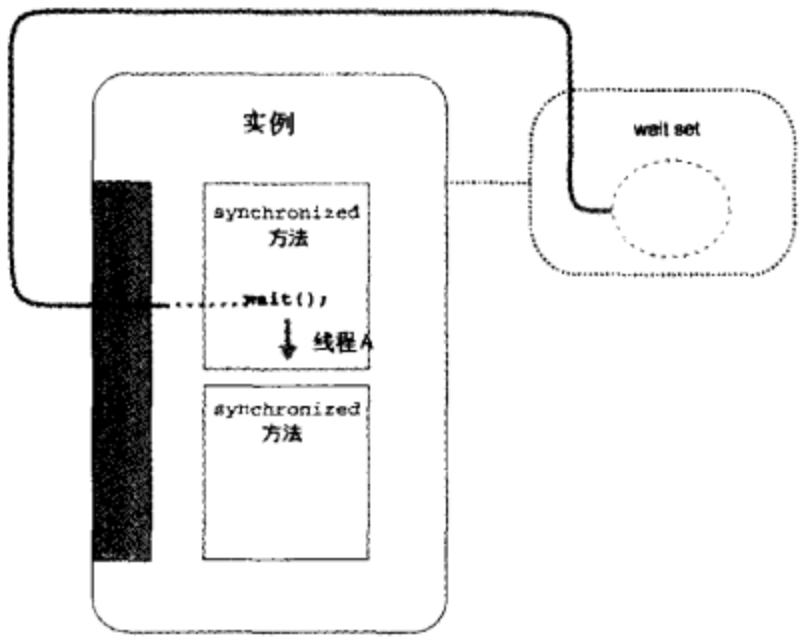


图 I1-19　已经退出 wait set 的线程 A 获取锁定，进行 wait 的下一个操作

线程必须有欲调用的实例的锁定，才能执行 notify 方法，这点跟 wait 方法一样（也是规则）。

▶▶ 补充说明：notify 后的线程

被 notify 唤醒的线程并不是在 notify 的一瞬间重新开始执行。因为在 notify 的那一刻，执行 notify 的线程还握着锁定不放，所以其他线程无法获取该实例的锁定（见图 I1-17）。

▶▶ 补充说明：notify 如何选择线程

假设执行 notify 方法时，wait set 里正在等候的线程不只 1 个。规格里并没有注明此时该选择哪一个线程。究竟是选择等待线程列表中的第一个、随机选择或是另以其他方式选择，则依 Java 处理系统而异。因此在写程序时，程序属性最好不要写成会因所选线程而有所变动。

notifyAll 方法——从 wait set 拿出所有线程

使用 notifyAll（通知全体）方法时，会将所有在 wait set 里苦等的线程都拿出来。假设现在的情况是：

```
obj.notifyAll();
```

则会唤醒所有留在实例 obj 的 wait set 里的线程。

若实例方法写成：

```
notifyAll();
```

则其意义同：

```
this.notifyAll();
```

故这个语句所在的方法的实例（this）的 wait set 里的线程会全部放出来。

图 I1-20、图 I1-21 是实例比较 notify 方法跟 notifyAll 方法两者操作的差异。notify 只唤醒 1 个线程，而 notifyAll 则是唤醒所有线程。两者的差别就只有这里。

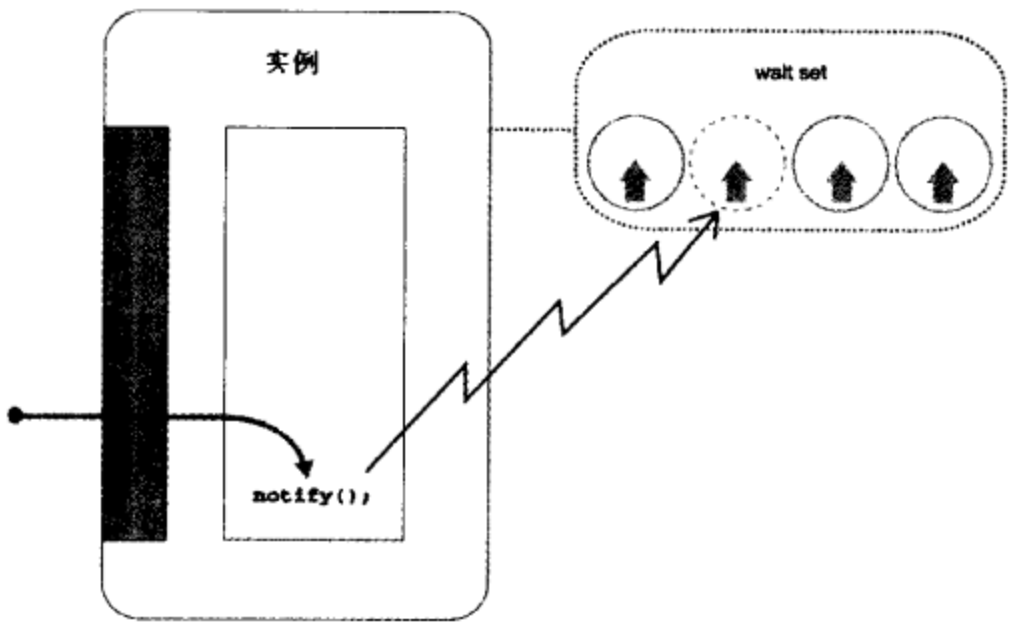


图 I1-20　notify 方法只唤醒 1 个线程，然后该线程退出 wait set

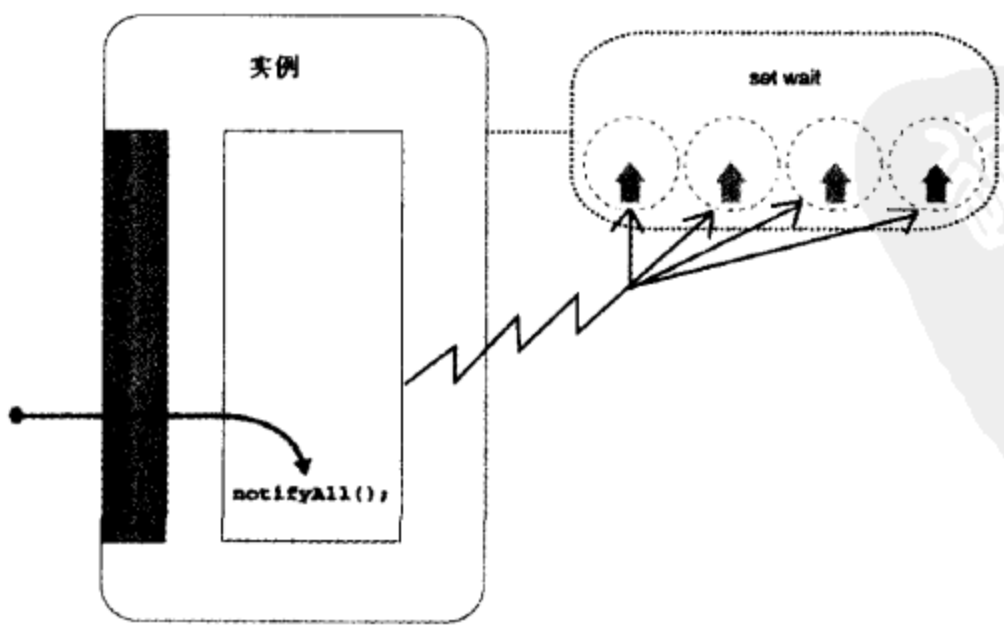


图 I1-21　notifyAll 方法是唤醒所有线程，让它们全部退出 wait set

跟 wait 方法和 notify 方法一样，线程必须获取欲调用实例的锁定，才能调用 notifyAll 方法。

被唤醒的线程便开始去获取刚才进入 wait 时释放掉的锁定，那么，这个锁定现在是在谁手中呢？没错，锁定就是在刚才执行 notifyAll 方法的线程手里。因此即使所有线程都已经退出 wait set，但它们仍然在去获取锁定的状态下，还是有阻挡。要等到刚才执行 notifyAll 方法的线程释放出锁定之后，其中 1 名幸运儿才能实际执行。

▶▶ 补充说明：要是没有锁定呢？

若没有锁定的线程去调用 wait、notify 或 notifyAll 时，便会抛出异常 java.lang.IllegalMonitorStateException。

▶▶ 补充说明：该用 notify 方法还是 notifyAll 方法？

notify 方法和 notifyAll 方法两者非常相似，到底该用哪一个？老实说，这个选择有点难。

选择 notify 的话，因为要唤醒的线程比较少，程序处理速度当然比 notifyAll 略胜一筹。

但选择 notify 时，若这部分处理得不好，可能会有程序挂掉的危险性。一般说来，选择 notifyAll 所写出来的程序代码会比选择 notify 可靠。

除非你能确定程序员对程序代码的意义和能力限度一清二楚，否则选择 notifyAll 应该比较稳扎稳打。选择 notify 时所派生问题的案例则留在第 5 章的练习问题 5-8 再讨论。

wait、notify、notifyAll 是 Object 类的方法

wait、notify、notifyAll 都是 java.lang 的 Object 类的方法，不是 Thread 类固有的方法。

请各位再回想一下 wait、notify、notifyAll 的作用：

- obj.wait( )是把现在的线程放到 obj 的 wait set；
- obj.notify( )是从 obj 的 wait set 里唤醒 1 个线程；
- obj.notifyAll ( )是唤醒所有在 obj 的 wait set 里的线程。

换句话说，把 wait、notify、notifyAll 三者均解释为对实例的 wait set 的操作，会比说它们是对线程的操作更贴切。由于所有实例都会有 wait set，所以 wait、notify、notifyAll 才会是 Object 类的方法。

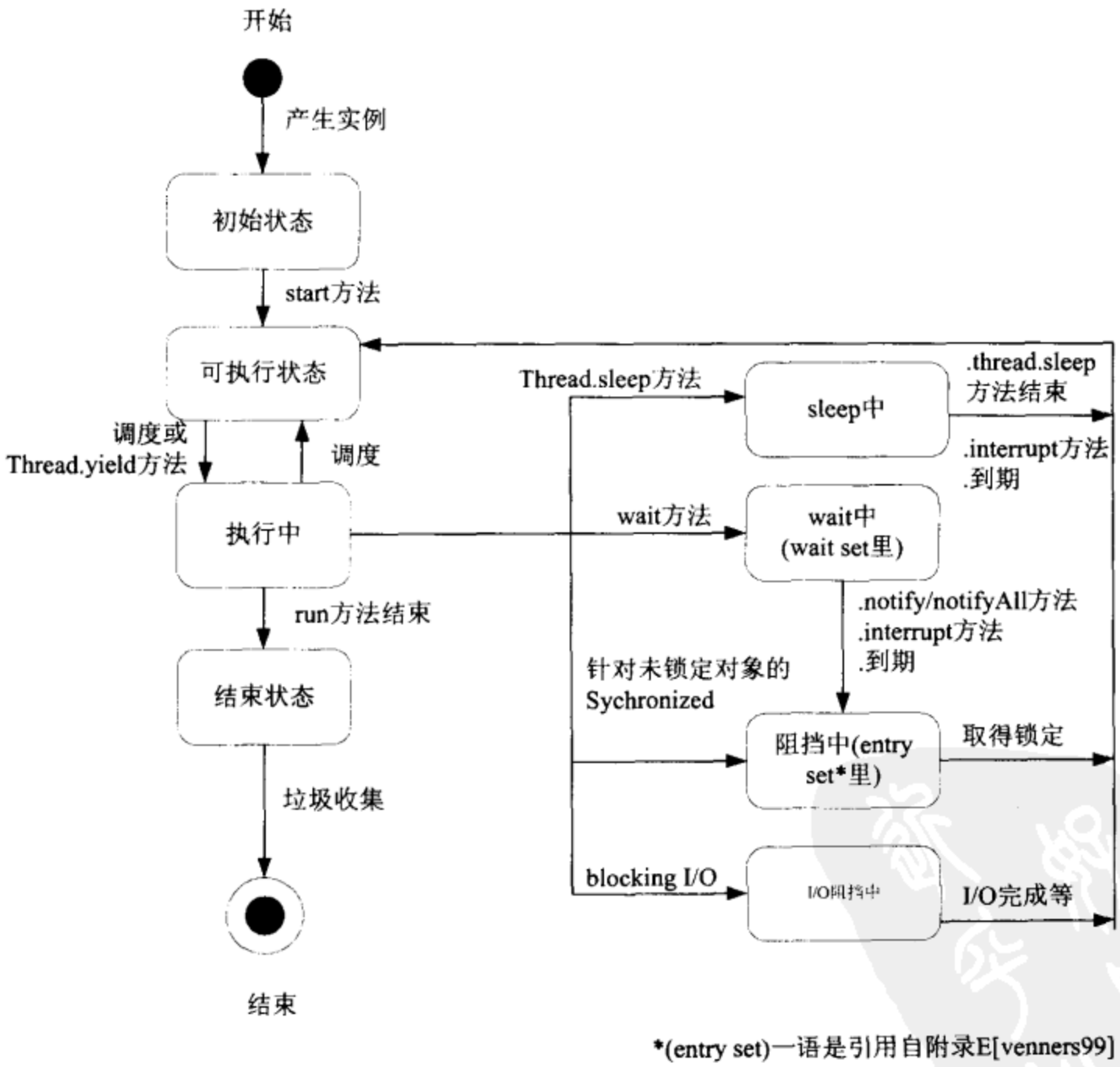
▶▶ 补充说明：wait、notify、notifyAll 也是 Thread 类的方法

wait、notify 和 notifyAll 三者确实不是 Thread 类固有的方法。不过，因为 Object 类是 Java 所有类的祖先类，所以 wait、notify、notifyAll 也是 Thread 类的方法。

有关 wait、notify 和 notifyAll 的使用方法，留待 Guarded Suspension（第 3 章）另行详细说明。

线程的状态移转

现在将线程的状态移转图整理如下（见图 I1-22），请各位读者参考。



*(entry set)一语是引用自附录E[venners99]

图 I1-22　线程的状态移转图

以上状态移转图的制作是参考附录 E〔Lea〕〔JIS〕〔Holub00〕及“Java World”杂志 2002 年 4 月号专栏。

跟线程有关的其他话题

以下跟线程有关的话题会在相关章节继续说明：

- 取消线程处理的中断
  (Interrupt、isInterrupted、interrupted、InterruptedException)
  →第 5 章
- 线程的优先级
  (setPriority、getPriority)
  →附录 C
- 等候线程结束
  (join)
  →第 5 章、第 10 章、

▶▶ 补充说明：“有阻挡”和“被阻挡”

有时候，线程会因为某种因素而无法继续进行下去。例如，当线程 A 欲执行 synchronized 方法时，若其他方法已获取相同实例的锁定，则线程 A 无法向前推进。这个状态称为“线程 A 有阻挡”。

部分读者可能有不同看法，或许认为既然它是因外在因素而停止，应该以被动语气写成“被阻挡”比较好。

本书称这个状态为“有阻挡”，不过“有阻挡”跟“被阻挡”两者意思相同，请别太拘泥于表现方式。

重点回顾

各位在本章学习到的 Java 语言的线程相关内容如下：

- 何谓线程
- 线程的启动
- 线程的暂时停止
- 线程的共享互斥
- 线程的协调

请各位尝试作答下列的练习问题，即可知道自己是否已确实理解本章内容。解答部分在本书附录 A，不过请不要直接翻过去看解答，先自己好好想一想喔。

练习问题

解答请见附录 A

问题 I1-1（基本知识）

Java 请详阅下列的文章叙述，若叙述正确打○，若不正确则打×。

（1）Java 的程序中，最少会执行一个线程。

（2）Thread 类的 run 方法被调用出来后，就会启动新的线程。

（3）Runnable 接口中声明有 start 方法与 run 方法。

（4）有时候多个线程都可以调用出同一个实例的方法。

（5）有时候多个线程都可以调用出同一个 Thread 类的实例（Instance）的方法。

（6）执行 sleep 方法后，在指定的时间内所有的线程都会暂停执行。

（7）某个线程在执行 synchronized 方法时，其他的所有线程都会停止执行。

（8）执行 sleep 方法后，该线程只有在指定的时间内才会进入 wait set。

（9）调用 wait 的语句一定要写在 sychronized 方法中才行。

（10）notifyAll 方法就是 java.lang.Object 类的实例方法。

问题 I1-2（共享互斥）

Java 在执行范例（见图 I1-4）时，为何字符串“Good!”和“Nice!”的交错排列是以单字为单位，而不会交错字母排列呢？例如：

```
GoNiod!ce!
```

或

```
GooNdi!ce!
```

问题 I1-3（线程的操作）

Java 若启动下列程序时，则屏幕上会输出 1000 个*记号后，再输出 1000 个+记号（见图 I1-23）。请问：为什么输出结果不是*和+交错混杂呢？

List I1-10 哪里有问题？（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new PrintThread("*").run();
        new PrintThread("+").run();
    }
}
```

List I1-11 输出指定字符串的 PrintThread 类（PrintThread.java）

```
public class PrintThread extends Thread {
    private String message;
```

```
    public PrintThread(String message) {
        this.message = message;
    }
    public void run() {
        for (int i = 0; i < 1000; i++) {
            System.out.print(message);
        }
    }
}
```

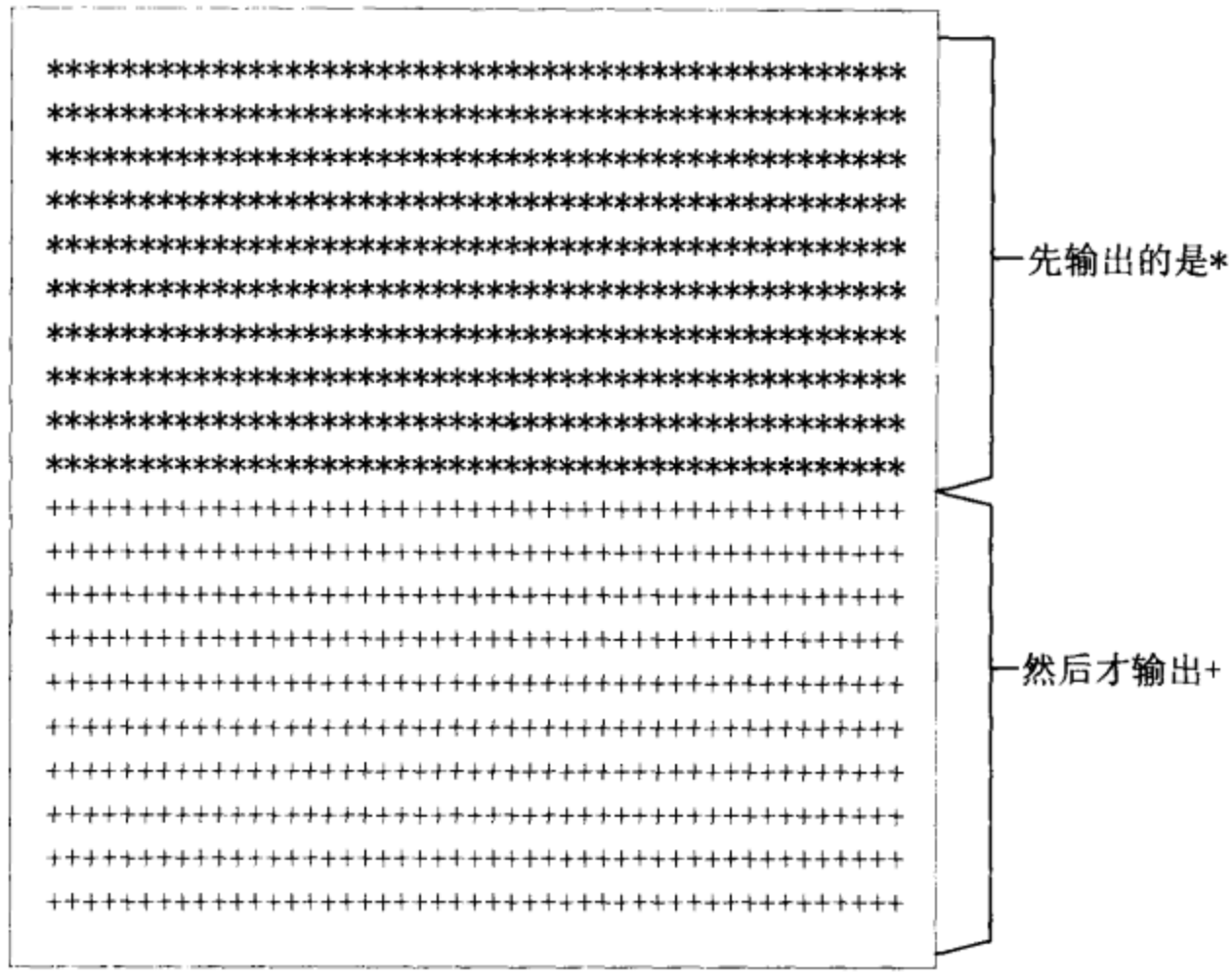


图 I1-23　执行结果

问题 I1-4（测知异常）

Java List I1-12 是沿用本章的 Bank 类（List I1-9），删除掉 synchronized 的部分，另加 1 个 check 方法，这样当可用余额变成负数时就会显示警告消息。试验证：若有 1 个以上的线程在使用这个银行（Bank 类）时，则可用余额会变成负数的可能性。

List I1-12　试验证可用余额可能会变成负数（Bank.java）

```
public class Bank {
    private int money;
    private String name;
    public Bank(String name, int money) {
```

```
    this.name = name;
    this.money = money;
}
public void deposit(int m) {
    money += m;
}
public boolean withdraw(int m) {
    if (money >= m) {
        money -= m;
        check();
        return true;
    } else {
        return false;
    }
}
public String getName() {
    return name;
}
private void check() {
    if (money < 0) {
        System.out.println("可用余额为负数! money =" + money);
    }
}
```

问题 I1-5（线程的暂时停止）

Java 现有某人试写 1 个程序，欲让已启动的线程暂时停止约 1s。但这个程序里面有错误，为什么错了呢？假设以下的 MyThread 同 List I1-2 所声明的 MyThread 类。

```
// 产生实例
MyThread t = new MyThread();
// 启动线程
t start();
// 暂时停止刚才启动的线程
try {
     t.sleep(1000);
} catch (InterruptedException e) {
}
```

问题 I1-6（共享互斥）

Java 假设现在如 List I1-13 一样声明 Something 类，变量 x、y 则表示 Something 类的不同实例。请详阅下面的组合，若该组合可供 1 个以上的线程同时执行，则打○；若该组合不允许 1 个以上的线程同时执行，则打×。

List I1-13 Something 类

```
public class Something {
    public                    void iA() {}
    public                    void iB() {}
    public       synchronized  void iSyncA() {}
    public       synchronized  void iSyncB() {}
    public static              void cA() {}
    public static              void cB() {}
    public static  synchronized   void cSyncA) {}
    public static  synchronized   void cSyncB() {}
}
```

（1）x.iA();
　　与
　　x.iA();

（2）x.iA();
　　与
　　x.iB();

（3）x.iA();
　　与
　　x.iSyncA();

（4）x.iSyncA();
　　与
　　x.iSyncA();

（5）x.iSyncA();
　　与
　　x.iSyncB();

（6）x.iSyncA();
　　与
　　y.iSyncA();

（7）x.iSyncA();
　　与
　　y.iSyncB();

（8）x.iSyncA();
　　与
　　Something.cA();

（9）x.iSyncA();
　　与
　　Something.cSyncA();

（10）Something.cSyncA();
　　与
　　Something.cSyncA();

（11）Something.cSyncA();
　　与
　　Something.cSyncB();

（12）x.cSyncA();
　　与
　　y.cSyncB();

Introduction 2

多线程程序的评量标准

多线程程序的评量标准

如果你对一个程序的评价方式还停留在“这个程序写得不错”、“这个程序写得太差”的程度，这样很难看出你是否确实分析了程序的构造。不要只是“好不好”，应该要有一个“好在哪里，不好在哪里”的评量标准。

本章是根据 Doug Lea 的分类方式，探讨多线程程序的评量标准。进一步的详细内容可参考附录 E[Lea]。

安全性——不损坏对象

对象损坏是一种比喻手法，因为对象是内存上的虚拟实体，不可能有实际损坏的问题。对象损坏则是指对象的状态不符合设计师的设计原意，通常是对象的字段所取得的值并非预期值。

以前面 Introduction1 介绍过的银行账户处理为例，若设计师本来没有想过会有负数的银行账户余额变成负数时，就是表示银行账户的对象已损坏。

可容 1 个以上线程使用而不危及安全性的类称为 thread-safe（线程安全）的类。类函数库里有 thread-safe 的类，也有非 thread-safe 的类，因此当多线程程序在利用到类函数库时要特别注意。例如，java.util.Vector 类是 thread-safe 的类，但 java.util.ArrayList 类就不是 thread-safe 的类。一般来说，该类是否为 thread-safe 均记载在 API 参考数据内。

▶▶ 补充说明：thread-safe 和 thread-compatible

java.util.ArrayList 类不是 thread-safe，但只要有做适当的共享互斥，还是可以放心使用。附录 E 的〔Bloch〕把这种情况称为 thread-compatible（线程兼容）。详细内容可参见〔Bloch〕的第 52 项说明。

生存性——进行必要的处理

生存性（liveness）是指也许不是现在但一定会进行必要的处理。这也是可正常使用的程序的必要条件之一（liveness 一词也有人翻译成“活性”的意思）。

不见得满足了不损坏对象的条件，就一定是好程序。讲得夸张点，假设现在有一个程序执行到一半突然停止。此时，既然程序没有进行任何处理，对象状态就没有变化；而对象状态没有变化，当然也不会异常，在逻辑上，这个情况也满足了维护“安全性”的条件。但是一个不能动的程序根本没有用，虽然有还是等于没有啊。“也许不是现在但一定会进行必要的处理”，这就是生存性。

有时候，安全性跟生存性刚好互相抵触，例如，越想提高安全性就会相对

降低生存性。最典型的例子就是死锁（deadlock），这是一种 1 个以上的线程在彼此等待对方释放锁住的状况。详细说明请参见 Single Threaded Execution（第 1 章）。

复用性——可再利用类

复用性（reusability）是指可再利用类。这并不是必备的条件，但却是提高程序质量的重点条件。

如果 A 类原来是一个已确认可正常使用的软件程序的一部分，而且 A 类又可切出来当作零件，这个 A 类就具有高度的复用性。

在写多线程程序时，若能巧妙地将线程的共享互斥结构和方针隐藏在类里，这就是一个具有高度复用性的程序。

性能——能快速、大量进行处理

性能（performance）是指能快速、大量进行处理。这也不是必备的条件，但仍是欲提高程序质量时应考虑的重点部分。

影响性能优劣的因素又分好几种，下面是根据 Lea 的性能分类摘录出来的主要部分：

数据吞吐量（throughput）是指在一定时间内所能完成的处理量。能处理的数量越多，则表示数据吞吐量越高。

响应性（responsiveness）是指从发出要求到收到反应的时间。所费时间越短，则程序的响应性越高。以 GUI 为例，到处理“结束”的所费时间可能比到处理“开始”的所费时间更重要。前者是指实际处理所花费的时间，后者则是程序第一时间的反应返回到用户那端所花费的时间。简单来说，“点选按键后都没反应，等到 10 秒后画面上才出现“处理完毕”的消息”跟“点选按键后立即显示“处理中……”的消息”两者相比较之后，后者的响应性显然比较高。无论一个处理要花多少时间才能处理完，“立即能收到响应”才会让用户比较放心。这个响应性又称为等待时间（latency）。

容量（capacity）是指可同时处理的数量。例如，服务器可同时处理的客户端数量或文件数量等。

其他像是效率（efficiency）、可伸缩性（scalability）、退化（degradation）等，都可作为性能的评量标准。

有时候，这些因素可能会互相抵触（也可说是有得必有失的关系）。举例来说，如果想要较高的数据吞吐量，通常会相对降低响应性。有点像是为了提高工作性能而目不转睛、专心打字时，要是有人开口打招呼，打字员的响应就会慢半拍。这个响应慢半拍等于是响应性变低的意思。另外，提高安全性的措施

可能会降低程序的性能（如数据吞吐量变少）。这有点像是为了不想搞迷糊，故意减少一次同时处理的工作量，此时，在一定时间内能处理的工作量自然就会降低。

评量标准的总结

程序既不能损害对象，也一定要进行必要的处理。设计多线程程序时，必须维持程序的安全性（safety）和生存性（liveness）。

满足这两个必要条件之后，再来考虑如何提高复用性（reusability）和性能（performance）。

根据下列评量标准分析各个 pattern：

- 安全性与生存性——必要条件；
- 复用性与性能——提高程序质量。

请注意一件事，在 pattern 中特别把程序设计上的评量标准称为约束力（force）。约束力是指设计师所面对的“规定”或“压力”。尤其是跟同时发生有关的 pattern，安全性、生存性、复用性和性能都应该算是相当重要的约束力吧！

重点回顾

各位在本章学习了多线程程序的评量标准，而且这些评量标准往往都是有得必有失，很难鱼与熊掌兼得。

下一章开始将正式开始介绍各个 pattern，喔！在这之前可别忘了利用下面的练习问题测验自己是否已经确实理解。

练习问题

解答请见附录 A

问题 I2-1（评量标准）

Java 请参考本章的评量标准，评估以下的叙述内容是否得当。

（1）在方法中一概都加上 synchronized，是不是就万无一失了呢？

（2）synchronized 方法中不是变成了无穷循环了吗？这样的程序是不是不好？

（3）由于程序的错误而陷入无穷循环的线程即使启动 100 个，也由于线程只是不断执行而已，所以应该没有什么关系。

（4）此服务器一次只能连接一个客户端，请将此服务器调整得更好一点。

（5）这个查找程序够烂。一旦启动就得要等到全部查找完毕为止才能取消（Cancel）。

（6）这样的话，类 A 与类 B 双方必须注意线程互斥控制的影响，这样是不是不太好？

问题 I2-2（throughput）

Java 若线程数量变成 2 倍，throughput 会不会也变成 2 倍呢？

第1章

Single Threaded Execution——能通过这座桥的，只有一个人

Single Threaded Execution Pattern

这里有一条独木桥。因为桥身非常的细，一次只能容许一个人经过。当一个人还没有走到桥的另一头，下一个人不可以过桥。如果桥上同时有两个人，桥身就会变成碎片而掉落河里了。

这一章要学习的是 Single Threaded Execution Pattern。

Single Threaded Execution 是指“以 1 个线程执行”的意思。就像细独木桥只能允许一个人通行一样，这个 Pattern 用来限制同时只让一个线程运行。

这个 Single Threaded Execution Pattern 将会是多线程程序设计的基础，请务必学好它。

Single Threaded Execution 有时候也称为 Critical Section（临界区；危险区域）或 Critical Region（同前）。Single Threaded Execution 是把视点放在运行的线程（过桥的人）上所取的名称，而 Critical Section 或 Critical Region 则是把视点放在执行的范围（桥身）上所取的名称。

范例程序 1：不使用 Single Threaded Execution Pattern 的范例

首先，我们先来看一个应该要使用 Single Threaded Execution Pattern 而没使用的程序范例。这个程序的用意是要实际体验多线程无法正确执行的程序，会发生什么现象。

在这里要写的程序，是要模拟 3 个人频繁地经过一个只能容许一个人经过的门（见图 1-1）。当人通过门的时候，这个程序会在计数器中，递增通过的人数。另外，还会记录通过的人的“姓名与出生地”。

程序使用到的类如表 1-1 所示。



图 1-1　一次只能通过一个人的门

表 1-1　类一览表

名　　称	解　　说
Main	创建一个门，并操作 3 个人不断地穿越门的类
Gate	表示门的类，当人经过时会记录姓名与出生地
UserThread	表示人的类，只负责处理不断地在门间穿梭通过

Main 类

Main 类（List 1-1）用来创建一个门（Gate），并让 3 个人（UserThread）不断通过。创建 Gate 类的实例，并将这个实例丢到 UserThread 类的构造器作为参数，告诉人这个对象“请通过这个门”。

有下面 3 个人会通过这个门：

- Alice——Alaska（阿拉斯加）出生地
- Bobby——Brazil（巴西）出生地
- Chris——Canada（加拿大）出生地

为了便于对应两者之间的关系，笔者在此故意将姓名与出生地设成相同的开头字母。

在主线程中，先创建 3 个 UserThread 类的实例，并以 start 方法启动这些线程。

List 1-1　Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing Gate, hit CTRL+C to exit.");
        Gate gate = new Gate();
        new UserThread(gate, "Alice", "Alaska").start();
        new UserThread(gate, "Bobby", "Brazil").start();
        new UserThread(gate, "Chris", "Canada").start();
    }
}
```

并非线程安全(thread-safe)的 Gate 类

Gate 类（List 1-2）表示人所要通过的门。

counter 字段表示目前已经通过这道门的“人数”。name 字段表示通过门的行人的“姓名”，而 address 字段则表示通过者的“出生地”。

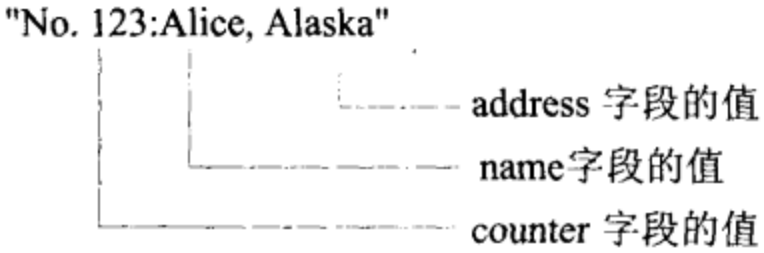
pass 是穿越这道门时使用的方法。在这个方法中，会将表示通过人数的 counter 字段的值递增 1，并将参数中传入行人的姓名与出生地，分别拷贝到 name 字段与

address 字段中。

```
this.name = name;
```

这行赋值(assignment)语句中，左方的 name 是这个实例的字段，而右方的 name 则是方法的参数。

toString 方法，会以字符串的形式返回现在门的状态。使用现在的 counter、name、address 各字段的值，创建字符串。例如下面这样的一行字符串：



check 方法，用来检查现在门的状态（最后通过的行人的记录数据）是否正确。当人的姓名（name）与出生地（address）第一个字符不相同时，就断定记录是有问题的。当发现记录有问题时，就显示出下面的字符串：

```
***** BROKEN *****
```

并紧接着调用 toString 方法显示出现在门的状态。broken 是“损坏”的意思。

这个 Gate 类，在单线程时可以正常运行，但在多线程下就无法正常执行。List 1-2 的 Gate 类是缺乏安全性的类，并不是线程安全（thread-safe）的类。

List 1-2 非线程安全的 Gate 类（Gate.java）

```
public class Gate {
    private int counter = 0;
    private String name = "Nobody";
    private String address = "Nowhere";
    public void pass(String name, String address) {
        this.counter++;
        this.name = name;
        this.address = address;
        check();
    }
    public String toString() {
        return "No." + counter + ": " + name + ", " + address;
    }
    private void check() {
        if (name.charAt(0) != address.charAt(0)) {
            System.out.println("***** BROKEN ***** " + toString());
```

```
        }
    }
}
```

UserThread 类

UserThread 类（List 1-3）表示不断穿越门的行人。这个类声明成 Thread 类的子类。

gate 字段表示所要通过的门，myname 字段表示姓名，而 myaddress 字段表示出生地。因为各字段通过构造器进行初始化以后，就不会再次赋值了，所以设置为 final。将不想被重复赋值的字段设置成 final，是撰写程序的好习惯。因为如果将字段预先设置成 final，就算不小心在程序里写了重复赋值的程序代码，在编译程序的时候也会被检查出来。这种声明字段时不设置初始值，而在构造器中初始化的形式，在 Java 里称为 blank final（空的 final）。

run 方法一开始会显示自己的姓名与 BEGIN 字样。接着马上以 while 进入无穷循环，在循环里面反复地调用 pass 方法。也就是说，这个人只会在门里不断穿梭通过。

List 1-3 UserThread 类（UserThread.java）

```
public class UserThread extends Thread {
    private final Gate gate;
    private final String myname;
    private final String myaddress;
    public UserThread(Gate gate, String myname, String myaddress)
{
        this.gate = gate;
        this.myname = myname;
        this.myaddress = myaddress;
    }
    public void run() {
        System.out.println(myname + " BEGIN");
        while (true) {
            gate.pass(myname, myaddress);
        }
    }
}
```

执行看看……果然出错了

执行这个程序时，会因为时间点不同，而产生不同的结果，图 1-2 是一种可能

的结果。

```
Testing Gate, hit CTRL+C to exit.
Alice BEGIN
Bobby BEGIN
Chris BEGIN
***** BROKEN ***** No.1010560: Bobby, Brazil
***** BROKEN ***** No.1753974: Alice, Alaska
***** BROKEN ***** No.2179746: Bobby, Canada
***** BROKEN ***** No.3440418: Bobby, Brazil
***** BROKEN ***** No.3864350: Chris, Canada
***** BROKEN ***** No.5948274: Bobby, Brazil
***** BROKEN ***** No.6289207: Chris, Canada
***** BROKEN ***** No.7969382: Alice, Alaska
(以下省略。以 Ctrl+C 结束程序)
```

图 1-2　执行范例

现在就仔细研究图 1-2，来思考多线程程序需要面对的问题。

◆ *Gate 类并非线程安全*

首先，Alice、Bobby、Chris 创建时的 BEGIN 各自显示出来后，就开始不断出现***** BROKEN *****的消息了。这是 Gate 类的检查用方法 check 所输出的字符串，当最后通过的人留下的姓名与出生地第一个字母不同的时候就会显示出这个消息。如果仔细观察的话，您会发现即使姓名与出生地的第一个字母都相同的时候，还是会显示这段消息，关于这点就先留到后面的部分再解释。

至少我们知道 List 1-2 的 Gate 类实例，在多线程的环境下使用时，与我们所期待的状态并不相同。List 1-2 的 Gate 类这时并不安全，我们称它为非线程安全（thread-safe）的类。

◆ 测试并无法证明安全性

我们在图 1-2 的结果，还能发现什么问题呢？请仔细看 counter 的值。一开始显示 BROKEN 的时候，counter 的值是 1010560。也就是，发生第一个错误，已经有 100 万人以上经过了。

在这里，因为 UserThread 类的 run 方法内跑的是无穷循环，所以才检查到错误。但如果只是简单的几次测试，不，就算是测试几万次，可能也不会找到错误。

多线程程序设计中，这就是一个较为困难的地方。如果测试时找到错误，表示写好的程序并不安全。但是，就算测试没有找到错误，也不表示程序一定是安全的。当测试次数不够、时间点不对，就可能检查不到问题。一般来说，操作测试并不足

以证明程序的安全性。操作测试（执行实验）所得到的结果，只不过表示“也许是安全的”可能性比较高。

◆ 调试消息也不可靠

不知道读者有没有注意到图 1-2 中另外一个不可思议的现象。请看下面这一行：

```
***** BROKEN ***** No.1010560: Bobby, Brazil
```

这里既然出现了 BROKEN 的错误消息，姓名（Bobby）与出生地（Brazil）的开头字母不是应该要不相同才对吗？可是显示出 BROKEN 消息了，而调试消息却好像是正确的。

图 1-2 的执行结果，姓名与出生地的开头字母真的不同的，其实只有这么一行：

```
***** BROKEN ***** No.2179746: Bobby, Canada
```

会发生这种现象，是因为某个线程正在执行 check 方法时，其他线程正不停地调用 pass 方法，name 字段与 address 字段的值已经又被更改了。

这也是多线程程序设计中较困难的地方。若调试消息的程序本身就并非线程安全，或许会显示出错误的调试消息。

▶▶ 补充说明：重新审查程序代码

连执行测试与调试消息，都无法保证程序的安全性，那要怎么办才好呢？这个时候就要重新审查程序代码。由多个人仔细阅读程序源代码，检查是否会产生问题，这是确保程序安全性最有效的方法。

为什么会出错呢

那么，接下来就要开始研究使用 List 1-2 的 Gate 类时，之所以会显示出 BROKEN 的原因。这是因为 pass 方法可被多个线程调用的关系。pass 方法是下面 4 行语句程序代码所组成的：

```
this.counter++;
this.name = name;
this.address = address;
check();
```

为了在解说时简单一点，现在只考虑两个线程（Alice 与 Bobby）。两个线程调用 pass 方法时，上面 4 行语句可能会是交错依次执行的[注 1]。如果交错的情况是图 1-3 这样，那调用 check 方法的时候，name 的值会是"Alice"，而 address 的值会是"Brazil"。这时就会显示出 BROKEN 了。

注 1：在这里是以语句当作线程的基本操作单位，事实上线程可能是以更小的单位在切换的。

或者说交错的情况如图 1-4 所示，则调用 check 方法的时刻，name 的值是"Bobby"，而 address 的值会是"Alaska"。这个时候也会显示出 BROKEN。

线程 Alice	线程 Bobby	this.name 的值	this.address 的值
this.counter++;	this.counter++;	(之前的值)	(之前的值)
	this.name = name;	"Bobby"	(之前的值)
this.name = name;		"Alice"	(之前的值)
this.address = address;		"Alice"	"Alaska"
	this.address=address;	"Alice"	"Brazil"
check();	check();	"Alice"	"Brazil"
		***** BROKEN *****	

图 1-3　线程 Alice 与线程 Bobby 调用 pass 方法的执行状况（1）

线程 Alice	线程 Bobby	this.name 的值	this.address 的值
this.counter++;	this.counter++;	(之前的值)	(之前的值)
this.name = name;		"Alice"	(之前的值)
	this.name = name;	"Bobby"	(之前的值)
	this.address=address;	"Bobby"	"Brazil"
this.address = address;		"Bobby"	"Alaska"
check();	check();	"Bobby"	"Alaska"
		***** BROKEN *****	

图 1-4　线程 Alice 与线程 Bobby 调用 pass 方法的执行状况（2）

上述哪一种状况，都使字段 name 与 address 的值出现非预期的结果。

通常，线程是不会去考虑其他线程，而自己只会一直不停地跑下去的。“线程 Alice 现在执行到的位置正指定 name 结束，还没指定 address 的值”，而线程 Bobby 对此情况并不知情。

范例程序 1 之所以会显示出 BROKEN，是因为线程并没有考虑到其他线程，而将共享实例的字段改写了。

对 name 字段来说，有两个线程在比赛，赢的一方会先将值改写。对 address 来说，也有两个线程在比赛谁先将值改写。像这样子引发竞争（race）的状况，我们称为 race condition。有 race condition 的情况时，就很难预测各字段的值了。

以上是没有使用 Single Threaded Execution Pattern 时所发生的现象。

范例程序 2：使用 Single Threaded Execution Pattern 的范例

那么，我们就来将 List 1-2 的 Gate 类修改成线程安全的版本吧。其他的类则不需要修改。

线程安全的 Gate 类

List 1-4 是线程安全的 Gate 类。需要修改的有两个地方，在 pass 方法与 toString 方法前面都加上 synchronized。这样 Gate 类就成为线程安全的类了。

List 1-4 线程安全的 Gate 类（Gate.java）

```
public class Gate {
    private int counter = 0;
    private String name = "Nobody";
    private String address = "Nowhere";
    public synchronized void pass(String name, String address) {
        this.counter++;
        this.name = name;
        this.address = address;
        check();
    }
    public synchronized String toString() {
        return "No." + counter + ": " + name + ", " + address;
    }
    private void check() {
        if (name.charAt(0) != address.charAt(0)) {
            System.out.println("***** BROKEN ***** " + toString());
        }
    }
}
```

执行结果如图 1-5 所示，无论等多久都没有显示 BROKEN 消息。请按【Ctrl+C】键结束程序执行。这个执行结果也并不能证明 Gate 类的安全性，但是我们可以说安全的可能性很高。

```
Testing Gate, hit CTRL+C to exit.
Alice BEGIN
Bobby BEGIN
Chris BEGIN
（无论等多久都没有 BROKEN。以 Ctrl+C 结束程序）
```

图 1-5　执行结果

synchronized 所扮演的角色

为什么只要在 pass 方法与 toString 方法加上 synchronized，就不会显示 BROKEN 了呢？

如前一节后面所说的，之所以会显示 BROKEN，是因为 pass 方法内的程序可能会被多个线程穿插执行。

synchronized 方法，能够保证同时只有一个线程可以执行它。这句话的意思是说：线程 Alice 执行 pass 方法的时候，线程 Bobby 就不能调用 pass 方法。在线程 Alice 执行完 pass 方法之前，线程 Bobby 会在 pass 方法的入口处被阻挡下。当线程 Alice 执行完 pass 方法之后，将锁定解除线程 Bobby 才可以开始执行 pass 方法。

所以只要将 pass 方法声明成 synchronized 的方法，就绝对不会出现图 1-3 或图 1-4 这类的情况。而一定是图 1-6 或图 1-7 的样子。

线程 Alice	线程 Bobby	this.name 的值	this.address 的值
【获取锁定】			
this.counter++;		(之前的值)	(之前的值)
this.name = name;		"Alice"	(之前的值)
this.address = address;		"Alice"	"Alaska"
check();		"Alice"	"Alaska"
【解除锁定】			
	【获取锁定】		
	this.counter++;	"Alice"	"Alaska"
	this.name = name;	"Bobby"	"Alaska"
	this.address=address;	"Bobby"	"Brazil"
	check();	"Bobby"	"Brazil"
	【解除锁定】		

图 1-6　线程 Alice 与线程 Bobby 调用 synchronized 的 pass 方法时的执行状况（1）

线程 Alice	线程 Bobby	this.name 的值	this.address 的值
	【获取锁定】		
	this.counter++;	(之前的值)	(之前的值)
	this.name = name;	"Bobby"	(之前的值)
	this.address=address;	"Bobby"	"Brazil"
	check();	"Bobby"	"Brazil"
	【解除锁定】		
【获取锁定】			
this.counter++;		"Bobby"	"Brazil"
this.name = name;		"Alice"	"Brazil"
this.address = address;		"Alice"	"Alaska"
check();		"Alice"	"Alaska"
【解除锁定】			

图 1-7　线程 Alice 与线程 Bobby 调用 synchronized 的 pass 方法时的执行状况（2）

而 toString 方法需要加上 synchronized 的理由，与 check 方法不加上 synchronized 的理由，则留到本章末的练习问题 1-3 研究。

Single Threaded Execution Pattern 的所有参与者

相信读者已经在范例程序中抓住 Single Threaded Execution Pattern 的感觉了，接下来就脱离具体的程序代码，使用更一般的表达方式，来观察、归纳 Single Threaded Execution Pattern。

◆ *SharedResource（共享资源）参与者*

Single Threaded Execution Pattern 中，有担任 SharedResource（共享资源）角色的类出现。在范例程序 2 中，Gate 类就是这个 SharedResource 参与者。

SharedResource 参与者是可由多个线程访问的类。SharedResource 会拥有一些方法，而这些方法又分为下面两类：

- SafeMethod——从多个线程同时调用也不会发生问题的方法；
- UnsafeMethod——从多个线程同时调用会出问题，而需要加以防护的方法。

safeMethod（安全的方法）并不需要特别处理。

unsafeMethod（不安全的方法）则是被多个线程同时执行时，可能会使实例的状态产生矛盾的方法。所以必须加以防卫，使同时不能有多个线程同时执行这个方法。

在 Single Threaded Execution Pattern 中，我们将 unsafeMethod 加以防卫，限制同时只能有 1 个线程可以调用它。在 Java 语言中，只要将 unsafeMethod 定义成 synchronized 方法，就可以实现这个目标。

这个必须只让单线程执行的程序范围，我们称为临界区（critical section）。

类图如图 1-8、时序图如图 1-9 所示。

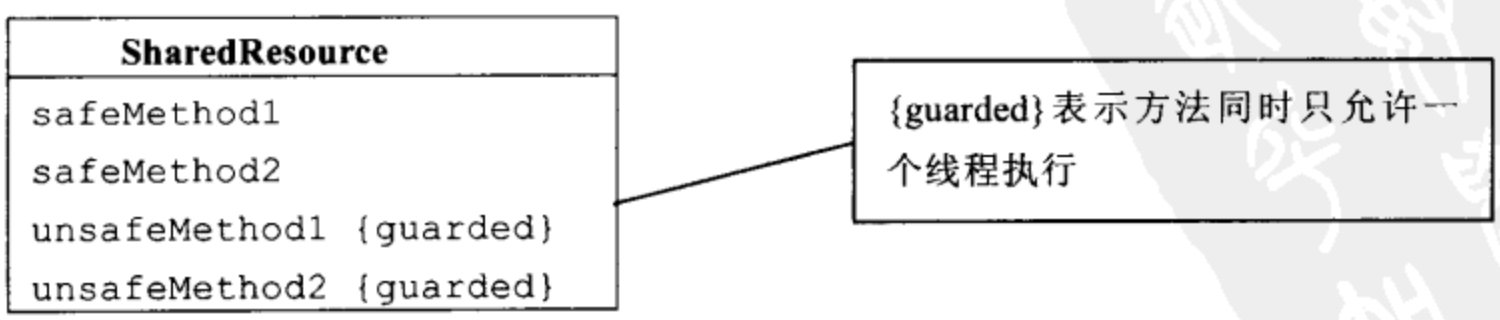


图 1-8　Single Threaded Execution Pattern 的类图

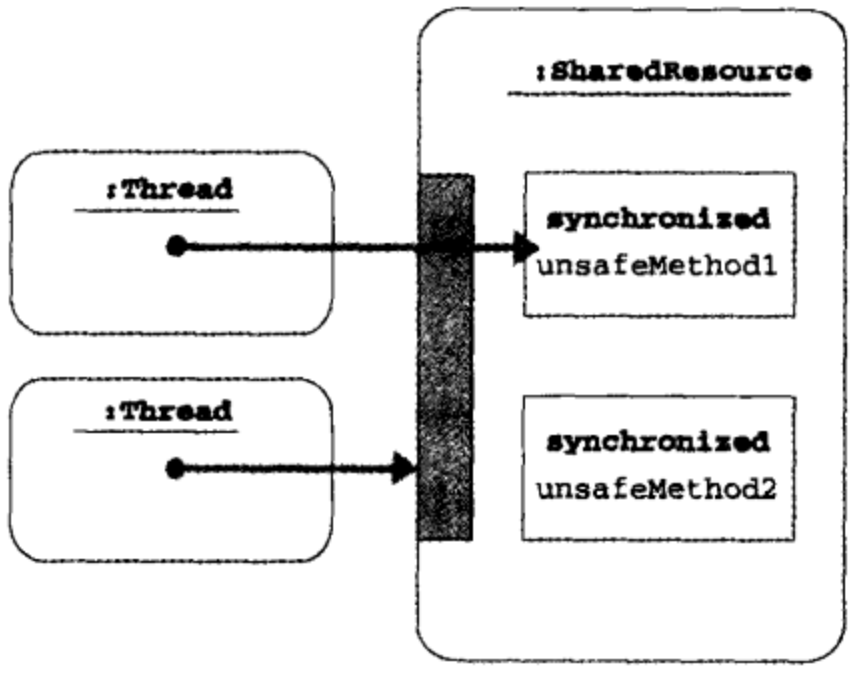


图 1-9　Single Threaded Execution Pattern 的时序图

扩展思考方向的提示

何时使用（适用性）

Single Threaded Execution Pattern 该在什么情况下使用呢？

◆ 多线程时

单线程程序，并不需要使用 Single Threaded Execution Pattern。因此，也不需要使用到 synchronized 方法。

即使在单线程程序中使用 synchronized 方法，也不会对程序的安全性造成危害。但是调用 synchronized 方法会比调用一般的方法多花一些时间，所以会使程序性能略微降低。

在单线程程序中使用 synchronized 方法，就好像在外独居的人，即使一个人在家中还是将厕所的门锁住的意思一样。既然是一个人住，即使不锁门，也不需要担心有人会突然打开厕所的门了。

◆ 数据可被多个线程访问的时候

会需要使用 Single Threaded Execution Pattern 的情况，是在 SharedResource 的实例可能同时被多个线程访问的时候。

就算是多线程程序，如果所有线程完全地独立运行，那也没有使用 Single Threaded Execution Pattern 的必要。我们将这个状态称为线程互不干涉（*interfere*）。

有些管理多线程的环境，会帮我们确保线程的独立性，这种情况下这个环境的用户就不必考虑需不需要使用 Single Threaded Execution Pattern。

◆ 状态可能变化的时候

当 SharedResource 参与者状态可能变化的时候，才会有使用 Single Threaded Execution Pattern 的需要。

如果实例创建了以后，从此不会再改变状态，也没有使用 Single Threaded Execution Pattern 的必要。

第 2 章所要介绍的 Immutable Pattern 就像这种情况。在 Immutable Pattern 中，实例的状态不会改变，所以是不需要用到 synchronized 方法的一种 Pattern。

◆ 需要确保安全性的时候

只有需要确保安全性的时候，才会需要使用 Single Threaded Execution Pattern。

例如，Java 2 的集合架构类多半并非线程安全。这是为了在不考虑安全性的时候，可以让程序的运行速度较高。

所以用户需要考虑自己要用的类需不需要考虑线程安全再使用。

▶▶ 补充说明：线程安全的包装方法

Java 2 还提供了下面这些“包装(Wrapper)”方法，用来让集合架构类可以线程安全地运行。

- synchronizedCollection 方法
- synchronizedList 方法
- synchronizedMap 方法
- synchronizedSet 方法
- synchronizedSortedMap 方法
- synchronizedSortedSet 方法

细节部分请参阅 java.util.Collections 类的 API 参考文件。

生命性与死锁

接着要思考的是 Single Threaded Execution Pattern 的生命性。

使用 Single Threaded Execution Pattern 时，可能会有发生死锁（deadlock）的危险。

所谓的死锁，是指两个线程分别获取了锁定，互相等待另一个线程解除锁定的现象。发生死锁时，哪个线程都无法继续执行下去，所以程序会失去生命性。

来举个例子吧。假设 Alice 与 Bobby 同吃一个大盘子所盛放的意大利面。盘子的旁边只有一支汤匙与一支叉子，而要吃意大利面时，同时需要用到汤匙与叉子。

只有一支的汤匙，被 Alice 拿去了，而只有一支的叉子，却被 Bobby 拿走了。就造成以下的情况：

- 握着汤匙的 Alice，一直等着 Bobby 把叉子放下。
- 握着叉子的 Bobby，一直等着 Alice 把汤匙放下。

这么一来 Alice 与 Bobby 只有面面相觑，就这样不动了。像这样，多个线程僵持不下，使程序无法继续运行的状态，就称为死锁。

Single Threaded Execution 达到下面这些条件时，可能会出现死锁的现象。

（1）具有多个 SharedResource 参与者。

（2）线程锁定一个 SharedResource 时，还没解除前就去锁定另一个 SharedResource。

（3）获取 SharedResource 参与者的顺序不固定（和 SharedResource 参与者是对等的）。

回过头来看前面吃不到意大利面的两个人这个例子。

（1）多个 SharedResource 参与者，相当于汤匙与叉子。

（2）锁定某个 SharedResource 参与者后，就去锁定其他 SharedResource。就相当于握着汤匙而想要获取对方的叉子，或握着叉子而想要获取对方的汤匙这些操作。

（3）SharedResource 角色是对等的，就像“拿汤匙→拿叉子”与“拿叉子→拿汤匙”两个操作都可能发生。也就是说在这里汤匙与叉子并没有优先级。

（1）、（2）、（3）中只要破坏一种条件，就可以避免死锁的发生。具体的程序代码留到本章末的练习问题 1-6 研究。

可重用性与继承异常

这里要思考 Single Threaded Execution Pattern 的可重用性。

假设现在有人写了一个 SharedResource 参与者的子类，当 SharedResource 的字段开放给子类访问时，会有撰写子类的程序员可能写出没有防卫的 unsafeMethod。也就是说好不容易考虑安全性而写出的 SharedResource 参与者，还是有可能在子类化时丧失安全性。也就是说，包括子类，若非所有 unsafeMethod 都定义成 synchronized，就无法保证 SharedResource 参与者的安全性。

在面向对象程序设计中，负责进行子类化的“继承”担任了重要的角色。但是，对多线程程序设计来说，继承也会引起许多麻烦的问题。我们通常称之为继承异常（inheritance anomaly）。

临界区的大小与执行性能

一般来说，Single Threaded Execution Pattern 会使程序执行性能低落的原因有两个：

◆ 理由(1)：获取锁定要花时间

进入 synchronized 方法时，要获取对象的锁定，这个操作会花一点时间。

若能减少 SharedResource 参与者的数量，就能减少需要获取的锁定数，可以减少性能低落的幅度。

◆ 理由(2)：线程冲突时必须等待

当线程 Alice 执行临界区内的操作时，其他要进入临界区的线程会被阻挡。这个状况称之为冲突（conflict）。当冲突发生时，线程等待的时间就会使整个程序的性能往下掉。

尽可能缩小临界区范围，以减少出现线程冲突的机会，可抑制性能的降低。

相关 Pattern

Single Threaded Execution Pattern 与多线程、并发性相关的许多 Pattern 都有关联。

Guarded Suspension Pattern (第 3 章)

Single Threaded Execution Pattern 中，造成线程等待的条件是“是否有其他线程正在执行保护住的 unsafeMethod”。

而 Guarded Suspension Pattern 中，造成线程等待的条件则是“对象的状态到底适不适当”。

另外，利用 Guarded Suspension Pattern 时，部分“检查对象状态”的操作会用到 Single Threaded Execution Pattern。

Read-Write Lock Pattern (第 6 章)

Single Threaded Execution Pattern 中，只要有一个线程执行着保护住的 unsafeMethod 时，其他所有想执行这个方法的线程都要等待执行结束。

而 Read-Write Lock Pattern 中，多个线程可以同时执行 read 方法。而这时要等待的就只有想要执行 write 方法的线程。

另外，利用 Read-Write Lock Pattern 时，在“检查线程种类与个数”的部分，也会使用到 Single Threaded Execution Pattern。

Immutable Pattern (第 2 章)

Single Threaded Execution Pattern 中，unsafeMethod 必须保护成只允许一个线程执行。

而 Immutable Pattern 的 Immutable 参与者，其对象的状态不会改变。因此，不需要去保护任何方法。换言之，Immutable Pattern 所有的方法都是 safeMethod。

Thread-Specific Storage Pattern (第 11 章)

Single Threaded Execution Pattern 中，多个线程会访问同一个 SharedResource 参与者。于是，我们需要将方法保护起来管制交通。

而 Thread-Specific Storage Pattern 则是对每个线程确保其特有的领域，因为这块特有的领域只有 1 个线程可以访问，所以不需要保护方法。

进阶说明：关于 synchronized

在进阶说明中，笔者将对 Java 的 synchronized 作更深入的解说。

synchronized 语法与 Before/After Pattern

无论是 synchronized 方法或 synchronized 块，都像 List 1-5、1-6 一样，是由“{”与“}”所括住的块构成的。

List 1-5 synchronized 方法

```
synchronized void method() {
    ...
}
```

List 1-6 synchronized 块

```
synchronized (obj) {
    ...
}
```

上面两者，都可以视为是在“{”处获取锁定，并在“}”处解除锁定。在下面我们对 synchronized 的程序代码，与明确操作“锁定”的程序代码加以比较。假设有一个 lock 方法用来获取锁定，并有一个解除锁定用的 unlock 方法存在。

List 1-7 明确操作锁定的程序代码

```
void method() {
    lock();
    ...
    unlock();
}
```

您是不是认为“虽然程序代码看起来复杂点，不过与使用 synchronized 的程序代码没有太大的差异吧”呢？不，其实它们有着极大的差异。在上面的程序代码中

省略的部分，是有可能会出现问题的。如果在调用 lock 与 unlock 方法之间，存在有 return 语句的话，那锁定可能就不会被解除了（List 1-8）。

List 1-8　有 return 的话，锁定没被解除

```
void method() {
    lock();
    if (条件式)
        return;    // 这里有 return 的话，unlock()就不会被调用了
    }
    unlock();
}
```

您或许会认为“那小心不要有 return 语句就好了”。但其实问题不只有 return 而已，异常处理也是一个问题。调用的方法（或是调用的方法所调用的方法）抛出异常时，锁定也有可能没有解除。

List 1-9　doMethod 抛出异常的话，锁定没被解除

```
void method() {
    lock();
    doMethod;  // 这里抛出异常的话，unlock()就不会被调用了
    unlock();
}
```

相对地，synchronized 方法和 synchronized 块，无论碰到 return 或是异常，都会确实解除锁定。

如果是成对的 lock()、unlock()方法，“想要在 lock()被调用后，无论发生什么事，都要调用到 unlock()”，就要像下面这样使用 finally（“最后”的意义）块。

List 1-10　在 lock()被调用后，无论发生什么事都要调用到 unlock()

```
void method() {
    lock();
    try {
        ...
    } finally {
        unlock();
    }
}
```

List 1-10 的代码部分，无论出现 return、抛出异常或是发生其他什么事，都会执行到 finally 的部分，这是 Java 的规定。

这种 finally 的使用方法，是 Before/After Pattern（事前/事后 Pattern）的一种实现方式。关于 Before/After Pattern，会在第 6 章的练习问题 6-5 再次说明。

这个 synchronized 在保护什么

请读者阅读程序代码的时候，看到 synchronized 时，就要思考：

“这个 synchronized 是在保护什么？”

无论是 synchronized 方法或是 synchronized 块，synchronized 势必保护着“某个东西”。例如，在 List 1-4 中我们将 pass 定义成 synchronized 方法。这个 synchronized 所保护的是 Gate 类的 counter、name、address 这些字段。使这些字段不会被多个线程同时访问。

确认“保护什么”之后，接下来应该要思考的是：

“其他的地方也有妥善保护到吗？”

若这些字段还在其他许多地方使用着，这里使用 synchronized 小心保护，但其他地方并没有做好保护措施，那其实这个字段还是没被保护的。就像就算小心翼翼地把大门跟后门都锁得好好的，如果窗户敞开，还是没有意义一样。

在这里，请仔细看 List 1-4。pass 方法与 toString 方法的确以 synchronized 保护着字段。但 check 方法也有用到 name 字段与 address 字段，却没有定义成 synchronized。这会不会变成敞开的窗户呢？

答案是不会。因为只有 pass 方法会去调用 check 方法，而 pass 方法已经设置成 synchronized 了。而且 check 方法又被设置成 private，所以不会有其他类调用这个方法。于是，check 方法不需要设置成 synchronized（设置成 synchronized 也没有关系，不过可能会降低程序的性能）。

synchronized 就像是门窗上施加的锁。当您在一片窗上上了锁，就应该仔细确认其他门窗是不是也不小心上锁了。所有可由多个线程共享，并会访问字段的方法，都应该设置成 synchronized 加以保护。

该以什么单位来保护呢

这里要对前一段所说的“保护着什么”，做更进一步地探讨。我们可不可以在 List 1-4 中，加上下面这样的方法呢？

```
public synchronized void setName(String name) {
    this.name = name;
}
public synchronized void setAddress(String address) {
    this.address = address;
}
```

这些方法的确都设置成 synchronized 了。可是，加上这些方法以后，Gate 类就不安全了。

因为 Gate 类，姓名跟出生地非得合在一起赋值不可。我们将 pass 方法设置成 synchronized，也就是为了不要让多个线程穿插执行赋值的操作。然而，如果定义出 setName、setAddress 这样的方法，线程对字段赋值的操作就被分散了。要保护 Gate 类时，若不将字段合在一起保护，是没有意义的。

获取谁的锁定来保护的呢

前面说过看到 synchronized 的时候，要先思考“synchronized 是在保护什么”。接下来要更进一步地去思考：

“获取谁的锁定来保护的呢？”

要调用 synchronized 实例方法（instance method）的线程，一定会获取 this 的锁定。一个实例的锁定，同一时间内只能有一线程可以得到。因为这个惟一性，我们才能使用 synchronized 来做到 Single Threaded Execution Pattern。

如果实例不同，那锁定也不同了。虽然我们说“使用 synchronized 来保护”，但如果有多个相异实例，那多个线程仍然可以分别执行不同实例的 synchronized 方法（参考图 I1-12）。

使用 synchronized 块的时候，特别需要考虑“获取谁的锁定来保护的呢”这种情况。因为 synchronized 块需要明确地指明要获取的是哪个对象的锁定。例如：

```
synchronized (obj) {
    ...
}
```

这样的程序代码中，obj 就是我们所要获取锁定的对象。请小心这个对象不可以写错，获取错误对象的锁定，就好像是想要保护自己的家，却反而去锁上邻居的门一样。

原子的操作

synchronized 方法同时只有一个线程可以执行。当一个线程正在执行 synchronized 方法时，其他线程都不能进入这个方法。也就是这个 synchronized 方法所进行的操作，从多线程的角度来看，是“原子的操作（atomic operation）”。atom 是物理学中“原子”的意思，本意是“不可分割的东西”。

为了让范例程序的 Gate 类线程安全，我们将 pass 定义成 synchronized 方法（List 1-4）。也就是说 pass 方法就是所谓原子的操作。

long 与 double 并不是原子的

其实，Java 语言规格中，一开始就定义了一些原子的操作。例如，char、int 这

些基本类型（primitive type）的赋值与引用操作是原子的。另外，对象等引用类型（reference type）的赋值与引用操作也是原子的。因为本来就定义成不可分割，就算没有加上 synchronized，也不会被分割。

例如，这里有一个 int 类型的字段 n，而有某个线程进行：

```
n = 123;
```

这样的赋值操作，而前后有别的线程也进行：

```
n = 456;
```

这样的赋值操作。这时最后 n 的值不是 123 就会是 456[注2]。并不用担心两个值的位模式会混在一起。

笔者刚刚说基本类型的指定、引用操作不可分割，但其实是有例外的。Java 语言的规格上，long 与 double 的指定、引用操作并非不可分割。

例如有一个 long 类型的 longField 字段，某个线程进行：

```
longField = 123L;
```

这样的指定操作，而同时有另一个线程执行：

```
longField = 456L;
```

这样的指定操作。之后 longField 的值会是什么，是无法保证的。也许是 123L，也可能是 456L，或许是 0L，甚至还可能是 31415926L。当然，这里所说的只是“Java 语言规格”而已。实际上大部分的 Java 执行环境现在都将 long 与 double 当作原子的操作来实现，但这也只能说是部分 Java 执行环境的实现就是了。

既然指定、引用 long、double 的操作是可以分割的，若要在线程间共享 long、double 的字段，那对字段进行操作时，就必须使用 Single Threaded Execution Pattern。最简单的方法，就是在 synchronized 方法内进行操作。

还有一种方法是将不使用 synchronized，而在声明字段时，加上 volatile 关键字。在字段前加上 volatile 关键字，所有对这个字段的操作就成为不可分割。

总而言之，我们可以得到下面这些结论。

- 基本类型、引用类型的指定、引用是原子的操作；
- 但是 long 与 double 的指定、引用是可以分割的；
- 要在线程间共享 long 或 double 的字段时，必须在 synchronized 中操作，或是声明成 volatile。

关于 volatile 关键字的角色，并不是只有对 long、double 进行原子的指定、引用。详细的说明请参考附录 B 的“Java 内存模型”。

注 2：这个说法严格来说是不正确的。当线程正在指定字段的值时，若这个字段不是 volatile 也没有以 synchronized 同步化，或许其他线程并无法马上看到指定的结果。这不是可不可以分割（atomicity）的问题，而是其他线程能不能看到（visibility）的问题。详见附录 B“Java 的内存模型”。

重点回顾

这一章我们学习了 Single Threaded Execution Pattern。

当我们修改了由多个线程共享的变量，变量就会失去安全性。所以，我们应该仔细找出变量状态并不安定的范围，将这个范围设置为临界区。并对临界区施加保护，限制同时只能有一个线程执行它。

Java 语言中，可以使用 synchronized 关键字定义临界区，保护多个线程所共享的字段。

这就是 Single Threaded Execution Pattern。

进阶说明中也对看到程序代码中出现 synchronized 时，应该要注意的地方作了一些说明。

那么，就请读者练习一些问题吧。

练习问题

解答请见附录 A

问题 1-1（使错误更容易发生）

Java 范例程序 1 的执行范例（图 1-2）中，非线程安全的 Gate 类（List 1-2）找到第一个错误时，counter 的值已经是 1010560 了。也就是要找到一个错误，可能需要调用 pass 方法 100 多万次。请尝试改写 List 1-2 的 Gate 类，使找出错误时，counter 的值能够更小。

问题 1-2（private 字段的角色）

Java 在本文部分的程序代码中，无论是 List 1-2、List 1-4，字段都声明成 private。

```
private int counter = 0;
private String name = "Nobody";
private String address = "Nowhere";
```

为什么要将字段设置成 private 呢？ 若这些字段是 protected 或 public，会发生什么问题吗？请用类安全性的观点加以分析。

问题 1-3（synchronized 的理由）

Java 范例程序 2 的 Gate 类（List 1-4）中，有下面 3 种方法：

（1）pass 方法——是 synchronized 方法

（2）toString 方法——是 synchronized 方法

（3）check 方法——不是 synchronized 方法

关于（1）的 pass 方法需要定义成 synchronized 方法的理由，与（3）的 check

方法不需要定义成 synchronized 方法的理由，在本文中都已经解释过了。那么，请思考（2）的 toString 方法需要定义成 synchronizedsynchronized 方法的理由。

问题 1-4（看到一半的程序代码）

Java List 1-11 是 Point 类的程序代码前半段。请就已读取的范围，判断下面这些与 Point 类相关的描述，若是正确的描述回答○，不正确的则回答×。

（1）无法创建 Point 类的子类。

（2）要指定到 Point 类的 x 字段的语句只能写在 Point 类的构造器（contructor）以及方法。

（3）关于有 Point 类的实例，可以执行这里读取的 move 方法的是一次一个的线程。

（4）Point 类即使由数个线程来使用也很安全。

（5）只要将 Point 类剩下的方法全部都读取，就可以判断是否发生死锁。

List 1-11　程序代码看了一半，它在说些什么呢（Point.java）

```
public final class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public synchronized void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
(读到这里为止)
```

问题 1-5（确认安全性）

Java 下面的 SecurityGate 类（List 1-12）是模拟机密设施的入口。进入（enter）时，人数（counter）就会递增 1；退出（exit）时，人数就会递减 1。使用获取计数值（getCounter）方法，可以得知现在停留在设施内的人数。

现在各方法都不是 synchronized，请问当这个类在多线程环境下使用时是否安全？

List 1-12　这个 SecurityGate 是否安全（SecurityGate.java）

```
public class SecurityGate {
    private int counter;
    public void enter() {
```

```
        counter++;
    }
    public void exit() {
        counter--;
    }
    public int getCounter() {
        return counter;
    }
}
```

问题 1-6（避免死锁的发生）

Java 在此尝试以程序确认本文中说明过的死锁（deadlock）问题。在这里我们定义出表 1-2 这些类。

表 1-2　死锁实验用的类一览表

名　　称	解　　释
Main	构建汤匙和叉子，并操作 Alice 与 Bobby 两个人
Tool	表示餐具（汤匙和叉子）的类
EaterThread	左手拿起餐具，右手也拿起餐具，开始用餐的类

下面是程序代码（List 1-13～List 1-15）。Main 类中构建出 Tool 类的实例汤匙与叉子，并传入 EaterThread 类以启动线程。

List 1-13　Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing EaterThread, hit CTRL+C to
exit.");
        Tool spoon = new Tool("Spoon");
        Tool fork = new Tool("Fork");
        new EaterThread("Alice", spoon, fork).start();
        new EaterThread("Bobby", fork, spoon).start();
    }
}
```

List 1-14　Tool 类（Tool.java）

```
public class Tool {
    private final String name;
    public Tool(String name) {
```

```
        .
        this.name = name;
    }
    public String toString() {
        return "[ " + name + " ]";
    }
}
```

EaterThread 类中，当线程启动后，会不断地重复调用 eat 方法。

在 eat 方法内进行了下面的操作：

（1）使用外侧的 synchronized 块，锁定左手（lefthand）获取的餐具。

（2）程序以下面的方式，表达左手拿起餐具的操作：

```
Alice takes up [ Spoon ] (left).
(Alice 拿起汤匙)
```

（3）使用外侧的 synchronized 块，锁定右手（righthand）获取的餐具。

（4）程序以下面的方式，表达右手拿起餐具的操作：

```
Alice takes up [ Fork ] (right).
(Alice 拿起叉子)
```

（5）接着开始进餐。

```
Alice is eating now, yam yam!
(Alice 正吃着。卡滋卡滋！)
```

（6）程序以下面的方式，表达右手放下餐具的操作：

```
Alice puts down [ Fork ] (right).
(Alice 放下叉子)
```

（7）退出内侧的 synchronized 块后，会解除右手（righthand）餐具的锁定。

（8）程序以下面的方式，表达左手放下餐具的操作：

```
Alice puts down [ Spoon ] (left).
(Alice 放下汤匙)
```

（9）退出外侧的 synchronized 块后，会解除左手（lefthand）餐具的锁定。

List 1-15 EaterThread 类（EaterThread.java）

```
public class EaterThread extends Thread {
    private String name;
    private final Tool lefthand;
    private final Tool righthand;
    public EaterThread(String name, Tool lefthand, Tool righthand){
        this.name = name;
        this.lefthand = lefthand;
```

```
        this.righthand = righthand;
    }
    public void run() {
        while (true) {
            eat();
        }
    }
    public void eat() {
        synchronized (lefthand) {
            System.out.println(name + " takes up " + lefthand + "
            (left).");
            synchronized (righthand) {
              System.out.println(name + " takes up " + righthand +
              " (right).");
              System.out.println(name + " is eating now, yam yam!");
              System.out.println(name + " puts down " + righthand
              + " (right).");
                 }
              System.out.println(name + " puts down " + lefthand +
              " (left).");
            }
        }
    }
```

直接执行这个程序，就会得到如图 1-10 所示的结果，程序会形成死锁的状态。

```
Testing EaterThread, hit CTRL+C to exit.
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right).
Alice is eating now, yam yam!
Alice puts down [ Fork ] (right).
Alice puts down [ Spoon ] (left).
Bobby takes up [ Fork ] (left).
Bobby takes up [ Spoon ] (right).
Bobby is eating now, yam yam!
Bobby puts down [ Spoon ] (right).
Bobby puts down [ Fork ] (left).
(中略)
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right).
Alice is eating now, yam yam!
```

（续上图）

```
Alice puts down [ Fork ] (right).
Alice puts down [ Spoon ] (left).
Bobby takes up [ Fork ] (left).
Bobby takes up [ Spoon ] (right).
Bobby is eating now, yam yam!
Bobby puts down [ Spoon ] (right).
Bobby puts down [ Fork ] (left).
Bobby takes up [ Fork ] (left).
Bobby takes up [ Spoon ] (right).
Bobby is eating now, yam yam!
Bobby puts down [ Spoon ] (right).
Bobby puts down [ Fork ] (left).
Alice takes up [ Spoon ] (left).        ← Alice 拿了 Spoon
Bobby takes up [ Fork ] (left).         ← Bobby 拿了 Fork
(在这里停住不动。按 Ctrl+C 结束程序。)
```

图 1-10　执行范例（发生死锁的状况）

程序到后面

```
Alice takes up [ Spoon ] (left).
Bobby takes up [ Fork ] (left).
```

的时候停了下来。的确如前文所说，是停在 Alice 握着汤匙，而 Bobby 握着叉子的地方不动了。

那么接下来就是问题了。请自由改写这个程序，避免掉发生死锁的可能性。

问题 1-7（创建 mutex，互斥）

> 这个问题使用到第 3 章的 Guarded Suspension Pattern，读者可以看完第 3 章后再来解答这个问题。

某人正思考着若不使用 synchronized，有没有其他的方法可以做到 Single Threaded Execution Pattern。而他写下了 List 1-16 所示的 Gate 类（使用到了第 6 章问题 6-5 所介绍的 Before/After Pattern）。

那么，接下来就是问题了。请创建 List 1-16 中所要使用的 Mutex 类。

顺带一提，像 Mutex 类这种用来进行共享互斥的机制，一般称为 mutex。mutex 是 mutual exclusion（互斥）的简称。

List 1-16　Gate 类（Gate.java）

```
public class Gate {
    private int counter = 0;
```

```
    private String name = "Nobody";
    private String address = "Nowhere";
    private final Mutex mutex = new Mutex();
    public void pass(String name,String address){//并非 synchronized
        mutex.lock();
        try {
            this.counter++;
            this.name = name;
            this.address = address;
            check();
        } finally {
            mutex.unlock();
        }
    }
    public String toString() { // 并非 synchronized
        String s = null;
        mutex.lock();
        try {
            s = "No." + counter + ": " + name + ", " + address;
        } finally {
            mutex.unlock();
        }
        return s;
    }
    private void check() {
        if (name.charAt(0) != address.charAt(0)) {
            System.out.println("***** BROKEN ***** " + toString());
        }
    }
}
```

第2章

Immutable——想破坏它也没办法

Immutable Pattern

java.lang.String 是用来表示字符串的类。String 类并没有提供任何改变字符串属性的方法。也就是说，String 实例所代表的字符串属性，绝对不会改变。

因为如此，String 类的方法，都不需要 synchronized。无论有多少线程如何对 String 访问，因为实例的内部状态不会改变，所以线程并不需要考虑共享互斥。

这一章中，要学习的就是 Immutable Pattern。

immutable 是“永恒的”、“不会改变的”的意思。在 Immutable Pattern 中，有着能够保证实例状态绝不会改变的类（immutable 类）。因为访问这个实例时，可以省去使用共享互斥机制所会浪费的时间，故若能妥善运用，将能提高程序的性能。

immutable 的反义词是 mutable（易变的或无常的）。设计类或是研究已有类的时候，请读者留意“这个类是 immutable 还是 mutable”，也就是说，请意识到这个类的不变性（immutability）。String 就是一个 immutable 的类。

范例程序

接下来就使用 Immutable Pattern，写一个简单的程序吧。类的说明如表 2-1 所示。

表 2-1　类一览表

名　　称	解　　释
Person	表示人的类
Main	运行测试用的类
PrintPersonThread	用来显示 Person 实例的线程类

使用 Immutable Pattern 的 Person 类

List 2-1 中所枚举的 Person 类，是用来代表人的类，具有姓名（name）、地址（address）等字段。

Person 类字段的值，都只能通过构造器来设置。类定义有用来引用字段值的 getName 方法与 getAddress 方法，但是没有定义 setName、setAddress 这类用来更改字段值的方法。

这时，即使有多个线程同时访问相同实例，Person 类也是安全的。Person 类的每个方法，都可以被多个线程同时执行。Person 类有 getName、getAddress、toString 等 3 种方法，都不需要定义成 synchronized。

Person 类被声明成 final。也就是说，Person 类不允许别人定义它的子类。虽然这不是 Immutable Pattern 的必要条件，但这是预防子类可能修改字段值的一种安全措施。

Person 类的字段 name 与 address 的访问权限都是 private。也就是说，这些字段只有类内部看的到。这也不是 Immutable Pattern 的必要条件，而是预防子类擅自修改字段值的安全措施。而且 Person 类的字段 name 与 address 都声明成 final。也就是说，只要 name 与 address 的值第一次指定后，从此就不能再指定其他的值了。这也不是 Immutable Pattern 的必要条件，而是可以明确表达程序员的意图，就算不小心写了指定的程序代码，编译器会替我们找出错误。

List 2-1 Person 类（Person.java）

```
public final class Person {
    private final String name;
    private final String address;
    public Person(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public String toString() {
        return "[ Person: name = " + name + ", address = " + address
+ " ]";
    }
}
```

Main 类

Main 类（List 2-2）会构造一个 Person 类（Alice），并以 3 个线程（PrintPersonThread）访问它。

List 2-2 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Person alice = new Person("Alice", "Alaska");
        new PrintPersonThread(alice).start();
```

```
        new PrintPersonThread(alice).start();
        new PrintPersonThread(alice).start();
    }
}
```

PrintPersonThread 类

PrintPersonThread 类（List 2-3）用来不停地显示构造器中传入的 Person 类实例。显示出的字符串是由下面的表达式所构成的：

```
Thread.currentThread().getName() + " prints " + person
```

其中：

```
Thread.currentThread().getName()
```

是用来获取自己这个线程的名称。这里的 Thread.currentThread，使用来获取现在线程的方法[注1]。也就是说，这个方法可以获取调用 currentThread 方法的线程本身。currentThread 是 Thread 类的类方法。而这里使用的 getName 则是 Thread 类的实例方法，用来获取线程的名称。

另外，

```
... + " prints " + person
```

的部分，与

```
... + " prints " + person.toString()
```

是等意的。

当字符串与实例表达式以+运算符连接的时候，会自动调用实例表达式的 toString 方法（这是 Java 的规定）。

List 2-3 PrintPersonThread 类（PrintPersonThread.java）

```
public class PrintPersonThread extends Thread {
    private Person person;
    public PrintPersonThread(Person person) {
        this.person = person;
    }
    public void run() {
        while (true) {
          System.out.println(Thread.currentThread().getName() + "
prints " + person);
        }
    }
}
```

注 1：正确的说法是“用来获取现在的线程所对应的 java.lang.Thread 类实例的方法”。

执行范例如图 2-1 所示。的确有多个线程在调用 toString 方法，但（理所当然）值并没有损坏。就算想破坏它也没有办法。

```
Thread-2 prints [ Person: name = Alice, address = Alaska ]
Thread-2 prints [ Person: name = Alice, address = Alaska ]
Thread-2 prints [ Person: name = Alice, address = Alaska ]
Thread-2 prints [ Person: name = Alice, address = Alaska ]
Thread-0 prints [ Person: name = Alice, address = Alaska ]
Thread-0 prints [ Person: name = Alice, address = Alaska ]
Thread-0 prints [ Person: name = Alice, address = Alaska ]
Thread-1 prints [ Person: name = Alice, address = Alaska ]
Thread-1 prints [ Person: name = Alice, address = Alaska ]
Thread-1 prints [ Person: name = Alice, address = Alaska ]
Thread-1 prints [ Person: name = Alice, address = Alaska ]
Thread-0 prints [ Person: name = Alice, address = Alaska ]
Thread-0 prints [ Person: name = Alice, address = Alaska ]
(以下省略。以 CTRL+C 结束程序。)
```

图 2-1　执行范例结果

范例程序的类图如图 2-2 所示。

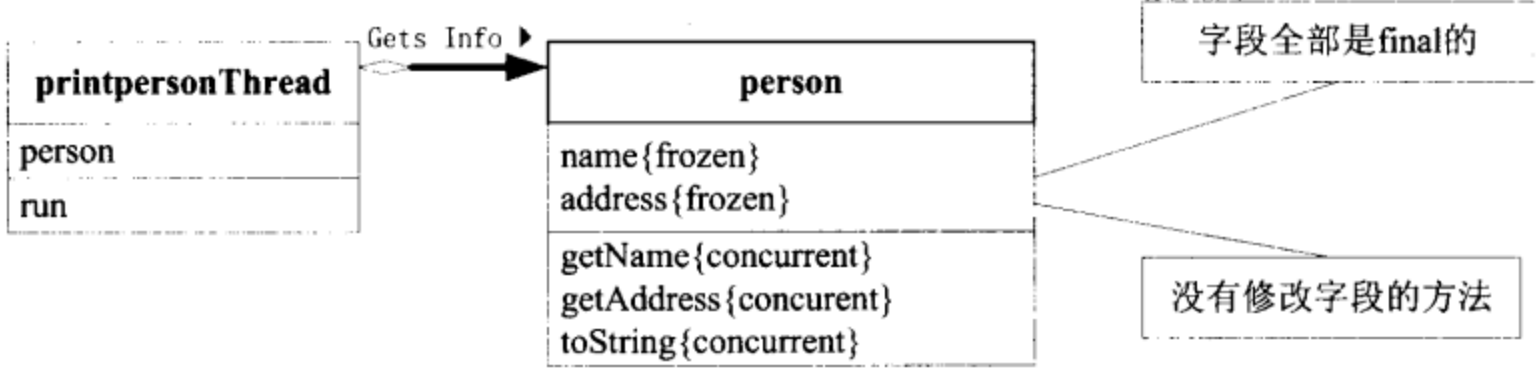


图 2-2　范例程序的类图

在图 2-2 中，字段名称加上{ frozen }的限制（frozen 意指“冻结的”）。这是 UML 中表现“建立实例并初始化字段后，就不能再次更改其值”的方式。在 Java 语言中，则是以“final 字段”做到。

另外，方法名称也注明{concurrent}（concurrent 是“并发”的意思）。这是 UML 中表现“多个线程同时执行也无妨”的方式。在 Java 语言中则是以“不需要加上 synchronized 的方法”（没有加上 synchronized 也要能够正确执行）做到。

请回想 Single Thread Execution Pattern（第 1 章）。Single Thread Execution Pattern 中，我们必须将可能修改、引用实例状态的地方设置为临界区间，使这个区域只有一个线程可以通行。但像 Person 类这样，实例的状态绝对不会改变时，情况就不一样了。即使有多个线程同时对这个实例进行操作，实例也不会出错。因为实例的状

态绝不会改变，所以实例就不需要使用 synchronized 保护了。即使想要去破坏实例，也没有办法。

使用 Immutable Pattern 的类（immutable 类）不需要使用 synchronized 方法进行共享互斥。因为不使用 synchronized 方法也能保障其安全性。讲到目前为止，好像尽是一些优点。不过，要保证 immutability 其实比想象中困难，所以需要特别注意（本章末的练习问题 2-4 会详细解说）。

Immutable Pattern 的所有参与者

Immutable Pattern 的所有参与者如下：

◆ *Immutable（不变的）参与者*

Immutable 参与者是一个字段值无法更改的类，也没有任何用来更改字段值的方法。当 Immutable 参与者的实例建立后，状态就完全不再变化。这时，我们就不用对 Immutable 参与者使用 Single Threaded Execution Pattern（第 1 章）。也就是说，Immutable 参与者的方法并不需要声明成 synchronized。

范例程序的 Person 类就是 Immutable 参与者（见图 2-3 和图 2-4）。

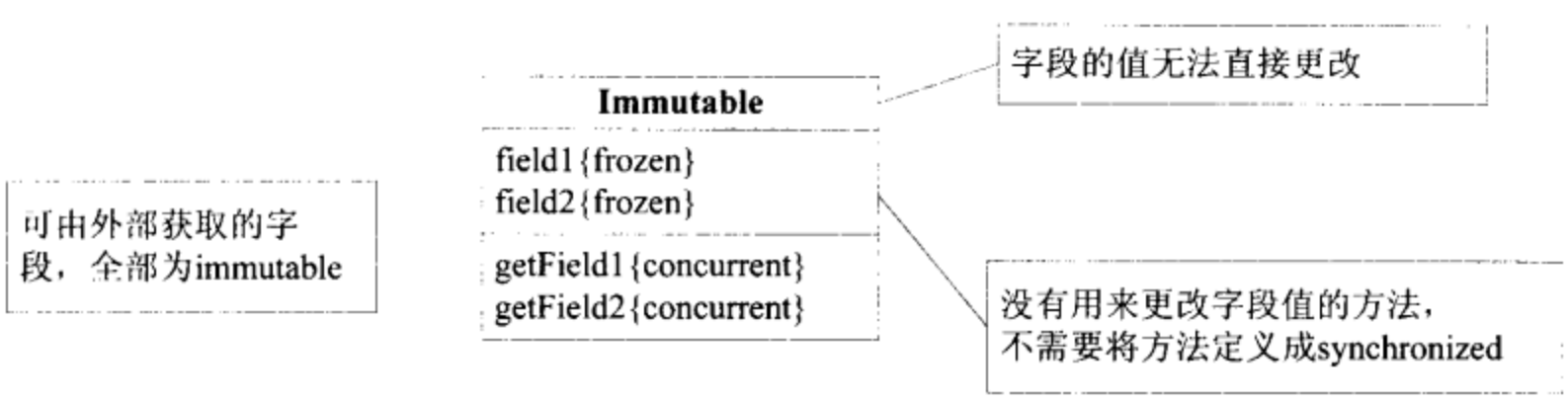


图 2-3　Immutable Pattern 的类图

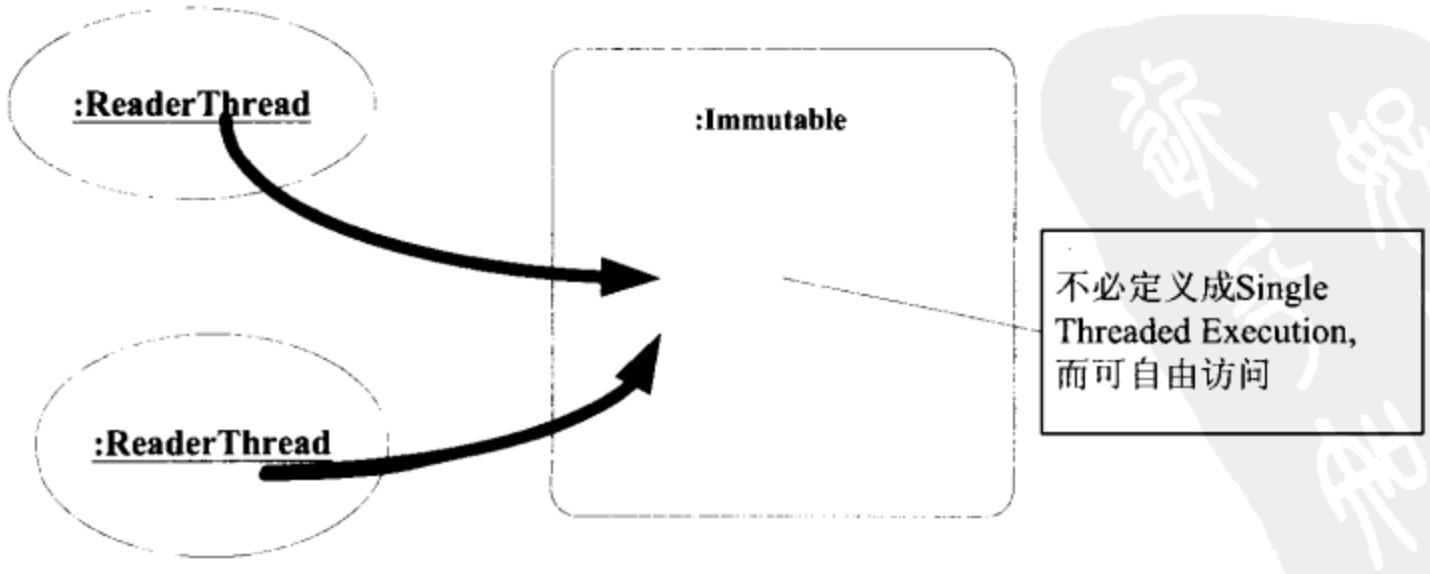


图 2-4　Immutable 参与者可被多个线程自由访问（时序图）

扩展思考方向的提示

何时使用（适用性）

Immutable Pattern 该在什么情况下使用呢？ 这个问题换个角度看，我们还会想问当设计者碰到什么情况，要去意识到实例的不变性呢？

◆ 当实例产生后，状态不再变化时

如同前文所说，当实例产生后，实例的状态就不再变化是首要条件。所谓实例的状态，是由字段的值所决定的，所以将字段设置成 final，并且不要定义 setter 方法则是重点所在（所谓 setter 方法，是指 setName、setAddress 类用来改变字段值的方法的总称）。

然而，这样还是不够的。就算字段都是 final，也没有 setter 方法，还是有可能定义出非 immutable 的类。因为就算字段的值不会改变，字段所属的实例还是有可能改变。具体的范例我们将在练习问题 2-4 讨论。

◆ 实例需要共享，而且访问很频繁时

使用 Immutable Pattern 的优点，在于“不需要使用 synchronized 保护”。不需要使用 synchronized 来保护最大的优点就是可在不丧失安全性与生命性的前提下，提高程序的执行性能。若实例由多数的线程所共享，而且访问可能会很频繁时，Immutable Pattern 就能发挥极大的优点。省下 synchronized 能对性能作多少贡献，我们留到练习问题 2-4 来实验。

思考成对的 mutable 类与 immutable 类 [执行性能]

关于执行性能，我们再进一步深入思考。假设现在有一个类可能由许多线程访问，并使用了 synchronized 保护其数据。如果这个类具有 setter 方法，那 Immutable Pattern 就不成立了。

但是，我们仔细检查程序代码后，发现实际上这个 setter 方法根本没有用到。所以我们就将字段设置成 final，并删除 setter 方法，仔细注意遵守不变性的原则，或许可以将类改成符合 Immutable Pattern 的条件。如果成功的话，就能使程序性能提高，值得好好庆祝一番。

很可惜的，假设我们在程序代码里发现，setter 方法还是有被用到。这种时候就无法配合 Immutable Pattern 了。可是，也不要就此心灰意冷。接下来我们可以仔细观察整个程序如何运用这个类。或许有分为使用 setter 方法的情况，与不使用 setter

方法的情况。如果这两种情况区分得很清楚，那我们是否可能将这个类二分为 mutable 类与 immutable 类呢？ 接下来，再设计成可由 mutable 类的实例构造出 immutable 类的实例，并也可从 immutable 类的实例构造出 mutable 类的实例。这样一来，immutable 类的部分就可以使用 Immutable Pattern 了。

来看个具体的例子吧。Java 的标准类链接库里，就具有成对的 mutable 类与 immutable 类。例如 java.lang.StringBuffer 类与 java.lang.String 类。StringBuffer 类是用来表示字符串的 mutable 类，StringBuffer 所表示的字符串可以自由地修改。为了确保改写时的安全性，进行改写操作时，都有适当地使用 synchronized 加以保护。而 String 则是用来表示字符串的 immutable 类。String 实例所表示的字符串无法做任何修改。所有对 String 类进行操作的方法都不是 synchronized，所以引用的速度会比较快[注2]。

StringBuffer 类具有一个以 String 作为参数的构造器。另外，String 类也有一个以 StringBuffer 作为参数的构造器。也就是说，StringBuffer 的实例与 String 的实例是可以互相转换的。

Java 语言中对“字符串”这一个概念，就提供了成对的 StringBuffer 类与 String 类。当属性可能频繁修改时，就使用 StringBuffer 类；而属性没有修改必要，只会引用其值时，就使用 String 类。将类的角色做妥善的区分。

为了保护类的不变性 [复用性]

不变性是很微妙的性质。有的时候只要稍微修改程序代码，可能就会失去不变性。以 Immutable Pattern 为前提，删除程序代码的 synchronized 后，如果又失去不变性，会使类的安全性也丧失了，故需要特别注意。在程序的注释与 API 说明文件里，必须明确地记述其不变性。如果自己或别人日后遗忘这个类不变性的重要性，而把字段的 final 删除掉、定义出 setter 方法，就失去不变性了。请注意即使没有 setter 方法，只要字段的访问权限不是 private，字段的值就有被其他类更改的风险。

另外，下列这些修改操作也有可能危及类的不变性。这些操作乍看之下并没有什么问题，但实际上却需要十分注意。具体的问题我们留到练习问题再来思考。

- 直接以 getter 方法返回字段所保存的实例
  →练习问题 2-4
  getter 方法是指用来返回字段值的方法。
- 将构造器的参数传入的实例，直接存进字段里
  →练习问题 2-5

注 2：要由多个字符串建立出新的字符串，StringBuffer 类的速度会比 String 类快。

标准类链接库里使用到的 Immutable Pattern

这里将介绍一些 Java 的标准类链接库里使用到的 Immutable Pattern 类。

◆ 表示字符串的 java.lang.String 类

java.lang.String 是 immutable 类。
故实例建立后，字符串的属性就不会再有变化。

◆ java.lang.Integer 等类

java.lang.Integer、java.lang.Short 这些基本类型的包装类（wrapper class；参考表 2-2）都是 immutable 类。当实例声明后，所包装（wrap）的值就不会再改变。

表 2-2 包装类

基本类型	包装类
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	Java.lang.Void

▶▶ 补充说明：java.lang.Void 类

java.lang.Void 与其他包装类不同，无法建立实例。这个类是用来保存表示基本类型 void 的 Class 对象的实例，使用在 reflection 与 serialization 等机制上。详细用法已超出本书的范围，请参考 API 说明文件。

◆ 表示颜色的 java.awt.Color 类

java.awt.Color 类是 immutable 类。
当实例建立后，所代表的颜色就不会再改变。
而 Color.red、Color.blue 这些代表特定颜色的实例，也可以共享使用。

◆ java.awt.Point 类不是 immutable

java.awt.Point 是表示“点”的类，但不是 immutable。在 API 说明文件中，查

询 Point 类的方法，虽然没有 set 开头的方法名称，但还是有 move 这类的方法。因为这些方法会改变 Point 的字段值，所以 Point 类不是 immutable。

相关 Pattern

Single Threaded Execution Pattern（第 1 章）

Immutable Pattern 实例的状态不会改变，所以没有保护的必要。

另一方面，Single Threaded Execution Pattern 的做法，则是当一个线程正在更改实例的状态时，不允许其他线程来访问这个实例。这个时候，会是下面两种情况的其中之一：

- 写入与写入的冲突（write-write conflict）
  当一个线程正在更改实例的状态时（write），有其他线程也试图更改其状态（write）所发生的冲突。
- 读取与写入的冲突（read-write conflict）
  当一个线程正在获取实例的状态时（read），有其他线程试图更改其状态（write）时所发生的冲突。

Immutable Pattern 状态不会变化（write），所以不会发生 write-write conflict，也不会发生 read-write conflict。Immutable Pattern 可能出现的只有 read-read 的情况，但这不算是 conflict。

Read-Write Lock Pattern（第 6 章）

如同前文所述，Immutable Pattern 在多个线程间只会出现 read-read 的情况。所以多个线程可以自由地访问实例。

Read-Write Pattern 也是利用了 read-read 不会引起 conflict 的技巧。Read-Write Pattern 中，进行 read 的线程与进行 write 的线程是分开来考虑的。而只在出现 write-write、read-write 等 conflict 的情况下，进行线程的互斥，而 read-read 的时候就不做互斥了。这也能使程序的性能提高。

Flyweight Pattern（参考附录 E[GoF][Yuki01]）

Immutable Pattern 中，实例的状态不会改变。所以多个线程可以共享一个实例。

Flyweight Pattern 则是为了提高内存的执行性能而共享实例。故 Immutable Pattern 与 Flyweight Pattern 可以同时使用。

进阶说明：final

Java 的 final 有许多不同的用途，意义也多少有点不同。在这里就稍微整理一下吧。

◆ *final 类*

当类声明成 final 时，这个类无法延伸。也就是说不能定义 final 类的子类。

因为 final 类的类无法定义子类，所以 final 类声明的方法也不会被覆盖（override）。

◆ *final 方法*

若将实例方法声明成 final 时，这个方法无法被子类所覆盖（override）。若将类方法声明成 final 时，这个方法无法被子类所隐藏（hide）。

Template Method Pattern（参考附录 E [GoF][Yuki01]）中，就有将模板（template）方法设置成 final 方法的内容。

▶▶ 补充说明：覆盖与隐藏

实例方法被子类的方法覆盖时，实际被调用的方法是在执行时决定的。

类方法被子类的方法隐藏时，实例被调用的方法是在编译时决定的。

详细内容请参考附录 E[JLS]。

◆ *final 字段*

final 的字段，在这一章中一直出现。

final 的字段的值只能指定一次。

要将值指定给 final 变量字段，有下列两种方法：

第一种是在声明字段时就直接赋初始值。

```
class Something {
    final int value = 123;
}
```

另外一种则是在构造器中将值赋给字段（blank final）。

```
class Something {
    final int value;
    Something() {
        this.value = 123;
    }
}
```

要将值赋给 final 实例的字段，有下面两种方法：

第一种是在声明字段时就直接赋初始值。

```
class Something {
    static final int value = 123;
}
```

另一种则是在 static 块（静态初始化子）中将值赋给字段（blank final）。

```
class Something {
    static final int value;
    static {
        value = 123;
    }
}
```

如同前面已经反复说明的，final 的字段无法使用 setValue 类的 setter 方法再次赋其他的值。

◆ *Final 变量与 final 参数*

局部变量与方法的参数，都可以声明成 final。final 的变量的值只能够指定一次。

final 参数的值则是一次都不能指定，因为当方法被调用时，已经有值指定进去了。

重点回顾

这一章我们学习了 Immutable Pattern。

当一个类的实例声明后，状态就完全不再改变。这时就算有多个线程同时调用这个类的方法也无所谓。故方法没有 synchronized 的必要。这样一来，可在不丧失安全性与生命性的前提之下，提高程序性能。

这就是 Immutable Pattern。

Immutable Pattern 保护类的方式不是 synchronized，而是 immutability（不变性）。

而保护类的 immutability，就是程序设计者的工作了。请务必挑战看看后面的练习问题。

练习问题

解答请见附录 A

问题 2-1（基础知识的确认）

Java 下列语句中，正确的请回答○，不正确的请回答×。

（1）java.lang.String 类是 immutable 的类。

（2）java.lang.StringBuffer 类是 immutable 的类。
（3）变成 final 的字段无法被指定两次。
（4）变成 private 的字段可以从该类与子类直接连接。
（5）将方法设为 synchronized 不会导致什么问题，所以应该尽量都加上 synchronized。

问题 2-2（immutability 的确认）

Java 某个人看到 String 类的 API 说明文件中，有 replace 这个方法后，说了下面这句话：

“replace 方法，会取代字符串中的字符。也就是说，执行 List 2-4 这样的程序，结果如图 2-5 所示一样，BAT 的'B'被取代为'C'，最后显示出的是 CAT。这样 String 类还能说是符合 Immutable Pattern 吗？ 我看不是吧。”

请针对上面这个意见提出您的反驳。

List 2-4 String 真的是 immutable 吗（Main.java）

```
public class Main {
    public static void main(String[] args) {
        String s = "BAT";
        System.out.println(s.replace('B', 'C')); // 'B'取代成'C'
    }
}
```

```
CAT
```

图 2-5 执行结果

问题 2-3（执行性能比较）

Java List 2-5 是测试以 Immutable Pattern 取代 synchronized 后，能提升多少性能的程序。Synch 类的 toString 是 synchronized 方法，而 NotSynch 的 toString 则不是 synchronized 方法。程序在 Main 类里反复地调用这些方法。请在您的计算机环境中，比较这两者执行时所相差的时间吧。在实际开始执行之前，请先估计可能会差多少。

在这个程序中，使用 System.currentTimeMillis()表达式以毫秒为单位获取现在的时刻。调用方法的次数定义在 CALL_COUNT，请配合您的环境做必要的修改。

List 2-5 使用 Immutable Pattern 可以提升多少性能呢（Main.java）

```
public class Main {
    private static final long CALL_COUNT = 1000000000L;
    public static void main(String[] args) {
        trial("NotSynch", CALL_COUNT, new NotSynch());
```

```
        trial("Synch",   CALL_COUNT, new Synch());
    }
    private static void trial(String msg, long count, Object obj)
{
        System.out.println(msg + ": BEGIN");
        long start_time = System.currentTimeMillis();
        for (long i = 0; i < count; i++) {
            obj.toString();
        }
        System.out.println(msg + ": END");
        System.out.println("Elapsed time = " + (System.currentTimeMillis()
- start_time) + "msec.");
    }
}

class NotSynch {
    private final String name = "NotSynch";
    public String toString() {
        return "[ " + name + "]";
    }
}

class Synch {
    private final String name = "Synch";
    public synchronized String toString() {
        return "[ " + name + "]";
    }
}
```

问题 2-4（没有 setter 方法，却不是 immutable）

Java 下面的 UserInfo 类（List 2-6）可用来表达用户的信息。这个类的 info 字段是 private final 的，也没有 setter 方法，可是 UserInfo 类并称不上是 immutable。请问为什么？

List 2-6　为什么不是 immutable（UserInfo.java）

```
public final class UserInfo {
    private final StringBuffer info;
    public UserInfo(String name, String address) {
        this.info = new StringBuffer("<info name=\"" + name + "\"
```

```
address=\"" + address + "\" />");
    }
    public StringBuffer getInfo() {
        return info;
    }
    public String toString() {
        return "[ UserInfo: " + info + " ]";
    }
}
```

问题 2-5（判断是不是 immutable）

Java 某个人正在定义用来表示画面上直线的 Line 类（List 2-7）。请问这个 Line 类是不是 immutable。

List 2-7 Line 是不是 immutable 类（Line.java）

```
public class Line {
    private final Point startPoint;
    private final Point endPoint;
    public Line(int startx, int starty, int endx, int endy) {
        this.startPoint = new Point(startx, starty);
        this.endPoint = new Point(endx, endy);
    }
    public Line(Point startPoint, Point endPoint) {
        this.startPoint = startPoint;
        this.endPoint = endPoint;
    }
    public int getStartX() { return startPoint.getX(); }
    public int getStartY() { return startPoint.getY(); }
    public int getEndX() { return endPoint.getX(); }
    public int getEndY() { return endPoint.getY(); }
    public String toString() {
        return "[ Line: " + startPoint + "-" + endPoint + " ]";
    }
}
```

List 2-8 Line 类所使用到的 Point 类（Point.java）

```
public class Point {
    public int x;
    public int y;
```

```
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
```

问题 2-6（用 mutable 的实例制作出 immutable 的实例）

Java 下面的 MutablePerson 类（List 2-9）与 ImmutablePerson 类（List 2-10）是本章中曾提到的成对的 mutable 类与 immutable 类。

MutablePerson 类的实例产生后，其字段值还可以变化（mutable）。而 ImmutablePerson 类的实例产生后，字段的值就不会变化了（immutable）。

MutablePerson 的 getName 方法与 getAddress 方法不是 public 也不是 private，而是使用默认的访问权限（只有同一个包（package）内可以访问）。这只是为了让 ImmutablePerson 类可以使用 getName 方法与 getAddress 方法访问数据（所以这两个类都放在相同的 person 包里）。

那么，接下来就是问题了。这两个类中，还存在有一些安全性上的漏洞，请问是在哪里呢？

List 2-9　MutablePerson 类（MutablePerson.java）

```
package person;

public final class MutablePerson {
    private String name;
    private String address;
    public MutablePerson(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public MutablePerson(ImmutablePerson person) {
        this.name = person.getName();
        this.address = person.getAddress();
    }
    public synchronized void setPerson(String newName, String
```

```
newAddress) {
        name = newName;
        address = newAddress;
    }
    public synchronized ImmutablePerson getImmutablePerson() {
        return new ImmutablePerson(this);
    }
    String getName() { // Called only by ImmutablePerson
        return name;
    }
    String getAddress() { // Called only by ImmutablePerson
        return address;
    }
    public synchronized String toString() {
        return "[ MutablePerson: " + name + ", " + address + " ]";
    }
}
```

List 2-10 ImmutablePerson 类（ImmutablePerson.java）

```
package person;

public final class ImmutablePerson {
    private final String name;
    private final String address;
    public ImmutablePerson(String name, String address) {
        this.name = name;
        this.address = address;
    }
    public ImmutablePerson(MutablePerson person) {
        this.name = person.getName();
        this.address = person.getAddress();
    }
    public MutablePerson getMutablePerson() {
        return new MutablePerson(this);
    }
    public String getName() {
        return name;
    }
```

```
    public String getAddress() {
        return address;
    }
    public String toString() {
        return "[ ImmutablePerson: " + name + ", " + address + " ]";
    }
}
```

第3章

Guarded Suspension——要等到我准备好喔

Guarded Suspension Pattern

在家里正当换衣服的时候，忽然客厅门铃响了，原来是邮递员送信来了。因为衣服换到一半不能开门，所以就先说出“请等我一下！”，请邮递员在外面稍等。当衣服换完以后，再把门打开，说句“让您久等了”……

这一章要学习的是 Guarded Suspension Pattern。

guarded 是“被保护着的”、“被防卫着的”的意思，suspension 则是“暂停”的意思。当现在并不适合马上执行某个操作时，就要求想要执行该操作的线程等待，这就是 Guarded Suspension Pattern。

Guarded Suspension Pattern 会要求线程等候，以保障实例的安全性。正如同我们请邮递员在门外等候，以保障自己的隐私权一样。

Guarded Suspension Pattern 还有 guarded wait、spin lock 等各种称呼，详见后面的说明部分。

范例程序

首先我们要看的是范例程序。这里所写的程序，是一个线程（ClientThread）对另外一个线程（ServerThread）传递请求（Request）的实例。这是非常简单的线程沟通。类的说明如表 3-1 所示。

表 3-1　类一览表

名　　称	说　　明
Request	用来表示“请求”的类
RequestQueue	依次存放请求以待使用的类
ClientThread	送出请求的类
ServerThread	接受请求的类
Main	操作测试用的类

范例程序的顺序图如图 3-1 所示。

这张图里面，:ClientThread 与:ServerThread 的边框以粗线表示。粗边框的长方形表示该对象与线程有相关性。意即该对象可以主动地调用方法（这是 UML 规定的表示法）。我们将:ClientThread 与:ServerThread 称为主动对象（active object），并将:RequestQueue 称为被动对象（passive object）。

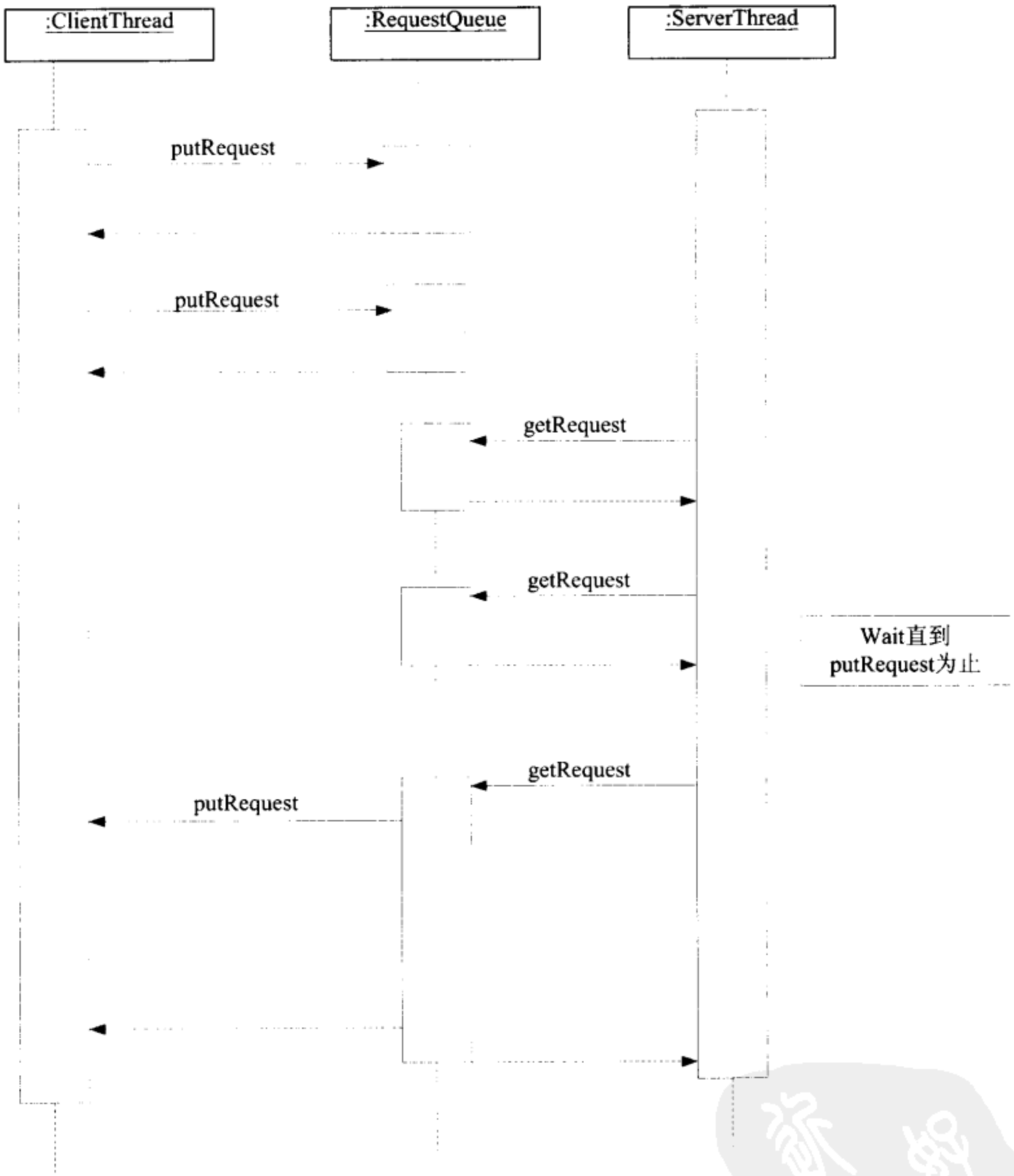


图 3-1　范例程序的顺序图

Request 类

Request 类（List 3-1）是用来表示请求。虽说是请求，但在这里只不过是用来作为从 ClientThread 传递给 ServerThread 的实例，所以没有定义特殊的处理。Request 是一个只有存放名称（name 字段）的类。

List 3-1 表示请求的 Request 类（Request.java）

```
public class Request {
    private final String name;
    public Request(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public String toString() {
        return "[ Request " + name + " ]";
    }
}
```

RequestQueue 类

RequestQueue 类（List 3-2）是用来依次存放请求的类。这个类拥有 getRequest 与 putRequest 两个方法。

◆ *getRequest 方法*

getRequest 方法会从 RequestQueue 里所存放的请求中，将最早传入的一个取出，作为返回值。如果没有任何一个请求存在，就等待其他线程 putRequest。

◆ *putRequest 方法*

使用 putRequest 方法，可以增加 1 个请求。当线程想要在 RequestQueue 中新增一个 Request 的实例时，就会调用这个方法。

也就是说，RequestQueue 可使用 putRequest 方法塞入 Request 的实例，并可用 getRequest 方法按塞入的顺序获取 Request 的实例。像这样的结构，我们通常称之为队列（queue），或是 FIFO（First In First Out）。例如，在银行的服务柜台前排队，也是队列的一种。

List 3-2 存放请求用的 RequestQueue 类（RequestQueue.java）

```
import java.util.LinkedList;

public class RequestQueue {
    private final LinkedList queue = new LinkedList();
    public synchronized Request getRequest() {
        while (queue.size() <= 0) {
```

```
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return (Request)queue.removeFirst();
    }
    public synchronized void putRequest(Request request) {
        queue.addLast(request);
        notifyAll();
    }
}
```

这个 List 3-2 在后面还会详细地讨论，在这里就只先提示一些关键的地方。

- getRequest、putRequest 都是 synchronized 方法；
- getRequest 的前面有 while 语句，用来测试条件是否成立；
- 在 while 的块中会调用 wait；
- 在 while 语句之后，才进行实例想做的处理（removeFirst）；
- 在 putRequest 里，要调用 notifyAll。

ClientThread 类

ClientThread 类（List 3-3）是用来表示送出请求的线程。ClientThread 拥有 RequestQueue 的实例（requestQueue），会不断调用 putRequest。请求的名称依次命名为"No.0"、"No.1"、"No.2"、……。

为了动态变动送出请求（调用 putRequest）的时机，所以使用 java.util.Random 类，产生范围在 0 以上而未满 1000 的随机数，sleep 这个长度的时间（单位为 ms）。

List 3-3 送出请求用的 ClientThread 类（ClientThread.java）

```
import java.util.Random;

public class ClientThread extends Thread {
    private Random random;
    private RequestQueue requestQueue;
    public ClientThread(RequestQueue requestQueue, String name,
long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }
```

```
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Request request = new Request("No." + i);
             System.out.println(Thread.currentThread().getName()
+ " requests " + request);
            requestQueue.putRequest(request);
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

ServerThread 类

ServerThread 类（List 3-4）是用来接受请求的线程。这个类也拥有 RequestQueue 的实例（requestQueue）。ServerThread 类会调用 getRequest 方法获取请求。

与 ClientThread 相同，ServerThread 类也使用随机数动态地 sleep。

List 3-4 接受请求用的 ServerThread 类（ServerThread.java）

```
import java.util.Random;

public class ServerThread extends Thread {
    private Random random;
    private RequestQueue requestQueue;
    public ServerThread(RequestQueue requestQueue, String name,
long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
            Request request = requestQueue.getRequest();
            System.out.println(Thread.currentThread().getName()
+ " handles  " + request);
            try {
                Thread.sleep(random.nextInt(1000));
```

```
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Main 类

在 Main 类之中（List 3-5）首先会建立 RequestQueue 的实例（requestQueue）。并建立名为 Alice 的 ClientThread 实例和名为 Bobby 的 ServerThread 实例，对两者都传入 requestQueue 对象，并 start 之。

3141592L 及 6535897L 只是用来作为随机数的起始值，并无特别意义。

List 3-5 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        RequestQueue requestQueue = new RequestQueue();
        new ClientThread(requestQueue, "Alice", 3141592L).start();
        new ServerThread(requestQueue, "Bobby", 6535897L).start();
    }
}
```

执行范例如图 3-2 所示。我们可以看到 Alice 会一直送出请求（requests），Bobby 则不停地处理请求（handles）。

```
Alice requests [ Request No.0 ]        ←Alice 送出请求 No.0
Bobby handles  [ Request No.0 ]        ←Bobby 处理请求 No.0
Alice requests [ Request No.1 ]        ←Alice 送出请求 No.1、No.2
Alice requests [ Request No.2 ]
Bobby handles  [ Request No.1 ]        ←Bobby 处理请求 No.1、No.2
Bobby handles  [ Request No.2 ]
Alice requests [ Request No.3 ]        ←Alice 送出请求 No.3
Bobby handles  [ Request No.3 ]        ←Bobby 处理请求 No.3
Alice requests [ Request No.4 ]        ←Alice 送出请求 No.4
Bobby handles  [ Request No.4 ]        ←Bobby 处理请求 No.4
Alice requests [ Request No.5 ]        ←Alice 送出请求 No.5、No.6
Alice requests [ Request No.6 ]
Bobby handles  [ Request No.5 ]        ←Bobby 处理请求 No.5、No.6
Bobby handles  [ Request No.6 ]
(以下省略。CTRL+C 结束程序)
```

图 3-2　执行范例

java.util.LinkedList 类的操作

在详细阅读 List 3-2 前，在这里先就 queue 字段的类型——java.util.LinkedList 类作简单的解说。LinkedList 是用来表示串状连接在一起的对象集合体的类。在范例程序中，使用了下面 3 种方法：

◆ *Object removeFirst()*

取出链表前面第一个元素，并将这个元素返回。如果一个元素都不剩，会抛出 java.util.NoSuchElementException 异常（意指“除此元素之外”）。

◆ *void addList(Object obj)*

在链表的尾端加上一元素 obj。

◆ *int size()*

返回链表连接的元素数量。

使用 java.util.LinkedList 类，可以简单的实现出队列。不过，这个类并不是线程安全的（这是为了使运行速度提高）。

仔细分析 getRequest 方法

接下来我们就来彻底地分析 List 3-2 的 getRequest 方法吧：

```
public synchronized Request getRequest() {
    while (queue.size() <= 0) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    return (Request)queue.removeFirst();
}
```

◆ 施加条件加以防卫

首先，请先思考 getRequest 方法应该做到的“目的操作”是什么。这个方法的目的是“从 queue 里取出一个 Request 的实例”。也就是说执行

```
queue.removeFirst()
```

这段语句。不过，这一行程序要能安全地执行，必须要满足 queue.size() > 0 的条件才行。条件是“元素数要大于 0”，意即“元素有 1 个以上”。像这样，我们将

这种一定要满足的条件，称为 Guarded Suspension Pattern 的警戒条件（guard condition）。

```
queue.size() > 0                    … 警戒条件
```

仔细看 getRequest 方法的 while 语句中的条件式后会发现，while 语句中的条件式是警戒条件的逻辑否定。程序就是靠这个 while 语句，来保证 removeFirst 方法被调用的时候，警戒条件一定会成立。

满足警戒条件的逻辑否定时——也就是警戒条件不成立时——就绝对不会进入 while 的下一个语句（调用 removeFirst）。

◆ 不等待的情况、等待的情况

当线程抵达 while 语句的时候，会分为满足警戒条件与不满足警戒条件两种情况。

当警戒条件成立时，线程不会进入 while 块，而会直接进行到下一个语句，调用 removeFirst 方法。这个时候不会执行到 wait，所以线程不会等待。

当警戒条件不成立时，线程会进入 while 块中，执行 wait 开始等待。只有当警戒条件不成立时，线程才会执行 wait。

◆ 执行 *wait* 等待条件的变化

当警戒条件不成立时，线程就会执行 wait 开始“等待”的操作。那么，线程是在等待什么呢？

“当然一定是在等待 notify/notifyAll 嘛！”没错，的确如此。调用 wait 后的线程，除非被 notify/notifyAll，不然都会在等待区里持续等待的状态。不过，这里要思考的是更深一层的意义。线程真正所等待的，是实例状态的变化。线程之所以进入等待状态，是因为警戒条件不成立。是被这个警戒条件挡住，才无法向前的。其实线程是很想要继续前进的。等待中的线程，正等待着条件的变化。也就是说，线程等待着警戒条件的成立。

好像前面说了一大堆理所当然的废话，事实上这个部分真的请多加留意。因为当您了解“线程在等待什么”，您就知道何时应该“调用 notify/notifyAll 了”。

正在 wait 的线程，期望当警戒条件成立时，能收到一声 notify/notifyAll。因为只有收到 notify/notifyAll，才能继续执行后面的语句。

◆ 前进到 *while* 的下一个语句时能够保证的事

那么，我们继续看 getRequest 方法。

while 语句的后面，就是主要想要进行的操作了。这时，while 语句的警戒条件，一定是成立的。

也就是说，调用 removeFirst 的时候，

```
queue.size() > 0                    ... 警戒条件
```

这个表达式一定会满足。因为“queue 中的元素数大于 0”，所以 removeFist 方法绝不会抛出 NoSuchElementException 异常。

这就是 getRequest 方法。

下面是前面说明的架构稍作整理的结果：

```
while (“警戒条件”的逻辑否定) {
        使用 wait 等待;
}
进行“目的操作”;
```

在进行真正的操作前，警戒条件一定会满足。这种在进行某个操作前一定要达到的条件，我们称为事前条件（precondition）。警戒条件就是“目的操作”的事前条件。

仔细分析 putRequest 方法

接着是 putRequest 方法，这个方法比较短，可以分析得比较快。

```
public synchronized void putRequest(Request request) {
    queue.addLast(request);
    notifyAll();
}
```

执行 addLast 方法，在 queue 的最后面新增一个请求（request）。

这时，queue 内所存放的元素至少会有 1 个。所以下面的表达式为真。

```
queue.size() > 0
```

前面的 getRequest 中，wait 中的线程所等待的是什么呢？对，正是这个条件——警戒条件——的成立。所以在这里，我们就调用 notifyAll。

这就是 putRequest。

synchronized 的意义

我们已经看完了 getRequest 方法与 putRequest 方法。不过前面没有特别提及，这两者都是 synchronized 方法。

看到 synchronized，就要想到如前面所说的，思考“这个 synchronized 是在保护什么？”。这个 synchronized 所保护的是 queue 字段（LinkedList 的实例）。例如保护 getRequest 方法中的这两个操作：

- 判断 queue 字段的元素数是否大于 0；
- 从 queue 字段取出 1 个元素。

一定“只有惟一的线程在进行”。这是 Single Threaded Execution Pattern（第 1 章）。

wait 与锁定

针对还不习惯 Java 的 wait 方法的读者，这里再稍微加以补充说明。

当某个线程试图去执行某个实例的 wait 方法时，这个线程必须获取该实例的锁定。若像范例程序这样在 synchronized 方法中调用 wait 方法，在执行 wait 的时候，的确线程正获取 this 的锁定。

然而，当线程执行 this 的 wait 方法后，会进入 this 的等待区，这时，线程就会解除 this 的锁定。

而线程可能会因为 notify、notifyAll 或 interrupt 退出等待区。不过，在实际进行下一个语句之前，必须再次获取 this 的锁定才行。

上述线程的行为，在前面已经详细解说过了。如果您忘记了，请翻到前面再复习一下。这不是 Java 线程的进阶知识，而是基础知识。请务必牢牢记在脑海里。

Guarded Suspension Pattern 的所有参与者

Guarded Suspension Pattern 的所有参与者如下：

◆ *GuardedObject（被防卫的对象）参与者*

GuardedObject 参与者是一个拥有被防卫的方法（guardedMethod）的类。当线程执行 guardMethod 时，只要满足警戒条件，就会马上执行。但警戒条件不成立时，就要开始等待。警戒条件的成立与否，会随 GuardedObject 参与者的状态变化。

GuardedObject 参与者除了 guardedMethod 以外，可能还会有用来更改实例状态（特别是用来更改警戒条件）的方法（stateChangingMethod）。

在 Java 语言中，是使用 while 语句与 wait 方法来实现 guardedMethod 的。而使用 notify/notifyAll 方法来实现 stateChangingMethod。

在范例程序中，RequestQueue 类就是 GuardedObject 参与者（见图 3-3）。getRequest 方法与 putRequest 方法则分别是 guardedMethod 与 stateChangingMethod（见图 3-4）。

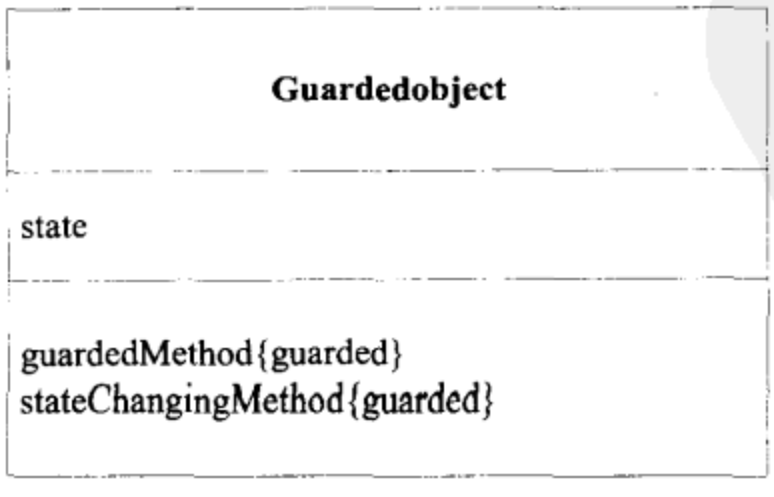


图 3-3　Guarded Suspension Pattern 的类图

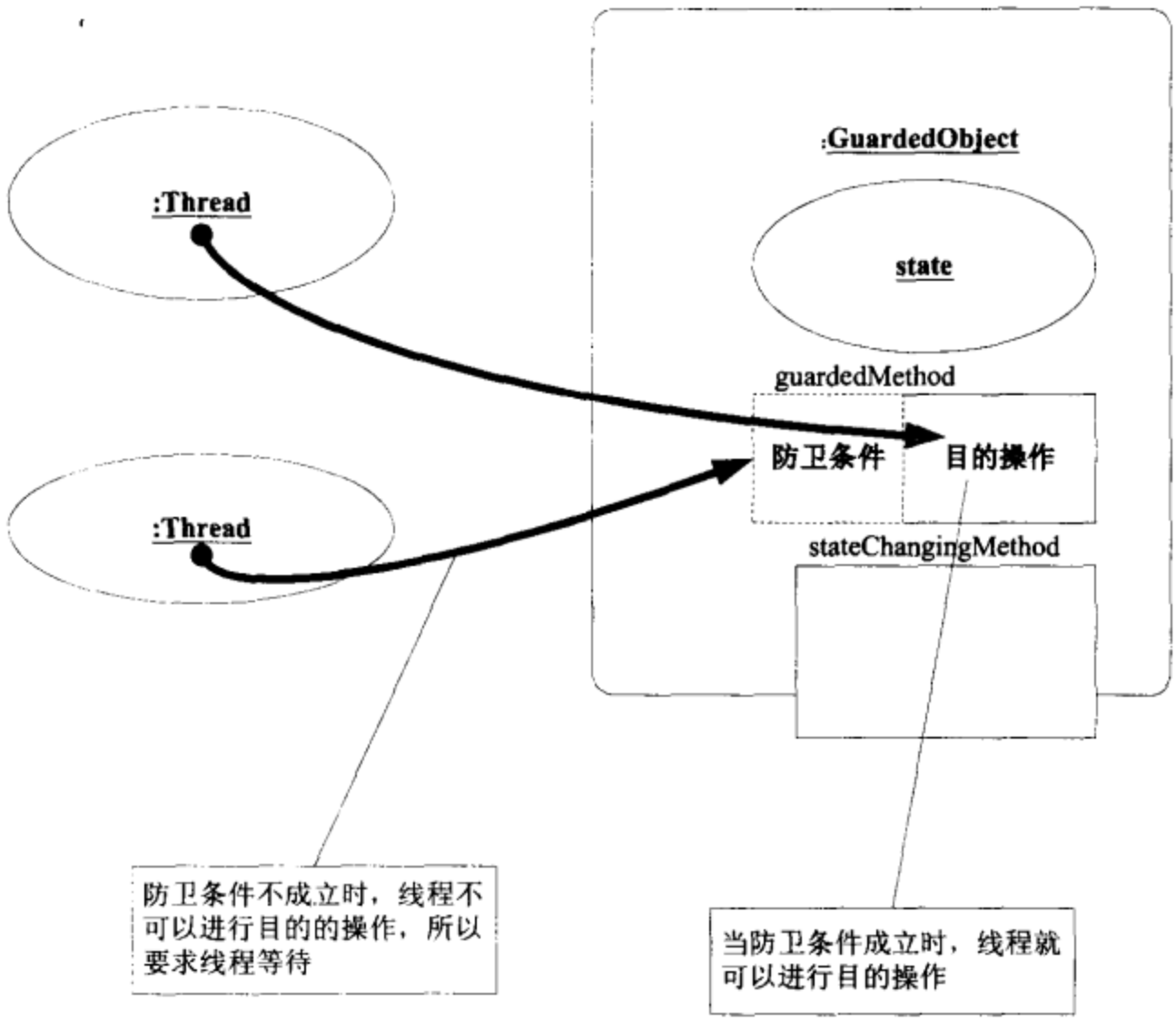


图 3-4　Guarded Suspension Pattern 的时序图（警戒条件不成立时）

扩展思考方向的提示

有条件的 synchronized

Single Threaded Execution Pattern（第 1 章）中，只要有一个线程进入临界区间，其他的线程就不能进入，而被请求等待。

而 Guarded Suspension Pattern 中，线程要不要等待，则是依警戒条件决定的。Guarded Suspension Pattern 是附加条件的 Single Threaded Execution Pattern。也就是“有条件的 synchronized”。

多线程版的 if

当然，单线程的程序，是不需要 Guarded Suspension Pattern 的。单线程中，操作主体的线程只有一个，如果这个惟一的线程进入等待状态，其他的线程都无法改变实例的状态。所以，即使实例“现在”的状态不理想，要线程等待的话，这个实例的状态也不过是一直保持不理想罢了。

所以一般来说，单线程的程序，警戒条件会使用 if 语句处理。

从这个角度看来，Guarded Suspension Pattern 就像是“多线程版的 if”。

忘记更改状态与生命性

wait 中的线程会在 notify/notifyAll 的时候重新测试警戒条件。无论被 notify/notifyAll 多少次，如果还是没有满足警戒条件，那线程会随着 while 循环再度进入 wait。

如果程序写错，忘记修改 GuardedObject 参与者的状态，那警戒条件无论经过多久都不会成立。这种时候，就算进行再多次的 notify/notifyAll，线程也不会向前执行，程序就失去生命性了。

所以我们可能会想将 wait 经过一段很长的时间还没有被 notify/notifyAll 的现象，认定为一种错误，而想要中断程序运行。想要在一定时间后中断操作时，我们可以在调用 wait 方法时，在参数里指定终止（timeout）的时间。详见 Balking Patten（第 4 章）的进阶说明。

wait 与 notify/notifyAll 的责任 [复用性]

如果仔细阅读范例程序的话，会发现只有 RequestQueue 类有用到 wait/notifyAll。在 ClientThread、ServerThread、Main 等其他类里都没有出现 wait/notifyAll。Guarded Suspension Pattern 的实现是封闭在 RequestQueue 类里的。

之所以隐藏住 wait/notifyAll，是为了 RequestQueue 类的复用性。使用 RequestQueue 类的一方，并不需要考虑 wait/notifyAll 的问题，只要调用 getRequest 方法与 putRequest 方法就行了。

各种各样的称呼

与 Guarded Suspension Pattern 类似的处理，有各种不同的称呼。但随着参考文献与文本语句上的不同，相同的名称有时候也代表不同的意义。所以请注意下面介绍的操作与名称，只有参考程度。

共通的特征是下面 3 点：

- 有循环的存在
- 有条件的测试
- 因为某种原因在“等待”

◆ *guarded suspension*

意义是“被阻挡而暂时中断执行”。这个名称并没有表现出实现方法。

◆ *guarded wait*

意义是“被阻挡而等待”。大致上是线程使用 wait 等待，等到被 notify/notifyAll 后再次测试条件的实现方法。使用 wait 等待的时间，其实是停止在等待区里停止执行，所以不会浪费到 Java 执行环境的处理时间。

◎等待端的范例

```
while (!ready) {
    wait();
}
```

◎唤醒端的范例

```
ready = true;
notifyAll();
```

◆ *busy wait*

“忙碌地等待”的意思。线程不使用 wait 等待，而使用 yield（尽可能把优先级交给其他线程），并不断测试条件的实现方法。因为等待中的线程也持续运行着，所以会浪费 Java 虚拟机的时间。yield 是 Thread 类的类方法[注1]。

◎等待端的范例

```
while (!ready) {
    Thread.yield();
}
```

◎唤醒端的范例

```
ready = true;
```

◆ *spin lock*

“旋转而锁定”的意思。表现出条件成立前 while 循环不断“旋转”的样子。spin lock 有时意思与 guarded wait 相同（参考附录 E [Holub00]），有时则与 busy wait 相同。另外，有时候则是指一开始时使用 busy wait，之后再切换成 guarded wait 的方式（参考附录 E [Lewis00]）。另外，有些以硬件实现的同步机制，也称为 spin lock。

◆ *polling*

“进行舆论调查”的意思。反复检查某个事件是否发生，当发生时，就进行对应处理的方式（参考附录 E [Lea]）。

注 1：Thread.yield 不会解除锁定，所以这段程序代码不可以写在 synchronized 里。而 ready 字段必须声明成 volatile。

相关 Pattern

Guarded Suspension Pattern 用于许多与并发性相关的 Pattern 中。

Single Threaded Execution Pattern（第 1 章）

“测试警戒条件”与“测试后更改状态(设置)”的部分，都使用了 Single Threaded Execution Pattern。一连串的测试、设置操作，都必须保证只有单线程可以执行。因为在测试后、设置前时，不能让其他线程插进来测试。

Balking Pattern（第 4 章）

当警戒条件不成立时，Guarded Suspension Pattern 中，线程会等到警戒条件成立为止。

而 Balking Pattern 中，线程不会等待警戒条件，而会直接退出。

Producer-Consumer Pattern（第 5 章）

Producer-Consumer Pattern 中，Producer 参与者放置数据时，与 Consumer 参与者获取数据时，有用到 Guarded Suspension Pattern。

Future Pattern（第 9 章）

Future Pattern 中，若要获取目的信息时，而目的信息还没准备好时，会使用 Guarded Suspension Pattern 进入等待。

重点回顾

这一章我们学习了 Guarded Suspension Pattern。

Guarded Suspension Pattern 中，有一个具有状态的对象。而这个对象只有在自己的状态合适的时候，才会让线程进行目的处理。

于是，首先我们以“警戒条件”来表示对象的合适状态。并在目的处理前，测试现在使用满足警戒条件。只有在警戒条件满足时，才会进行目的处理；而警戒条件不成立时，就会等待到成立为止。

Java 语言中使用 while 语句来测试条件，使用 wait 方法来等待。而当条件更改后，使用 notify/notifyAll 方法发出通知。

这就是 Guarded Suspension Pattern。
那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 3-1（基础知识的确认）

Java 关于范例程序的操作，请仔细阅读下列语句，正确的请回答○，不正确的请回答×。

（1）getRequest 与 putRequest 分别由不同的线程来调用。
（2）RequestQueue 的实例可以做出两个来。
（3）从 getRequest 当中调用 removeFirst 方法时，queue.size() > 0 的值一定是 true。
（4）从 getRequest 当中调用 wait 方法时，queue.size() > 0 的值一定为 false。
（5）ClientThread 的线程在执行 putRequest 时，ServerThread 的线程不会进行任何处理。
（6）从 getRequest 调用出 wait 方法的线程，会释放 lock，进入 queue 的 waitset。
（7）putRequest 方法中的 notifyAll();这个语句跟写成 queue.notifyAll();的意思一样，只是后者比较冗长。

问题 3-2（notifyAll 的位置）

Java 假设范例程序中，RequestQueue 类（List 3-2）的 putRequest 方法改如 List 3-6 所示（在执行 addList 之前就执行 notifyAll）。请问这时 RequestQueue 类可以安全执行吗？为什么呢？

List 3-6　先执行 notifyAll 的 putRequest 可以安全执行吗

```
public synchronized void putRequest(Request request) {
    notifyAll();
    queue.addLast(request);
}
```

问题 3-3（加上调试输出）

Java 范例程序的执行范例（见图 3-2）中，的确 ServerThread（List 3-4）会依次处理 ClientThread（List 3-3）所送出的请求，可是从程序结果上，我们看不出来 wait 方法与 notifyAll 方法是否有如同我们所期待的被调用。
请在 RequestQueue 类加上调试输出，使我们可以一目了然程序是否有照我们的期待运行。

问题 3-4（似是而非的 Guarded Suspension Pattern）

Java RequestQueue 类（List 3-2）的 getRequest 方法若分别写成下面（1）～（4）的版本，各会出现什么问题。

（1）把 while 写成 if（List 3-7）

（2）synchronized 的范围只有 wait（List 3-8）

（3）try...catch 放在 while 的外面（List 3-9）

（4）以 Thread.sleep 取代 wait（List 3-10）

List 3-7　把 while 写成 if 会出问题吗

```
public synchronized Request getRequest() {
    if (queue.size() <= 0) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    return (Request)queue.removeFirst();
}
```

List 3-8　synchronized 的范围只有 wait 会出问题吗

```
public Request getRequest() {
    while (queue.size() <= 0) {
        try {
         synchronized (this) {
               wait();
            }
        } catch (InterruptedException e) {
        }
    }
    return (Request)queue.removeFirst();
}
```

List 3-9　把 try...catch 放在 while 的外面会出问题吗

```
public synchronized Request getRequest() {
    try {
        while (queue.size() <= 0) {
            wait();
```

```
        }
    } catch (InterruptedException e) {
    }
    return (Request)queue.removeFirst();
}
```

List 3-10 以 Thread.sleep 取代 wait 会出问题吗

```
public synchronized Request getRequest() {
    while (queue.size() <= 0) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
    return (Request)queue.removeFirst();
}
```

问题 3-5（两个 Guarded Suspension）

Java 有一个人想以范例程序为基础，来写“以交谈方式互相传送请求的两个线程”。

TalkThread 类（List 3-11）拥有两个 RequestQueue（List 3-2）的实例。一个是输入用（input），另一个是输出用（output）。TalkThread 类首先会从输入用的 RequestQueue，以 getRequest 方法获取一个请求（request1）。

接着，在 request1 名称的后面加上一个惊叹号（!），建立出新的请求（request2）。并使用输出用 RequestQueue 的 putRequest 方法，送出这个请求给交谈对象的线程。

建立两个上述 TalkThread 的实例，并共享两个输出、输入用的 RequestQueue 实例（互相颠倒）。这个人想着这只程序执行时，惊叹号应该持续增加的请求，应该会不断地持续交换着。于是，最后写出了 Main 类如 List 3-12 所示。

但实例执行程序时，执行结果如图 3-5 所示。Alice 与 Bobby 两个线程确实启动了，但什么都没有显示。这是为什么呢？再问，要怎样改写，才能达到我们所期待的请求交换呢？

List 3-11 交谈的线程（TalkThread.java）

```
public class TalkThread extends Thread {
    private RequestQueue input;
    private RequestQueue output;
    public TalkThread(RequestQueue input, RequestQueue output,
```

```
String name) {
        super(name);
        this.input = input;
        this.output = output;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName()    +
":BEGIN");
        for (int i = 0; i < 20; i++) {
            // 获取对方的请求
            Request request1 = input.getRequest();
            System.out.println(Thread.currentThread().getName()
+ " gets  " + request1);

            // 加上一个惊叹号(!)，传给对方
            Request request2 = new Request(request1.getName() + "!");
            System.out.println(Thread.currentThread().getName()
+ " puts  " + request2);
            output.putRequest(request2);
        }
        System.out.println(Thread.currentThread().getName()    +
":END");
    }
}
```

List 3-12 想让 Alice 与 Bobby 交谈，但不能正常执行的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        RequestQueue requestQueue1 = new RequestQueue();
        RequestQueue requestQueue2 = new RequestQueue();
        new  TalkThread(requestQueue1,  requestQueue2,  "Alice").
start();
        new  TalkThread(requestQueue2,  requestQueue1,  "Bobby").
start();
    }
}
```

```
Alice:BEGIN
Bobby:BEGIN
(停在这里。CTRL+C 结束程序)
```

图 3-5　执行结果

问题 3-6（线程的取消）

Java 这个问题使用了 interrupt 方法。读者可以看完第 5 章的说明后再来解答这个问题。

这一章所介绍的范例程序，若不使用 Ctrl+C 强制结束，到程序执行完毕需要花上很长的时间。所以我们把 Main 方法改写成 List 1-13 的样子。这个 Main 方法在经过约 10 秒后，就会调用 ClientThread（List 3-3）与 ServerThread（List 3-4）的 interrupt 方法。可是，这个更改的操作还是无法使程序结束（见图 3-6）。请尝试不更改这个 Main 类，而试着修改其他类，让程序在约 10 秒后能够真的结束。

List 3-13　约 10 秒后会调用 interrupt 方法的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        // 启动线程
        RequestQueue requestQueue = new RequestQueue();
        Thread alice = new ClientThread(requestQueue, "Alice",
314159L);
        Thread bobby = new ServerThread(requestQueue, "Bobby",
265358L);
        alice.start();
        bobby.start();

        // 等待约 10 秒
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
        }

        // 调用 interrupt 方法
        System.out.println("***** calling interrupt *****");
        alice.interrupt();
        bobby.interrupt();
    }
}
```

```
Alice requests  [ Request No.15 ]
Bobby handles [ Request No.15 ]
Alice requests  [ Request No.16 ]
Bobby handles  [ Request No.16 ]
Alice requests [ Request No.17 ]
Bobby handles  [ Request No.17 ]
***** calling interrupt *****          ← 调用 interrupt 方法
Alice requests  [ Request No.18 ]      ← 可是线程不停下来
Bobby handles  [ Request No.18 ]
Alice requests [ Request No.19 ]
Bobby handles [ Request No.19 ]
Alice requests [ Request No.20 ]
(以下省略。CTRL+C 结束程序)
```

图 3-6　执行范例

第4章

Balking——不需要的话，就算了吧



Balking Pattern

我现在正在餐厅里，考虑要吃什么东西。

因为已经决定要吃什么了，所以我举手表明我想要点餐。注意到我举起手的服务生，就走向我这边。

这时，有另一位服务生也注意到我了。不过，她看到已经有一个人靠近我的座位，所以就不过来了……。

这一章要学习的是 Balking Pattern。

当现在不适合进行这个操作，或是没有必要进行这个操作时，就直接放弃进行这个操作而回去，这就是 Balking Pattern。

“balk”就是“退缩不前”的意思。棒球的“投手犯规；仰投”也是 balk 这个字。当跑垒员在垒时，投手站在投手板但“中途放弃”投球的动作，这种犯规行为，就称为 balk。

Balking Pattern 与 Guarded Suspension Pattern（第 3 章）一样，需要警戒条件。在 Balking Pattern 中，当警戒条件不成立时，就马上中断。而 Guarded Suspension Pattern 则是等待到可以执行时再去执行相应的操作。

范例程序

那接下来就来阅读使用 Balking Pattern 的简单程序吧。在这里要写的程序，是定期将现在数据的属性写入到文件里。

保存数据属性时，之前所保存的属性将会被覆盖，而只有最新的属性会被存下来。

另外，若现在的数据与上次保存的属性并无不同时，此时写入的操作就是多余了，所以就不保存了。也就是以“数据的属性有所不同”作为警戒条件，当没有不同时就放弃进行写入的操作（balk）。

请想象成文本处理软件的“自动保存功能”。所谓自动保存功能，是为了预防计算机忽然断电，或是软件忽然出错的危险，定期将数据保存到文件里的功能。这一章所介绍的范例程序，可说是自动保存功能的超级简化版。

表 4-1 是程序所使用到的类。

Data 类与 SaverThread 分别就像文本处理软件的文章与进行自动保存操作的线程。而 ChangerThread 类则是模仿一边修改文章，当满意的时候就执行存盘的用户。

表 4-1　类一览表

名　　称	说　　明
Data	表示可修改、可保存的数据的类
SaverThread	用来定期保存数据属性的类
ChangerThread	用来修改、保存数据属性的类
Main	操作测试用的类

Data 类

Data 类（List 4-1）代表现在的数据。

filename 字段是保存的文件名称。

content 字段是代表数据内容的字符串。此字符串就是写入文件的内容。

changed 字段则是表示自从上次保存之后，content 字段是否有修改过。当 changed 的值为 true 就是数据有修改过，而 changed 为 false 时，则是数据没有变动。配合 changed 值是 true/false，来决定要/不要对文件进行写入。像这种用来表示两种状态的字段或变量，我们一般称之为标识（flag）。在这里，changed 字段就是标识，我们的警戒条件就是这个标识必须为 true 值。

Data 类中，拥有用来修改数据的方法（change），以及用来要求保存数据的方法（save）。另外，还有实际用来保存文件时使用的方法（doSave）。

change 方法会将新的数据属性存入到 content 字段里。而将 changed 标识的值设置为 true，表示数据的更改。

save 方法会检查 changed 标识的值。当 changed 标识的值是 false 时，就表示属性没有更改，所以就马上 return（也就是不进行实际的保存操作）。这个 return 就相当于“balk”的操作。若 changed 标识是 true 时，就调用 doSave 方法将数据写入文件里。并将 changed 标识的值改为 false，表示 content 字段的属性与文件的内容一致了。

change 与 save 两者都是 synchronized 方法。我们有说过，当看到 synchronized 的时候，就要思考“synchronized 是在保护什么？”。这个 synchronized 所保护的，是 content 字段与 changed 字段。

在 doSave 方法中，使用 java.io.FileWriter 类，将 content 的内容写入文件里。若写入文件发生任何错误时，FileWriter 类的构造器、write 方法、close 方法都有可能抛出 java.io.IOException 异常。当这个异常抛出来时，程序会跳离 doSave 方法与 save 方法。这时，调用 doSave 的下一行语句（将 false 指定给 changed 标识）并不会执行到。changed 标识的值仍然会是 true，表现出数据内容与文件内容不相同的状态。

List 4-1 表示现在数据的类（Data.java）

```
import java.io.IOException;
import java.io.FileWriter;
import java.io.Writer;

public class Data {
    private final String filename; // 保存时的文件名
    private String content;        // 数据的内容
    private boolean changed;    // 修改后的内容还没保存的话，值为 true

    public Data(String filename, String content) {
        this.filename = filename;
        this.content = content;
        this.changed = true;
    }

    // 修改数据内容
    public synchronized void change(String newContent) {
        content = newContent;
        changed = true;
    }

    // 若数据有修改，就保存到文件里
    public synchronized void save() throws IOException {
        if (!changed) {
            return;
        }
        doSave();
        changed = false;
    }

    // 实际将数据保存到文件里用的方法
    private void doSave() throws IOException {
        System.out.println(Thread.currentThread().getName() + "
calls doSave, content = " + content);
        Writer writer = new FileWriter(filename);
        writer.write(content);
        writer.close();
    }
}
```

SaverThread 类

SaverThread 类（List 4-2）是用来定期要求保存数据的类。

在 run 方法中，以约 1000ms（约 1s）的间隔调用 Data 实例（data）的 save 方法。当文件的操作出现异常，抛出 java.io.IOException 异常时，这个线程将会停止执行。

List 4-2　用来定期保存数据的 SaverThread 类（SaverThread.java）

```
import java.io.IOException;

public class SaverThread extends Thread {
    private Data data;
    public SaverThread(String name, Data data) {
        super(name);
        this.data = data;
    }
    public void run() {
        try {
            while (true) {
                data.save();                    // 保存数据
                Thread.sleep(1000);             // 休息约 1 秒
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ChangerThread 类

ChangerThread 类（List 4-3）则是用来修改数据、保存文件的类。

启动后的线程会以"No.0"、"No.1"、"No.2"、……的字符串作为参数，反复调用 change 方法修改数据。之后，这个线程会“去做别的工作”（实际上是以 sleep 方法暂停随机数时间模拟而已）。每一次要更改数据的时候，都会调用 save 方法来保存数据。这是仿真“试图将刚才修改的数据反映在文件上”的操作。

注意：Data 类的 doSave 方法，每次都会重新建立文件。当新建文件时，文件的属性会瞬间消失，所以这个范例程序不能直接沿用到应用程序的自动保存功能上。实例实现自动保存功能时，必须先将文件备份保存起来。

List 4-3 用来修改、保存数据的 ChangerThread 类（ChangerThread.java）

```
import java.io.IOException;
import java.util.Random;

public class ChangerThread extends Thread {
    private Data data;
    private Random random = new Random();
    public ChangerThread(String name, Data data) {
        super(name);
        this.data = data;
    }
    public void run() {
        try {
            for (int i = 0; true; i++) {
                data.change("No." + i);                  //修改数据
                Thread.sleep(random.nextInt(1000));      //模拟去做别的事
                data.save();                             //明确地要求存档
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Main 类

Main（List 4-4）中，首先会建立 Data 的实例。接着将这个实例交给 ChangerThread 与 SaverThread 共享，并启动这两个线程。

执行范例如图 4-1 所示。有时候是 SaverThread 调用 doSave 方法，也有时候是 ChangerThread 在调用 doSave 方法（两者都必须通过 save 方法来调用）。

无论是哪一个线程调用的，请注意 content 字段的属性都没有“重复保存”（没有重复的号码）。因为当 content 的内容相同时，线程就会 balk 住，而不会调用到 doSave 方法。

List 4-4 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Data data = new Data("data.txt", "(empty)");
        new ChangerThread("ChangerThread", data).start();
        new SaverThread("SaverThread", data).start();
    }
}
```

```
SaverThread calls doSave, content = No.0
SaverThread calls doSave, content = No.1
ChangerThread calls doSave, content = No.2
SaverThread calls doSave, content = No.3
SaverThread calls doSave, content = No.4
ChangerThread calls doSave, content = No.5
SaverThread calls doSave, content = No.6
ChangerThread calls doSave, content = No.7
ChangerThread calls doSave, content = No.8
SaverThread calls doSave, content = No.9
ChangerThread calls doSave, content = No.10
ChangerThread calls doSave, content = No.11
(以下省略。CTRL+C 结束程序)
```

没有重复的号码

图 4-1　执行范例

范例程序的顺序图，如图 4-2 所示。

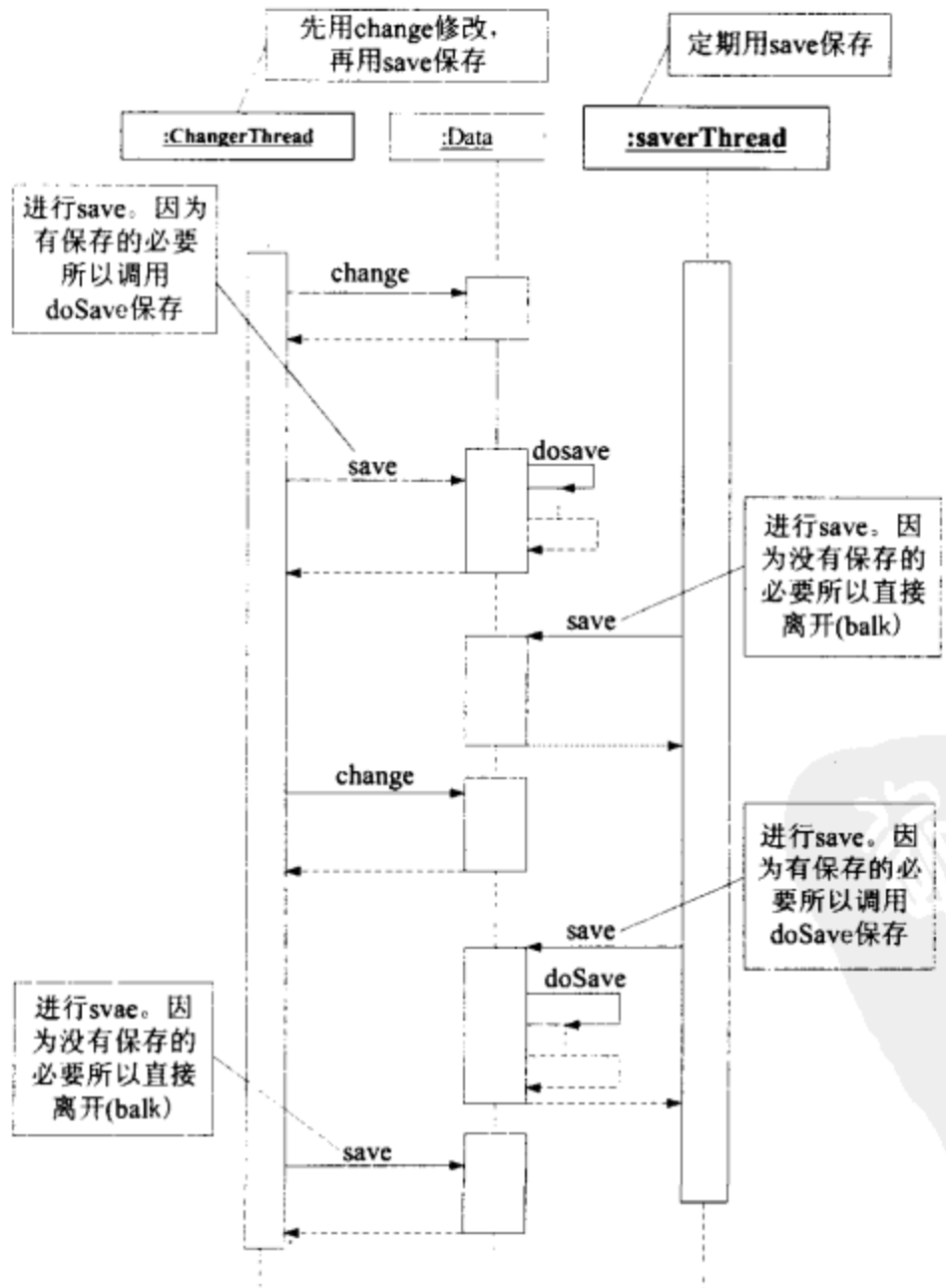


图 4-2　范例程序的顺序图

Balking Pattern 的所有参与者

Balking Pattern 的所有参与者如下：

◆ *GuardedObject（被警戒的对象）参与者*

GuardedObject 参与者是一个拥有被警戒的方法（guardedMethod）的类。当线程执行 guardMethod 时，只要满足警戒条件，就会执行实例的操作。但警戒条件不成立时，就不执行实例操作，而直接退出。警戒条件的成立与否，会随 GuardedObject 参与者的状态变化。

GuardedObject 参与者除了 guardedMethod 以外，可能还会有用来更改实例状态的方法（stateChangingMethod）。

在范例程序中，Data 类就是 GuardedObject 参与者。save 方法与 change 方法则分别是 guardedMethod 与 stateChangingMethod。

范例程序的警戒条件则是“changed 字段的值为 true”。

类图如图 4-3 所示，而时序图如图 4-4 所示。请与 Guarded Suspension Pattern（第 3 章）的时序图（图 3-4）比较。

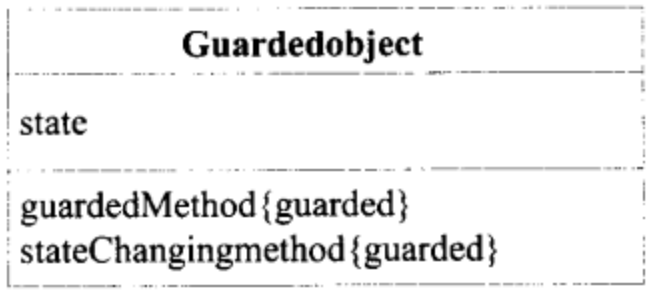


图 4-3　Balking Pattern 的类图

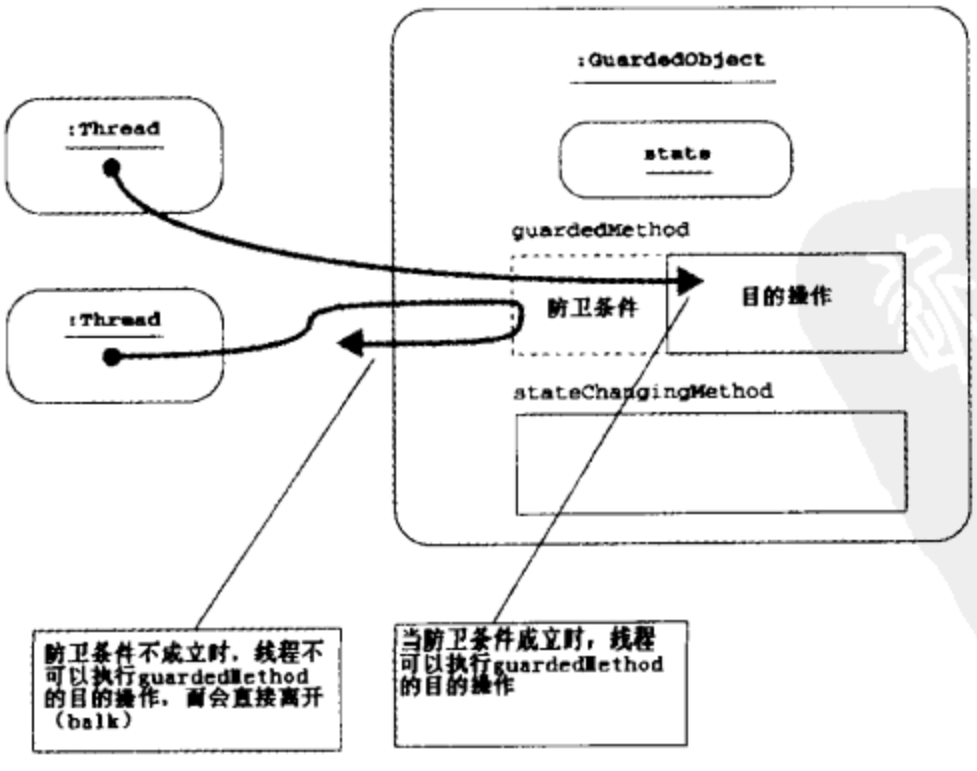


图 4-4　Balking Pattern 的时序图

扩展思考方向的提示

何时使用（适用性）

Balking Pattern 该在什么时机使用呢？ 也就是说，什么情况下，我们会想在操作前 balk 呢？

◆ 不需要刻意去执行的时候

范例程序中，当 content 字段的内容没有改变，就在 save 方法里 balk 住了。

balk 的原因是 content 的内容已经写在文件里了，没有必要大费工夫再去写一次。

像这样“不需要刻意去执行”的时候，就可以使用 Balking Pattern。在这里 balk 的原因，是为了提高程序的执行性能。

◆ 不想等待警戒条件成立时

Balking Pattern 的特征是“不等待”。当警戒条件不成立时，就直接退出，马上进入下一个工作。这种情况就适合使用 Balking Pattern。这是提高程序的响应性。

◆ 警戒条件只有第一次成立时

当“警戒条件只有第一次成立时”，也可以使用 Balking Pattern。

例如 List 4-5 的 Something 类。在这里使用 initialized 字段来表示是否已经初始化。调用 init 方法时，会先去检查 initialized 字段。

当 initialized 字段的值为 true 时，表示已经初始化了，所以就直接 return 退出（就是 balk；设计成 throw 出异常也可以）。

当 initialized 字段的值为 false 时，以 doInit 方法进行实际的初始化操作，接下来就将 initialized 字段设置为 true，记录初始化操作已经处理完毕了（这相当于没有 stateChangingMethod 方法的情况）。

List 4-5 只进行一次初始化操作的类（Something.java）

```
public class Something {
    private boolean initialized = false;
    public synchronized void init() {
        if (initialized) {
            return;
        }
        doInit();
```

```
        initialized = true;
    }
    private void doInit() {
        // 实际的初始化操作
    }
}
```

程序收录在下载文件夹 Balking/Others/Init1 内

List 4-5 中，有使用 if 条件式来检查警戒条件。

```
if (initialized) {
    return;
}
```

在这种地方，如果使用 Guarded Suspension Pattern，当然是错误的。

```
while (initialized) {
    wait();
}
```

因为 initialized 字段“只要变成 true 以后，就不会再变成 false 了”。也就是说，这个类“警戒条件不成立了以后，就绝对不会再次成立了”。

像这种只有第一次执行时警戒条件会成立的例子，经常出现在实例的初始化、终结处理等“无法执行两次以上的操作”中。练习问题中，也有（偷偷地）出现类似的问题，请等一下试着找看看。

像 Something 类的 initialized 字段这种“状态只会变化一次的变量”，我们有时候称为 latch（门闩）。这个门闩，只要锁上就不会再打开了。

表达 balk 结果的方式

在 guardedMethod 方法里因为 balk 而退出时，我们可用下面几种方式来表示 balk 的结果。

◆ 忽略 *balk* 的发生

最简单的方式，就是不通知调用端 balk 的发生。在范例程序中就是采用这种方式。

◆ 以返回值表达 *balk* 的发生

以 boolean 类型的返回值表示 balk 的发生。例如返回 ture 就是没有 balk，而有正常处理目的操作；返回 false 则表示发生 balk，目的操作没有进行。

若是返回参考类型的方法，也可以返回 null 表示 balk 的发生。

◆ 以异常表达 balk 的发生

也可以使用异常表达 balk 的发生。这时，balk 时就不是从方法 return 出来，而是从方法 throw 出异常。本章末的练习问题 4-5 就是这么做的。

相关 Pattern

Guarded Suspension Pattern（第 3 章）

Guarded Suspension Pattern 在 guardedMethod 的警戒条件不成立时，线程会等待。而 Balking Pattern 在警戒条件不成立时，线程会直接退出。

Observer Pattern（参考附录 E [GoF] [Yuki01]）

在多线程的环境下使用 Observer Pattern 时，可能会用到 Balking Pattern。当 Subject 参与者要通知 Observer 参与者状态的变化时，若 Observer 参与者的状态无法处理这个通知，通知处理会 balk 掉。

进阶说明：timeout

这一章的进阶说明要谈的是 timeout。

Balking Pattern 与 Guarded Suspension Pattern 的中间

这一章所学的 Balking Pattern 中，当警戒条件不成立时，会直接 balk 退出。

而第 3 章所学过的 Guarded Suspension Pattern，当警戒条件不成立时，会等待到成立为止。

在“balk 退出”与“等待到警戒条件成立为止”两种极端的处理方式之间，还有一种折衷的做法——“在条件成立为止之前，等待一定的时间”。等待一定的时间，看看警戒条件会不会成立，如果还是不成立的话，就 balk 退出。这种方式称为 guarded timed，或是简单地称为 timeout。

wait 的结束是什么时候

若调用 Java 的 wait 方法时加上参数，可以指定 timeout 的时间。例如，下面的语句，就是在实例 obj 上，调用 wait 方法，timeout 的时间是约 1000ms（约 1s）。

```
obj.wait(1000);     // 指定 timeout 时间为约 1000ms
```

执行这个语句，线程就会进入 obj 的等待区里暂停，并释放 obj 的锁定。只有发生下面的情况，才会从等待区里退出。

◆ 当 *notify* 方法执行时

对 obj 调用 notify 方法，线程被唤醒时。

但等待区里若有多条线程时，只有一条线程会被唤醒。哪一条线程会被唤醒，Java 规格里没有明确定义。

◆ 当 *notifyAll* 方法执行时

对 obj 调用 notifyAll 方法时。

notifyAll 会把实例的等待区里所有的线程全部唤醒。无论是 notify 还是 notifyAll，线程被唤醒以后，还是要重新获取 obj 的锁定。

◆ 当 *interrupt* 方法执行时

对这条线程调用 interrupt 方法时。

被 interrupt 时，等待区里的线程会（与被调用 notify、notifyAll 时一样）重新获取 obj 的锁定，并抛出 InterruptedException 异常。

notify、notifyAll 两个方法是对实例调用的，而 interrupt 则是对线程调用的。关于中断，将在后面说明。

◆ 发生 *timeout* 时

当 wait 方法参数中设置的 timeout 时间（上面的例子是约 1000ms）已经到了的时候。与 notify、notifyAll 的时候一样，需要重新获取 obj 的锁定。

那么，看完上面的说明后，或许读者已经察觉这一件事——我们没有办法识别 wait 方法是被 notify、notifyAll，还是自己 timeout 了。所以要实现 guarded timed，程序代码会需要稍微复杂一点，下一段中将详细解说。

Allen Holub 的著作（参考附录 E [Holub00]）的最后一章，下了“If I Were King（如果我是神）”这样的标题，举出了一些 Java 语言多线程的问题（标题的意思是说“如果我是神的话，我实在想要改掉 Java 的这些问题”）。其中之一，就是“wait 的 timeout 无法识别”的问题。

guarded timed 的实现

那么我们就来尝试实现 guarded timed 的程序吧。首先，我们先定义 timeout 用的异常 TimeoutException（List 4-6）。在这里，我们将 TimeoutException 声明成

InterruptedException 的子类（参考附录 E [Lea]）。也就是我们将 timeout 视为取消的一种。

List 4-6 TimeoutException 类（TimeoutException.java）

```
public class TimeoutException extends InterruptedException {
    public TimeoutException(String msg) {
        super(msg);
    }
}
```

程序收录在下载文件夹 Balking/Others/Timeout 内

Host 类（List 4-7）拥有更改状态用的 setExecutable 方法与执行用的 execute 方法。execute 的警戒条件是“当 ready 字段为 ture”。execute 方法的 timeout 时间则是 timeout 字段（单位为 ms）。timeout 的时候会 throw 出 TimeoutException。

当 wait 的参数传入 0 的时候，就表示没有 timeout 时间（timeout 的时间为无限长），wait 的参数如果传入负数的话，会抛出 IllegalArgumentException（参数不合法的异常），写程序时请务必小心不要出现这些情况（Host 类中，若 timeout 的值为负数，都当作和 0 一样来处理）。

List 4-7 具有 timeout 的 Host 类（Host.java）

```
public class Host {
    private final long timeout;     //timeout 值
    private boolean ready = false; //如果可以执行方法的话则为 true

    public Host(long timeout) {
        this.timeout = timeout;
    }

    // 更改状态
    public synchronized void setExecutable(boolean on) {
        ready = on;
        notifyAll();
    }

    // 评断状态后执行它
    public synchronized void execute() throws InterruptedException,
TimeoutException {
        long start = System.currentTimeMillis();   // 开始时刻
```

```
        while (!ready) {
            long now = System.currentTimeMillis();  //现在时刻
            long rest = timeout - (now - start);    //剩下的等待时间
            if (rest <= 0) {
                throw new TimeoutException("now - start = " + (now
- start) + ", timeout = " + timeout);
            }
            wait(rest);
        }
        doExecute();
    }

    // 实际的处理操作
    private void doExecute() {
        System.out.println(Thread.currentThread().getName() + "
calls doExecute");
    }
}
```

程序收录在下载文件夹 Balking/Others/Timeout 内

Main 类（List 4-8）中，设置 timeout 为约 10000ms（约 10s），建立 Host 类的实例。调用 Host 的 execute 方法，而故意不去调用 setExecutable(true)的话，就会像图 4-5 这样发生 timeout。这个范例中，出现了 5ms 左右的误差。

List 4-8 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Host host = new Host(10000);
        try {
            System.out.println("execute BEGIN");
            host.execute();
        } catch (TimeoutException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

程序收录在下载文件夹 Balking/Others/Timeout 内

```
execute BEGIN                                          ← 马上显示
TimeoutException: now - start = 10005, timeout = 10000
                                                       ← 约 10 秒后显示
        at Host.execute(Host.java:22)
        at Main.main(Main.java:6)
```

图 4-5　执行范例

synchronized 没有 timeout，也不能中断

来思考下面两种线程的状态（关于线程的状态移转过程请参考图 I 1-22）。

（1）synchronized 处想要获取锁定，但被阻挡住的状态。

（2）执行 wait 而进入等待区的状态。

这两种状态，线程都是暂时停住的，可是也有所不同。

（1）在 synchronized 处想要获取锁定，但被阻挡住的状态

无法使状态（1）中的线程 timeout。这是因为无论是 synchronized 方法或 synchronized 块，都没有办法设置 timeout 的值。

另外，即使对状态（1）中的线程调用 interrupt，也没办法抛出 InterruptedException 异常。而必须在获取锁定，进入 synchronized 了以后，调用 wait、sleep、join 等会意识到现在是不是中断状态的方法，或是使用 isInterrupted 方法或 interrupted 方法自己检查是不是中断状态，自己 throw 出去（参考附录 E [Holu00]）。

另外，附录 E 的[Venners99]中，将状态（1）的线程称为“停留在入口区（entry set）”。这是 Java 语言规格里没有出现的名词，不过可用来与等待区比较，是可帮助理解的概念。

（2）执行 wait 而进入等待区的状态

将状态（2）中的线程 timeout 的方法，就是本章进阶说明“guarded timed 的实现”中将要介绍。

另外，对状态（2）中的线程调用 interrupt 的话，会抛出 InterruptedException 异常。

重点回顾

这一章我们学习了 Balking Pattern。

Balkingn Pattern 中，有一个具有状态的对象。而这个对象只想在自己的状态合适时，才进行线程目的操作；当状态不合适时，就不进行目的操作了。

于是，首先我们以“警戒条件”来表示对象的合适状态。并在目的操作前，测

试现在使用满足警戒条件。只有在警戒条件满足时，才会进行目的操作；而警戒条件不成立时，就会直接从方法里退出。

Java 语言中，使用 if 语句来测试警戒条件。balk 的时候，可以使用 return 方法退出，或是使用 throw 抛出异常。警戒条件的测试，要使用 synchronized 放进临界区间里。

这就是 Balking Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 4-1（基础知识的确认）

Java 下列语句中，正确的请回答○，不正确的请回答×。

（1）只有 SaveThread 能调用出 save 方法。

（2）将 synchronized 全部删除，将 changed 字段改成 volatile，还是可以执行相同的操作。

（3）change 方法的话，一定要先指定到 content 字段，再将 changed 设为 true。

（4）doSave 方法不是 synchronized 方法。但是，由于 doSave 方法只能从 synchronized 方法来调用，因此来到 doSave 方法的线程一定有 this 的 lock。

（5）doSave 方法不会同时被两个线程所调用。

问题 4-2（范例程序操作的认知 1）

Java 请看范例程序的执行范例（图 4-1），因为调用 doSave 方法的时候，content 字段的值都没有重复，由此可以得知没有多余的文件写入操作。可是由执行结果无法清楚的看到 balk 的操作（不调用 doSave 而 return 退出的样子）。

（1）请在程序加上调试输出，使 balk 的操作可以明显得知。

（2）请故意删除 save 方法的 synchronized，测试是否会出现多余的文件操作。为了使多余的文件操作更容易发生，加入 sleep 也无妨。

问题 4-3（范例程序操作的认知 2）

Java 范例程序的操作中，changed 字段什么时候会是 true 呢？ 请在范例程序的程序图（图 4-2）中，标示出 changed 字段为 true 的范围。

问题 4-4（使用 timeout 检查死锁）

Java 请尝试改写 *Guarded Suspension Pattern*（第 3 章）的范例程序中出现的 RequestQueue 类（List 4-10），让它可以检查死锁的发生。当经过 30 秒还没有满足警戒条件时，请让它抛出 LivenessException（List 4-9）。

注意：严格来说这不是为了检查出死锁。只不过以长时间无法达到警戒条件，断定程序缺乏生命性而已。

▶▶ 补充说明：RuntimeException 与 Exception 使用上的区别

List 4-9 中，LivenessException 声明成 java.lang.RuntimeException 的子类。这是因为缺乏生命性的状态，是在执行中发现，且是无法恢复的状态。

若是进行某些处理，而发生可以修复的异常状态，我们会使用 java.lang.Exception 子类的异常来表现。而解决异常状态的程序代码则写在 catch 块里。

但异常状态无法解决时，就会声明成 java.lang.RuntimeException 的子类。通常是程序员的失误。

关于异常的设计，参见附录 E [Bloch]的第 40 项。

List 4-9 LivenessException 类（LivenessException.java）

```
public class LivenessException extends RuntimeException {
    public LivenessException(String msg) {
        super(msg);
    }
}
```

List 4-10 第 3 章的 RequestQueue 类（RequestQueue.java）

```
import java.util.LinkedList;

public class RequestQueue {
    private final LinkedList queue = new LinkedList();
    public synchronized Request getRequest() {
        while (queue.size() <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return (Request)queue.removeFirst();
    }
    public synchronized void putRequest(Request request) {
        queue.addLast(request);
        notifyAll();
```

```
    }
}
```

问题 4-5（异常 IllegalThreadException）

Java 有个人正在练习线程的用法。

他建立了一个简单的 TestThread 类（List 4-11），并从 Main 类（List 4-12）调用它。

因为 Main 类的 while 是无限循环，所以推测这支程序应该会不停地输出

```
BEGIN..................................................END
```

无限次，然而程序却只有显示了一次，就抛出 IllegalThreadStateException（表示线程状态异常的异常）（见图 4-6）。

请问，这是为什么？

List 4-11 TestThread 类（TestThread.java）

```
public class TestThread extends Thread {
    public void run() {
        System.out.print("BEGIN");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("END");
    }
}
```

List 4-12 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new TestThread();
        while (true) {
            thread.start();
        }
    }
}
```

```
Exception in thread "main" java.lang.IllegalThreadStateException
←抛出异常
        at java.lang.Thread.start(Native Method)
        at Main.main(Main.java:5)
BEGIN.....................................END  ←但是只有显示一次而已
```

图 4-6　执行范例

第5章

Producer–Consumer——我来做，你来用



Producer-Consumer Pattern

这一章要学习的是 Producer-Consumer Pattern。

producer 是“生产者”的意思，是指产生数据的线程。而 consumer 是“消费者”的意思，意指使用数据的线程。

生产者必须将数据安全地交给消费者。虽然只是这样的问题，但当生产者与消费者在不同线程上运行时，两者的处理速度差将是最大的问题。当消费者要取数据时生产者还没建立出数据，或是生产者建立出数据时消费者的状态还没办法接收数据等等。

Producer-Consumer Pattern 是在生产者与消费者之间加入一个“桥梁参与者”。以这个桥梁参与者缓冲线程之间的处理速度差。

一般来说可能存在多个生产者与多个消费者，不过也有可能生产者与消费者都只有一个。当两方都只有一个时，我们也称之为 Pipe Pattern（参考附录 E [Grand]）。

范例程序

为了能了解 Producer-Consumer Pattern，就先来看范例程序吧。这个范例程序中，有 3 名厨师会一直做蛋糕放在桌上，而有 3 名客人会不停地吃。程序的行为如下所示：

- 厨师（MakerThread）会做蛋糕（String），放在桌子（Table）上。
- 桌上顶多可以放 3 个蛋糕。
- 桌上已经放满 3 个蛋糕，而厨师还要放上蛋糕的话，会等待到桌子上出现空间为止。
- 客人（EaterThread）会拿桌上的蛋糕去吃。
- 客人会依次取桌上的蛋糕。
- 桌上没有蛋糕时，客人又要拿蛋糕的话，会等待到蛋糕放到桌上为止。

类一览如表 5-1 所示。

表 5-1　类一览表

名　　称	说　　明
Main	操作测试用的类
MakerThread	表示厨师的类
EaterThread	表示客人的类
Table	表示桌子的类

Main 类

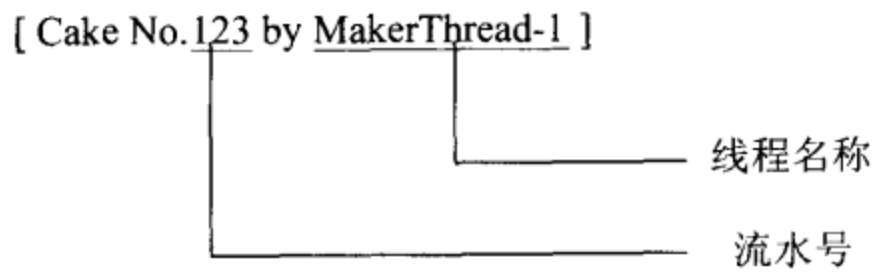
Main 类（List 5-1）会建立出桌子，并启动代表厨师、客人的线程。传进 MakerThread 与 EaterThread 构造器的谜样数字，只是拿来当作随机数种子，数值本身没有特别的意义。

List 5-1 驱动厨师与客人的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);    // 建立可以放置 3 个蛋糕的桌子
        new MakerThread("MakerThread-1", table, 31415).start();
        new MakerThread("MakerThread-2", table, 92653).start();
        new MakerThread("MakerThread-3", table, 58979).start();
        new EaterThread("EaterThread-1", table, 32384).start();
        new EaterThread("EaterThread-2", table, 62643).start();
        new EaterThread("EaterThread-3", table, 38327).start();
    }
}
```

MakerThread 类

MakerThread 类（List 5-2）是用来制作蛋糕并放到桌上的类，就是厨师。为了使程序简便起见，以蛋糕的“流水号”和制作蛋糕的“线程名称”组成的字符串来表示。



为了使程序结果容易理解，蛋糕的流水号是所有厨师共通的。于是在这里将流水号（id）声明成类字段。

建立出的蛋糕使用 Table 类的 put 方法放到桌子上。接下来会随机暂停一段时间（0ms 以上，未满约 1000ms）。这个暂停的操作是模拟“烘烤蛋糕所耗费的时间”。

MakerThread 类会一直不断持续“制作蛋糕→放到桌子上”的操作，是蛋糕的生产者。

List 5-2 建立蛋糕放在桌子上的 MakerThread 类（MakerThread.java）

```
import java.util.Random;

public class MakerThread extends Thread {
    private final Random random;
    private final Table table;
    private static int id = 0; // 蛋糕的流水号（所有厨师相同）
    public MakerThread(String name, Table table, long seed) {
        super(name);
        this.table = table;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            while (true) {
                Thread.sleep(random.nextInt(1000));
                String cake = "[Cake No."+nextId() + "by "+getName()+"";
                table.put(cake);
            }
        } catch (InterruptedException e) {
        }
    }
    private static synchronized int nextId() {
        return id++;
    }
}
```

EaterThread 类

EaterThread 类（List 5-3）用来表示从桌上拿蛋糕来吃的客人（虽说是吃，但在程序里是用 sleep 代替就是了）。

桌上的蛋糕可以使用 Table 类的 take 方法获取。接着与 MakerThread 一样，随机暂停一段时间。这个暂停的操作是模拟“吃蛋糕所花的时间”。

EaterThread 会不断持续“从桌上拿蛋糕→吃掉”的操作，是蛋糕的消费者。

List 5-3 从桌上拿蛋糕去吃的 EaterThread 类（EaterThread.java）

```
import java.util.Random;

public class EaterThread extends Thread {
    private final Random random;
```

```
    private final Table table;
    public EaterThread(String name, Table table, long seed) {
        super(name);
        this.table = table;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            while (true) {
                String cake = table.take();
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Table 类

Table 类（List 5-4）代表用来放置蛋糕的桌子。

蛋糕可以放置的个数是由构造器设置的。

范例程序中，蛋糕是以 String 的实例表示的。Table 类中声明有 String 的数组(String[])称为 buffer 字段，作为实际存储蛋糕的地方。

为了能够正确地放置（put）、获取（get）蛋糕，Table 还拥有 tail、head、count 这些 int 类型的字段。意义各自如下：

- tail 字段——表示下一个放置（put）蛋糕的地方；
- head 字段——表示下一个拿取（take）蛋糕的地方；
- count 字段——表示现在桌上蛋糕的个数。

Put 和 take 方法会在后文详细说明。

List 5-4　表示桌子的 Table 类（Table.java）

```
public class Table {
    private final String[] buffer;
    private int tail;   // 下一个 put 的地方
    private int head;   // 下一个 take 的地方
    private int count;  // buffer 内的蛋糕数
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
```

```
        this.tail = 0;
        this.count = 0;
    }
    // 放置蛋糕
    public synchronized void put(String cake) throws
InterruptedException {
        System.out.println(Thread.currentThread().getName() + "
puts " + cake);
        while (count >= buffer.length) {
            wait();
        }
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notifyAll();
    }
    // 获取蛋糕
    public synchronized String take() throws InterruptedException
{
        while (count <= 0) {
            wait();
        }
        String cake = buffer[head];
        head = (head + 1) % buffer.length;
        count--;
        notifyAll();
        System.out.println(Thread.currentThread().getName() + "
takes " + cake);
        return cake;
    }
}
```

这支程序的执行范例如图 5-1 所示。

注意点看的话不难发现，MakerThread 们放在桌子上的蛋糕，会被 EaterThread 们依次（No.的次序）拿走。线程更详细的行动，会在本章末的练习问题 5-2 讨论。

```
MakerThread-1 puts [ Cake No.0 by MakerThread-1 ]    ←摆出 No.0 的蛋糕
MakerThread-2 puts [ Cake No.1 by MakerThread-2 ]    ←摆出 No.1 的蛋糕
MakerThread-3 puts [ Cake No.2 by MakerThread-3 ]    ←摆出 No.2 的蛋糕
EaterThread-1 takes [ Cake No.0 by MakerThread-1 ]   ←拿走 No.0 的蛋糕
EaterThread-2 takes [ Cake No.1 by MakerThread-2 ]   ←拿走 No.1 的蛋糕
EaterThread-3 takes [ Cake No.2 by MakerThread-3 ]   ←拿走 No.2 的蛋糕
```

（续上图）

```
MakerThread-2 puts [ Cake No.3 by MakerThread-2 ]    ←摆出 No.3 的蛋糕
MakerThread-1 puts [ Cake No.4 by MakerThread-1 ]    ←摆出 No.4 的蛋糕
MakerThread-3 puts [ Cake No.5 by MakerThread-3 ]    ←摆出 No.5 的蛋糕
EaterThread-1 takes [ Cake No.3 by MakerThread-2 ]   ←拿走 No.3 的蛋糕
EaterThread-3 takes [ Cake No.4 by MakerThread-1 ]   ←拿走 No.4 的蛋糕
MakerThread-2 puts [ Cake No.6 by MakerThread-2 ]    ←摆出 No.6 的蛋糕
EaterThread-2 takes [ Cake No.5 by MakerThread-5 ]   ←拿走 No.5 的蛋糕
EaterThread-3 takes [ Cake No.6 by MakerThread-2 ]   ←拿走 No.6 的蛋糕
MakerThread-3 puts [ Cake No.7 by MakerThread-3 ]    ←摆出 No.7 的蛋糕
EaterThread-1 takes [ Cake No.7 by MakerThread-3 ]   ←拿走 No.7 的蛋糕
MakerThread-1 puts [ Cake No.8 by MakerThread-1 ]    ←摆出 No.8 的蛋糕
MakerThread-2 puts [ Cake No.9 by MakerThread-2 ]    ←摆出 No.9 的蛋糕
EaterThread-3 takes [ Cake No.8 by MakerThread-1 ]   ←拿走 No.8 的蛋糕
EaterThread-2 takes [ Cake No.9 by MakerThread-2 ]   ←拿走 No.9 的蛋糕
MakerThread-3 puts [ Cake No.10 by MakerThread-3 ]   ←摆出 No.10 的蛋糕
EaterThread-3 takes [ Cake No.10 by MakerThread-3 ]  ←拿走 No.10 的蛋糕
MakerThread-1 puts [ Cake No.11 by MakerThread-1 ]   ←摆出 No.11 的蛋糕
EaterThread-1 takes [ Cake No.11 by MakerThread-1 ]  ←拿走 No.11 的蛋糕
MakerThread-2 puts [ Cake No.12 by MakerThread-2 ]   ←摆出 No.12 的蛋糕
EaterThread-2 takes [ Cake No.12 by MakerThread-2 ]  ←拿走 No.12 的蛋糕
(以下省略。CTRL+C 结束程序)
```

图 5-1　执行范例

分析 put 方法

接着要分析的是用来摆出蛋糕的 put 方法。

◆ *throws InterruptedException*

声明 put 方法可能会抛出 InterruptedException。看到 throws InterruptedException 时，请自行解读成“这个方法可以取消”。详见下面的进阶说明。

◆ *Guarded Suspension Pattern*

put 方法中使用了 Guarded Suspension Pattern（第 3 章）。而警戒条件就是 while 条件的逻辑否定，如下所示：

```
!(count >= buffer.length)        …警戒条件
```

这等价于下面的表达式：

```
count < buffer.length            …警戒条件
```

也就是“现在桌上的蛋糕数，低于可以放置的总上限”，简单地说就是“还有可以用来摆放蛋糕的空间”。以这个表达式作为 put 方法的警戒条件，大家应该都可以

理解吧。

或许将警戒条件改写成下面的形式，更容易看出是“还有可以用来摆放蛋糕的空间”：

```
buffer.length - count > 0        …警戒条件
```

上面的表达式中，“以桌上可以摆的数量上限，减掉现在已经摆的个数”，剩下的就是“可以摆放蛋糕的空间”，这个数字必须大于0。

◆ *tail 与 count 的更新*

tail（尾巴）字段存放的是下一次放置蛋糕的位置（数组 buffer 的索引）。所以放置蛋糕的操作是以下面的语句表示的：

```
buffer[tail] = cake;
```

把蛋糕放上去了以后，需要将 tail 前进到“下一次放置蛋糕的位置”。基本上只要将 tail 加 1 就可以了，不过如果超过 buffer 的长度，就必须回到0。也就是下面这样：

```
tail++;
if (tail >= buffer.length) {
    tail = 0;
}
```

这个操作使用也可以使用求余数的%运算符，写成下面这样，意义是一样的：

```
tail = (tail + 1) % buffer.length;
```

另外，桌上的蛋糕也会增加，所以 count 的值也要加 1。

◆ *notifyAll*

上面的程序代码已经将蛋糕放在桌子上了。因为桌子的状态变化了，所以要执行 notifyAll，把 wait 中的线程全部唤醒。

分析 take 方法

接着要分析的是 List 5-4 的 take 方法。大致上是 put 方法的相反操作。

◆ *throws InterruptedException*

take 方法也会抛出 Interrupted Exception。也就是可以取消的方法。

◆ *Guarded Suspension Pattern*

take 方法也使用了 Guarded Suspension Pattern（第 3 章）。而警戒条件，也就是 while 条件的逻辑否定，是下面的表达式：

```
!(count <= 0)                …警戒条件
```

这与下面的表达式同义：

```
count > 0                                    ...警戒条件
```

也就是说“现在桌上的蛋糕数大于 0”，即“蛋糕至少有 1 个”，相信以这个运算是作为拿蛋糕用的 take 方法的警戒条件，这也是大家都可以接受的。

◆ *head 与 count 的更新*

head（头）字段则表示下一次拿取蛋糕的位置。拿走蛋糕以后，head 也要前进。在这里与 put 方法一样，使用余数运算符：

```
head = (head + 1) % buffer.length;
```

另外，因为拿走了一个蛋糕，所以 count 的值减去 1。顺序图如图 5-2 所示。

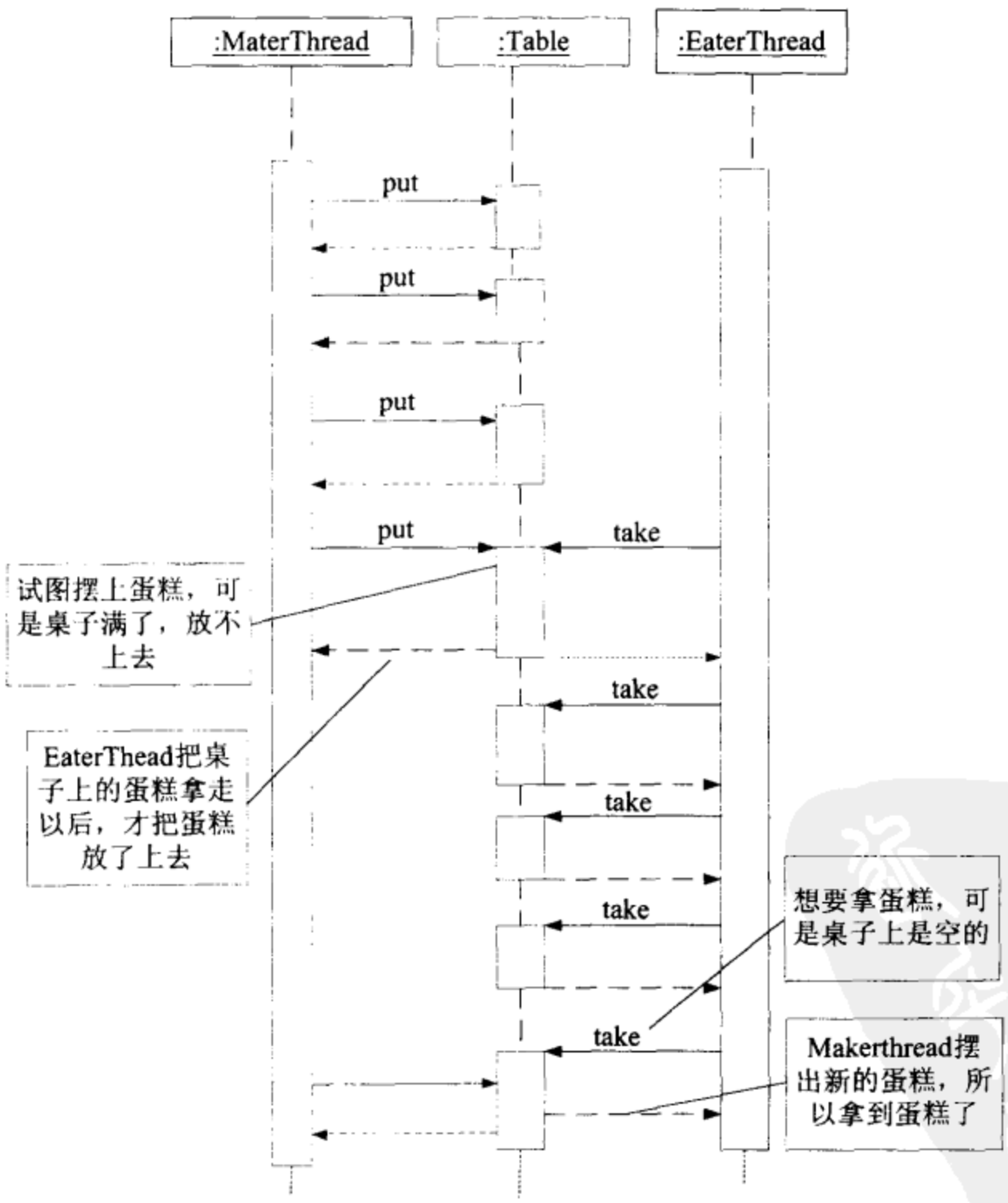


图 5-2　范例程序的顺序图（MakerThread 与 EaterThread 只画一个人）

◆ *notifyAll*

上面的程序代码已经将蛋糕拿走了。因为桌子的状态变化了，所以要执行notifyAll，把 wait 中的线程全部唤醒。

Producer-Consumer Pattern 的所有参与者

Producer-Consumer Pattern 的所有参与者如下：

◆ *Data 参与者*

Data 参与者由 Producer 参与者所建立，并由 Consumer 参与者所使用。
在范例程序中，Data 参与者是 String 类（蛋糕）。

◆ *Producer（生产者）参与者*

Producer 参与者会建立 Data 参与者，传递给 Channel 参与者。在范例程序中，由 MakerThread 担任 Producer 参与者的角色。

◆ *Consumer（消费者）参与者*

Consumer 参与者会从 Channel 参与者获取 Data 参与者。在范例程序中，由 EaterThread 担任 Consumer 参与者的角色。

◆ *Channel（通道）参与者*

Channel 参与者会从 Producer 参与者处接收 Data 参与者，并保管起来。并应 Consumer 参与者的要求，将 Data 参与者传送出去。为了确保安全性，Producer 参与者与 Consumer 参与者要对访问进行共享排斥。

当 Producer 参与者将 Data 参与者传给 Channel 参与者的时候，若 Channel 参与者的状态无法接受 Data 参与者，这时 Producer 参与者会在 Channel 的状态变成可以接受之前，保持等待状态。

当 Consumer 参与者从 Channel 参与者获取 Data 参与者的时候，若 Channel 参与者没有 Data 参与者可以给 Consumer 参与者，这时 Consumer 参与者在 Channel 可以传送 Data 参与者之前，保持等待状态。

当程序存在有多个 Producer 参与者与 Consumer 参与者时，为了使处理不互相干涉，Channel 参与者也要进行共享互斥。

像这样，Channel 参与者介于 Producer 参与者、Consumer 参与者之间，担任传递 Data 参与者中继站、通道的角色。

在范例程序中，Table 类就扮演 Channel 参与者的角色。

Producer-Consumer Pattern 的类图如图 5-3 所示。

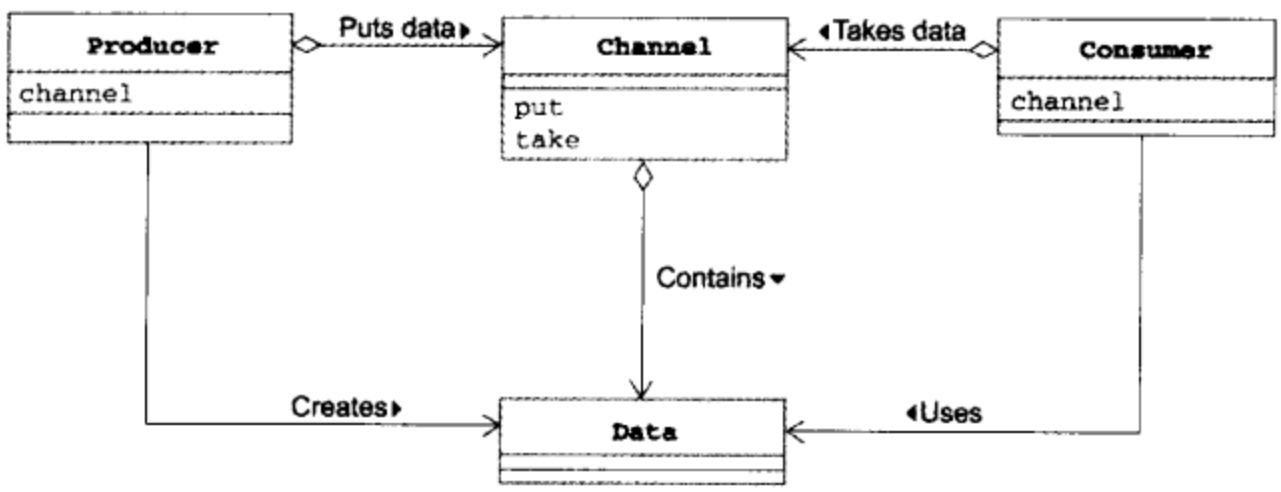


图 5-3　Producer-Consumer Pattern 的类图

扩展思考方向的提示

保护安全性的 Channel 参与者 [复用性]

Producer-Consumer Pattern 中，肩负保护安全性的使命的是 Channel 参与者。Channel 参与者进行线程之间的共享互斥，让 Producer 参与者能正确地将 Data 参与者送到 Consumer 参与者的手上。

范例程序中，Table 类的 put 方法与 take 方法都使用了 Guarded Suspension Pattern。但 MakerThread 类与 EaterThread 类都不想依于 Table 类的详细实现。也就是说，MakerThread 不必理会其他线程，只要调用 put 方法就好；而 EaterThread 也是只要会调用 take 方法就可以了。使用 synchronized、wait、notifyAll 这些考虑到多线程操作的程序代码，全部隐藏在 Channel 参与者 Table 类里面。

不能直接传递吗

Producer-Consumer Pattern 中，Producer 参与者为了将 Data 参与者传送给 Consumer 参与者，设计了一个 Channel 参与者放在中间。这里可能会“为什么不能让 Producer 参与者直接调用 Consumer 参与者的方法”这样的疑问油然而生。所以接下来我们就比较直接调用方法，与通过 Channel 参与者传递有什么不同。

◆ 直接调用方法

Consumer 参与者想要获取 Data 参与者，通常是想要使用 Data 参与者作某些处理。若 Producer 参与者直接调用 Consumer 参与者的话，进行处理的就不是 Consumer 参与者的线程，而是 Producer 参与者的线程了。

这样一来，这个处理所花的时间，也得由 Producer 线程来负担，因此制作下一个数据的操作也必须延迟。这样会使程序的响应性变差。

直接调用对方的方法，就好像厨师制作蛋糕直接交给客人，等客人吃完才开始做下一个（或是比喻成厨师在自己吃掉自己做的蛋糕）。

◆ 通过 Channel 参与者传递

而通过 Channerl 参与者时呢？Producer 参与者将 Data 参与者传给 Channel 参与者后，不用等待 Consumer 参与者，而可以马上开始制作下一个 Data 参与者。Producer 参与者可以不断地制作新的 Data 参与者。Producer 参与者的操作不会受到 Consumer 参与者的影响。

当然，虽说可是不断地制作新的 Data 参与者，但也受到 Channel 参与者的存储量限制。当 Channel 参与者无法负荷时，就不能再增加 Data 参与者了。这个“负荷”，就是我们下面要介绍的。

Channel 参与者负荷派生的问题

范例程序中，桌上可以摆放的蛋糕最多可以有 3 个。厨师最多可以在桌上摆出 3 个蛋糕，如果要摆第 4 个以上，就必须等待到客人拿走蛋糕才行。如果客人慢吞吞地吃，那厨师就要等上很久。

如果增加桌上可以摆放的蛋糕数量呢？这么一来，就算客人吃得慢一点，厨师也可以不用等待，可以不断做出新蛋糕摆到桌上。桌上的蛋糕数（buffer 字段的元素数），会直接影响到缓冲 MakerThread 与 EaterThread 间处理速度差的程度。

当然，如果客人吃蛋糕的平均速度比厨师做出的蛋糕的平均速度低，那桌上留下的蛋糕会越来越多，一段时间后还是会达到 buffer 字段的上限。

像 Guarded Suspension Pattern（第 3 章）的范例程序，使用 java.util.LinkedList，就可以实现出可存储的实例数没有上限的 Channel 参与者了。但是即使这样做，在 EaterThread 平均速度较慢的情况下，最后（虽然可能要等很长一段时间）还是会因为内存不足，无法分配出存放蛋糕的实例。

要以什么顺序传递 Data 参与者

Channel 参与者会从 Producer 参与者中接受 Data 参与者，并传递给 Consumer 参与者。当 Channel 参与者拥有多个 Data 参与者时，要以什么顺序传递给 Consumer 参与者呢？ 下面比较各种决定传递顺序的方式。

◆ 队列——最先收到的先传

最常见的做法，是把最先收到的传出去。这种方法我们在 Guarded Suspension

Pattern（第 3 章）里也碰过。我们将这种方法称为 FIFO（First In First Out），先进先出或称为队列（queue）。

范例程序的 Table 类，也是使用队列。Guarded Suspension Pattern 的范例程序 RequestQueue 类（List 3-2）是使用 java.io.LinkedList 类实现出队列的。而本章的范例程序则是使用数组来实现队列。无论是怎样实现的，只要是先进先出，就是队列。

◆ 堆栈——最后收到的先传

与队列相反的，最后收到的东西先传出去。这称之为 LIFO（Last In First Out），后进先出，又称为堆栈（stack）（stack 本意是“堆放”的意思）。

Producer-Consumer Pattern 中很少使用堆栈。不过日常生活倒经常使用。例如请想象家里厨房中盘子堆放的情况。洗好的盘子总是一个一个往上堆，而要使用时则是从上面往下拿；最后放上去的盘子，会是第一个拿来用的。这就是堆栈。

◆ 优先队列——“优先”的东西先传

比较常用的方法，还有优先队列（priority queue）可想。这是在 Channel 参与者中对 Data 参与者以某些条件给予优先次序，而优先性高的先传给 Consumer 参与者。决定优先次序的方法就琳琅满目了。

队列、堆栈可以说是优先队列的特例，Data 参与者收到时间较早者优先性高的优先队列称为队列，而 Data 参与者收到时间较晚者优先性高的优先队列称为堆栈。

[中间者的存在]隐含的意义

Producer-Consumer Pattern 中，放在 Producer 参与者和 Consumer 参与者之间的中间者——Channel 参与者，具有重要的意义。因为 Channel 参与者的存在，Producer 参与者和 Consumer 参与者这多条线程，才能保持协力、合作。因为 Channel 参与者的中介，才能实现线程的协调合作。

而您还记得在 Single Threaded Execution Pattern（第 1 章）中曾经说的“synchronized 是在保护什么”吗？当时我们说过，思索线程的共享互斥问题时，把观察切入点放在“保护着什么”上面，会比较容易找到问题的症结。

上面所说的思维若整理成口诀，可以得到这两句：

- 线程的合作要想“放在中间的东西”
- 线程的互斥要想“应该保护的东西”

仔细地思考不难发现，合作与互斥其实是表里一体的。线程为了协调合作，所以必须进行共享互斥，使共享的东西不会损坏。而线程的共享互斥，也是为了让线程合作才进行的。也就是说，合作与互斥本来就具有深远的关系。因此，上述的口诀，可说是多线程程序中，解决问题的重要线索。

只有一个 Consumer 参与者时会如何

Producer-Consumer Pattern 通常都考虑多 Producer 参与者和多 Consumer 参与者之间的数据传递。在这里，我们想思考的是如果 Consumer 参与者只有一个的时候，又会是什么情况。也就是“多 Producer 参与者和单一 Consumer 参与者”。

有多个 Producer 参与者，也就是有多条线程可以产生出 Data 参与者。而产生的所有 Data 参与者，都会存放在 Channel 参与者里。

Consumer 参与者为惟一时，处理 Channel 参与者里存放的 Data 参与者的线程只有一条。如果 Consumer 参与者为复数时，就必须小心 Consumer 参与者的线程之间互相不能发生干涉。但如果 Consumer 参与者只有一条，就不需要烦恼 Consumer 参与者的线程之间的干涉了[注1]。

具体来说，也就是只有 Consumer 参与者的线程会访问的范围，就不需要考虑共享互斥了。这样可以提升程序的性能。

Producer 参与者为复数，而 Consumer 参与者为单数的状况，就是所谓将多线程的处理简化成单一线程。

Swing（JFC）framework 中，事件处理的部分，就是使用这个方法（多 Producer 参与者和单一 Consumer 参与者）。进行 Swing 事件处理的线程，称为（the）event-dispatching thread（事件分配处理线程）。这个线程相当于从 Channel 参与者里的事件队列取出事件来处理的 Consumer 参与者。event-dispatching thread 的数量只有一个。这种设计方式，可以使事件处理的程序好写得多，而且处理的速度也很快。

关于 event-dispatching thread，在 Worker Thread Pattern（第 8 章）的进阶说明还会有更详细的解说。

相关 Pattern

Mediator Pattern（参考附录 E [GoF] [Yuki01]）

Mediator Pattern 中，多个 ConcreteColleague 参与者间并不是直接进行沟通，而是通过中介的 ConcreteMediator 参与者控制、调解操作的进行。

而 Producer-Consumer Pattern 的 Channel 参与者也是在进行控制、调解 Producer 参与者和 Consumer 参与者的操作。但 Producer-Consumer Pattern 中控制、调解的操作，并不是调用 Producer 参与者和 Consumer 参与者的方法，而是以对多条线程进行共享互斥来做到的。

注 1：当然，Consumer 参与者和 Producer 参与者之间的干涉还是需要注意的。

Worker Thread Pattern（第 8 章）

Producer-Consumer Pattern 的焦点集中在 Producer 参与者把数据传给 Consumer 参与者的经过。

而 Worker Thread Pattern 中，Client 参与者也会把数据传给 Worker 参与者。不过 Worker Thread Pattern 则把焦点放在共有处理数据的线程，与如何降低线程启动时所花费的时间。

Command Pattern（参考附录 E [GoF] [Yuki01]）

Producer-Consumer Pattern 中，Producer 参与者传递给 Consumer 参与者的 Data 参与者，有时候会用到 Command Pattern。

Strategy Pattern（参考附录 E [GoF] [Yuki01]）

Producer-Consumer Pattern 中，决定 Channel 参与者传递 Data 参与者给 Consumer 参与者的顺序时，可以使用 Strategy Pattern。

进阶说明：InterruptedException 异常

可能会花一些时间，但是可以取消掉

习惯编写 Java 多线程程序以后，就会很习惯去注意方法的后面有没有接着 throws InterruptedException。当方法接着 throws InterruptedException 时，表示这个方法内（或是这个方法所调用的方法内），可能会抛出 InterruptedException 异常。

这通常告诉我们下面这两件事：

- 这是“需要花点时间”的方法
- 这是“可以取消”的方法

用一句话来说，后面接着 throws InterruptedException 的方法，是可能会花一点时间，但是可以取消的方法。

下面就要解释这句话具体的意义：

后面接着 throws InterruptedException 的方法

Java 的标准类链接库中，后面接着 throws InterruptedException 的方法的代表选手，有下面这 3 位：

- java.lang.Object 类的 wait 方法
- java.lang.Thread 类的 sleep 方法
- java.lang.Thread 类的 join 方法

◆ 需要花点时间的方法

执行 wait 方法的线程，会进入等待区等待被 notify/notifyAll。在等待的期间，线程不会活动。故需要花费等待被 notify/notifyAll 的时间。

执行 sleep 方法的线程，会暂停执行参数内所设置的时间。这也是花费时间的方法。

执行 join 方法的线程，会等待到指定的线程结束为止。也就是会花费直到指定线程结束之前的这段时间。

像这样，上面 3 个方法，分别要等待“被 notify/notifyAll、设置的时间、指定的线程结束”，的确都是“需要花点时间”的方法。

◆ 可以取消的方法

因为需要花时间的操作，会降低程序的响应性，所以我们会希望像下面这样可以在中途放弃（取消）执行这个方法。

- 取消 wait 方法等待 notify/notifyAll 的操作
- 取消 sleep 方法等待设置长度时间的操作
- 取消 join 方法等待其他线程结束的操作

取消了以后要做什么事呢？这不同的程序就会有不同的规格了，可能结束线程或是通知用户取消的事实，也有可能放弃现在的处理而进入下一个操作。

sleep 方法与 interrupt 方法

以 sleep 方法为例，接下来我们要考虑的是如何取消。

线程 Alice 像下面这样，使用 sleep 线程暂停着。

```
Thread.sleep(604800000);
```

在这里，我们要取消线程 Alice 的等待状态（604800000ms 可是一个礼拜喔）。

因为线程 Alice 暂停着，所以取消的操作的必须由其他线程来执行的（这项线程就称做 Bobby 吧）。

线程 Bobby，会执行下面的语句，要求线程 Alice 放弃等待操作。

```
alice.interrupt();
```

在这里，变量 alice 是线程 Alice 对应到的 Thread 的实例。

在这里所使用的 interrput 方法，是 Thread 类的实例方法。执行 interrupt 方法时，并不需要获取 Thread 实例的锁定。任何线程在任何时刻，都可以调用其他线程的

interrupt 方法。

当 sleep 中的线程被调用 interrupt 方法时，就会放弃暂停的状态，并抛出 InterruptedException 异常。丢出异常的，是 Alice 线程。

这么一来，线程 Alice 的控制权，就交给捕捉这个异常的 catch 块了。

wait 方法与 interrupt 方法

当线程 Alice 以 wait 方法等待时，与 sleep 一样可以取消。使用 interrupt 方法，可以对 wait 中的线程传达“不用等 notify / notifyAll 了，从等待区出来吧”的意图。

线程 Bobby 执行下面的语句后，与 sleep 时一样，线程 Alice 会抛出 InterruptedException 异常。

```
alice.intereupt();
```

但是当线程 wait 时，要小心锁定的问题。线程在进入等待区时，会把锁定解除。当对 wait 中的线程调用 interrupt 时（即等待被取消的线程），会先重新获取锁定，再抛出 InterruptedException。获取锁定之前，并无法抛出 InterruptedException 异常。

▶▶ 补充说明： notify 方法与 interrupt 方法

notify 方法与 interrupt 方法的操作，对 wait 中的线程调用时意义有点接近，但是仍有差异如下：

notify/notifyAll 是 java.lang.Object 类的方法，是该实例的等待区调用的。而不是对线程直接调用。notify / notifyAll 方法所唤醒的线程，会前进到 wait 的下一个语句。另外，执行 notify / notifyAll 方法，获取类的实例。

interrupt 是 java.lang.Thread 的方法，是对该线程直接调用的。当被 interrupt 的线程正在 sleep 或 wait 时，会抛出 InterruptedException 异常。执行 interrupt（取消其他线程），不需要获取该线程的锁定。

join 方法与 interrupt 方法

当线程以 join 方法等待其他线程结束时，一样可以使用 interrupt 方法取消之。因为调用 join 方法不需要获取锁定，故与 sleep 时一样，会马上跳到 catch 块里。

interrupt 方法只是改变中断状态而已

或许有些读者会认为“调用 interrupt 后，目的线程就会自己抛出 InterruptedException 异常”，但这其实是误解。事实上，interrupt 这个方法只会改变线程的中断状态而已。

所谓中断状态（interrupt status），是用来表示这个线程有没有被中断的状态。

线程 Alice 在执行 sleep、wait、join 时，线程 Bobby 调用了 Alice 的 interrupt 方法，的确这个时候线程 Alice 里会有 InterruptedException 异常抛出来。但这其实是在 sleep、wait、join 这些方法内部会不断检查中断状态的值，而自己抛出的 InterruptedException。

如果线程 Alice 正在执行的是 1+2 的计算操作或 a = 123 这类的指定操作。那么，就算这时 Bobby 调用 Alice 的 interrupt 方法，线程 Alice 也不会抛出 InterruptedException 异常，而会继续处理自己后续的操作。事实上并不只是计算跟指定而已，即使是 for 语句、while 语句、if 语句、调用方法，都不会去检查中断状态。所以线程 Alice 不会抛出 InterruptedException，而会一直继续着自己的操作。当线程 A 终于执行到 sleep、wait、join 这些方法时，才马上抛出 InterruptedException。

若没有调用 sleep、wait、join 这些方法，或是没有在线程里自己检查中断状态，自己抛出 InterruptedException，那 InterruptedException 是不会抛出来的。

下面就要说明检查线程中断状态的方法，如图 5-4 所示。

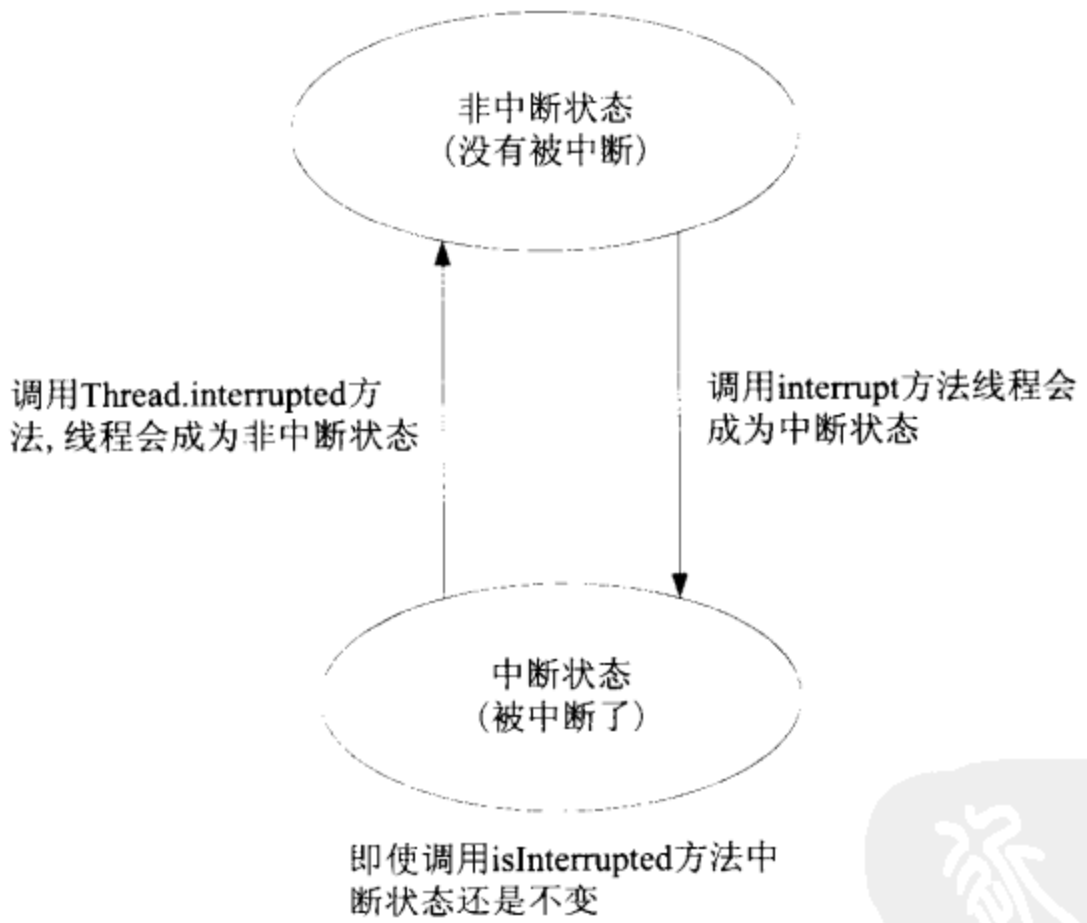


图 5-4　线程的中断状态

isInterrupted 方法——检查中断状态

Thread 类的实例方法 isInterrupted 方法可用来检查指定线程的中断状态。当线程为中断状态时，会返回 true；而线程为不是中断状态（非中断状态）时，会返回 false。isInterrupted 不会自己改变中断状态。

Thread.interrupted 方法——检查并清除中断状态

而 Thread.interrupted 方法会检查现在线程的中断状态，并清除之（Thread 类的类方法）。当现在的线程为中断状态时，就返回 true，若非中断状态时，则返回 false。调用 Thread.interrupted 方法后，中断状态会被清除（设置为非中断状态）。

除了这个方法以外，没有其他用来清除中断状态的方法。Thread.interrupted 方法检查的对象是现在的线程（也就是自己这条线程），所以我们无法清除其他线程的中断状态。

▶▶ 补充说明：interrupt 方法与 interrupted 方法

interrupt 方法与 interrupted 方法外观类似，但意义并不相同。请注意结尾有无“ed”。

interrupt 方法是将线程切换到中断状态的方法。

interrupted 方法是检查并清除中断状态的方法。

Thread 类的 stop 方法不能使用

这章的进阶说明讨论了 interrupt 方法的取消等待。

然而，Thread 类里还有 stop 这个 deprecated 方法（不建议使用的方法）。这个方法本来是用来强制结束还在运行中的线程。不过，现在因为 stop 方法可能破坏安全性，所以不能使用。这是因为调用 Thread 类的 stop 方法时，就算线程正在执行临界区间的内容，也可以结束掉线程。以安全性的观点来看，这是有危险性的。临界区间的内容应该要一连串作完，不该间断。

另外 java.applet.Applet 类还有一个 stop 方法，这与 Thread 类的 stop 方法没有关系。

重点回顾

这一章我们学习了 Producer-Consumer Pattern。

这个 Pattern 中，有着建立 Data 参与者的线程（Producer 参与者）与使用 Data 参与者的线程（Consumer 参与者）在运行。而现在要从 Producer 参与者，将 Data 参与者传递给 Consumer 参与者。

这时，我们在 Producer 参与者与 Consumer 参与者之间，安插了 Channel 参与者。并委托由 Channel 参与者来保管想要传递的 Data 参与者。Channel 参与者的工作可说是 Data 参与者的中继站、桥梁及沟通管路。

因为可能会有多个线程使用到 Channel 参与者，所以 Channel 参与者中必须进行共享互斥。在 Channel 参与者中，“从 Producer 参与者接收 Data 参与者”与“传送 Data

参与者给 Consumer 参与者”的部分使用了 Guarded Suspension Pattern（第 3 章）。

因为这样的设计，使线程之间可以安全地沟通。若 Channel 参与者可囤积 Data 参与者的空间足够，可作为 Producer 参与者与 Consumer 参与者处理速度差的缓冲。

这就是 Producer-Consumer Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 5-1（基础知识的确认）

Java 下列语句中，正确的请回答○，不正确的请回答×。

（1）MakerThread 类的构造器中所写的 super(name)调用出 Thread 类的构造器。

（2）MakerThread 类的 nextId 方法变成 Synchronized，是由于 EaterThread 类也在调用它的原因。

（3）在桌上没有放蛋糕时调用 take，线程就会去获取 Table 的实例的 lock，而进行 block。

（4）如果桌上没有任何蛋糕时，count 字段的值就是 0。

（5）桌上放满了蛋糕，再也放不下时，count 字段的值等于 buffer.length-1。

（6）head 字段的值不会超过 buffer.length。

问题 5-2（停住的程序）

Java 尝试将范例程序的 Main 类（List 5-1）改写如下（List 5-5，但如执行范例（见图 5-5）所示，程序却停下来了。请问发生了什么问题呢？

List 5-5　不能正常执行的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new MakerThread("Alice", new Table(3), 31415).start();
        new EaterThread("Bobby", new Table(3), 92653).start();
    }
}
```

```
Alice puts [ Cake No.0 by Alice ]
Alice puts [ Cake No.1 by Alice ]
Alice puts [ Cake No.2 by Alice ]
Alice puts [ Cake No.3 by Alice ]
(停在这里。CTRL+C 结束程序)
```

图 5-5　执行结果

问题 5-3（单纯的疑惑）

Java 某个对 Single Threaded Execution Pattern（第 1 章）与 Guarded Suspension Pattern（第 3 章）不甚熟悉的人看完这一章，出现下面这样单纯的疑惑：

“take 方法只不过是“没有蛋糕的时候就等，有蛋糕就拿”而已，不是吗？为什么一定需要到 synchronized 方法呢？实在是令人不解。”

请回答这个问题。

问题 5-4（加上调试输出）

Java 请在 Table 类（List 5-4）加上调试用的输出，使线程是否有在进行 wait 等待能更加一目了然。

问题 5-5（加上清除蛋糕的方法）

Java 请在范例程序的 Table 类中（List 5-4），加上用来清除桌上所有蛋糕的 clear 方法。clear 方法没有参数，返回值是 void。

问题 5-6（结束线程）

Java 改写范例程序的 Main 类（List 5-1），使程序能在开始执行约 10 秒后结束掉所有线程。然后程序就此结束，但是请不要更改到 Main 类以外的部分。

问题 5-7（使繁重的工作可以取消）

Java Host 类（List 5-6）execute 方法，会按参数 count 指定的次数，连续调用 doHeavyJob 方法。doHeavyJob 是很繁重的工作（线程会花很多时间处理的工作），而且还取消不掉。当 count 参数传入的数字一大，线程光要从退出 exexute 方法，就要执行很长一段的时间。

请改写 Host 类的 execute 方法，使它可以中途取消。

List 5-6　繁重的工作（Host.java）

```
public class Host {
    public static void execute(int count) {
        for (int i = 0; i < count; i++) {
            doHeavyJob();
        }
    }
    private static void doHeavyJob() {
        // 下面的程序代码
        // 是用来取代“无法取消的繁重工作”
        // （停留约 10 秒的循环）
        System.out.println("doHeavyJob BEGIN");
        long start = System.currentTimeMillis();
        while (start + 10000 > System.currentTimeMillis()) {
```

```
            // busy loop
        }
        System.out.println("doHeavyJob END");
    }
}
```

问题 5-8（notify 无法正常运行的情况）

Java List 5-7 将范例程序 Table 类（List 5-4）的 notifyAll 改成 notify。使用这个类，有可能出现蛋糕无法传递的情况。

请思考问题的原因，并写出程序证明您的想法。

List 5-7　不用 notifyAll，而用 notify 的 Table 类（Table.java）

```
public class Table {
    private final String[] buffer;
    private int tail;  // 下一个 put 的地方
    private int head;  // 下一个 take 的地方
    private int count; // buffer 内的蛋糕数
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
    // 放置蛋糕
    public synchronized void put(String cake) throws
InterruptedException {
        System.out.println(Thread.currentThread().getName() + "
puts " + cake);
        while (count >= buffer.length) {
            wait();
        }
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notify();
    }
    // 获取蛋糕
    public synchronized String take() throws InterruptedException
{
```

```
        while (count <= 0) {
            wait();
        }
        String cake = buffer[head];
        head = (head + 1) % buffer.length;
        count--;
        notify();
        System.out.println(Thread.currentThread().getName() + "
takes " + cake);
        return cake;
    }
}
```

问题 5-9（智力激荡）

Java 在这里休息一下，来玩个智力激荡转换心情吧。

Something 类（List 5-8）声明了一个 method 方法。请问这个方法有什么功能呢？

List 5-8　这是做什么的（Something.java）

```
public class Something {
    public static void method(long x) throws InterruptedException
{
        if (x != 0) {
            Object object = new Object();
            synchronized (object) {
                object.wait(x);
            }
        }
    }
}
```

第6章

Read-Write Lock
——大家想看就看吧，不过看的时候不能写喔



Read-Write Lock Pattern

老师写在黑板上的说明，全班的学生都一起看着。当大家都还在看的时候，老师想要擦掉黑板，写上下一段的说明。这时候学生们出声了：“老师，我们还在看！请先不要擦！”于是老师就等待大家都看完。

这一章所要学习的是 Read-Write Lock Pattern。

当线程“读取”实例的状态时，实例的状态不会变化。会使实例状态变化的，只有线程对线程“写入”的操作而已。把视点放在实例状态变化的角度来看时，“读取”与“写入”本身是不同的东西。

Read-Write Lock Pattern 将读取与写入分开来处理。在读取数据之前，必须获取用来读取的锁定。而要写入的时候，则必须获取用来写入的锁定。

因为进行读取时，实例的状态不会改变，所以，就算有多个线程在同时读取也没有关系。但，有人在读取的时候，不可以做写入的操作。

写入的时候，实例的状态就会改变。于是，当有一个线程在写入的时候，其他线程就不可以有进行读取或写入。

一般来说，进行共享互斥会使程序性能变差，但将写入的共享互斥与读取的共享互斥分开来思考，就可以提升程序的性能。

Read-Write Lock Pattern 也有人称为 Readers and Writers Pattern（参考附录 E [Lea]）、Reader/Writer Lock Pattern（参考附录 E [Holub00][Lewis00]）、Readers/Writer Lock Pattern（参考附录 E [Lewis00]）。其中 Readers/Writer Lock Pattern 这个名称，特别强调读取端（Readers）是复数的。

范例程序

接着我们来看看使用 Read-Write Lock Pattern 的范例程序吧。在这里要写的范例程序，是有多个线程，会对 Data 类的实例进行读取或写入的操作。

类一览如表 6-1 所示。

表 6-1　类一览

名　　称	说　　明
Main	操作测试用的类
Data	可读可写的类
WriterThread	进行写入操作的类
ReaderThread	进行读取操作的类
ReadWriteLock	提供读、写锁定的类

范例程序的类图如图 6-1 所示。

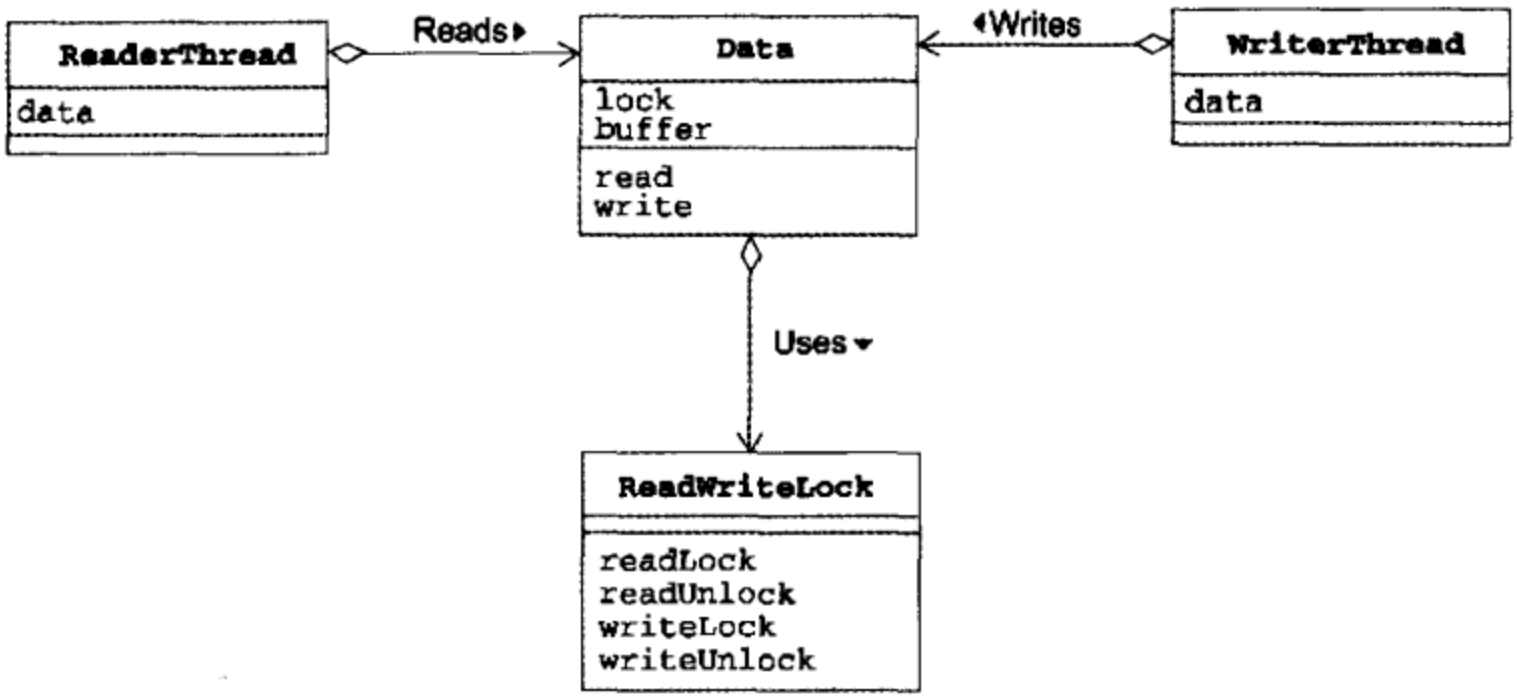


图 6-1　范例程序的类图

Main 类

Main 类（List 6-1）首先会构建出一个 Data 类的实例。再建立并启动用来“读取”这个 Data 实例的线程（ReaderThread）与用来“写入”这个 Data 实例的线程（WriterThread）。

在这里启动 6 条读访问的线程，与 2 条写入用的线程。

List 6-1　Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Data data = new Data(10);
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new WriterThread(data, "ABCDEFGHIJKLMNOPQRSTUVWXYZ").start();
      new WriterThread(data, "abcdefghijklmnopqrstuvwxyz").start();
    }
}
```

Data 类

Data 类（List 6-2）是可以进行读取（read）与写入（write）的类。

buffer 字段是实际写入对象的 char 数组。

lock 字段是这个 Pattern 的主角——ReadWriteLock 的实例。

构造器会初始化 buffer 字段，分配出一个 char 数组，而长度由参数指定。并以*字符填满 buffer，*在这里表示初始值。

read 方法是“读取”的操作。实际的读取操作是由 doRead 方法进行的，doRead 的前后夹着 lock.readLock()与 lock.readUnlock()。lock.readLock()表示获取读访问的锁定，而 lock.readUnlock()表示解除读访问的锁定。

夹住 doRead 的地方，使用了 try...finally 结构。这是因为在调用了 lock.readLock()以后，无论 doRead 里面发生了什么事，都还是要调用 lock.readUnlock()的关系。这个部分将在 Before/After Pattern 的实现。这个 Pattern 的主要结构如下所示：

```
前置处理（获取锁定）
try {
    实际的操作
} finally {
    后续处理（解除锁定）
}
```

在这里，请您回想 synchronized 块。synchronized 块是这样子使用的：

```
synchronized (锁定用的实例) {
    实际的操作
}
```

这两种写法，在概念上都是一样的结构。两者都会执行

```
获取锁定
  ↓
实际的操作
  ↓
解除锁定
```

这一连串的操作。

所以，看到 synchronized，就思考它在保护的是什么这个原则，在这里应该也适用。lock.readLock()～lock.readUnlock()在保护的是什么呢？在这里所保护的是 buffer 字段。

write 方法与 read 方法的结构一样。只不过是下面这几行取代掉而已：

```
lock.readLock()     →   lock.writeLock()
doRead()            →   doWrite(c)
lock.readUnlock()   →   lock.writeUnlock()
```

doRead 方法会进行实际的读取操作。在这里，会拷贝 buffer 的内容，分配一个新的 char 数组（newbuf），并将 newbuf 返回。

doWrite 方法会进行实际的写入操作。在这里，会使用参数 c 所传入的字符，将

buffer 的内容填满。“使用传入的字符把数组填满”这种操作虽然没有什么意义，不过为了使程序的执行结果更容易理解，此处就采用这种方式。

slowly 方法则是用来模拟需要花点时间的工作。在这里会让程序 sleep 约 50ms。doRead 方法与 doWrite 方法都有调用这个 slowly 方法，不过调用的方法有点不一样。每调用一次 doRead，slowly 也会被调用一次。而每调用 doWrite 一次，程序会在每对 buffer 写入一个字时，就调用一次 slowly。也就是说这个类是假定写入所花的时间会比读取来的长。

List 6-2 Data 类（Data.java）

```
public class Data {
    private final char[] buffer;
    private final ReadWriteLock lock = new ReadWriteLock();
    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
    }
    public char[] read() throws InterruptedException {
        lock.readLock();
        try {
            return doRead();
        } finally {
            lock.readUnlock();
        }
    }
    public void write(char c) throws InterruptedException {
        lock.writeLock();
        try {
            doWrite(c);
        } finally {
            lock.writeUnlock();
        }
    }
    private char[] doRead() {
        char[] newbuf = new char[buffer.length];
        for (int i = 0; i < buffer.length; i++) {
            newbuf[i] = buffer[i];
```

```
        }
        slowly();
        return newbuf;
    }
    private void doWrite(char c) {
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = c;
            slowly();
        }
    }
    private void slowly() {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
}
```

▶▶ 补充说明： return 与 finally

List 6-2 中，在 read 方法的 try 块中执行了 return 语句，就连这种情况下，从方法退出之前，finally 的内容还是会被执行。

▶▶ 补充说明：数组的拷贝

doRead方法里，使用for语句来拷贝数组的内容，这是为了方法拿到与doWrite作比较。要拷贝数组的内容时，使用 java.lang.System.arraycopy 方法是比较普通的方式。

WriterThread 类

WriterThread 类（List 6-3）表示用来对 Data 的实例写入的线程。构造器参数 filter 是一个字符串，程序会从这个字符串依次每次取出一个字符 write 到 Data 的实例。每 write 一次以后，就随机 sleep 一段 0ms 以上（约 3000ms 以下的时间）。nextchar 方法是用来获取下一个要写入的字符。

这个类没有任何与共享互斥相关的程序代码。

List 6-3 WriterThread 类（WriterThread.java）

```
import java.util.Random;

public class WriterThread extends Thread {
    private static final Random random = new Random();
    private final Data data;
    private final String filler;
    private int index = 0;
    public WriterThread(Data data, String filler) {
        this.data = data;
        this.filler = filler;
    }
    public void run() {
        try {
            while (true) {
                char c = nextchar();
                data.write(c);
                Thread.sleep(random.nextInt(3000));
            }
        } catch (InterruptedException e) {
        }
    }
    private char nextchar() {
        char c = filler.charAt(index);
        index++;
        if (index >= filler.length()) {
            index = 0;
        }
        return c;
    }
}
```

ReaderThread 类

ReaderThread 类（List 6-4）表示用来读取的线程。这个类会重复调用 data.read()，并显示出读取到的 char 数组。String.valueOf 是 String 类的类方法，用来将参数（这里是 char[]）转换成字符串。

这个类也没有任何与共享互斥相关的程序代码。

List 6-4 ReaderThread 类（ReaderThread.java）

```
public class ReaderThread extends Thread {
    private final Data data;
    public ReaderThread(Data data) {
        this.data = data;
    }
    public void run() {
        try {
            while (true) {
                char[] readbuf = data.read();

System.out.println(Thread.currentThread().getName() + " reads "
+ String.valueOf(readbuf));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

ReadWriteLock 类

那么，终于到真正的重头戏 ReadWriteLock 类（List 6-5）了。这是用来提供读访问锁定与写入用锁定的类。这个类看上去很简单，但其中又蕴含了巧妙的玄机，需要仔细地阅读。

为了保护安全性，我们必须防止下面两种冲突：

- “读取”和“写入”的冲突（read-write conflict）；
- “写入”和“写入”的冲突（write-write conflict）。

“读取”与“读取”之间并不会冲突。

为了不要引起冲突，所以我们首先要思考的，是可以获取读取锁定、写入锁定的条件。有下面（1）～（4）的情况：

◆ 当线程想要获取读访问的锁定时……

（1）已经有线程在进行写入时，等待。

理由：不等待的话，会引起 read-write conflict。

（2）已经有线程在进行读取时，不等待。

理由：read-read 不会造成 conflict。

◆ 当线程想要获取写入用的锁定时…

（3）已经有线程在进行写入时，等待。

理由：不等待的话，会引起 write-write conflict。

（4）已经有线程在进行读取时，等待。

理由：不等待的话，会引起 read-write conflict。

会引起冲突的情况，整理成表格如表 6-2 所示。

表 6-2　什么情况下会发生冲突呢

	读　　取	写　　入
读取	不会冲突	“读取”和“写入”的冲突 （read-write conflict）
写入	“读取”和“写入”的冲突 （read-write conflict）	“写入”和“写入”的冲突 （write-write conflict）

ReaderWriter 类就可以满足这些条件。首先，我们先来确认字段的意义。

readingReaders 字段表示实例正在读取的线程数量，也就是通过 readLock 后，还没有通过 readUnlock 的线程数量。readingReaders 的值一定大于或等于 0。

waitingWriters 字段表示调用 writeLock 时，就进入 wait 状态的线程。关于这个字段的意义，我们会在本章末的练习问题 6-6 讨论。

writingWriters 字段表示实际正在进行写入操作的线程数量，也就是通过 writeLock 后，还没有通过 writeUnlock 的线程数量。writingWriters 字段的值不是 0 就是 1，绝不会是 2 以上（一瞬间都不会）。

preferWriter 字段为是个标识，当值为 true 时，表示“写入优先”，而 false 时则表示“读取优先”。这是为了使线程的生命性不会降低所作的功劳。关于这个字段的意义，我们会在本章末的练习问题 6-6 讨论。

List 6-5 中注释所写的（A）、（B）、（C），是用来强调出字段递增（++）与递减（--）处的对应关系。

例如请看 readingReaders 字段（A）。在 readLock 方法的最后面递增，而在 readUnlock 方法的最前面递减。readingReaders 字段所表示的是实例正在读取的线程数量（通过 readLock 后，还没有通过 readUnlock 的线程数量），相信这样读者就很容易了解该进行递增与递减操作的位置。

接下来，请看 waitingWriters 字段（B）。waitingWriters 在 writeLock 方法最前面递增，并在 finally 块里递减。喔，这里出现 Before/After Pattern 了。

再来是 writingWriters 字段（C）。writingWriters 字段在 writeLock 方法的最后面递增，而在 writeUnlock 方法的最前面递减。writingReaders 所表示的是实例正在写入的线程数量（也就是通过 writeLock 后，还没有通过 writeUnlock 的线程数量），

这样就确信程序没有问题了。

List 6-5 ReadWriteLock 类（ReadWriteLock.java）

```
public final class ReadWriteLock {
    private int readingReaders = 0;     // (A)…实际正在读取的线程数量
    private int waitingWriters = 0;     // (B)…正在等待写入的线程数量
    private int writingWriters = 0;     // (C)…实际正在写入的线程数量
    private boolean preferWriter = true; // 写入优先的话，值为 true

   public synchronized void readLock() throws InterruptedException {
      while(writingWriters>0||(preferWriter && waitingWriters > 0)) {
          wait();
       }
       readingReaders++;                // (A) 实际正在读取的线程数量加 1
    }

    public synchronized void readUnlock() {
       readingReaders--;                // (A) 实际正在读取的线程数量减 1
       preferWriter = true;
       notifyAll();
    }

    public synchronized void writeLock() throws InterruptedException {
       waitingWriters++;                // (B) 正在等待写入的线程数量加 1
       try {
          while (readingReaders > 0 || writingWriters > 0) {
             wait();
          }
       } finally {
          waitingWriters--;             // (B) 正在等待写入的线程数量减 1
       }
       writingWriters++;                // (C) 实际正在写入的线程数量加 1
    }

    public synchronized void writeUnlock() {
       writingWriters--;                // (C) 实际正在写入的线程数量减 1
       preferWriter = false;
       notifyAll();
    }
}
```

执行结果

执行范例程序后，会出现图 6-2 所示的结果。这是 ReaderThread 将读到的内容显示出来的样子。当 WriterThread 在写入的时候，画面显示会忽然停下来一段时间，接下来显示出的数据可以观察到内容改变了。

在这里，reads 右方的 10 个字符永远是一样的对吧。这表示我们成功防止了线程的冲突，保障了程序的安全性。如果安全性不佳的话，可能就会有不同的字符混杂在一起了。

关于这点将在练习问题 6-2 讨论。

```
Thread-1 reads **********
Thread-2 reads **********
Thread-3 reads **********
Thread-4 reads **********
Thread-5 reads **********
Thread-6 reads **********
Thread-2 reads aaaaaaaaaa
Thread-3 reads aaaaaaaaaa
Thread-4 reads aaaaaaaaaa
Thread-5 reads aaaaaaaaaa
Thread-1 reads aaaaaaaaaa
Thread-1 reads aaaaaaaaaa
Thread-0 reads aaaaaaaaaa
Thread-3 reads aaaaaaaaaa
(中略)
Thread-0 reads LLLLLLLLLL
Thread-2 reads LLLLLLLLLL
Thread-5 reads LLLLLLLLLL
Thread-3 reads LLLLLLLLLL
Thread-4 reads LLLLLLLLLL
Thread-1 reads LLLLLLLLLL
Thread-0 reads LLLLLLLLLL
Thread-4 reads MMMMMMMMMM
Thread-2 reads MMMMMMMMMM
Thread-5 reads MMMMMMMMMM
Thread-3 reads MMMMMMMMMM
Thread-1 reads MMMMMMMMMM
Thread-0 reads MMMMMMMMMM
Thread-4 reads kkkkkkkkkk
Thread-5 reads kkkkkkkkkk
(CTRL+C 结束程序)
```

图 6-2　执行范例

检验警戒条件

我们回过头来看 List 6-5 的 ReadWriteLock 类。

相信读者已经注意到，readLock 方法与 writeLock 方法都使用了 Guarded Suspension Pattern（第 3 章）。阅读 Guarded Suspension Pattern 的关键，则在于警戒条件。我们马上来看看吧。

注意：以下的说明故意省略 waitingWriters 字段与 preferWriter 字段的解说。关于这两个字段的意义，将在本章末的练习问题 6-6 讨论。

◆ *readLock 方法*

当线程实际进行读取的操作前，会调用 readLock 方法。当线程从这个方法退出，就表示可以开始进行实际进行读取的操作了。

那么，可以开始实际进行读取操作的警戒条件该是什么呢？有其他线程也正读取并不会造成影响，但绝对不能有线程正在执行写入的操作。如果有其他线程正在写入，而这里开始读取的话，就会发生 read-write conflict 了。于是，警戒条件是“没有线程正在执行写入的操作”。以程序表达的话，就是下面的样子：

```
writingWriters <= 0     … 警戒条件
```

while 语句的条件式是警戒条件的逻辑否定，所以写成：

```
writingWriters > 0 … 警戒条件的逻辑否定
```

从 readLock 方法退出之前，要将实际进行读取的线程数量 readingReaders 递增（++）。

◆ *writeLock 方法*

当线程实际进行写入的操作前，会调用 writeLock 方法。线程从这个方法退出，就表示可以开始进行实际进行写入的操作了。

那么，可以开始实际进行写入操作的警戒条件该是什么呢？与 readLock 方法的警戒条件有一点不同。这里，无论有其他线程正在读取，或正在写入，都不可以放行。

如果有其他线程正在读取，会发生 read-write conflict；而如果有其他线程正在写入，则会发生 write-write conflict 了。于是，警戒条件是“没有线程正在执行读取或写入的操作”。以程序表达的话，就是下面的样子：

```
readingReaders <= 0 && writingWriters <= 0     … 警戒条件
```

while 条件式是警戒条件的逻辑否定，所以写成：

```
readingReaders > 0 || writingWriters > 0  … 警戒条件的逻辑否定
```

从 writeLock 方法退出之前，要将实际进行写入的线程数量 writingWriters 递增（++）。

Read-Write Lock Pattern 的所有参与者

Read-Write Lock Pattern 的所有参与者如下：

◆ *Reader（读取者）参与者*

Reader 参与者会对 SharedResource 参与者进行 read。范例程序中，Reader 参与者是 ReaderThread 类。

◆ *Writer（写入者）参与者*

Writer 参与者会对 SharedResource 参与者进行 write。范例程序中，Writer 参与者是 WriterThread 类。

◆ *SharedResource（共享资源）参与者*

SharedResource 参与者代表 Reader 参与者与 Writer 参与者所共享的资源。SharedResource 参与者会提供有不会改变内部状态的操作（read），与会改变内部状态的操作（write）。范例程序中，SharedResource 参与者是 Data 类。

◆ *ReadWriteLock（读写用锁定）参与者*

ReadWriteLock 参与者提供了对 SharedResource 参与者进行 read 操作与 write 操作时需要的锁定。为了完成 read 操作，提供了 readLock 与 readUnlock，为了完成 write 操作，提供了 writeLock 与 writeUnlock。范例程序中，ReadWriteLock 参与者是 ReadWriteLock 类（见图 6-3～图 6-5）。

注意：有时候我们也用 acquire（获取）来表示获取锁定，以 release（释放）来表示解除锁定。

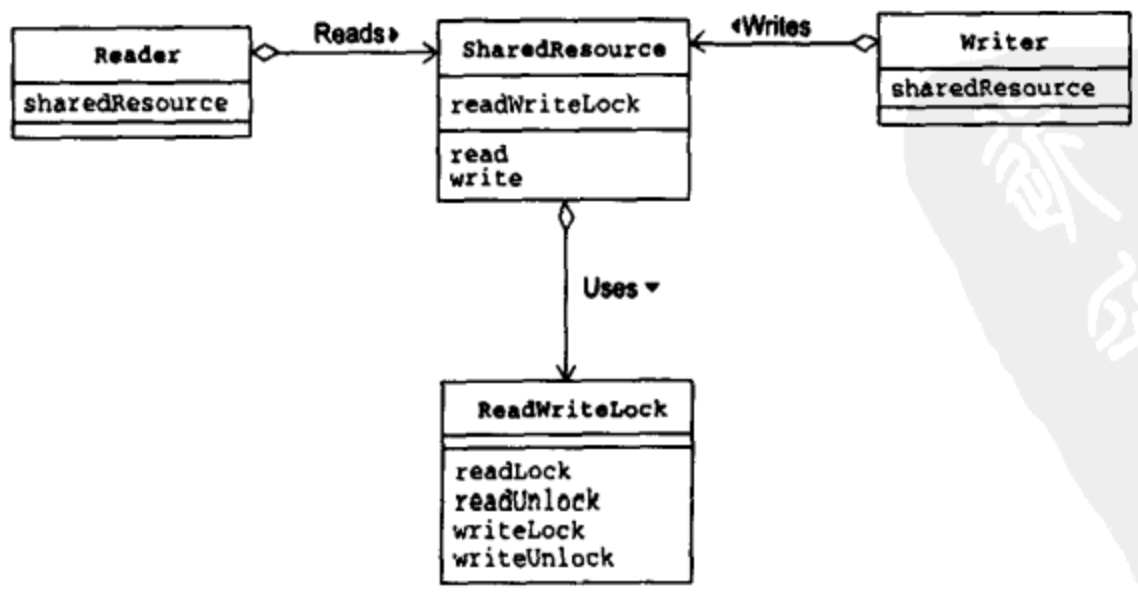


图 6-3　Read-Write Lock Pattern 的类图

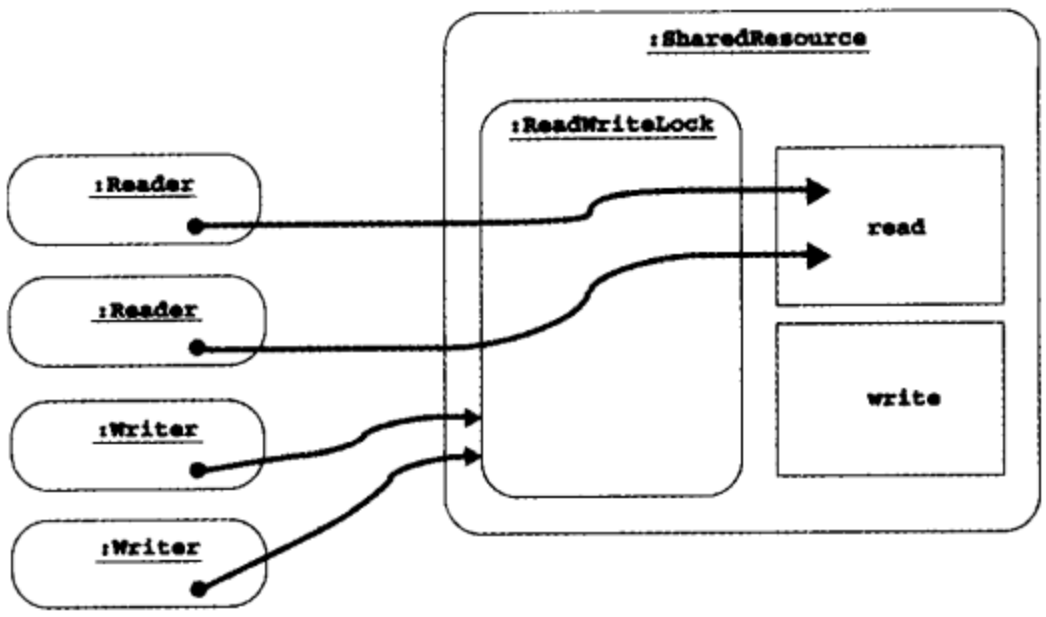


图 6-4　Read-Write Lock Pattern 的时序图

（当 Reader 参与者在读取，Writer 参与者在等待时）

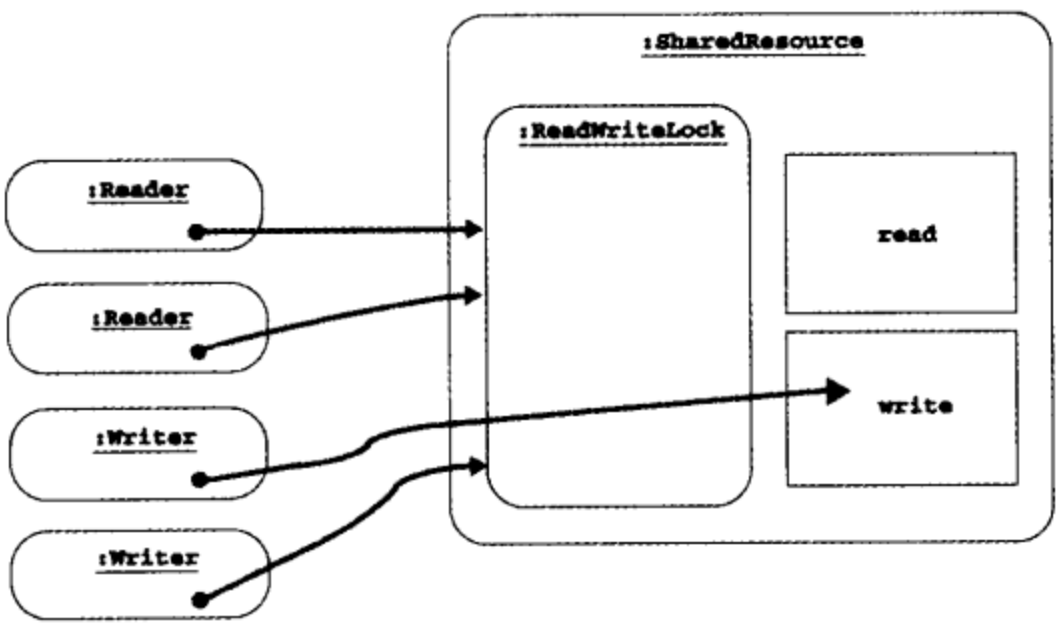


图 6-5　Read-Write Lock Pattern 的时序图

（当有一个 ReaderWriter 参与者在写入，Reader 参与者与其他的 Writer 参与者在等待时）

扩展思考方向的提示

利用同时“读取”不会冲突的特性，提高程序的性能

Read-Write Lock Pattern 利用了“读取”操作彼此不会冲突的特性。读取的操作不会改变 SharedResource 参与者的状态，所以不需要进行共享互斥。多个 Reader 参与者可以同时进行 read，以提升程序的性能。

适合读取操作繁重时

单纯使用 Single Threaded Execution Pattern（第 1 章）时，就连 read 的操作，

一次也只有一条线程可以执行。如果 read 的操作特别繁重（花时间），那使用 Read-Write Lock Pattern 会比 Single Threaded Execution Pattern 来得好。

但因为 Read-Write Lock Patter 的程序比 Single Threaded Execution Pattern 要来的复杂，若 read 的操作很简单（不花时间）时，使用 Single Threaded Execution Pattern 可能性能反而较高。

适合读取比写入次数频繁时

Read-Write Lock Pattern 的优点在于 Reader 参与者之间不会起冲突。不过，当写入（write）的操作频率较高时，Writer 参与者会经常阻挡 Reader 参与者的进行，这样无法展现 Read-Write Lock Pattern 的优点。

锁定的意义

使用 synchronized，可以获取实例的锁定。Java 程序的每个实例各有一个“锁定”，同一个实例的锁定无法由两个以上的线程所获取。这是制订在 Java 语言规格里，实现在 Java 执行环境里的机制。因为是一开始就提供的功能，我们称为所谓物理性的锁定。Java 程序无法去改变这种锁定的行为。

而相对地，这一章的“读访问锁定”、“写入用锁定”这些“锁定”，与使用 synchronized 获取的锁定意义上并不同。这不是 Java 语言制订的机制，而是程序员自己实现的。这就是所谓的逻辑上的锁定。当程序员改写 ReadWriteLock 类时，就可以更改锁定的行为机制。

在 ReadWriteLock 类中，提供了“读访问锁定”、“写入用锁定”等两种逻辑上的锁定。但是实例实做这两种逻辑上的锁定，其实只用到了一个物理性的锁定。这个物理性的锁定，就是 ReadWriteLock 实例的锁定。

▶▶ **补充说明：分离物理性的锁定**

关于分离物理上的锁定，可参考附录 E 的[Lea]与[Holub00]等著述。

相关 Pattern

Immutable Pattern（第 2 章）

Immutable Pattern 不对线程进行共享互斥，就能提升程序的性能。之所以能去掉共享互斥，是因为 Immutable 参与者的内部状态绝不会改变。

而 Read-Write Pattern 则是只去掉 Reader 参与者之间的共享互斥，以求提高程

序性能。因为 SharedResource 参与者的内部状态，只有在进行写操作时才会变化，所以 Reader 参与者之间可以不用进行共享互斥。Read-Write Lock Pattern 的 SharedResource 参与者，可说是“在不作写入操作时是 immutable 的”。

Single Threaded Execution Pattern（第 1 章）

Single Threaded Execution Pattern 则限制程序中一段特定的部分，同时只能有一个线程可以执行。

而 Read-Write Lock Pattern 则限制程序中一段特定的部分，同时只能有一个 Writer 参与者线程可以执行。

Guarded Suspension Pattern（第 3 章）

Read-Write Lock Pattern 的 ReadWriteLock 参与者，使用到了 Guarded Suspension Pattern。

Before/After Pattern（参考附录 E [Lea]）

为了忘记 Read-Write Lock Pattern 忘记解除锁定，需与 Read-Write Lock Pattern 配合使用。

Strategized Locking Pattern（参考附录 E [POSA2]）

Read-Write Lock Pattern 中，以活用 Reader 参与者与 Writer 参与者特征的共享互斥方式，提升程序的性能。

而 Strategized Locking Pattern 将同步的机制参数化，能进行更有可伸缩性的共享互斥。

重点回顾

这一章我们学习了 Read-Write Lock Pattern。

Read-Write Lock Pattern 中，有一个提供 read 与 write 两种操作的 SharedResource 参与者。read 操作不会改变 SharedResource 参与者的状态，而 write 则会。

Reader 参与者会进行 read 操作，Writer 参与者会进行 write 操作，而 read 操作执行得比 write 频繁很多。这时，若我们以对 Reader 参与者、Writer 参与者进行相同的共享互斥操作是一种浪费。因为 Reader 参与者之间没有共享互斥的必要。

然而，完全不作共享互斥又很危险。当 Reader 参与者进行 read 时，必须要求 Writer 参与者的 write 操作等待才行。而 Writer 参与者进行 write 操作时，Reader 参

与者与其他 Writer 参与者也有等待的必要。

于是，我们引进了 ReadWriteLock 参与者的概念，对 read、write 操作分别提供的锁定。进行上述复杂的共享互斥操作。

这样一来，就能兼顾 SharedResource 参与者的安全性，也能使程序的性能提高。尤其是当 read 操作繁重时，可使程序性能大幅地上升。

实现时，会使用到 Guarded Suspension Pattern（第 3 章）来进行共享互斥，所以必须妥善思考警戒条件。

Java 语言中，使用 finally 可以避免忘记解除锁定。

这就是 Read-Write Lock Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 6-1（基础知识的确认）

Java 请就您对范例程序的认知，在下列语句中，正确的请回答○，不正确的请回答×。

（1）doWrite 方法不会同时被数个线程来执行。

（2）doRead 方法不会同时被数个线程来执行。

（3）某个线程正在执行 doWrite 方法时，readingReaders 字段的值一定是 0。

（4）某个线程正在执行 doRead 方法时，writingWriters 字段的值一定是 0。

问题 6-2（完全不使用到锁定的情况）

Java List 6-6 修改了范例程序的 Data 类（List 6-2），使程序完全没用到 ReadWriteLock 类。请问执行程序 List 6-6，会出现什么结果呢？

List 6-6 不使用 ReadWriteLock 类的 Data 类（Data.java）

```
public class Data {
    private final char[] buffer;
    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
    }
    public char[] read() throws InterruptedException {
        return doRead();
```

```
    }
    public void write(char c) throws InterruptedException {
        doWrite(c);
    }
    private char[] doRead() {
        char[] newbuf = new char[buffer.length];
        for (int i = 0; i < buffer.length; i++) {
            newbuf[i] = buffer[i];
        }
        slowly();
        return newbuf;
    }
    private void doWrite(char c) {
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = c;
            slowly();
        }
    }
    private void slowly() {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
}
```

问题 6-3（执行性能）

Java List 6-7 修改了范例程序的 Data 类（List 6-2），使程序完全没用到 ReadWriteLock 类。但以 synchronized 取代原来 ReadWriteLock 类的角色。请检验 List 6-2 与 List 6-7 的执行性能有什么改变。

List 6-7　以 synchronized 取代 ReadWriteLock 类的 Data 类（Data.java）

```
public class Data {
    private final char[] buffer;
    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
```

```
    }
    public synchronized char[] read() throws InterruptedException
{
        return doRead();
    }
    public synchronized void write(char c) throws InterruptedException {
        doWrite(c);
    }
    private char[] doRead() {
        char[] newbuf = new char[buffer.length];
        for (int i = 0; i < buffer.length; i++) {
            newbuf[i] = buffer[i];
        }
        slowly();
        return newbuf;
    }
    private void doWrite(char c) {
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = c;
            slowly();
        }
    }
    private void slowly() {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
}
```

问题 6-4（复用性）

Java 范例程序 Data 类（List 6-2）与 ReadWriteLock 类（List 6-5）中，关于程序复用性，有下列的问题。请修改范例程序，改善这些问题。

（1）Data 类包含了实例进行读写的操作。若要进行其他的读写操作，需要建立新的类，进行与 Data 类一样的同步处理操作。

（2）ReadWriteLock 类包含了警戒条件。当想要更改警戒条件的原则时，还需要建立新的类，进行与 Data 类一样的读写操作。

（3）ReadWriteLock 类的方法都是 public 的，有被 Data 以外的类调用的危险。

问题 6-5（Before/After Pattern）

Java Before/After Pattern（参考附录 E [Lea]）可以用来保证方法的执行顺序。使用 Java 实现 Before/After Pattern 时，通常会像下面这样写：

```
before();
try {
    execute();
} finally {
    after();
}
```

上例中，调用 execute()之前一定会调用 before 方法，并能确保 after 方法也一定会被调用到。例如 Data 类（List 6-2）的 read 方法、write 方法中，就用到了这个 Before/After Pattern。

那么，若将 Data 类（List 6-2）改写成下面 List 6-8 的形式，会发生什么问题呢？（将 readLock 方法或 writeLock 方法也写在 try 块里）。

List 6-8 Data 类（Data.java）

```
public class Data {
    private final char[] buffer;
    private final ReadWriteLock lock = new ReadWriteLock();
    public Data(int size) {
        this.buffer = new char[size];
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = '*';
        }
    }
    public char[] read() throws InterruptedException {
        try {
            lock.readLock();
            return doRead();
        } finally {
            lock.readUnlock();
        }
    }
    public void write(char c) throws InterruptedException {
        try {
            lock.writeLock();
            doWrite(c);
        } finally {
            lock.writeUnlock();
        }
```

```
    }
    private char[] doRead() {
        char[] newbuf = new char[buffer.length];
        for (int i = 0; i < buffer.length; i++) {
            newbuf[i] = buffer[i];
        }
        slowly();
        return newbuf;
    }
    private void doWrite(char c) {
        for (int i = 0; i < buffer.length; i++) {
            buffer[i] = c;
            slowly();
        }
    }
    private void slowly() {
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
}
```

问题 6-6（preferWriter 字段的作用）

Java 若将范例程序 ReadWriteLock 类（List 6-5）中的 preferWriter 字段和 waitingWriter 字段删除，简化成 List 6-9 的程序。但实例执行程序后，结果如图 6-6 所示，数据几乎一直没有变化（执行结果会与 Java 执行环境有关），这是为什么呢？另外，请思考范例程序 preferWriter 字段和 waitingWriter 字段的意义。

List 6-9　简化的 ReadWriteLock 类（ReadWriteLock.java）

```
public final class ReadWriteLock {
    private int readingReaders = 0;     // (a)…实际正在读取的线程数量
    private int waitingWriters = 0;     // (b)…实际正在写入的线程数量

    public synchronized void readLock() throws InterruptedException {
        while (writingWriters > 0) {
            wait();
        }
        readingReaders++;               // (a) 实际正在读取的线程数量加 1
    }
    public synchronized void readUnlock() {
```

```
        readingReaders--;          // (a) 实际正在读取的线程数量减 1
        notifyAll();
    }

    public synchronized void writeLock() throws InterruptedException {
        while (readingReaders > 0 || writingWriters > 0) {
            wait();
        }
        writingWriters++;          // (b) 实际正在写入的线程数量加 1
    }

    public synchronized void writeUnlock() {
        writingWriters--;          // (b) 实际正在写入的线程数量减 1
        notifyAll();
    }
}
```

```
Thread-4 reads **********
Thread-5 reads **********
Thread-2 reads **********
Thread-1 reads **********
Thread-0 reads **********
Thread-3 reads **********
Thread-4 reads **********
Thread-5 reads **********
Thread-2 reads **********
Thread-0 reads **********
Thread-1 reads **********
Thread-3 reads **********
(以下省略。CTRL+C结束程序)
```

长时间读到的字符都没有改变

图 6-6　执行范例

问题 6-7（确认理解是否正确）

Java 回答这个问题请先做完问题 6-6，了解 perferWriter 字段的用途后再开始作答。

对于具有 perferWriter 字段的范例程序，请回答下列问题：

（1）当 WriterThread 的线程执行 writeUnlock 时，在 this（ReadWriteLock 类的实例）上 wait 的只有 ReaderThread 的线程。

（2）当 ReaderThread 的线程执行 readUnlock 时，在 this（ReadWriteLock 类的实例）上 wait 的只有 WriterThread 的线程。

第7章

Thread-Per-Message——这个工作交给你了

Thread-Per-Message Pattern

上司把文件交给下属，指示“能帮我传真这份文件吗？”。妻子对先生说“老公，帮忙倒一下垃圾喔”。一个人将工作拜托别人去做，是很常见的事情。拜托别人“这个工作交给你了”以后，就回来继续做自己的事。

这一章中要学习的是 Thread-Per-Message Pattern。

per 是“每一”的意思。所以 thread per message 解释过来就是“每个消息一个线程”。message 在这里可以看作是“命令”或“请求”的意思。对每个命令或请求，分配一个线程，由这个线程执行工作，这就是 Thread-Per-Message Pattern。

使用 Thread-Per-Message Pattern 时，“委托消息的一端”与“执行消息的一端”会是不同的线程。也就像是委托消息的线程，对执行消息的线程说“这个工作交给你了”。

范例程序

接下来就来研读使用 Thread-Per-Message Pattern 的范例程序吧。在这里要写的范例程序中，Main 类会委托 Host 类显示文字。Host 类会启动一个线程，来处理这项委托的工作。启动的线程，会使用 Helper 类实际进行显示操作。类的说明如表 7-1 所示。

表 7-1　类一览表

名　　称	说　　明
Main	用来对 Host 送出显示文本要求的类
Host	用来对收到的要求建立线程的类
Helper	提供显示文本功能的被动类

范例程序的类图如图 7-1 所示。

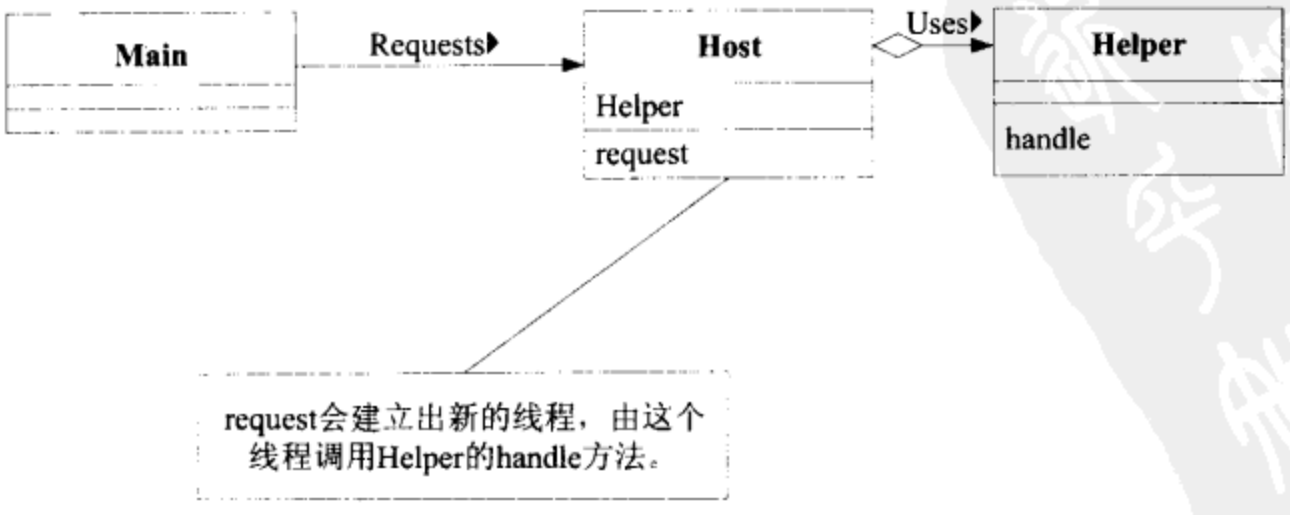


图 7-1　范例程序的类图

Main 类

Main 类（List 7-1）首先会建立出 Host 类的实例。而调用 Host 实例的 request 方法。下面这一行语句，表示连续显示 10 次 A 字符。

```
host.request(10, 'A');
```

为了容易从结果了解线程执行的状况，程序在 main 方法的最前面和最后面加上调试用的消息。

List 7-1 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("main BEGIN");
        Host host = new Host();
        host.request(10, 'A');
        host.request(20, 'B');
        host.request(30, 'C');
        System.out.println("main END");
    }
}
```

Host 类

Host 类（List 7-2）中，在 request 方法里启动新的线程。实际的操作由该线程负责。

List 7-2 中，使用了 Java 的匿名内部类（anonymous inner class），建立 Thread 子类的实例，并使用这个实例启动线程。

可能有些读者不习惯匿名内部类的语法，其实习惯的话，这并不会很困难。请仔细观察下面的程序。

```
new Thread() {
    public void run() {
        helper.handle(count, c);
    }
}.start();
```

把这行程序看作是类声明、建立实例、启动线程的操作写在一起，就好了解多了[注1]。将上面的程序省略成下面这样，就看得出来是类声明的其中一部分（声明 run 方法的部分）：

注 1：匿名内部类是将类的声明与建立实例的操作写在一起。不过并不是在执行方法时才建立出类文件。匿名内部类与一般类相同，都会在编译时产生出类文件。

```
        ... {
        public void run() {
            helper.handle(count, c);
        }
    } ...
```

而上面省略掉的部分，则看得出来是建立出 Thread 的实例，并启动线程：

```
new Thread()
        ...
 .start();
```

List 7-2 中，使用这个匿名内部类，进行下面的操作：

- 声明一个重载 run 方法的 Thread 的子类；
- 建立这个类的实例；
- 调用这个实例的 start 方法，启动线程。

请注意，在匿名内部类的 run 方法中，我们用到了 request 方法的参数 count 与 c。像这样，当我们要在匿名内部类中，用到方法的参数或局部变量时，必须将变量声明成 final。如果 count 与 c 不是 final，会出现编译错误。

List 7-2 Host 类（Host.java）

```
public class Host {
    private final Helper helper = new Helper();
    public void request(final int count, final char c) {
        System.out.println("    request(" + count + ", " + c + ")
BEGIN");
        new Thread() {
            public void run() {
                helper.handle(count, c);
            }
        }.start();
        System.out.println("    request(" + count + ", " + c + ")
END");
    }
}
```

Helper 类

Helper 类（List 7-3）提供了 handle 方法，用来将字符显示指定的次数。为了使显示速度放慢（表现出 handle 需要一些处理时间），在 slowly 方法里使用了 Thread.sleep 方法。

List 7-3 Helper 类（Helper.java）

```
public class Helper {
    public void handle(int count, char c) {
```

```
        System.out.println("        handle(" + count + ", " + c +
") BEGIN");
        for (int i = 0; i < count; i++) {
            slowly();
            System.out.print(c);
        }
        System.out.println("");
        System.out.println("        handle(" + count + ", " + c +
") END");
    }
    private void slowly() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
}
```

执行范例如图 7-2 所示。注意看调试消息的话，会发现在 handle 方法结束前，request 方法已经结束了。

执行 main 方法的主线程，调用完 host 的 request 方法后，就马上结束了。送出“帮忙显示一下 A 喔”“帮忙显示一下 B 喔”“帮忙显示一下 C 喔”这些指示，自己就结束了。无论 handle 方法多花时间，都不会影响 request 方法的响应性。request 方法不会等待 handle 方法执行完毕，而会马上退出。

```
main BEGIN
   request(10, A) BEGIN
   request(10, A) END          ←在handle(10, A)BEGIN之前request(10, A)先结束了
   request(20, B) BEGIN
   request(20, B) END          ←在handle(20, B)BEGIN之前request(20, B)先结束了
   request(30, C) BEGIN
   request(30, C) END          ←在handle(30, C)BEGIN之前request(30, C)先结束了
main END                       ← 主线程在这里结束
     handle(10, A) BEGIN       ←其他线程开始执行handle(10, A)
     handle(20, B) BEGIN       ←其他线程开始执行handle(20, B)
     handle(30, C) BEGIN       ←其他线程开始执行handle(30, C)
ABCABCABCABCABCABCABCACBACBA ←三个线程的内容交错
     handle(10, A) END         ←handle(10, A)的操作结束
BCBCBCBCBCBCBCBCBCBCB          ←两个线程的内容交错
     handle(20, B) END         ←handle(20, B)的操作结束
CCCCCCCCCCC                    ←最后一个线程的输出
     handle(30, C) END         ←handle(30, C)的操作结束
```

图 7-2　执行范例（显示内容会按执行时机多少有所变化）

范例程序的顺序图如图 7-3 所示。调用 request 方法的线程会马上回到原处，而 handle 方法是别的线程在处理的。

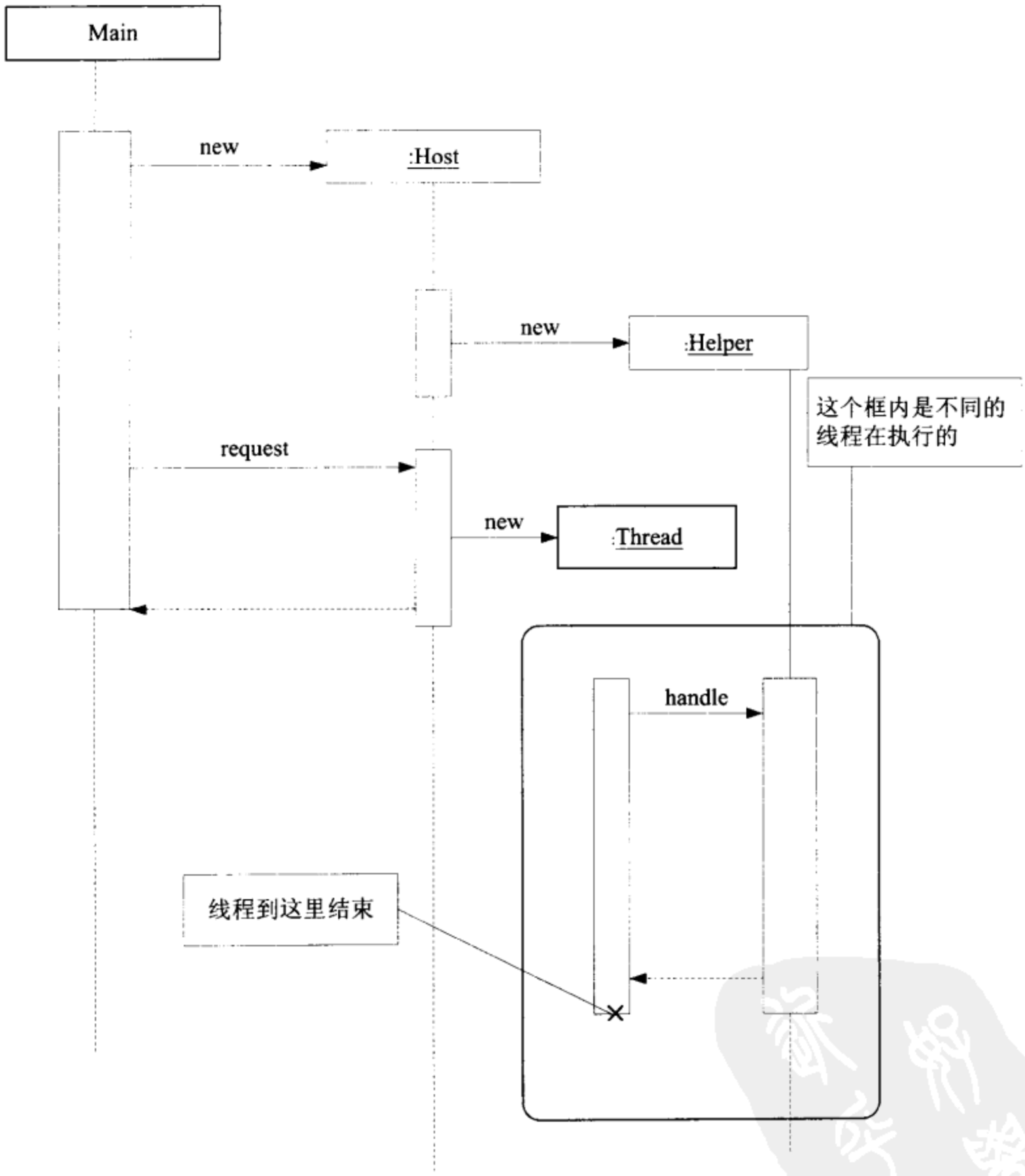


图 7-3　范例程序的顺序图

范例程序的时序图则如图 7-4 所示。从图中可以看得出来，在 Host 类里会产生出新的线程。

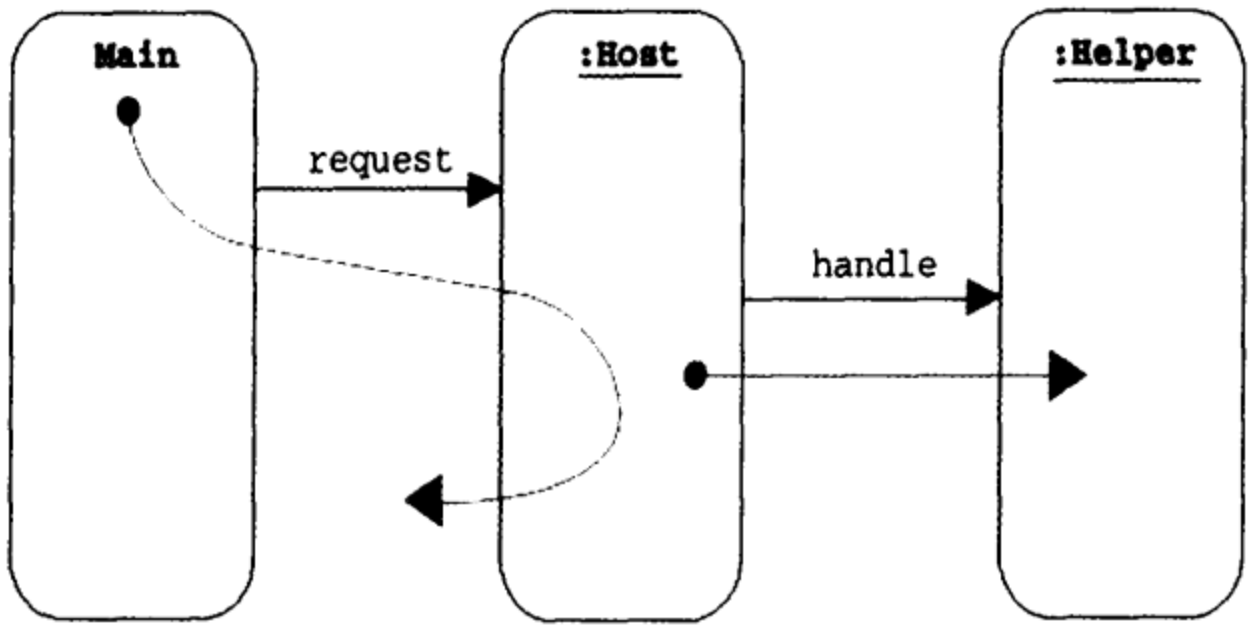


图 7-4　范例程序的时序图

另外，如果在 Host 类内不产生新的线程，而以同一个线程调用 Helper 类的话，时序图则如图 7-5 所示。比较图 7-4、图 7-5，应可以明显看出之间的差异。图 7-5 的程序会出现怎样的结果，我们留在本章末的练习问题 7-2 实现。

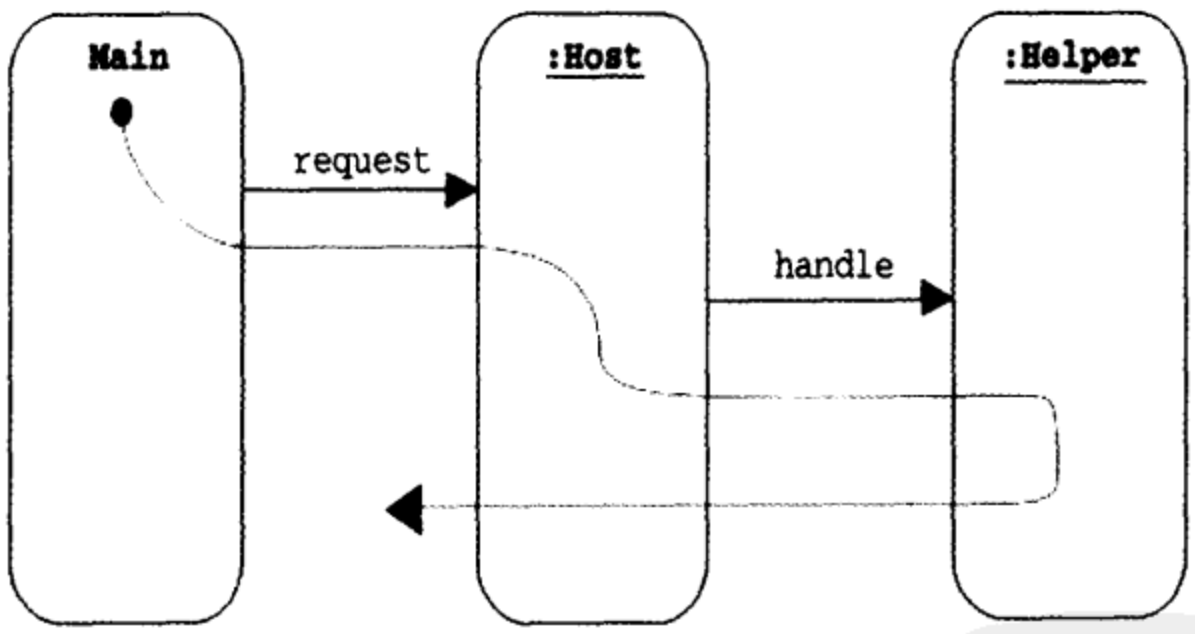


图 7-5　Host 里不建立新线程时的时序图

Thread-Per-Message Pattern 的所有参与者

Thread-Per-Message Pattern 的所有参与者如下：

◆ *Client（委托人）参与者*

Client 参与者会对 Host 参与者送出请求（request）。Client 参与者并不知道 Host 参与者会如何实现这个请求。范例程序中，Client 参与者是 Main 类。

◆ *Host 参与者*

当 Host 参与者收到 Client 参与者的请求（request），会建立新的线程并启动它。这个新的线程，会使用 Helper 参与者，“处理（handle）”这个请求。范例程序中，Host 参与者是 Host 类。

◆ *Helper（帮助者）参与者*

Helper 参与者会对 Host 参与者提供处理（handle）请求的功能。Host 参与者所建立的线程，会使用 Helper 参与者。范例程序中，Helper 参与者是 Helper 类（见图 7-6）。

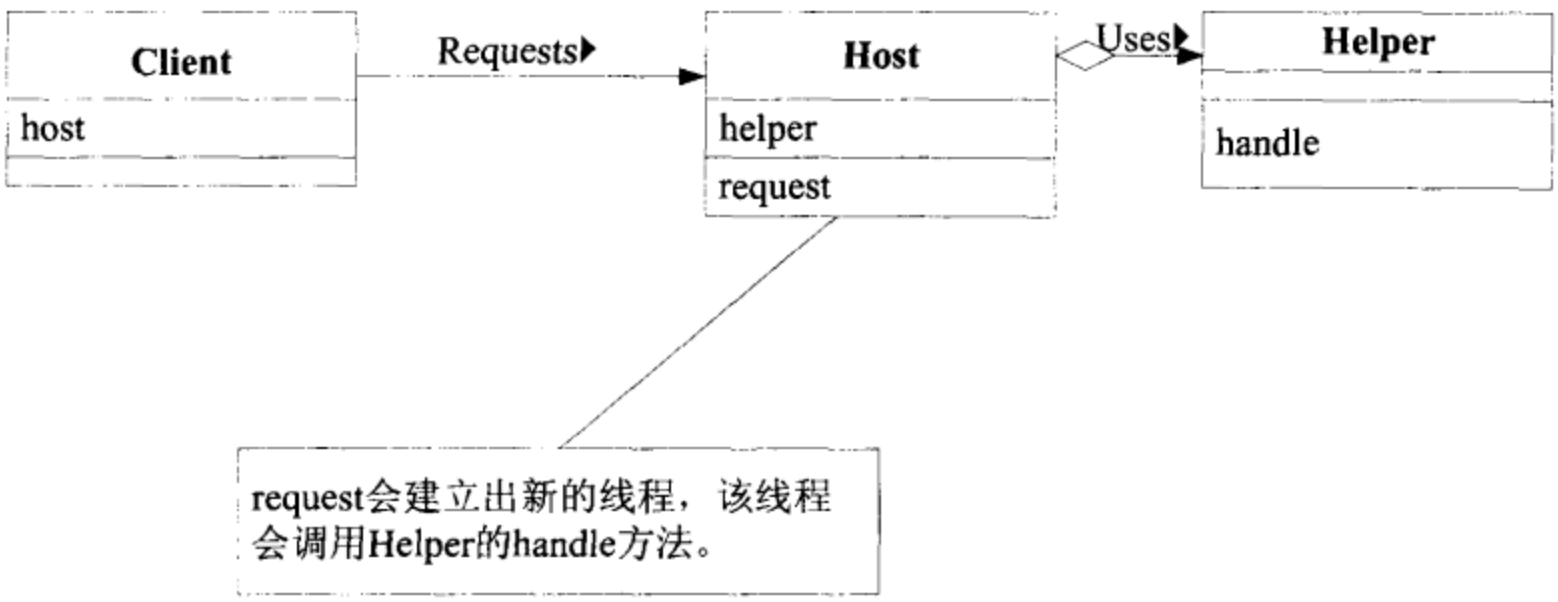


图 7-6　Thread-Per-Message Pattern 的类图

扩展思考方向的提示

提升响应性，降低延迟时间

使用 Thread-Per-Message Pattern，Host 参与者对 Client 参与者的响应性会提高，延迟时间会下降。尤其是当 handle 的操作很花时间时，或是 handle 的操作需要等待输出入（I/O）时，效果特别明显。为了感受到响应性，本章末的练习问题 7-5 会实际思考 GUI 应用程序的范例。

使用 Thread-Per-Message Pattern 时，在 Host 参与者里，会启动新的线程。因为启动线程需要花一些时间，所以对希望用来提升响应性的 Thread-Per-Message Pattern 而言，“handle 的操作需要花的时间”与“启动线程需要花的时间”是鱼与熊掌不能兼得的关系。

为了降低启动线程所需的时间，可使用 Worker Pattern（第 8 章）。

适合在操作顺序无所谓时使用

Thread-Per-Message Pattern 中，handle 方法执行的顺序，并不一定是调用 request 方法的顺序。所以操作顺序有意义时，不适合使用 Thread-Per-Message Pattern。

不需要返回值的时候

Thread-Per-Message Pattern 中，request 方法不会等待 handle 方法执行结束。也就是说，request 方法拿不到 handle 方法的返回值。所以，Thread-Per-Message Pattern 只能在不需要返回值的时候使用。例如，通知事件的发生等等。

需要得知处理结果时，可使用 Future Pattern（第 9 章）。

应用在服务器的制作

为了使服务器可以处理多数的请求，可以使用 Thread-Per-Message Pattern。客户端送达的请求，由主线程来接收。而实际处理该请求，则交给其他线程负责，服务器的主线程回到继续等待其他客户端请求的状态。本章末的练习问题 7-6，就要来实现一个使用 Thread-Per-Message Pattern 的迷你 Web 服务器。

调用方法＋启动线程→传送消息

通常调用出普通方法时，会执行完方法里的所有操作，控制权才会回来。Thread-Per-Message Pattern 里，因为 request 也是普通的方法，所以也是执行完 request 方法的内容，控制权才回来。但请思考“request 方法有进行真正想要作的操作（显示字符串）了吗？”并不是从 request 方法回来，就表示字符串显示完了。request 是期待的操作开始处理的触发器（Trigger），但不会等待到执行结束。

这个事实我们该怎么描述呢？的确“在 request 里启动了新的线程”是正确的原因，但这不是惟一的说法。请注意在这里利用了调用方法与启动线程两个操作，实现“传送异步消息”的操作[注2]。这是比较宏观一点观察程序的解释。

在之前的章节，也有出现过这样的观点。例如请回想 Guarded Suspension Pattern（第 3 章），那个 Pattern 也曾视为“多线程版的 if”。当时是使用 while、wait 与 notifyAll 来实现出“多线程版的 if”。

而 Read-Write Lock Pattern（第 6 章）也出现了相似的行为。因为 synchronized

注 2：硬要说的话，或许可以当他它是“多线程版的 Proxy Pattern”或“多线程版的 Adapter Pattern”。关于 Proxy Pattern 与 Adapter Pattern，请参考附录 E 的[GoF]与[Yuki01]。

的“物理性”锁定欠缺可伸缩性，所以我们使用 synchronized，自己实现设计出“读访问的锁定与写入用的锁定”。

▶▶ **补充说明：Pattern 与程序语言**

本书中，以 Java 语言为题材来介绍与多线程有关的 Pattern。于是，思考程序的时候，都被 Java 语言所绑住了。但是，Pattern 本来是不依存于特定语言的。本书所介绍的程序，充其量只能代表“以 Java 语言要如何实现”的范例罢了。我们是利用程序语言提供的功能，来实现出 Pattern。

若是依存于特定语言的，通常不称之 Pattern，而称为 idiom（惯用句）。

相关 Pattern

Future Pattern（第 9 章）

Thread-Per-Message Pattern，会启动新的线程来进行处理。但送出请求的线程不会等待实际的操作完成，故无法获取处理的结果。

Future Pattern 会事先建立用来存放返回值的实例（Future 参与者）。当处理实际操作的线程结束时，会将返回值写入 Future 参与者。当送出请求的线程需要用到返回值时，才从 Future 参与者读取。

Worker Thread Pattern（第 8 章）

Thread-Per-Message Pattern 中，每次提出请求时启动一条线程。

Worker Thread Pattern 则预先启动足够数量的线程，重复使用这些线程，提升程序的执行性能。

进阶说明：进程与线程

这本书最前面（Introduction1“Java 语言的线程”）中，曾说过线程是“执行程序的主体”。但操作系统（OS）的进程（process），也可以说是执行程序的主体。这个进阶说明，就要探讨进程与线程的差异。

进程与线程之间的关系，会因为平台的差异（OS、硬件），有极大的不同。即使在相同的平台，进程与线程之间的关系也会因为 Java 虚拟机（Java Virtual Machine，JVM）的实现方式而有所不同。不过一般而言，我们可以说一个进程里可建立多条的线程。

◆ 线程间的内存是共享的

进程与线程最大的差异在于内存能否共享。

通常每个进程所拥有的内存空间是各自独立的。进程不能擅自读取、改写其他进程的内存空间。因为进程的内存空间是各自独立的，所以进程无须担心被其他进程破坏的危险。

线程则是共享内存的[注3]。所以我们经常会使用一条线程在内存上写入数据，而由其他的线程来读取。在这里所说“共享相同的内存”，在 Java 中就是“共享相同的实例”。Java 的实例存放在内存上，而多条线程可以同时对这个实例读写。

因为线程间的内存是共享的，所以线程之间的沟通可以使用很自然、简单的方式做到。而因为同一个实例可由多个线程同时访问，所以需要正确地进行共享互斥。

◆ 线程间的 *context-switch* 较容易

进程与线程另一个差异，在于 context-switch 的繁重程度。

要切换执行中的进程时，进程必须将现在自己的状态（context 信息）存储下来，并将下一个要开始执行的进程以前所保留的 context 数据读回来。这个信息的切换操作（context switch）需要花费一些时间。

切换执行中的线程时，线程与进程一样，需要进行 context-switch 的操作。然而，线程所管理的 context 信息比进程要来的少[注4]，一般而言线程的 context-switch 比进程的 context-switch 快得多。

所以，需要紧密地进行多项相关的工作时，线程通常会比进程来的实用。

重点回顾

这一章我们学习了 Thread-Per-Message Pattern。

Client 参与者会调用 Host 参与者的 request 方法送出请求。实际处理该请求的是 Helper 参与者的 handle 方法。但如果 Client 参与者调用 request 方法，而在 request 方法里又调用 handle 方法，在实际的操作处理结束之前不退出 handle 方法，结果等于也不会退出 request 方法的话，会使 request 的响应性降低。

于是，我们在 Host 参与者里启动一个新的线程来处理这个请求。而由这个新的线程来调用 handle 方法。这么一来，送出请求的线程就可以很快地从 request 方法里退出。

注 3：Java 的内存模型中，将内存分为主存储器与工作内存两种。可让线程共享的，只有主存储器的部分，详见附录 B。

注 4：这只能说是一般概况。线程与进程实际的关系，与 Java 执行环境的实现关系性很高。

也就是我们使用了这种方法，来实现了送出异步的消息。

这就是 Thread-Per-Message Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 7-1（基础知识的确认）

Java 请就您对范例程序的认知，阅读下列语句中，正确的请回答○，不正确的请回答×。

（1）每次调用出 request 方法时，都会启动新的线程。

（2）每次调用出 request 方法时，都会生成 Helper 类的实例。

（3）如果没有从 request 方法回来，就不会调用出 handle 方法。

（4）调用出 handle 方法，显示出文字的是主线程。

（5）slowly 方法中，延长 sleep 时间的话，从 request 方法回来的时间也会延长。

问题 7-2（不使用 Thread-Per-Message Pattern）

Java 请把范例程序改写成不使用 Thread-Per-Message Pattern 的版本。也就是说主线程会连实际的操作（handle）一起进行。并请将这个程序的执行结果与范例程序的执行结果（见图 7-2）加以比较。

问题 7-3（启动线程）

Java 某个人改写了范例程序的 Host 类（List 7-2）如 List 7-4 所示。这时执行结果会有何改变？ 为什么？

List 7-4 Host 类（Host.java）

```
public class Host {
    private final Helper helper = new Helper();
    public void request(final int count, final char c) {
        System.out.println("    request(" + count + ", " + c + ")
BEGIN");
        new Thread() {
            public void run() {
                helper.handle(count, c);
            }
        }.run();
        System.out.println("    request(" + count + ", " + c + ") END");
    }
}
```

问题 7-4（不使用匿名内部类）

Java 请改写范例程序的 Host 类（List 7-2）为不使用匿名内部类的版本。

提示：请尝试建立一个新的 HelperThread 类用来取代匿名内部类。

问题 7-5（提升响应性）

Java List 7-5～7-7 的程序，可以用来显示出图 7-7 般的 GUI 应用程序。单击右侧的[Execute]按钮时，Swing 的 framework 会去调用 actionPerformed 方法，在 actionPerformed 方法里，会调用 Service 类的 service 方法。

然而，因为 service 方法很花时间，这么一来要从 actionPerformed 方法退出，得等上一些时间。这会使按钮的反应、应用程序对用户的反应都慢下来。请改写 Service 类（List 7-7），提高这个类的响应性。

提示：请考虑用户连续单击按钮的情况。

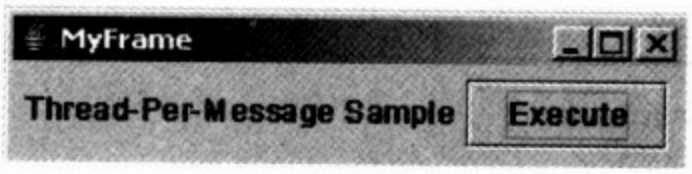


图 7-7　拥有[Execute]钮的 GUI 应用程序

List 7-5　Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

List 7-6　MyFrame 类（MyFrame.java）

```
import java.io.IOException;
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;

public class MyFrame extends JFrame implements ActionListener {
    public MyFrame() {
        super("MyFrame");
        getContentPane().setLayout(new FlowLayout());
```

```
        getContentPane().add(new JLabel("Thread-Per-Message Sample"));
        JButton button = new JButton("Execute");
        getContentPane().add(button);
        button.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        Service.service();
    }
}
```

List 7-7 Service 类（Service.java）

```
public class Service {
    public static void service() {
        System.out.print("service");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("done.");
    }
}
```

Java 问题 7-6（制作服务器）

List 7-8～List 7-10 的程序是一个会从 10 开始，每隔约 1 秒倒数 1，直到 0 为止的 Web 服务器。执行范例如图 7-8 所示[注5]。

这个服务器是以单线程运行的，所以同时只能应付一个 Web 浏览器。对一个浏览器倒数的 10 秒之间，其他浏览器都会被要求等待。

请使用 Thread-Per-Message Pattern，将它修改为可以同时应付多个 Web 浏览器

注 5：这里无视 Web 浏览器（客户端）传来的 HTTP 请求，而机械性地对客户端吐出 HTML 数据。

的服务器。

执行要按下面的步骤进行：

（1）编译

```
javac Main.java
```

（2）执行

```
java Main
```

（3）在同一台计算机打开两个 Web 浏览器

（4）在一个 Web 浏览器中，访问 http://127.0.0.1:8888/，程序会开始倒数

（5）还没倒数完之前，若以另一个浏览器窗口也访问 http://127.0.0.1:8888/，目前的程序，会需要等另一边倒数完才会开始

List 7-8 Main 类（Main.java）

```
import java.io.IOException;

public class Main {
    public static void main(String args[]) {
        try {
            new MiniServer(8888).execute();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

List 7-9 MiniServer 类（MiniServer.java）

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.IOException;

public class MiniServer {
    private final int portnumber;
    public MiniServer(int portnumber) {
        this.portnumber = portnumber;
    }
    public void execute() throws IOException {
        ServerSocket serverSocket = new ServerSocket(portnumber);
        System.out.println("Listening on " + serverSocket);
        try {
            while (true) {
```

```
                System.out.println("Accepting...");
                Socket clientSocket = serverSocket.accept();
                System.out.println("Connected to " + clientSocket);
                try {
                    Service.service(clientSocket);
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {

            e.printStackTrace();
        } finally {
            serverSocket.close();
        }
    }
}
```

List 7-10 Service 类（Service.java）

```
import java.net.Socket;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.DataOutputStream;

public class Service {
    private Service() {
    }
    public  static  void  service(Socket  clientSocket)  throws
IOException {
        System.out.println(Thread.currentThread().getName() + ":
Service.service(" + clientSocket + ") BEGIN");
        try {
            DataOutputStream out = new DataOutputStream(clientSocket.
getOutputStream());
            out.writeBytes("HTTP/1.0 200 OK\r\n");
            out.writeBytes("Content-type: text/html\r\n");
            out.writeBytes("\r\n");

out.writeBytes("<html><head><title>Countdown</title></head><body>");
            out.writeBytes("<h1>Countdown start!</h1>");
```

```
            for (int i = 10; i >= 0; i--) {
                System.out.println(Thread.currentThread()
                  .getName() + ": Countdown i = " + i);
                out.writeBytes("<h1>" + i + "</h1>");
                out.flush();
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
            out.writeBytes("</body></html>");
        } finally {
            clientSocket.close();
        }
        System.out.println(Thread.currentThread().getName() + ":
Service.service(" + clientSocket + ") END");
    }
}
```



图 7-8　执行范例（请在本地执行，以 Web 浏览器读取 http://127.0.0.1:8888/）

问题 7-7

Java 这个问题用来检验您对多线程的理解。Blackhole 类（List 7-11）具有 enter 与 magic 这两个类方法。enter 方法已经写好了，而 magic 方法还没写。

在 enter 方法中，有用来显示 Step 1、Step 2、Step 3 等字符串的语句。在显示 Step 1、Step 2 之间，调用了 magic 方法。

而现在的目标是调用 enter 方法时，会显示出

```
Step 1
Step 2
```

而再来的

```
Step 3
```

则不显示出来。请尝试写出 magic 方法，达到上面期待的程序行为。不过，在此不可以在 magic 方法里显示字符串。

执行程序使用的 Main 类如 List 7-12 所示，期待的执行结果如图 7-9 所示。

List 7-11 Blackhole 类（Blackhole.java）

```
public class Blackhole {
    public static void enter(Object obj) {
        System.out.println("Step 1");
        magic(obj);
        System.out.println("Step 2");
        synchronized (obj) {
          System.out.println("Step 3(never reached here)");//不会执行这里
        }
    }

    (magic 方法请您来完成)
}
```

List 7-12 Main 类（Main.java）

```
public class Main {
    public static void main(String args[]) {
        System.out.println("BEGIN");
        Object obj = new Object();
        Blackhole.enter(obj);
        System.out.println("END");
    }
}
```

```
BEGIN
Step 1
Step 2
(在这里停止不动。CTRL+C 结束程序)
```

图 7-9　执行结果

第8章

Worker Thread
——等到工作来，来了就工作

Worker Thread Pattern

这是某个工厂的故事。在这里，工人要组装塑料玩具模型。

工作的委托人，会将很多塑料玩具模型盒子搬去工厂，堆放在桌上。

工人必须逐个组装收到的塑料玩具模型。工人首先会去拿桌上堆放的模型盒子，并阅读盒内的说明书，并将模型组装起来。做完工作的工人，就去拿下一个盒子来组合。如果没有模型盒子了，就等待新的模型盒子送来……。

这一章要学习的是 Worker Thread Pattern。

worker 是“工人”的意思。Worker Thread Pattern 中，工人线程（worker thread）会依次抓一件工作来处理。当没有工作可作时，工人线程会停下来等待新的工作过来。

Worker Thread 也有人称为 Background Thread（背景线程）。另外，也有人把视点放在管理工人线程的地方，称之为 Thread Pool。

范例程序

接下来就来看 Worker Thread Pattern 的范例程序吧。程序里出现的类如表 8-1 所示。范例程序的操作如下：

ClientThread 类的线程，会对 Channel 类送出工作的请求（说是工作，在这里只是要求显示出请求者的名称与请求编号而已）。

Channel 类的实例管理着 5 个工人线程（WorkerThread）。每个工人线程都等待收到请求。

当工作的请求来临时，工人线程会从 Chaneel 里抓一件请求来处理。处理结束后，就回到 Channel 里，等待下一个工作啊。

表 8-1　类一览表

名　　称	说　　明
Main	操作测试用线程
ClientThread	用来送出工作请求的线程
Request	代表工作请求的类
Channel	接受工作请求，传给工人线程的类
WorkerThread	表示工人线程的类

范例程序的类图和顺序图，如图 8-1 和图 8-2 所示。

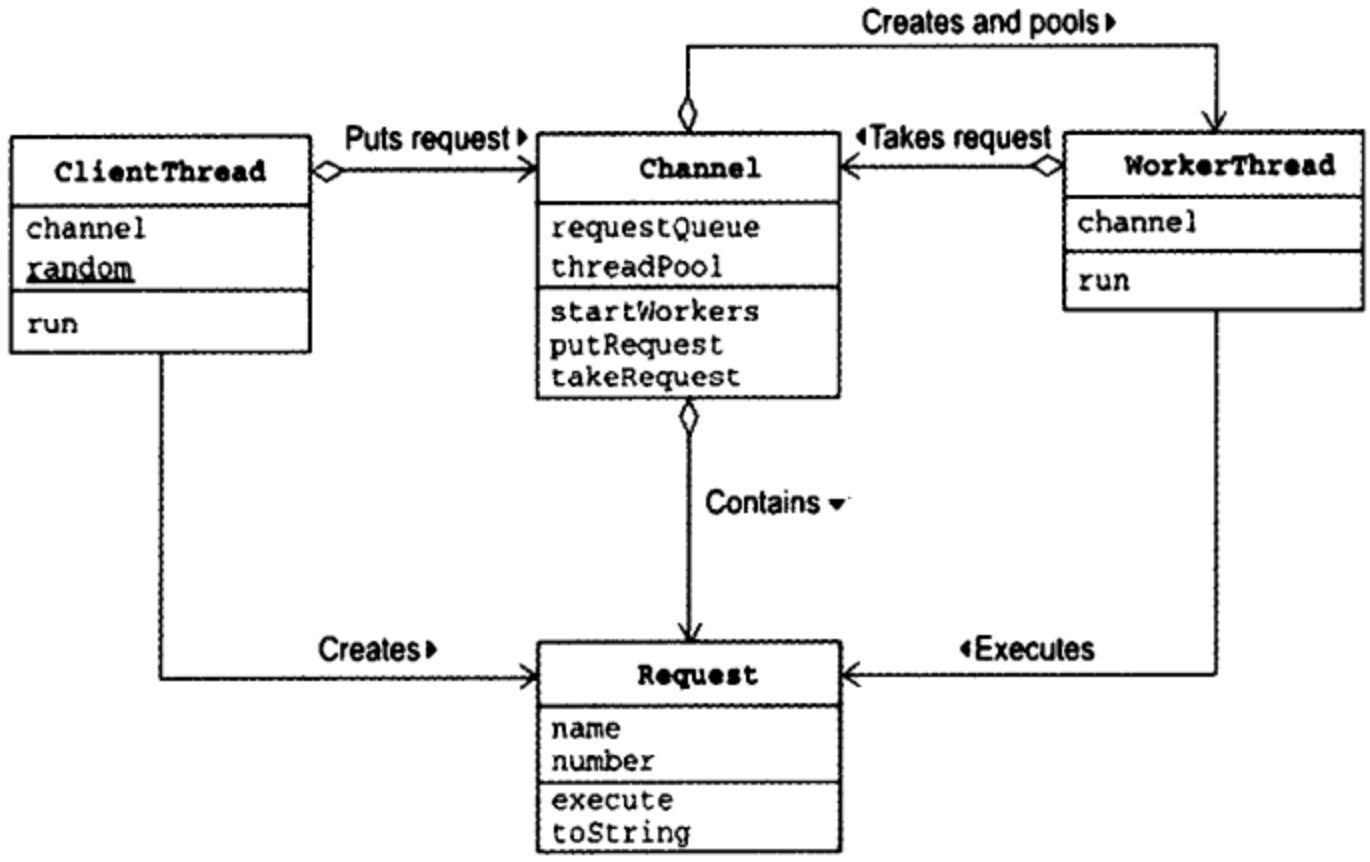


图 8-1　范例程序的类图

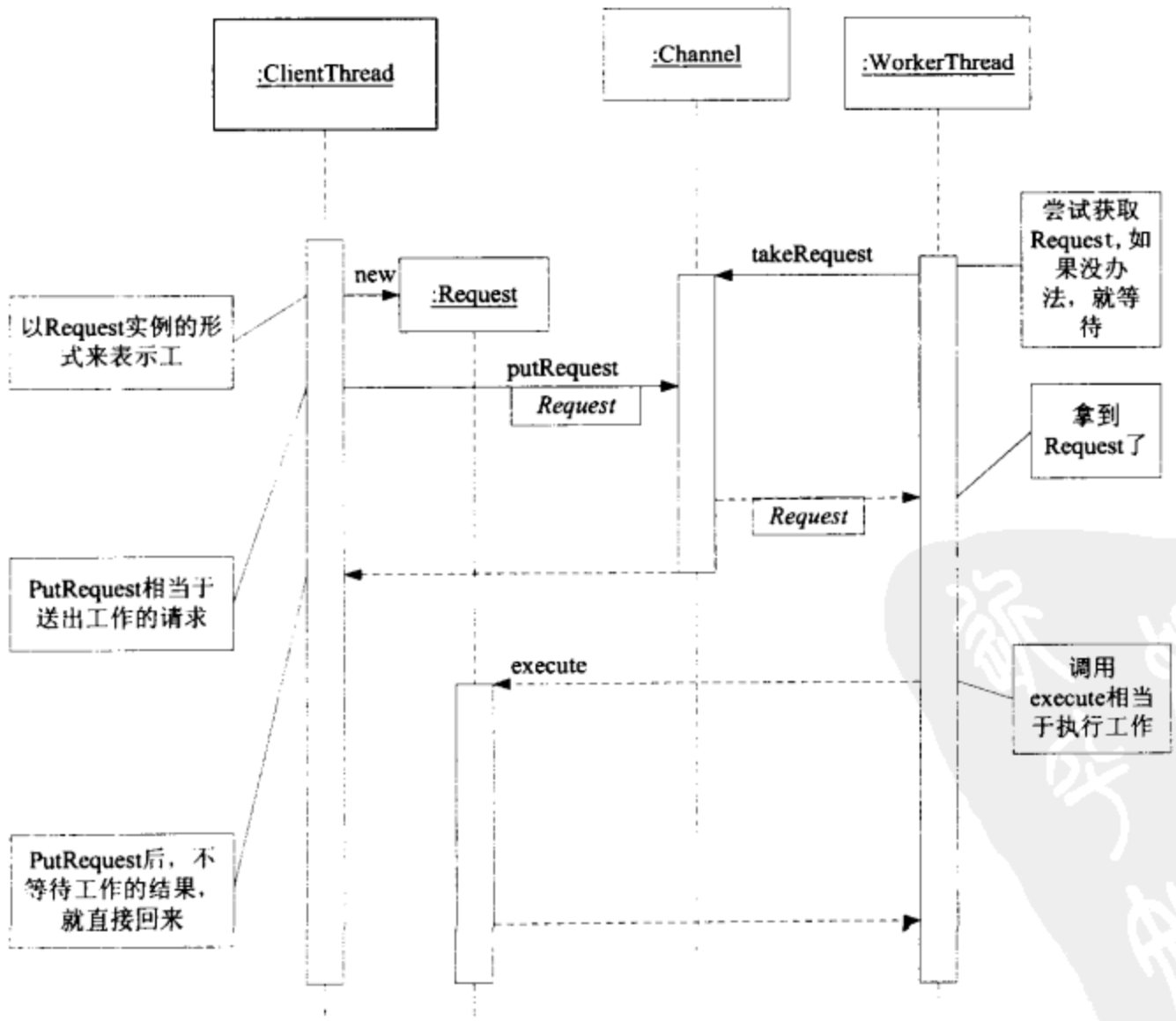


图 8-2　范例程序的顺序图

Main 类

Main 类（List 8-1）会建立 1 个拥有 5 条工人线程的 Channel 实例。并让 3 个 ClientThread 实例（Alice、Bobby、Chris）共享这个 Channel 实例。

List 8-1 测试用的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Channel channel = new Channel(5);   // 工人线程的数量
        channel.startWorkers();
        new ClientThread("Alice", channel).start();
        new ClientThread("Bobby", channel).start();
        new ClientThread("Chris", channel).start();
    }
}
```

ClientThread 类

ClientThread 类（List 8-2）用来送出工作请求。“送出工作请求”的行为，在范例程序中，对应到这样的操作：

- 建立 Request 实例；
- 将这个实例传给 Channel 类的 putRequest 方法。

为了使程序操作有些变化，程序使用了随机数 sleep 一些时间。

List 8-2 用来送出工作请求的 ClientThread 类（ClientThread.java）

```
import java.util.Random;

public class ClientThread extends Thread {
    private final Channel channel;
    private static final Random random = new Random();
    public ClientThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        try {
            for (int i = 0; true; i++) {
                Request request = new Request(getName(), i);
```

```
                channel.putRequest(request);
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Request 类

Request 类（List 8-3）用来表示工作请求。

name 字段是请求者的名称，number 字段是请求编号。这个范例程序中，name 的名字会是 Alice、Bobby、Chris 其中之一。number 的值则是 0、1、2……。execute 方法定义了这个请求的“操作”。说是操作，在范例程序里只是显示出线程名称（委托者与号码）而已。为了模拟这个“操作”需要花点时间，所以用了 sleep。

List 8-3 表示工作请求的 Request 类（Request.java）

```
import java.util.Random;

public class Request {
    private final String name; // 委托者
    private final int number;  // 请求编号
    private static final Random random = new Random();
    public Request(String name, int number) {
        this.name = name;
        this.number = number;
    }
    public void execute() {
        System.out.println(Thread.currentThread().getName() + "
executes " + this);
        try {
            Thread.sleep(random.nextInt(1000));
        } catch (InterruptedException e) {
        }
    }
    public String toString() {
```

```
        return "[ Request from " + name + " No." + number + " ]";
    }
}
```

Channel 类

Channel 类（List 8-4）可用来接受、传送工作请求，并保存工人线程。

Channel 类拥有 requestQueue 字段，用来接受、传送工作请求。这个字段会以队列的方式存储请求。要将请求放进队列里，必须使用 putRequest 方法，要从队列取出请求时，则要使用 takeRequest 方法。在这里用到了 Producer-Consumer Pattern（第 5 章），而 putRequest 方法与 takeRequest 方法的实现上，还用到了 Guarded Suspension Pattern（第 3 章）。

Channel 类还具有 threadPool 字段，用来存储工人线程。threadPool 是 WorkerThread 的数组。构造器会初始化 threadPool，并建立出 WorkerThread 的实例。数组的大小由 thread 定义。

工人线程各自命名成 Worker-0、Worker-1、Worker-2、……。

startWorkers 方法用来启动所有的工人线程。

List 8-4　接受工作请求、保存工人线程的 Channel 类（Channel.java）

```
public class Channel {
    private static final int MAX_REQUEST = 100;
    private final Request[] requestQueue;
    private int tail;   // 下一个 putRequest 的地方
    private int head;   // 下一个 takeRequest 的地方
    private int count;  // Request 的数量

    private final WorkerThread[] threadPool;

    public Channel(int threads) {
        this.requestQueue = new Request[MAX_REQUEST];
        this.head = 0;
        this.tail = 0;
        this.count = 0;

        threadPool = new WorkerThread[threads];
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i] = new WorkerThread("Worker-" + i, this);
```

```
    }
}
public void startWorkers() {
    for (int i = 0; i < threadPool.length; i++) {
        threadPool[i].start();
    }
}
public synchronized void putRequest(Request request) {
    while (count >= requestQueue.length) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    requestQueue[tail] = request;
    tail = (tail + 1) % requestQueue.length;
    count++;
    notifyAll();
}
    public synchronized Request takeRequest() {
        while (count <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        Request request = requestQueue[head];
        head = (head + 1) % requestQueue.length;
        count--;
        notifyAll();
        return request;
    }
}
```

WorkerThread 类

WorkerThread 类（List 8-5）代表工人线程。

工人线程可以执行工作。“执行工作”的操作，在这个范例程序中是以下面的方式做到的：

- 使用 takeRequest 方法，从 Channel 的实例获取 1 个 Request 的实例；
- 调用这个实例的 execute 方法。

工人线程启动后，就会永远不停地持续工作。也就是说不断重复“获取新的 Request 实例，并调用 execute 方法”。

Thread-Per-Message Pattern（第 7 章）中，是在每次执行工作的时候启动新的线程。但在 Worker Thread Pattern 中，工人线程会不断地重复执行新工作，所以不需要启动新的线程。

WorkerThread 的字段只有一个 channel（Channel 的实例），用来获取工作请求让自己进行。WorkerThread 并不知道请求具体的内容（显示字符串的工作），WorkerThread 只知道“Reqeust 类有 execute 这个方法”。

List 8-5 工人线程 WorkerThread（WorkerThread.java）

```
public class WorkerThread extends Thread {
    private final Channel channel;
    public WorkerThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        while (true) {
            Request request = channel.takeRequest();
            request.execute();
        }
    }
}
```

执行范例程序，可得到类似图 8-3 的结果。Worker-0～Worker-4 等 5 个 WorkerThread，会分担执行 Alice、Bobby、Chris 等 3 个 ClientThread 传来的请求。

送出请求的 ClientThread 与执行请求的 WorkerThread 之间并没有固定的关系。Alice 送出的请求 No.0 虽然是 Worker-0 执行的，但同样是 Alice 所送出的 No.1，却是 Worker-3 执行的，No.2 则是 Worker-2 执行的。

工人线程不会理会请求是谁送出的，只管执行收到的请求。

```
Worker-0 executes [ Request from Alice No.0 ]   Worker-0执行Alice的请求No.0
Worker-1 executes [ Request from Bobby No.0 ]   Worker-1执行Bobby的请求No.0
Worker-2 executes [ Request from Chris No.0 ]   Worker-2执行Chris的请求No.0
Worker-3 executes [ Request from Alice No.1 ]   Worker-3执行Alice的请求No.1
Worker-4 executes [ Request from Bobby No.1 ]   Worker-4执行Bobby的请求No.1
Worker-1 executes [ Request from Chris No.1 ]   Worker-1执行Chris的请求No.1
Worker-1 executes [ Request from Bobby No.2 ]   Worker-1执行Bobby的请求No.2
Worker-2 executes [ Request from Alice No.2 ]   Worker-2执行Alice的请求No.2
(CTRL+C 结束程序)
```

图 8-3　执行范例

Worker Thread Pattern 的所有参与者

Worker Thread Pattern 的所有参与者如下：

◆ *Client（委托人）参与者*

Client 参与者会建立 Request 参与者，传给 Channel 参与者。范例程序中，Client 参与者是 ClientThread 类。

◆ *Channel（通路）参与者*

Channel 参与者会从 Client 参与者获取 Request 参与者，传给 Worker 参与者。范例程序中，Channel 参与者是 Channel 类。

◆ *Worker（工人）参与者*

Worker 参与者会从 Channel 参与者获取 Request 参与者，并执行这份工作。当工作结束以后，会去拿取下一个 Request 参与者。范例程序中，Worker 参与者是 WorkerThread 类。

◆ *Request（请求）参与者*

Request 参与者用来表示工作。Request 参与者会存放执行这份工作所需要的数据。范例程序中，Request 参与者是 Request 类。

Worker Thread Pattern 的类图如图 8-4 和时序图如图 8-5 所示。

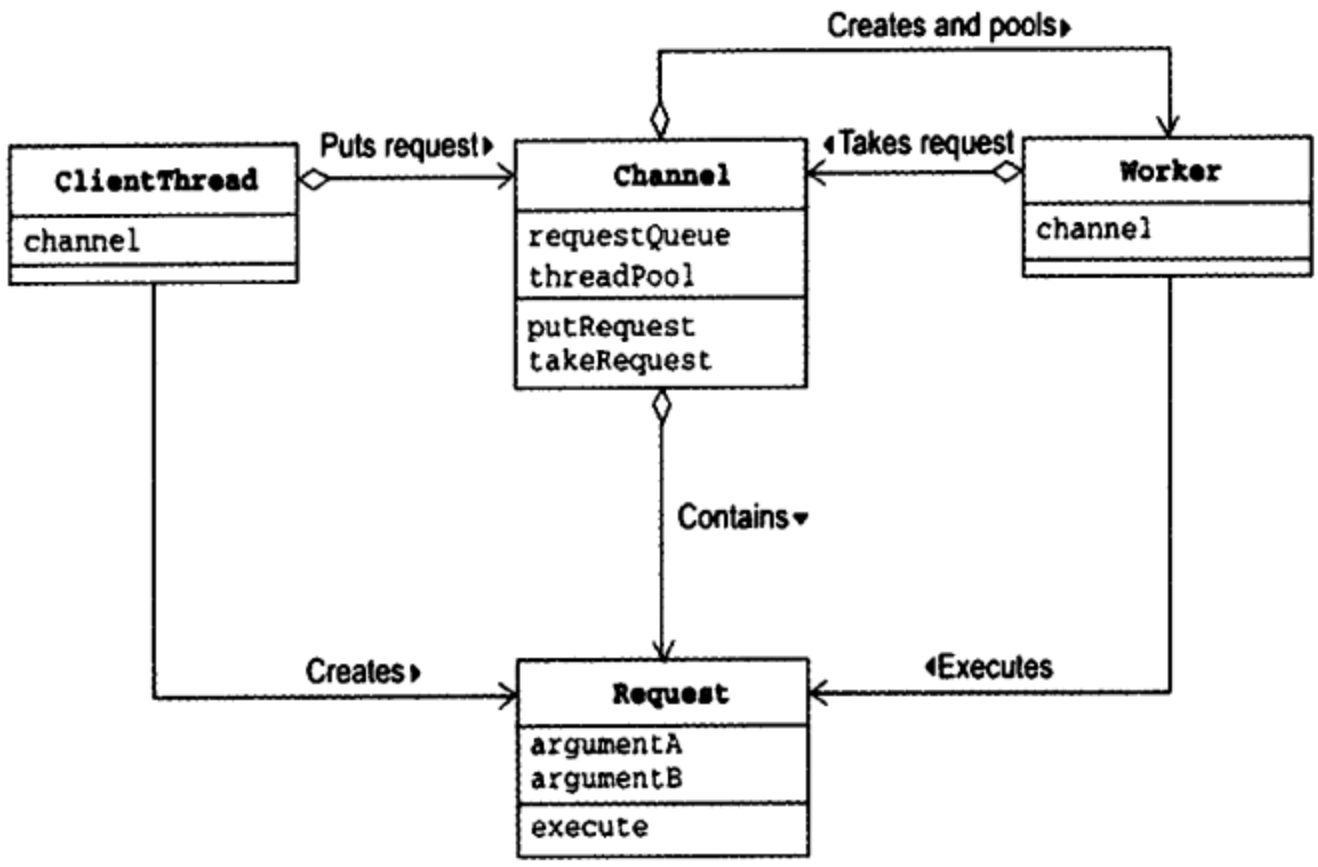


图 8-4　Worker Thread Pattern 的类图

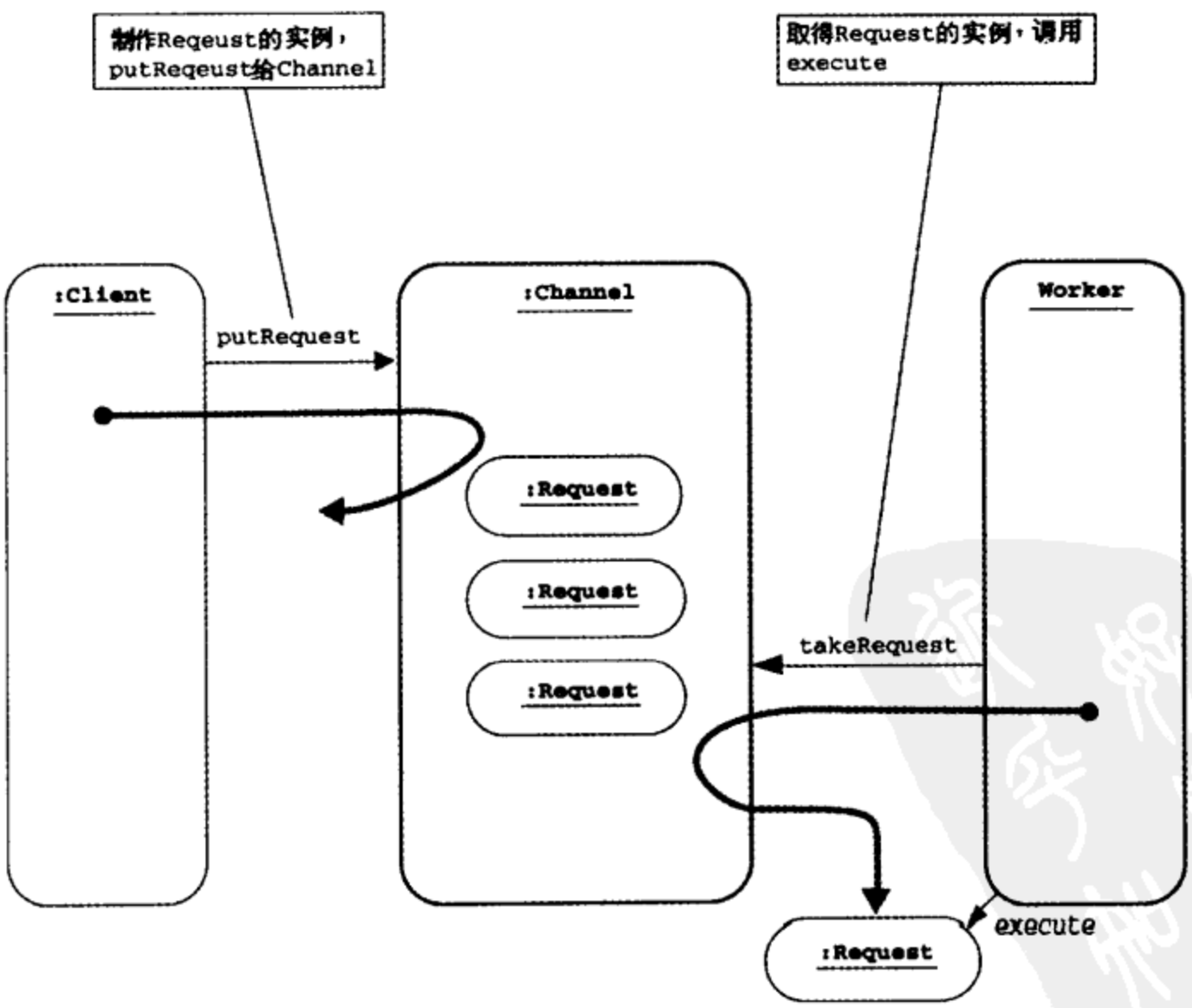


图 8-5　Worker Thread Pattern 的时序图

扩展思考方向的提示

启动线程是繁重的操作

如果可以把自己的工作交给别人做，自己就可以去做其他事情。线程也一样。如果可以把工作交给其他线程，自己就可以继续前进到下一个工作。这是Thread-Per-Message Pattern（第7章）的主题。

可是，启动线程是很花费时间的工作。所以，Worker Thread将重复使用线程、资源再利用也作为主题之一。

控制承载量

Worker Thread还有一个主题。就是承载量（capacity：容量、同时可以提供的服务量）的控制。

◆ *Worker参与者的数量*

就如范例程序所示，Worker参与者的数量可以自由的设置。范例程序中，Channel的构造器参数threads就是设置工人线程数量。会建立出threads个WorkerThread的实例。

提高Worker参与者的数量，可以提高并发处理的工作量。但如果准备的Worker参与者数量比同时间的工作量来的多，有些Worker参与者也派不上用场。多余的Worker参与者既不工作，只会占用内存。提高承载量，会增加消耗的资源。所以有必要配合软件实际的需要，调整Worker参与者的数量。

Worker参与者的数量，并不一定要在程序启动时决定，也可以像下面这样机动性变化：

- 最先从某个一定量的Worker参与者开始；
- 当工作量增加时，增加Worker参与者；
- 不过增加太多会用完内存，所以到达某个上限要停止增加；
- 相反地，工作减少时（也就是待命中的Worker参与者增加时），就结束掉一些Worker参与者。

◆ *Request参与者的数量*

Channel参与者中存放了Request参与者。当Worker参与者能不断一件接一件完成工作时，Channel里“堆放的”Request参与者理应不会大幅增加。但如果进来了一些超过Worker参与者处理能力的工作时，Channel参与者中的Request参与者

可能就会塞满。当 Channel 参与者被塞满时，要传送新的 Request 参与者给 Channel 参与者的 Client 参与者反而需要停下来等待了。以范例程序来说，就是 Channel 类的 putRequest 里的线程 wait 了。

增加 Channel 参与者可存放的 Request 参与者数，可缓冲 Client 参与者与 Worker 参与者的处理速度差。可是，如果存储太多 Request，也会占用大量的内存资源。所以又陷入了容量与资源之间鱼与熊掌的关系。这个部分也是 Producer-Consumer Pattern（第 5 章）出现过的课题。

invocation 与 execution 的分离

接下来我们在这里尝试比较 Worker Thread Pattern 的“工作请求”，与普通的“调用方法”有什么不同。

Client 参与者会送出工作的请求。工作的内容以 Request 的形式包装起来，传给 Channel 参与者。这个部分，对普通的调用方法来说，就是“评估自变量、启动方法”的操作。“评估自变量”的操作对应到“建立 Request 参与者”，“启动方法”的操作则大致对应到“传给 Channel 参与者”。

另一方面，Worker 参与者会进行工作。使用从 Channel 参与者拿来的 Request 参与者，进行实际的操作。这个部分，对普通的调用方法来说，可以对应到“执行方法”的部分。

普通的调用方法操作，“启动方法”与“执行方法”是连续进行的。当方法一调用，就会马上继续执行。普通的调用方法操作中，启动与执行是密不可分的。

但 Worker Thread Pattern 与 Thread-Per-Message Pattern（第 7 章）中，我们刻意将方法的启动与执行分开。启动方法的操作叫做 invocation（动词是 invoke），执行方法的操作则叫做 execution（动词是 execute）。Worker Thread Pattern 与 Thread-Per-Message Pattern，都可说是将方法的 invocation 与 execution 分离。关于 invocation 与 execution 的分离，也是 Command Pattern（参考附录 E [GoF][Yuki01]）的主题。

那么，我们要讨论的是，分离 invocation 与 execution 这件事又具有什么意义。

◆ 提高响应性

如果 invocation 与 execution 无法分离，当 execution 很花时间时，invocation 的操作也会受到牵累。但如果将 invocation 与 execution 事先分离，即使 execution 花时间，invocation 可以继续自己前进。这样能提高程序的响应性。

◆ 控制实行顺序

如果 invocation 与 execution 无法分离，一旦 invoke 出来，就必须直接把它 execute 完。但，如果将 invocation 与 execution 分离，execute 的顺序就可以与 invoke 的次

序无关。也就是说，我们可以对 Request 参与者设立优先性，控制 Channel 参与者传递 Request 参与者给 Worker 参与者的顺序。

◆ 可取消和可重复执行

若能分离 invocation 与 execution，就有办法做到“虽然 invocation 了，但将 execution 取消”的功能。

因为 invoke 后会产生 Request 参与者这个对象，只要把 Request 参与者保存下来，就可以做到重复 execute。

◆ 分散处理的第一步

因为 invocation 与 execution 分离了，所以 invoke 与 execute 的操作也较容易拆开在两部计算机上执行。相当于 Request 参与者的对象，可通过网络传送到另一台计算机。

Runnable 接口的意义

java.lang.Runnable 接口，常常用来作为 Worker Thread Pattern 的 Request 参与者。也就是说，建立一个实现 Runnable 接口的类的实例，作为表达工作内容的对象，并将它传给 Channel 参与者，“这件工作拜托你啰”。

“实现 Runnable 接口的类的实例”这种说法太长了，所以下面简称为“Runnable 对象”。

我们曾经在“启动线程”的地方学过使用 Runnable 接口的方法。只要将 Runnable 对象传进 Thread 类的构造器里，这个新启动的线程，就会调用 Runnable 对象的 run 方法。但 Runnable 接口的使用方式不止这一种。

Runnable 对象，可以作为方法的自变量传递、堆到队列里、通过网络传递、甚至存进文件里。而这样一个 Runnable 对象可以经过多次传来传去，最后传到某台计算机的某条线程上，才真正交付执行。

这时，Runnable 接口就可以看作是 GoF 的 Command Pattern（参考附录 E [GoF][Yuki01]）中的 Command 参与者。这一章的进阶说明中将要说明的 Swing 的 invokeLater 方法，就可说是这个实例。将想要进行的工作包装成 Runnable 对象的形式，再传给 invokeLater。

多态的 Request 参与者

范例程序中，ClientThread 所传给 Channel 的，是 Request 的实例。但 WorkerThread 并不知道 Request 类的详细内容。WorkerThread 只不过单纯地接收 Request 的实例，调用 execute 方法而已。

也就是说，如果我们建立 Request 类的子类，并将它的实例传给 Channel，

WorkerThread 也能正确调用这个实例的 execute 方法。以面向对象的用词来说，就是使用了多态（polymorphism）的特征。

Request 类用来表示工作，我们增加一个 Request 类的子类，就可以增加工作的种类。也就是我们可以建立出多态的（polymorphic）Request 参与者。

执行工作时所需要的信息，都定义在 Request 参与者里。所以即使建立出多态的 Request 参与者，增加工作的种类，Channel 参与者与 Worker 参与者都不需要修改。即使增加工作的种类，Worker 参与者都只是调用 Request 参与者的 execute 方法而已。

图 8-6 是建立多态的 Request 参与者时的类图。

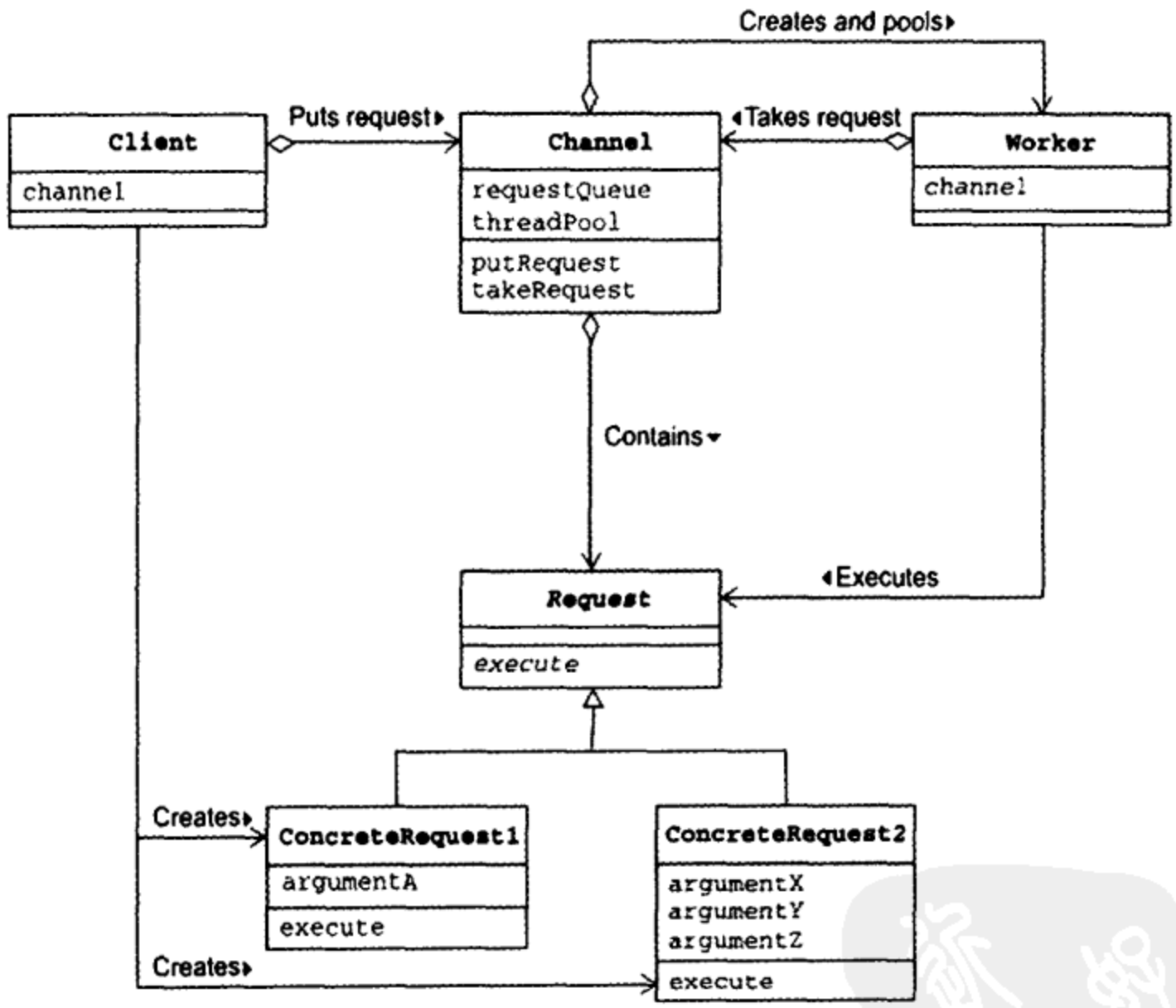


图 8-6　使用多态的 Request 参与者时的类图

只有一条 Worker 参与者

如果工人线程（Worker 参与者）只有一条，会发生什么情况呢？ 如同我们曾在 Producer-Consumer Pattern 时所说的，只有一条工人线程的话，工人线程处理的范围就变成单线程，而可以省略共享互斥的必要了。在本章的进阶说明会再详述。

相关 Pattern

Producer-Consumer Pattern（第 5 章）

Worker Thread Pattern 中传递 Request 参与者的部分，使用到了 Producer-Consumer Pattern。Worker Thread Pattern 的 Client 参与者与 Worker 参与者，分别对应到 Producer-Consumer Pattern 的 Producer Pattern 与 Consumer 参与者。

Worker Thread Pattern 的 Channel 参与者，对应到 Producer-Consumer Pattern 的 Channel 参与者。

Thread-Per-Message Pattern（第 7 章）

Thread-Per-Message Pattern 会将工作交付给新建立的线程，以提高程序的响应性。但会使程序需要耗费启动线程所需要的时间。

Worker Thread Pattern 与 Thread-Per-Message Pattern 一样，会将工作交给别的线程。但线程并不是每次重新启动，而是事先启动着，以工人线程的形式等待工作。这样可以节省启动线程所需的时间。

Command Pattern（参考附录 E [GoF][Yuki01]）

在 Worker Thread Pattern 中，Worker 参与者接受工作进行作业的部分，使用到了 Command Pattern。Worker Thread Pattern 的 Request 参与者，可对应到 Command Pattern 的 Command 参与者。

Future Pattern（第 9 章）

Client 参与者想要获取 Worker 参与者的执行结果时，需要使用到 Future Pattern。

Flyweight Pattern（参考附录 E [GoF][Yuki01]）

Flyweight Pattern 中，通过共享实例来节省内存。

Worker Thread Pattern 中则通过共享线程来节省启动线程所需的时间。不过即使是共享线程，也不表示同时将多件工作交给一条工人线程进行。所以描述成工人线程的“再利用”、“资源回收”或“重复使用”会比较贴切。当工人线程完成一件工作后，可以前进到其他工作。

Flyweight Pattern 则必须注意共享的实例所具有的信息。应该共享的信息

（intrinsic 的信息）可以存放在实例里，但不该共享的信息（extrinsic 的信息）就不该放到实例中。

Worker Thread Pattern 也要注意一样的问题。工人线程中，不应该拥有个别工作独有的信息。工作独有的信息应该放在 Request 参与者里。

Thread-Specific Storage Pattern（第 11 章）

Thread-Specific Storage Pattern 用来确保住线程独有的领域。

工人线程中，没有存放每件工作独有的信息，所以对工人线程不能使用 Thread-Specific Storage Pattern。

Active Object Pattern（第 12 章）

Worker Thread Pattern 有作为 Active Object Pattern 的一部分使用。

进阶说明：Swing 的 Event-Dispatching Thread

何谓 Event-Dispatching Thread

使用 Swing 建立 GUI 应用程序时，需要意识到(the) event-dispatching thread（事件分配线程）的运行。

使用 GUI 应用程序时，我们可以自由地点选按钮、移动鼠标光标等等。而 GUI 应用程序必须处理我们进行的这些操作。这里“点选按钮”、“移动鼠标光标”等现象我们称之为事件（event），以 Swing 的类的形式来表达。例如单击按钮，会产生 java.awt.event.ActionEvent 这个类的实例，移动鼠标的话，会产生 java.awt.event.MouseEvent 类的实例。这些实例都存放在 Swing 内部的事件队列（event queue）里。

看到这里为止，观察敏锐的读者可能已经在想，是不是 Swing 的“事件”正好对应到 Request 参与者，而“事件队列”对应到 Channel 参与者呢？正是如此。单击按钮、移动鼠标这些现象，在 Swing 内部会转换成名为“事件”的 Request 参与者，传给名为“事件队列”的 Channel 参与者。Client 参与者则是对应到管理鼠标、键盘等事件来源的部分，这个被 Swing 内部隐藏起来，从外面看不出来。

如果前面所说的您都了解，那要懂 event-dispatching thread 就不难了。因为 event-dispatching thread 就是工人线程（Worker 参与者）。

Event-dispatching thread 会从事件队列里取出一个事件，并执行这个事件。执行结束后，再回来事件队列，接着取出下一个事件。就一直不断重复这样的工

作。如果事件队列里没有任何事件，event-dispatching thread 就会停下来等待事件进来。

这正是 Worker Thread Pattern。

Event-dispatching thread 只有一条

Event-dispatching thread 有很多名称。它也被称作 the event dispatch thread，或是更精简的 the event thread。但无论是哪个名字，在英文里它总是以“定冠词 the＋单数形式”表达，这是因为(the) event-dispatching thread 在系统中是“独一无二”的。

不知读者是否还记得在 Producer-Consumer Pattern（第 5 章）中，我们曾经讨论过只有一个 Consumer 的情况。Worker Thread Pattern 的 Worker 参与者，就相当于 Producer-Consumer Pattern 的 Consumer 参与者。event-dispatching thread 只有一条，就是说工人线程（Worker 参与者）只有一个人。

Worker 参与者只有一条时，乍看之下会丧失多线程的优点，但是可以省略共享互斥的问题，反而带来极大的优点。因为 Swing 的 event-dispatching thread 设计成只有一条，所以 event-dispatching thread 所调用的方法，都不需要为工人线程进行共享互斥。

Event-dispatching thread 会调用 Listener

我们已经了解到 Event-dispatching thread 会去执行事件，接下来我们要具体思考事件要怎么“执行”。

Event-dispatching thread 进行的操作之一，是调用各种 Listener（收听者、监听器）的方法。

例如，单击按钮的时候，java.awt.event.ActionEvent 类的实例会被放进事件队列里。当 Event-dispatching thread 获取这个实例，会去调用“用来处理 ActionEvent 实例的对象”（Listener）的 actionPerformed 方法。Event-dispatching thread 其实并不知道这个方法里会做些什么事，event-dispatching thread 只是单纯地去调用 actionPerformed 方法而已。

而移动鼠标时呢？这个时候 java.awt.event.MouseEvent 类的实例会被塞进事件队列。Event-dispatching thread 获取这个实例要执行时，会调用用来处理 MouseEvent 的实例的对象（Listener）的 mouseMoved 方法。与 actionPerformed 一样，Event-dispatching thread 并不知道 mouseMoved 方法里具体的操作内容。

Event-dispatching thread 的顺序图和时序图，如图 8-7 和图 8-8 所示。

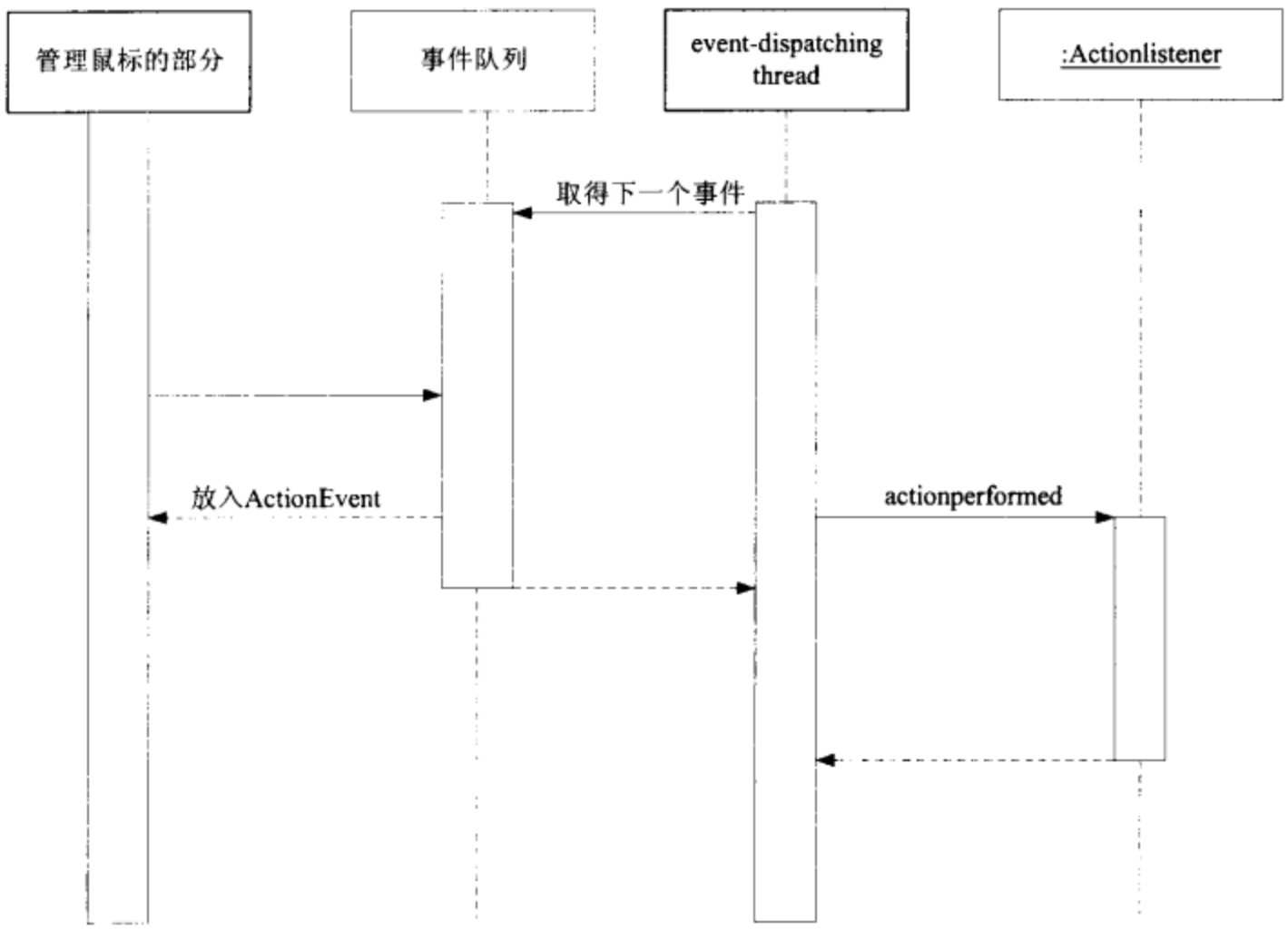


图 8-7　Event-dispatching thread 的顺序图（单击按钮时）

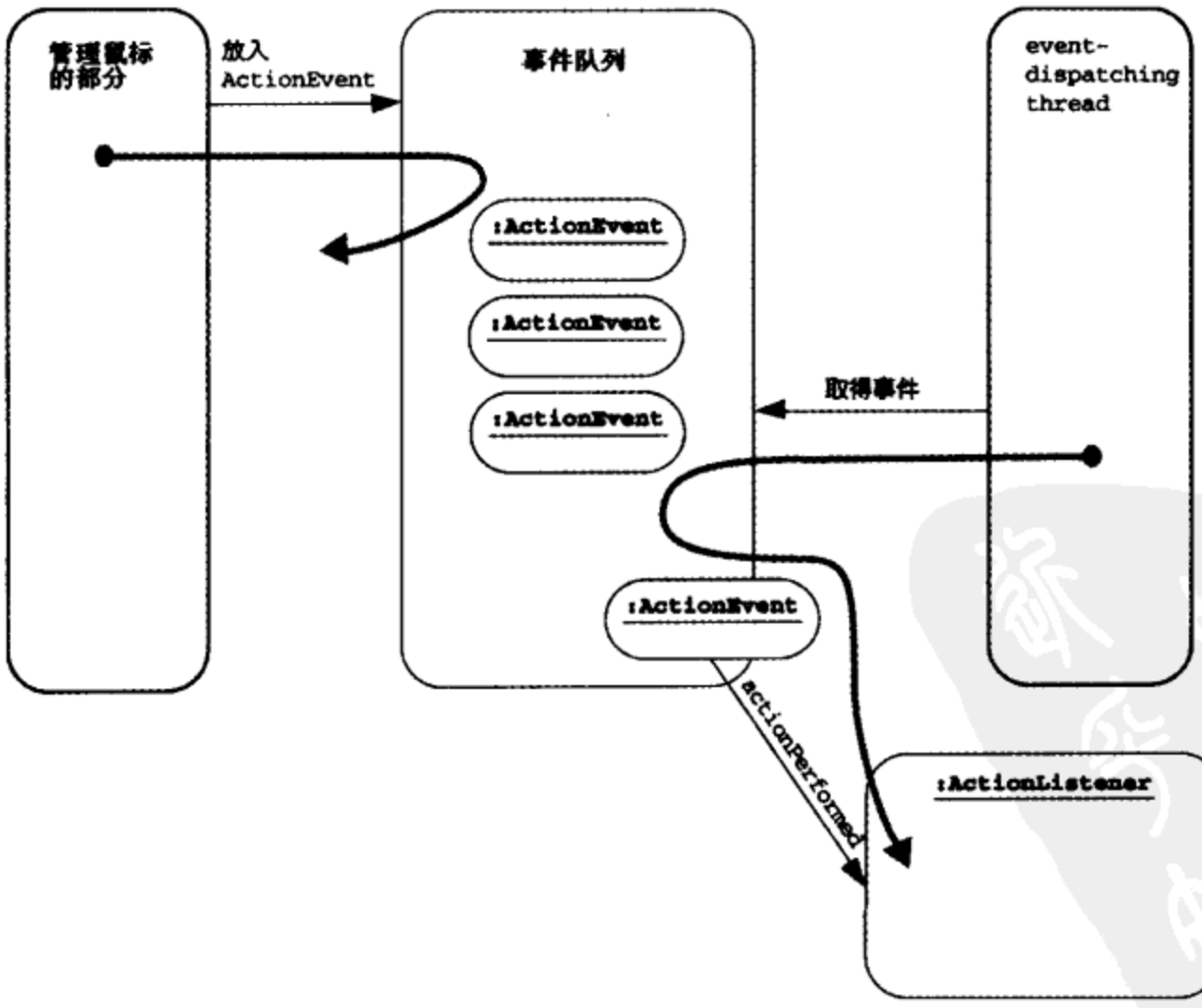


图 8-8　Event-dispatching thread 的时序图（单击按钮时）

注册 Listener 的意义

有用 Swing 撰写程序的人，应该都有写过按钮（JButton）这些组件（JComponent）“注册 Listener”的经验吧。具体来说，就是调用 addActionListener 方法或 addMouseListener 方法的操作。可能有很多人都并不清楚原因，而只是记住“反正就是要这样做”，而机械地作注册 Listener 的操作。

如果您了解前面所说的 Event-dispatching thread，那应该也能了解“注册 Listener”的意义。对组件注册 Listener，其实就是对组件设置当事件发生时，“Event-dispatching thread 所要调用的方法所在的实例”。

Event-dispatching thread 也处理画面的重绘

Event-dispatching thread 除了调用 Listener 的方法以外，还会调用画面重绘用的方法（update 或 paint）。

当我们想要重绘画面的时候，会去调用 repaint 方法。但调用 repaint 方法，其实并不会“马上”开始重绘。repaint 方法只会在内部记录“要重绘的区域”，实际上，重绘的操作还是由 Event-dispatching thread 另外处理的。

javax.swing.SwingUtilities 类

javax.swing.SwingUtilities 类中，收集了 Swing 相关的许多方便的方法。在这里介绍其中 3 个与 Event-dispatching thread 有关的方法（见表 8-2）。

表 8-2　javax.swing.SwingUtilities 类的方便的方法

名称	static void invokeAndWait(Runnable runnable) throws InterruptedException, InvocationTargetException
说明	由 Event-dispatching thread 执行 runnable.run()。当目前囤积的所有事件处理完，runnable.run()也执行完以后，控制权才会从 invokeAndWait()方法交回来。也就是说，runnable.run()的操作是同步执行的。当 runnable.run()抛出异常时，invokeAndWait()方法会抛出 InvocationTargetException 异常。runnable.run()所抛出的异常，可使用 InvocationTargetException 的 getTargetException 方法获取
名称	static void invokeLater(Runnable doRun)
说明	由 Event-dispatching thread 执行 runnable.run()。调用这个方法后，控制权会马上从 invokeLater()方法交回来。也就是说，runnable.run()是异步执行的。直到囤积的所有事件都执行完后，才会真正地被执行。通常会用在当 Event-dispatching thread 以外的线程想要访问 GUI 组件时
名称	static boolean isEventDispatchThread()
说明	当现在正执行的这条线程是 Event-dispatching thread，就返回 true，否则返回 false

◆ *invokeAndWait 方法*

invokeAndWait 方法，会执行参数中传入的 Runnable 对象。不过，执行这个对象的是 Event-dispatching thread。也就是说，使用 invodeAndWait 方法，可以将任意操作塞进 Swing 的事件队列。

invokeAndWait 是“启动并等待”的意思。如同其名，这个方法会等待参数中传入的 Runnable 对象执行完毕。也就是说，要等到调用 invokeAndWait 方法的时间点，事件队列中已经存放的所有事件都执行完，并且参数传入的 Runnable 对象的 run 方法也执行完毕以后，才会从 invokeAndWait 方法退出。

为什么需要这种方法呢？ 请回想前面说过“event-dispatching thread 只有一条，所以不需要共享互斥”。大部分的 Swing 的组件都假设只有 event-dispatching thread 这一个线程会对它访问，所以都不是线程安全的。也就是说，我们若以自己建立的线程（Event-dispatching thread 以外的线程）去调用组件的方法，其实是危险的。无论如何都想要调用组件的方法时，就该将要执行的操作的内容建立成 Runnable 对象，使用 invokeAndWait 方法（或是 invokeLater）方法交给 Event-dispatching thread 来调用。

◆ *invokeLater 方法*

invokeLater 方法的操作与 invokeAndWait 方法类似。不过 invokeLater 方法不会等待 Runnable 对象执行结束。只把塞进事件队列，就马上退出。

◆ *isEventDispatchThread 方法*

使用 javax.swing.SwingUtilities 类的 isEventDispatchThread 方法，可以检查现在的线程（调用 isEventDispatchThread 方法的线程）是不是 Event-dispatching thread。

Swing 的单线程规则

在这里要介绍 Swing 的单线程规则（The Single-Thread Rule）。这个守则可为这篇进阶说明所介绍过的内容作一个总结[注1]（见图 8-9）。

> 当 Swing 组件一旦被实现，可能改变组件状态的程序代码、相依于状态的程序代码，都必须交给 Event-dispatching thread 执行。

图 8-9 Swing 的单线程规则

所谓组件被实现（realized），是指组件处在已经可以调用 paint 方法的状态。具

注 1：在这段说明之中所提到的这项守则是引用来自于“Thread and Swing”（http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html）。

体来说，就是这个组件的 setVisible(true)方法、show()方法、pack()方法已经被调用，或是这个组件是已经被实现的组件的子组件。

我们可以简单地说是，当组件还在准备时，由其他的线程调用也没有关系。可是，一旦显示出来以后（设置为可显示以后），组件的方法就只能从 Event-dispatching thread 执行。

使用 Swing 开发程序时，必须遵守这个守则。这个守则是建立在 Event-dispatching thread 只有一条才得以成立的。程序员必须遵守这个守则，作为省去共享互斥、提高程序性能的代价。

另外，有一些方法即使在组件被实现后，仍然可由任何线程调用。表 8-3 是部分举例。

表 8-3　可从任何线程调用的方法举例

repaint
revalidate
addActionListener
removeActionListener
addMouseMotionListener
removeMouseMotionListener
其他可以执行 Listener 的新增及删除的方法全部

重点回顾

这一章我们学习了 Worker Thread Pattern。

使用 Thread-Per-Message Pattern（第 7 章）将“送出工作请求的线程”与“执行工作的线程”分离开来，可提高程序的响应性。

但是，每次送出工作请求时，都为这个工作启动新的线程也很浪费。

所以我们事先启动用来执行工作的线程（工人线程）备用。并使用 Producer-Consumer Pattern（第 5 章），将表示工作内容的实例传递给工人线程。

这么一来，工人线程会负责执行工作，就不需要一直启动新的线程了。

这就是 Worker Thread Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 8-1（基础知识的确认）

Java 请就您对范例程序的认识，阅读下列语句，正确的请回答○，不正确的请回答×。

（1）没有任何 request 的时候，WorkerThread 的线程会一直 sleep 下去。
（2）来自某个 ClientThread 的 request 再被执行(execute)时，来自同一个 ClientThread 的 request 就不会被执行(execute)。
（3）会调用 putRequest 方法的只有 ClientThread。
（4）会调用 takeRequest 方法的只有 WorkerThread。
（5）没有必要将 execute 方法设成 synchronized。

问题 8-2（使用 Thread-Per-Message Pattern 会发生什么事）

Java 请改写范例程序的 Channel 类（List 8-4），使得请求传进来时，就产生一个新的线程来处理请求。

问题 8-3（throughput）

Java 范例程序使用了 Worker Thread Pattern，上一个问题则使用了 Thread-Per-Message Pattern。请测试这两个程序的 throughput（单位时间可完成的处理量）有何差异。

问题 8-4（event-dispatching thread）

Java 我们不可以使用 Event-dispatching thread 来调用 javax.swing.SwingUtilities 类的 invokeAndWait 方法，为什么？

问题 8-5（在 GUI 上的应用）

Java 在这里想要建立一个单击[countUp]按钮后，会每隔一秒依次显示出 0、1、2、……、9 等数字的 GUI 应用程序。

启动 Main 类后，会建立出 MyFrame 的实例并显示出 GUI。单击窗口里显示的按钮后，就会调用 MyFrame 类（List 8-7）的 countUp 方法。在这个方法里，我们使用 label 字段（JLabel 类）的 setText 方法显示出数字，并使用 sleep 休息约 1 秒钟。

实际执行程序后，我们发现控制台（console）画面如图 8-10 所示，每隔约 1 秒就会显示出一行字符串。可是 GUI 的部分却不是 0、1、2、……这样递增，而是在经过约 10 秒后，忽然显示出 9（图 8-11～图 8-13）。

这是为什么呢？应该要如何修改呢？

List 8-6 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()    +
":BEGIN");
        new MyFrame();
        System.out.println(Thread.currentThread().getName()    +
```

```
":END");
    }
}
```

List 8-7 MyFrame 类（MyFrame.java）

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MyFrame extends JFrame implements ActionListener {
    private final JLabel label = new JLabel("Event Dispatching
Thread Sample");
    private final JButton button = new JButton("countUp");
    public MyFrame() {
        super("MyFrame");
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(label);
        getContentPane().add(button);
        button.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button) {
            countUp();
        }
    }
    private void countUp() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName()
          + ":countUp:setText(" + i + ")");
            label.setText("" + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
```

```
            }
        }
    }
}
```

```
main:BEGIN
main:END
AWT-EventQueue-0:countUp:setText(0)←每隔约 1 秒会显示出一行字符串，但
GUI 画面没有变化
AWT-EventQueue-0:countUp:setText(1)
AWT-EventQueue-0:countUp:setText(2)
AWT-EventQueue-0:countUp:setText(3)
AWT-EventQueue-0:countUp:setText(4)
AWT-EventQueue-0:countUp:setText(5)
AWT-EventQueue-0:countUp:setText(6)
AWT-EventQueue-0:countUp:setText(7)
AWT-EventQueue-0:countUp:setText(8)
AWT-EventQueue-0:countUp:setText(9)←执行到这里 GUI 画面终于改变了
```

图 8-10　执行结果



图 8-11　启动后的画面

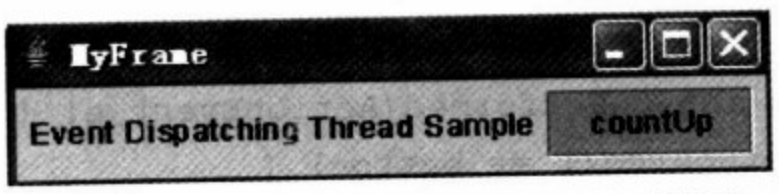


图 8-12　单击[countUp]按钮后的画面（就这样持续 10 秒）

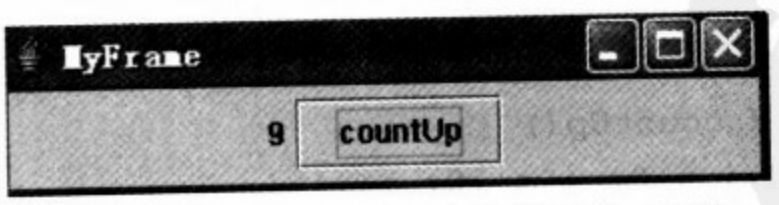


图 8-13　最后的画面（直接显示出“9”）

问题 8-6（结束工人线程）

Java 为了使范例程序能在约 5 秒后自动结束，我们将 Main 类（List 8-1）修改如下（List 8-8）。请配合这个类，进行下列的修改操作。

（1）在 Channel 类（List 8-4）加上 stopAllWorker 方法。这个方法可以用来结

束掉所有 Channel 类所管理的 WorkerThread 的线程。

（2）在 ClientThread 类（List 8-2）加上 stopThread 方法，这个方法可以用来结束掉 ClientThread 的线程。

List 8-8 修改后的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        Channel channel = new Channel(5);   // 工人线程个数
        channel.startWorkers();
        ClientThread alice = new ClientThread("Alice", channel);
        ClientThread bobby = new ClientThread("Bobby", channel);
        ClientThread chris = new ClientThread("Chris", channel);
        alice.start();
        bobby.start();
        chris.start();

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        alice.stopThread();
        bobby.stopThread();
        chris.stopThread();
        channel.stopAllWorkers();
    }
}
```

第9章

Future——先给您这张提货单

Future Pattern

我去蛋糕店买蛋糕。下订单以后，店员请我“请在天黑以后再来店取货”，并给我一张提货单。黄昏后，我就拿着这张提货单到蛋糕店取货。店员说了一声“先生您的蛋糕已经好了”，并把蛋糕交给了我。

这一章要学习的是 Future Patten。

future 是“未来”、“（经济学用语上）期货”的意思。

假设有一个执行起来需要花一些时间的方法，我们就不要等待执行结果出来了，而获取一张替代的“提货单”。因为获取提货单不需要花时间，这时这个“提货单”就是 Future 参与者。

获取 Future 参与者的线程，会在事后再去获取执行结果。就好像拿提货单去领取蛋糕一样。如果已经有执行结果了，就可以马上拿到数据。如果执行结果还没好，则继续等待到执行结果出现为止。

Future 参与者，可说是蛋糕的提货单、剧场的预售票、签名会的句柄，是将“未来”加以现金化的一种契约凭据。

建议读者在阅读本章之前，务必先看过 Thread-Per-Message Pattern（第 7 章）。

范例程序

我们先来看 Future Pattern 的范例程序。

在 Thread-Per-Message Pattern（第 7 章）中，我们已经研究过“收到每个请求时，建立一个新的线程”的范例程序。当时，我们送出的请求像是这样：

```
host.request(10, 'A');
```

只有请求，而不用获取返回值。

而 Future Pattern 中，送出请求后，马上就要获取返回值。也就是像：

```
Data data = host.request(10, 'A');
```

方法是有返回值（data）的。但是，这个返回值 data，并不是程序执行的结果。事实上，用来获取执行结果的程序，现在其实正在其他线程刚开始执行而已。也就是说，这个返回值，不是蛋糕本身，而是提货单（Future 参与者）。

一段时间以后，这时才像

```
data.getContent()
```

这样，使用 data 的 getContent 方法，获取执行结果。这就像是拿提货单去换蛋糕的操作。如果其他线程已经处理完毕了，那调用 getContent 方法的线程就可以马

上退出这个方法。如果还没处理完毕，就要等待到处理好为止。

笔者在这里稍微偷跑了一下，以上就是范例程序所表达的 Future Pattern 运行概要。类与接口的说明如表 9-1 所示。

表 9-1　类与接口一览表

名　　称	说　　明
Main	对 Host 送出请求，获取数据的类
Host	对请求返回 FutureData 实例的类
Data	表达数据访问方式的接口，FutureData 与 RealData 都实现这个接口
FutureData	ReadData 的提货单类，而 RealData 实例则由其他线程建立
RealData	表达实际数据的类，执行构造器需要花一些时间

范例程序的类图和顺序图，如图 9-1 和图 9-2 所示。

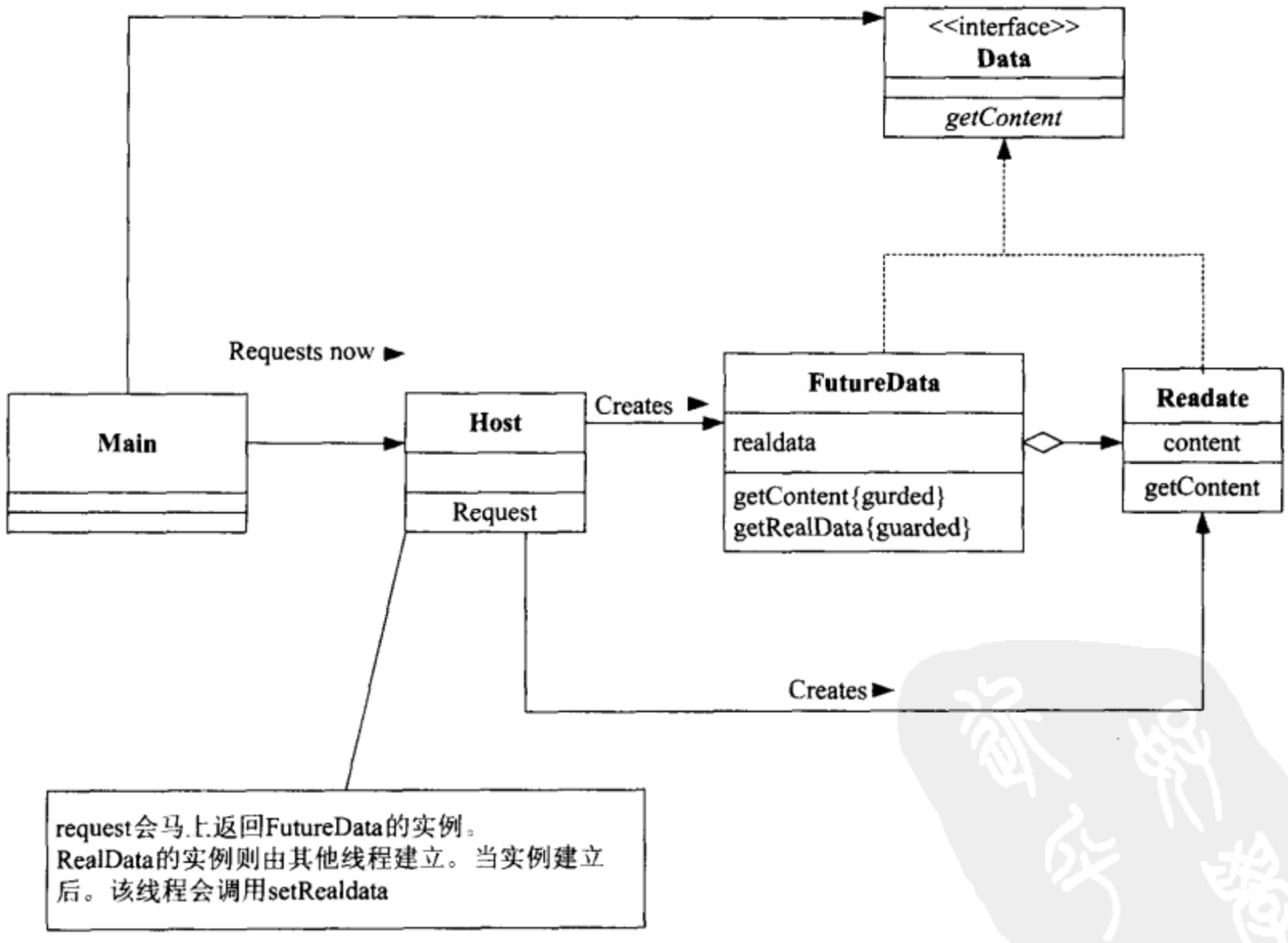


图 9-1　范例程序的类图

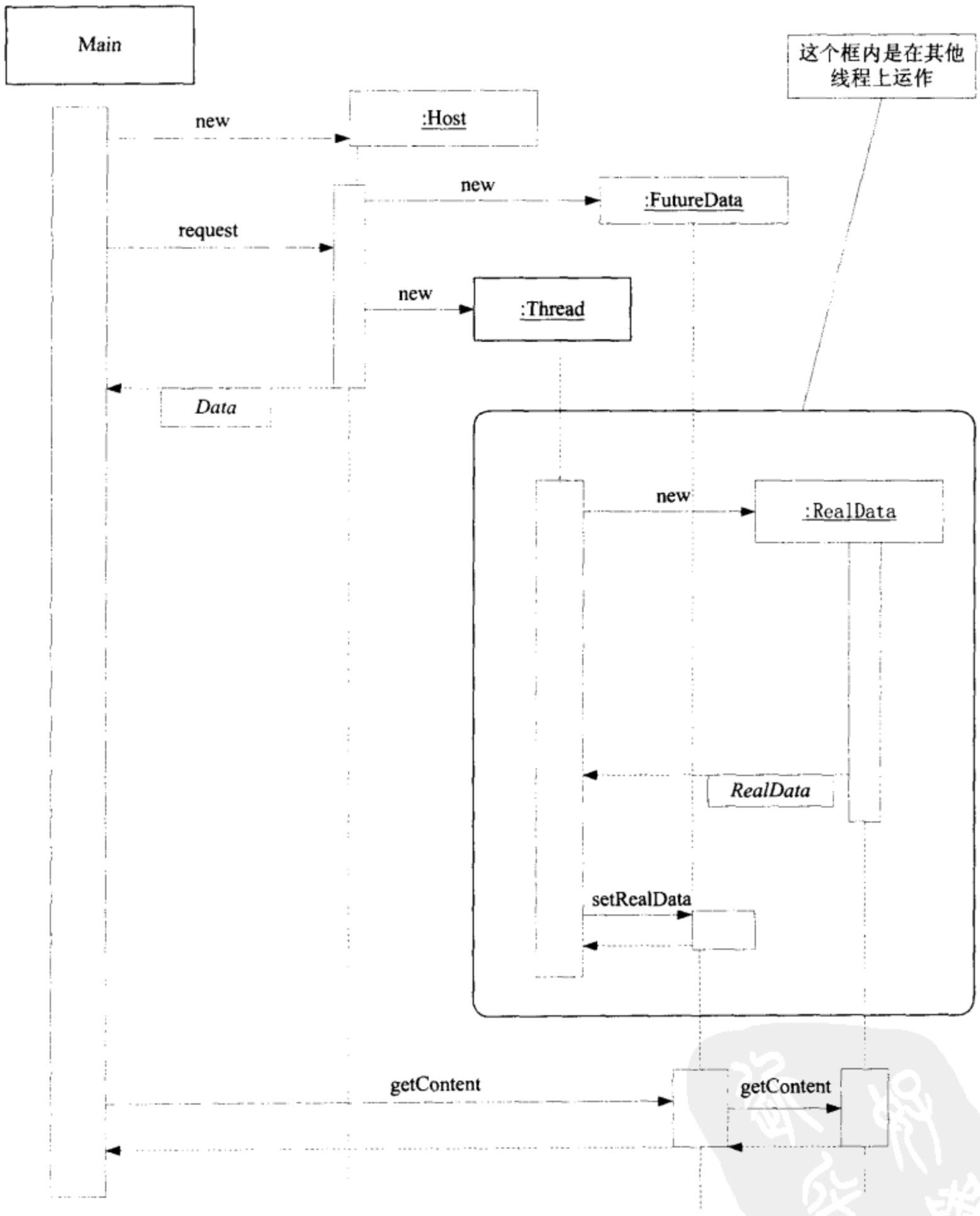


图 9-2　范例程序的顺序图

Main 类

Main 类（List 9-1）会调用 3 次 request 方法。而获取 3 个 Data 类型的返回值（data1、

data2、data3）。这些返回值，实际上是 FutureData 的实例，获取这些实例并不花费时间，也就相当于蛋糕的提货单。所以在这里我们获取了 3 张提货单。

之后，程序 sleep 约 2 秒。在这里是表现程序可以在这个时候继续做一些其他的工作。

之后，再对刚才收到的返回值 data1、data2、data3 使用 getContent 方法，获取我们真正想要的东西（request 方法处理的结果）。也就是拿提货单去领蛋糕。

List 9-1 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("main BEGIN");
        Host host = new Host();
        Data data1 = host.request(10, 'A');
        Data data2 = host.request(20, 'B');
        Data data3 = host.request(30, 'C');

        System.out.println("main otherJob BEGIN");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
        }
        System.out.println("main otherJob END");

        System.out.println("data1 = " + data1.getContent());
        System.out.println("data2 = " + data2.getContent());
        System.out.println("data3 = " + data3.getContent());
        System.out.println("main END");
    }
}
```

Host 类

Host 类（List 9-2）的 request 方法一开始会建立出 FutureData 的实例（future）。这个操作并不特别花费时间。future 实例会成为这个方法的返回值。

接着建立新的线程并启动，在这里产生 RealData 的实例（realdata）。产生出 RealData 的实例需要花费一些时间，不过是交给新的线程进行的。这里所谓“新的线程”，就像是做蛋糕的蛋糕师父。

新的线程会努力地产生出 RealData 实例（realdata）。当 realdata 建立后，就调

用 setRealData 方法，设置给 future。因为建立 realdata 需要花一些时间，所以这个设置操作在一段时间后（未来）才会执行。对 future 设置 realdata，就像把蛋糕准备好，等待客人来领取一样。

调用 request 方法的线程（主线程）启动新的线程以后，便马上退出，返回值则是 future 实例。这就好像是拿到提货单的客人，就不等待蛋糕做好，而马上回家一样。

总之，调用 request 的线程（也就是买蛋糕的客人），作了下面 3 个操作：

（1）建立 FutureData 的实例；

（2）为了建立 RealData 的实例，启动新的线程；

（3）取回 FutureData 实例，作为返回值。

这 3 个操作，（跟建立 RealData 实例的工作相较之下，）都不花费什么时间。所以调用 request 的线程可以马上退出这个方法。

List 9-2 Host 类（Host.java）

```
public class Host {
    public Data request(final int count, final char c) {
        System.out.println("    request(" + count + ", " + c + ")
BEGIN");
        // (1)  建立 FutureData 的实例
        final FutureData future = new FutureData();

        // (2)  为了建立 RealData 的实例，启动新的线程
        new Thread() {
            public void run() {
                RealData realdata = new RealData(count, c);
                future.setRealData(realdata);
            }
        }.start();

        System.out.println("    request(" + count + ", " + c + ")
END");

        // (3)  取回 FutureData 实例，作为返回值
        return future;
    }
}
```

▶▶ 补充说明：参数与局部变量是线程所拥有的

request 方法虽然不是 synchronized 的，但就算从多个线程同时调用，仍是安全的。这是因为 request 里使用的参数（count、c）与局部变量（future）都是调用 request 方法的线程所拥有的，而不是由多个线程所共有的。

▶▶ 补充说明：final 与匿名内部类

request 方法的参数（count、c）与局部变量（future）声明成 final，是为了在匿名内部类里使用。

Data 接口

Data（List 9-3）是用来表现数据的访问方式（getContent 方法）的接口。FutureData 类与 RealData 类都实现这个接口。

List 9-3 Data 接口（Data.java）

```
public interface Data {
    public abstract String getContent();
}
```

FutureData 类

FutureData 类（List 9-4）是表示“提货单”的类。

realdata 字段是用来存放未来将建立好的 RealData 实例。要将 RealData 指定给这个字段，要使用 setRealData 方法。

ready 字段表示是否已经有值赋给 realdata 字段了。当值为 ture 时，表示已经有值赋给 realdata 字段了（蛋糕做好了）。

setRealData 方法用来将 RealData 的实例赋给 realdata 字段。因为这时表示 RealData 的实例已经完成了，所以要将 ready 字段设置为 true，并调用 notifyAll，唤醒所有在 getContent 方法里等待的线程。

setRealData 方法，会被 Host 类的 request 方法中所建立的新线程调用。在这里使用了 Balking Pattern（第 4 章），防止 setRealData 被调用两次以上（这个措施并不是绝对必要的）。

getContent 方法用来获取实际的数据。因为必须等待 setRealData 方法设置 realdata，所以在这里使用 ready 作为防卫条件，使用到了 Guarded Suspension Pattern（第 3 章）。接着后面的

```
return realdata.getContent();
```

语句，则是返回实际的数据。FutureData 类 getContent 方法，将工作委托（delegation）给了 RealData 类的 getContent 方法。

如果 RealData 的实例已经设置了，那就会马上从 getContent 方法退出。如果 RealData 的实例还没有好，那就会 wait 到好了为止。在 wait 状态里的线程，会被 setRealData 方法里调用的 notifyAll 唤醒。

拿到提货单以后，如果马上就去领蛋糕，可能蛋糕还没有做好。这个时候客人还是需要在蛋糕店里等待。而 notifyAll，就是蛋糕店老板口中“让您久等了，这是您的蛋糕”的声音。

List 9-4 FutureData 类（FutureData.java）

```
public class FutureData implements Data {
    private RealData realdata = null;
    private boolean ready = false;
    public synchronized void setRealData(RealData realdata) {
        if (ready) {
            return;     // balk
        }
        this.realdata = realdata;
        this.ready = true;
        notifyAll();
    }
    public synchronized String getContent() {
        while (!ready) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return realdata.getContent();
    }
}
```

RealData 类

RealData 类（List 9-5）是建立出实例需要花点时间的类。构造器的工作是建立出一个字符串，包含 count 个 c 字符。为了表现出方法需要花费一些时间，在这里使用了 sleep。好不容易建立出的 String 实例，则存放在 content 字段里。

getContent 方法做的事，只有将 content 字段的值返回而已。

请注意这个类完全没有用到 synchronized。也就是说，这个类并不处理“等待到实例建立好为止”的线程控制操作。RealData 类并不需要考虑多线程环境。

Future Pattern 是对这种“需要花很多时间的普通类”加上“提货单”机制，使程序多线程化后仍可提高执行性能。

List 9-5 RealData 类（RealData.java）

```
public class RealData implements Data {
    private final String content;
    public RealData(int count, char c) {
        System.out.println("       making RealData(" + count + ",
" + c + ") BEGIN");
        char[] buffer = new char[count];
        for (int i = 0; i < count; i++) {
            buffer[i] = c;
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("       making RealData(" + count + ",
" + c + ") END");
        this.content = new String(buffer);
    }
    public String getContent() {
        return content;
    }
}
```

执行范例如图 9-3 所示。3 次的调用 request 方法的操作都是马上结束，这点与 Thread-Per-Message Pattern（第 7 章）的范例程序是一样的（参考图 7-2）。

不同的是，这个范例程序中，我们还能看到 RealData 的实例建立的状况。当主线程在进行其他操作时，我们可以看到 RealData 的实例一个接一个被建立好。

而接下来主线程调用了 getContent 方法。若 ReadData 已经建立好了，就会马上显示出内容（data1、data2）。如果还没有建立好，就会等待到建立好后，才显示出内容（data3）。

```
main BEGIN                                          ← 主线程开始
    request(10, A) BEGIN                            ← request(10, A)开始
    request(10, A) END                              ← request(10, A)马上结束
    request(20, B) BEGIN                            ← request(20, B)开始
    request(20, B) END                              ← request(20, B)马上结束
    request(30, C) BEGIN                            ← request(30, C)开始
        making RealData(10, A) BEGIN                ← RealData(10, A)开始建立
        making RealData(20, B) BEGIN                ← RealData(20, B)开始建立
   request(30, C) END                               ← request(30, C)马上结束
main otherJob BEGIN                                 ← 主线程进行其他操作，这段时间中...
        making RealData(30, C) BEGIN                ← RealData(30, C)开始建立
        making RealData(10, A) END                  ← RealData(10, A)建立好了
        making RealData(20, B) END                  ← RealData(20, B)建立好了
main otherJob END                                   ← 这时主线程其他操作做完了
data1 = AAAAAAAAAA                                  ← 显示出建立好的 RealData(10, A)
data2 = BBBBBBBBBBBBBBBBBBBB                        ← 显示出建立好的 RealData(20, B)
        making RealData(30, C) END                  ← 好不容易RealData(30, C)也建立好了
data3 = CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC              ← 显示出好不容易建立好的RealData(30, C)
main END                                            ← 主线程结束
```

图 9-3　执行范例（结果可能随每次执行有所变化）

Future Pattern 的所有参与者

Future Pattern 的所有参与者如下：

◆ *Client（委托人）参与者*

Client 参与者会向 Host 参与者送出请求（request）。Clinet 参与者会马上得到 VirtualData 参与者，作为这个请求的结果（返回值）。

然而，这里所收到的 VirtualData 参与者，实际上是 Future 参与者。也就是说，Future 参与者带着 VirturlData 参与者的面具。Client 参与者并不需要知道返回值是 RealData 参与者还是 Future 参与者。一段时间后，Client 参与者会通过 VirtualData 参与者进行操作。

在范例程序中，Client 参与者是 Main 类。

◆ *Host 参与者*

Host 参与者会建立出新的线程，开始建立 RealData 参与者。另一方面，会对

Client 参与者（以 VirturlData 参与者的形式）返回 Future 参与者。范例程序中，Host 参与者是 Host 类。

新的线程建立出 RealData 参与者后，会对 Future 参与者设置 RealData 参与者。

◆ *VirtualData（虚拟的数据）参与者*

VirtualData 是用来统一代表 Future 参与者和 RealData 参与者。范例程序中，VirtualData 参与者是 Data 接口。

◆ *RealData（实际数据）参与者*

RealData 参与者用来表示实际的数据。建立这个对象需要花一些时间。范例程序中，RealData 参与者是 RealData 类。

◆ *Future（期货）参与者*

Future 是 Host 参与者传给 Client 参与者，当作是 RealData 参与者的“提货单”使用的参与者。

Future 参与者相对于 Client 参与者而言，可以进行 VirtualData 参与者的行为。而实际上 Client 参与者对 Future 参与者操作时，若 RealData 参与者还没建立好，线程会以 wait 等待。如果 RealData 参与者已经建立好了的话，就不会等待了。Future 参与者会将操作委托给 Client 参与者。

范例程序中，Future 参与者是 FutureData 类。Future Pattere 的类图如图 9-4 所示。

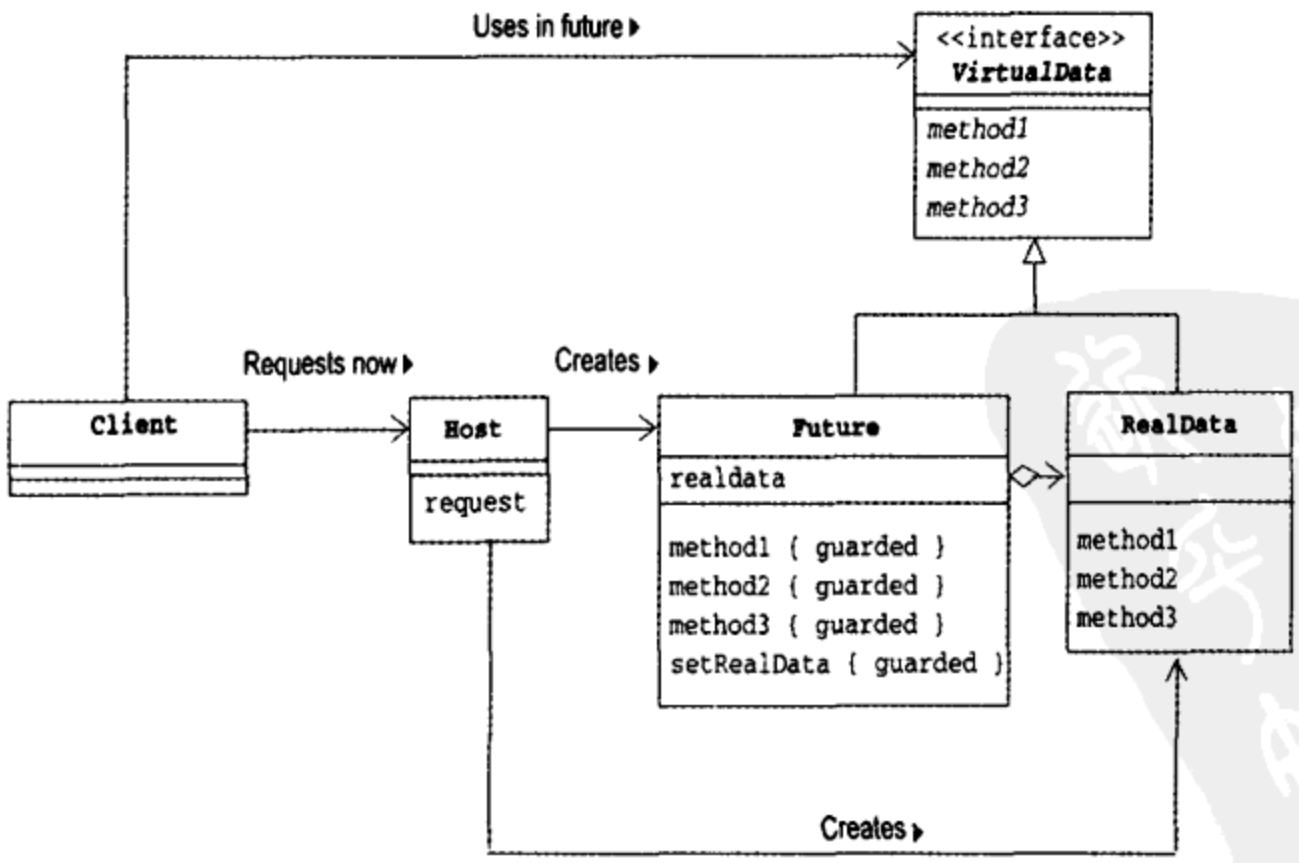


图 9-4　Future Pattern 的类图

扩展思考方向的提示

能够提升 throughput 吗

Thread-Per-Message Pattern（第 7 章）中，我们无法得到处理的结果。可是，Future Pattern 不只可以达到如同 Thread-Per-Message Pattern 的响应性，还可以获取处理的结果。

可是，可能有些读者会抱有疑问。“我可以接受程序的响应性能够提高这一点。但是，throughput 不是没改变吗？为了考虑在最后能收到所有的结果，那无论在单线程或是多线程下执行，等到所有结果都运算完，花的时间还不是一样。所以，虽然说是多线程，可是处理所花的总时间还不是没有变短。”

关于这个疑问，可说是一半对一半错的。

的确，程序写成多线程后，处理总时间并没有理由缩短。但问题在于“负担处理时间的是哪个线程”。

在单 CPU 上运行的 Java 执行环境上，把单纯的一连串运算以多线程处理，应该是不会提高 throughput。因为就算是由多个线程分担运算，最后还是由一个 CPU 在负责运算[注 1]。但是，如果加上 I/O（输出入）处理，就不一样了。例如，对硬盘访问的时候，并不是所有工作都是由 CPU 进行的。当硬盘读写数据的时候，CPU 只是在等待工作结束而已。对 CPU 而言，这时是“空闲时间”。若能将这些空闲时间分配给其他线程，继续其他的处理操作，是可能提高 throughput 的[注 2]。

本章末的练习问题 9-3，就要建立一支使用多线程从多个网页上获取数据的程序，实验对 throughput 有多少影响。

异步方法调用的“返回值”

Java 提供的方法调用，都是同步的（synchronous）。也就是说，调用方法之后，只能一直前进到方法执行完为止。

Thread-Per-Message Pattern（第 7 章）中，通过在方法里启动新的线程，仿真出类似异步的（asynchronous）方法调用。当然，就算使用 Thread-Per-Message Pattern，Java 的方法调用本身仍然是同步的。所以我们说是“仿真出”异步的方法调用。

只使用 Thread-Per-Message Pattern，并不能得到处理结果（也就是非同步方法

注 1：但有可能因为线程数的提高，使 OS 分配给 Java 执行环境整体的分割时间变长，要如何正确地描述 throughput 的变化，实际上是困难的。所以在这里写得比较概略一些。

注 2：Java 2 SDK, Standard Edition, v 1.4（J2SE 1.4）起提供的新 I/O 链接库（NIO）中，提供了非搁置的 I/O (nonblocking I/O)。使用这些技术，可以写出不会在 I/O 时停顿的程序。

调用的“返回值”)。而使用 Future Pattern 时，通过“稍后在设置真正的处理结果”，做到异步方法调用的“返回值”。

分离“准备返回值”与“使用返回值”

在 Worker Thread Pattern（第 8 章）中，我们讨论过“invocation 与 execution 的分离”。当时我们学到了如何在不同的线程上分别启动方法（invoke）与执行方法（execution）。

而 Future 则可以将“准备方法的返回值”与“使用方法的返回值”分离。例如范例程序中，建立 RealData 的操作就是“准备方法的返回值”，而调用 getContent 方法则是为了“使用方法的返回值”。并且“准备返回值”与“使用返回值”也可以在不同的线程上处理。

将调用方法的一连串动作，像慢动作播放般逐一拆解。接着将分解出来的每个独自的操作（启动、执行、准备返回值、使用返回值），尝试分配给不同的线程。我们把多线程当作道具使用，就是在做这些事。

说到这里，关于方法调用，还有一种相关情况，就是“抛出异常时不会得到返回值”的情况。关于这点，我们留到本章末练习问题 9-4 研究。

变形——不让人等待的 Future 参与者

范例程序中，当调用 FutureData 的 getContent 方法时，如果 RealData 的实例还没建立好，就使用 Guarded Suspension Pattern（第 3 章），“等到建立好为止”。因为会等待到建立好为止，所以从 FutureData 的 getContent 方法退出时，一定会得到需要的信息。

可是，getContent 方法事实上也可能设计成异步的。但不是在 getContent 方法里建立出新的线程那么麻烦。而是使用 Balking Pattern（第 4 章），“如果还没好，就马上退出”。既然 RealData 的实例还没做好，就先回去，稍微做一些自己的工作以后，再调用一次 getContent 方法试试看。

这是 Future Pattern 的一种变形。

变形——会改变的 Future 参与者

通常我们只会对 Future 参与者设置一次“返回值”。也就是说，Future 参与者是“只会改变一次状态的变量（latch；门闩）”。不过，有些时候我们会希望重复设置 Future 参与者的“返回值”。也就是说 Future 参与者随时都被设置成“在这个时刻恰当的返回值”。

例如，假设我们现在要从网络获取图片。一开始要获取图片的长宽尺寸，接下

来要获取概略的模糊图片，最后再获取清晰的图片。这种时候可能就适合使用会变化的 Future 参与者。

这也是一种 Future Pattern 的变形。

谁需要考虑到多线程[复用性]

接下来，我们回过头来看范例程序的 RealData 类。RealData 并没有考虑到多线程，是一个普通的类。RealData 会进行繁重的工作。如果 Host 类直接使用 RealData，程序的响应性会下降。所以在这章里，我们通过 Thread-Per-Message Pattern（第 7 章）与 Future Pattern，提高程序的响应性。

然而，请注意 RealData 相关的类（Data、Host、FutureData）里，哪些类有放入“多线程相关的处理”呢？

Data 接口，没有考虑多线程。

Host 类，使用 Thread-Per-Message Pattern，启动新的线程，所以是有考虑多线程的。

那么 FutureData 类呢？ FutureData 当然有考虑多线程。在 setRealData 方法与 getContent 方法里就使用到 Guarded Suspension Pattern（第 3 章）了。

那么，请重新看一次 RealData 类，这个类并没有考虑多线程。与多线程有关的部分，都在 Host 与 FutureData 类内解决，对 RealData 类则没有影响。因为这一点，所以我们可以将现有的类套用 Future Pattern。章末的练习问题 9-3，就要练习套用 Future Pattern。

▶▶ 补充说明：Open Call Pattern

关于“Host 类有考虑到多线程”这一点，还有一些需要说明的地方。Host 类的 request 方法虽然不是 synchronized，但是可以安全地由多条线程调用。这是因为 Host 类没有字段（也就是根本没有状态），所以就算同时被多条线程访问，也不会破坏安全性。可说是 Immutable Pattern（第 2 章）的特殊形式。这也可以说是 Open Call Pattern（参考附录 E[Lea]）的特殊形式。Open Call Pattern 是当 Host 具有状态时，只对状态更新的部分加以防御的 Pattern。

回调与 Future Pattern

当我们想要在处理结束后，得到返回值，还可以使用回调（callback）的方式。所谓回调，是指处理完毕后，由 Host 参与者所启动的线程去调用 Client 参与者的方法。不过这样做的话，Client 参与者也必须考虑多线程了。具体来说，就是 Client 参与者中必须加入一些程序代码，确保接收返回值时的安全性。

相关 Pattern

Thread-Per-Message Pattern（第 7 章）

Thread-Per-Message Pattern 是将花时间的工作交给新线程来作的 Pattern。当我们需要得到处理结果时，就要用 Future Pattern。

Builder Pattern（参考附录 E [GoF][Yuki01]）

Builder Pattern 会在事后去获取组合好的成果。

Future Pattern 会在开始组合的时刻，先建立一个 Future 参与者，之后再通过 Future 参与者去访问实际的成果。

Proxy Pattern（参考附录 E [GoF][Yuki01]）

Proxy Pattern 中，因为建立 RealSubject 参与者需要花一些时间，所以加入 Proxy 参与者作为代理者，尽可能代理一些工作。直到非需要 RealSubject 参与者不可的时候，再建立 RealSubject 参与者（Virtual Proxy Pattern 的情况）。

Future Pattern 中，因为建立 RealData 参与者需要花一些时间，所以加入 Future 参与者作为提货单。而由其他线程建立好 RealData 参与者。直到需要通过 Future 参与者获取 RealData 参与者时，才等待 RealData 参与者的完成。

Guarded Suspension Pattern（第 3 章）

通过 Future 参与者访问 RealData 参与者的方法，都使用了 Guarded Suspension Pattern 加以防卫。防卫条件是“ReadData 参与者已经建立好了”。

◆ *Balking Pattern（第 4 章）*

Future 参与者是“只能指定一次值的变量”，可视为 latch（门闩）的一种。若有必要防止对 Future 参与者重复设置 RealData 参与者，可使用 Balking Pattern。

重点回顾

这一章我们学习了 Future Pattern。

Thread-Per-Message Pattern（第 7 章）是将花费时间的工作交给别的线程，以提高程序的响应性。不过，如果我们需要得到别的线程所处理的结果时，就行不通了。

若同步执行需要花一些时间的操作，会使程序响应性降低。但是，如果异步地开始执行，却无法在第一时刻得知结果。

这种时候就要使用 Future Pattern。首先我们建立一个与处理结果具有相同接口（API）的 Future 参与者。接着，在开始处理时，先把 Future 参与者当作返回值返回。直到其他线程处理完以后，才将真正的结果设置给 Future 参与者。Client 参与者可通过 Future 参与者得到处理的结果。

使用这个 Pattern，可使响应性不降低，并能得到想要的处理结果。

这就是 Future Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 9-1（基础知识的确认）

Java 请就您对范例程序的了解，阅读下列语句，正确的请回答○，不正确的请回答×。

（1）调用出 request 方法后，一个新的线程就会启动。

（2）request 方法的返回值类型虽是 Data 接口，实际的返回值是 FutureData 的实例。

（3）会调用 SetRealData 方法的是主线程。

（4）执行 RealData 类的 getContent 方法的，是主线程。

（5）如果 request 方法被从数个线程调用出来的话，request 方法必须设为 synchronized。

问题 9-2（每条线程的工作）

Java 范例程序运行时，有几条线程在运行呢？（主线程除外）每条线程的工作各是什么？

问题 9-3（套用 Future Pattern）

Java 下面是一支指定 URL 以从网络上获取文件的程序。请将这支程序改成使用多线程的版本。

首先，我们建立出可以获取 Yahoo!网页的 Retriever 类。

```
Content content = Retriever.retriever("http://www.yahoo.com/");
```

接下来，我们定义了可以获取 byte 数组的 Content 类。

```
byte[] bytes = content.getBytes();
```

测试用的 Main 类如 List 9-6 所示。

获取文件的部分，则放在 content 这个包（package）里，其中 Retriever 类（List 9-7）与 Content 接口（List 9-8）对包外公开。而实际上用来获取文件的则是

SyncContentImpl 类（List 9-9）。

类图如图 9-5 所示，执行范例则如图 9-6 所示。

那么，接下来就是问题了。现在的 Retriever 类（List 9-7）使用 SyncContentImpl 类以单线程获取文件。请以这支程序作为参考，修改这个包，使程序可使用多线程获取文件。具体来说，就是建立出 AsyncContentImpl 类，将 Retriever 类改写成可以使用 AsyncContentImpl 类的版本。Main 类、Content 接口、SyncContentImpl 类请保持原封不动。

注意：一般来说，要开发公开包时，建议使用颠倒的网上域名作为包名称，以确保在世界上是惟一的（例如笔者的网域是 hyuki.com，就使用以 com.hyuki 为首的包名称）。不过，这里为求简单化，就不遵照这个规范，而使用 content 作为包名称。

注意：若您的计算机没有连接到因特网上，这支程序无法正常运行。另外，使用企业、学校内部的局域网络时，可能需要设置 HTTP 代理服务器。JDK 1.3、JDK 1.4 的 java 命令，可以下面方式设置之：

```
java -DproxySet=true -DproxyHost=代理服务器主机名称 -DproxyPorts=
端口号 Main
```

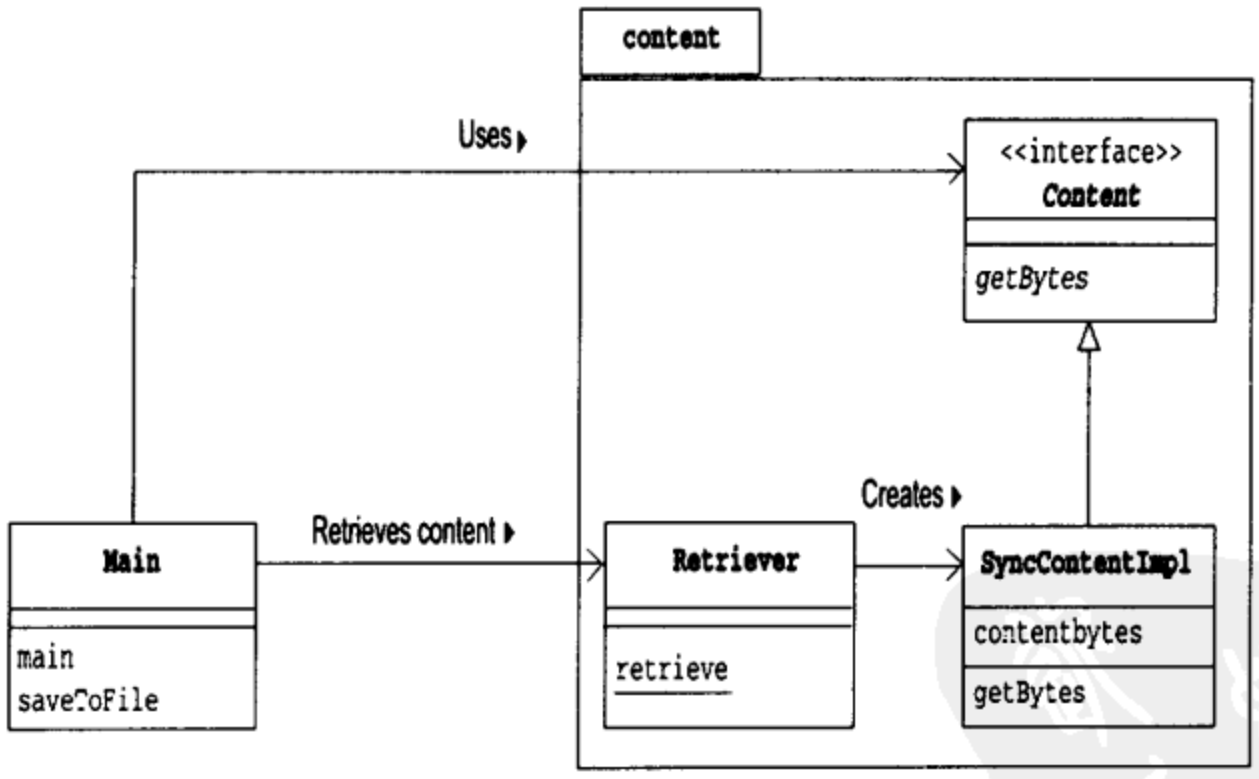


图 9-5　问题 9-3 的类图（单线程版）

List 9-6　Main 类（Main.java）

```
import content.Retriever;
import content.Content;

import java.io.IOException;
```

```
import java.io.FileOutputStream;

public class Main {
    public static void main(String args[]) {
        long start = System.currentTimeMillis();

        Content content1 = Retriever.retrieve("http://www.yahoo.com/");
        Content content2 = Retriever.retrieve("http://www.google.com/");
        Content content3 = Retriever.retrieve("http://www.hyuki.com/");

        saveToFile("yahoo.html", content1);
        saveToFile("google.html", content2);
        saveToFile("hyuki.html", content3);

        long end = System.currentTimeMillis();

        System.out.println("Elapsed time = " + (end - start) + "msec.");
    }

    // 将 content 的内容输出到名为 filename 的文件里
    private static void saveToFile(String filename, Content
content) {
        byte[] bytes = content.getBytes();
        try {
            System.out.println(Thread.currentThread().getName()
+ ": Saving to " + filename);
            FileOutputStream out = new FileOutputStream(filename);
          for (int i = 0; i < bytes.length; i++) {
              out.write(bytes[i]);
            }
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

List 9-7 Retriever 类（Retriever.java）

```
package content;

public class Retriever {
```

```
    public static Content retrieve(String urlstr) {
        return new SyncContentImpl(urlstr);
    }
}
```

List 9-8 Content 接口（Retriever.java）

```
package content;

public interface Content {
    public abstract byte[] getBytes();
}
```

List 9-9 SyncContentImpl 类（SyncContentImpl.java）

```
package content;

import java.io.DataInputStream;
import java.io.EOFException;
import java.net.URL;

class SyncContentImpl implements Content {
    private byte[] contentbytes;
    public SyncContentImpl(String urlstr) {
        System.out.println(Thread.currentThread().getName() + ":
Getting " + urlstr);
        try {
    URL url = new URL(urlstr);
    DataInputStream in = new DataInputStream(url.openStream());
    byte[] buffer = new byte[1];
    int index = 0;
    try {
              while (true) {
                  int c = in.readUnsignedByte();
                  if (buffer.length <= index) {
    byte[] largerbuffer = new byte[buffer.length * 2];
    System.arraycopy(buffer, 0, largerbuffer, 0, index);
    buffer = largerbuffer;
    // System.out.println("Enlarging buffer to " + buffer.length);
```

```
                }
                buffer[index++] = (byte)c;
                // System.out.print("Getting " + index + " bytes
from " + urlstr);
            }
        } catch (EOFException e) {
        } finally {
            in.close();
        }
        contentbytes = new byte[index];
        System.arraycopy(buffer, 0, contentbytes, 0, index);
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
  public byte[] getBytes() {
    return contentbytes;
  }
}
```

```
main: Getting http://www.yahoo.com/
main: Getting http://www.google.com/
main: Getting http://www.hyuki.com/
main: Saving to yahoo.html
main: Saving to google.html
main: Saving to hyuki.html
Elapsed time = 1903msec.
```

图 9-6　执行范例

问题 9-4（在 Future Pattern 处理异常）

Java 下面程序故意对 ReadData 类（List 9-5）的构造器的 count 参数传入负值。

```
new RealData(-1, 'N')
```

这么一来，程序会在执行时会抛出 java.lang.NegativeArraySizeException 异常（图 9-7）。因为负数长度的 char[]并无法分配。

假如在建立 RealData 实例时，发生了 NegativeArraySizeException 异常。这个异常，会从 request 方法里启动的新线程中抛出。也就是说，无论我们怎么在 request 方法与 getContent 方法里 try...catch，都无法捕捉到这个异常。

例如，像程序 List 9-10 这样调用 request，虽然抛出了 NegativeArraySizeException 异常，但程序会卡住不动。这是因为 main 线程在 getContent 里 wait 住了。

那么，接下来就是问题了。请改写范例程序，使得在建立 RealData 的实例中途发生异常时，未来调用 FutureData 类（List 9-4）的 getContent 方法时，可以得知异常的发生。

List 9-10 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("main BEGIN");
            Host host = new Host();

            Data data = host.request(-1, 'N');

            System.out.println("data = " + data.getContent());

            System.out.println("main END");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
main BEGIN
    request(-1, N) BEGIN
    request(-1, N) END
        making RealData(-1, N) BEGIN
java.lang.NegativeArraySizeException
        at RealData.<init>(RealData.java:5)
        at Host$1.run(Host.java:11)          ←程序停在这里
(CTRL+C 结束程序)
```

图 9-7　执行范例

第 10 章

Two-Phase Termination——快把玩具收拾好，去睡觉吧

Two-Phase Termination Pattern

小朋友总是把玩具散落一地。晚上到了睡觉时间时，妈妈会说“快把玩具收拾好，去睡觉吧”。这时小朋友就停下来不玩了，开始收拾玩具。

这一章要学习的是 Two-Phase Termination Pattern。

解释过来就是“两阶段终止”（见图 10-1）。这个 Pattern 用来进行结束的操作后，再终止线程。

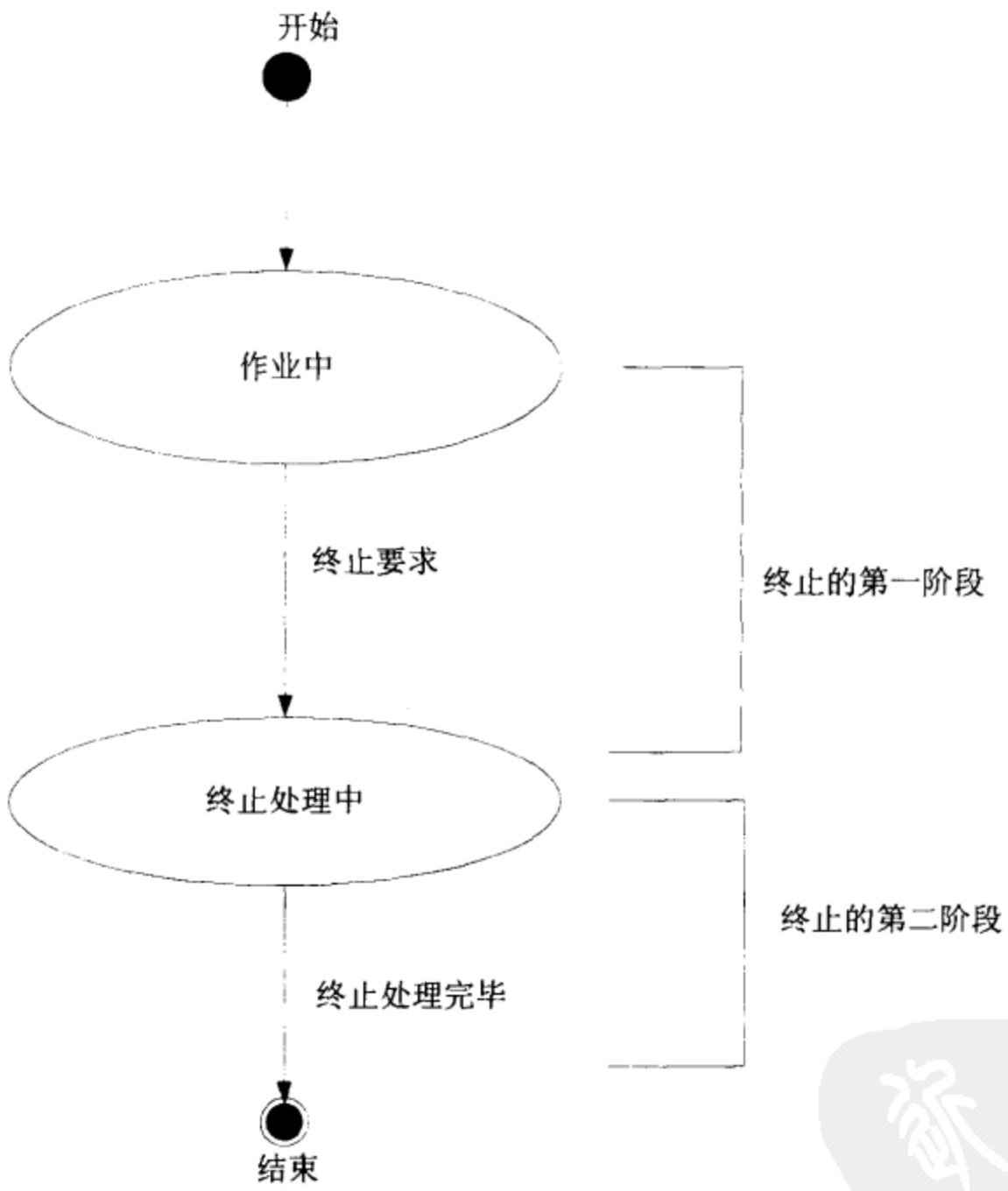


图 10-1　两阶段终止

我们将线程进行平常的处理的状态称为【作业中】。当希望结束这个线程时，则送出“终止请求”。接着这个线程，并不会马上结束，而会开始进行必要的刷新工作。这个状态称为【终止处理中】。从【作业中】改变成【终止处理中】是第一阶段。

【终止处理中】的状态时，不会进行平常的操作。虽然线程还在运行，但进行

的是终止处理。直到终止处理结束后，才真正结束线程。【终止处理中】的操作结束，是第二阶段。

先从【作业中】进入【终止处理中】状态，再真正结束掉线程。这就是 Two-Phase Termination Pattern。

主要考虑的关键因素如下：

- 安全地结束（安全性）；
- 一定会进行终止处理（生命性）；
- 收到终止请求后，要尽快开始终止处理（响应性）。

范例程序

接下来就来研究 Two-Phase Termination Pattern 的范例程序。

在这里要写的范例程序，有一条线程会每隔约 500 毫秒将计数递增 1，而我们要在约 10 秒后结束程序。虽然这个程序进行的操作实在没有意义，不过在这里为了使读者能看清楚 Two-Phase Termination Pattern 的架构，所以尽量以这种小程序为题材。类说明如表 10-1 所示。

表 10-1　类一览表

名　　称	说　　明
CountupThread	递增数字的线程
Main	操作测试用的类

类图如图 10-2 所示，顺序图则如图 10-3 所示。

类图中，字段名称与方法名称前面标示着加号(+)或减号(－)。加号表示 public，减号则表示 private。另外，{frozen}的限制表示 final、{concurrent}的限制则表示可以从多个线程同时访问。

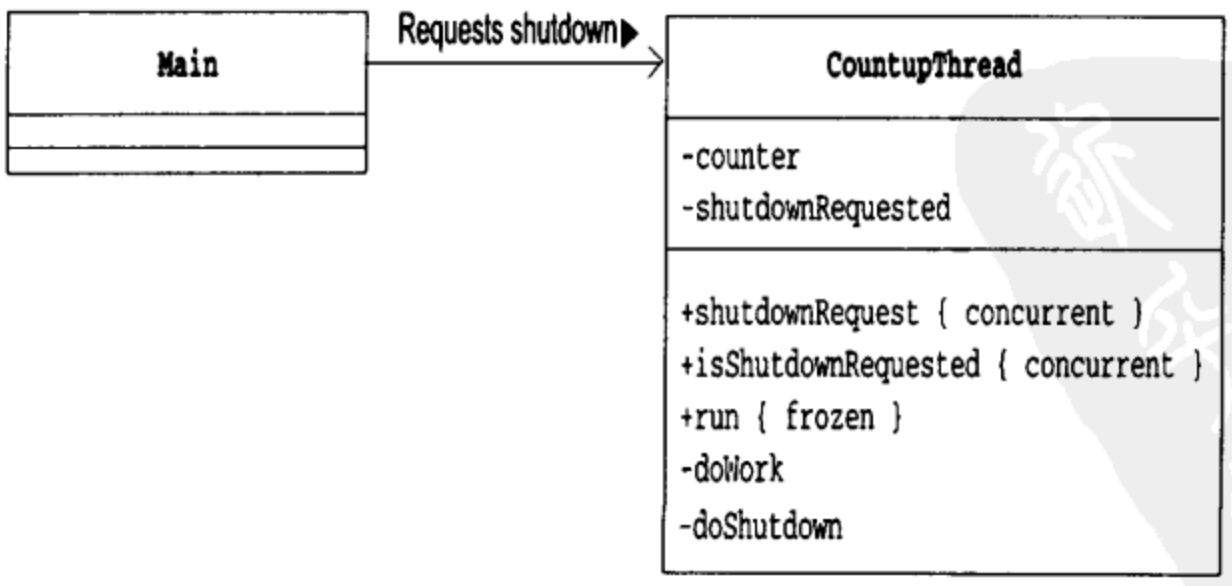


图 10-2　范例程序的类图

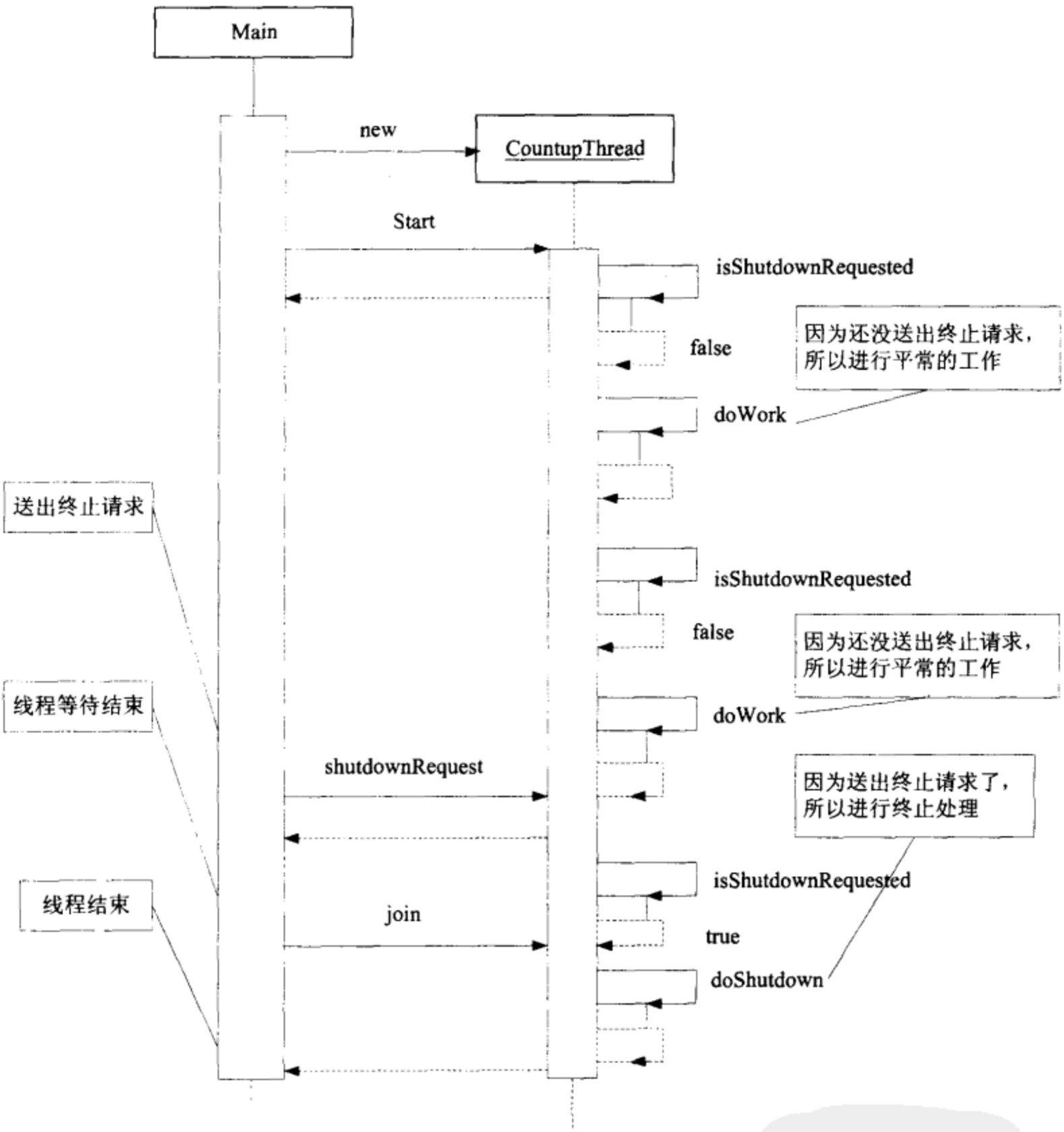


图 10-3　范例程序的顺序图

CountupThread 类

CountupThread 类（List 10-1）是一条会不断递增数值（0→1→2→3…）的线程。

counter 字段表示现在的值。

shutdownRequested 字段是表示终止请求是否已经送出的标识。以这个值判断是否前进到【终止处理中】。

shutdownRequest 方法用来对此线程送出终止请求。想要结束 CountupThread 线

程时，就要调用这个方法。

请注意 shutdownRequest 方法不只会将 shutdownRequested 设置成 ture，还会调用 interrupt 方法。使线程就算在 sleep 或是 wait 的状态下，也能结束掉。

shutdownRequest 方法没有 synchronized 的必要。因为 shutdownRequested 标识是一旦变成 true 就不会再变成 false 的 latch（门闩），所以就算从多条线程同时调用也无所谓[注1]。所以，shutdownRequest 方法限制成{concurrent}。

而 shutdownRequested 字段声明成 volatile 的理由，则留到本章末的练习问题 10-6 来讨论。

isShutdownRequested 方法则是用来检查 shutdownRequest 方法是否已经被调用（意即是否已经收到终止请求）。

run 方法则是实际的处理内容。只有在 shutdownRequested 的值为 false 时，会进行 while 循环。在循环中，会调用用来实际进行作业的 doWork 方法。并在 finally 块中，调用进行终止处理的 doShutdown 方法。

doWork 方法是用来进行实际操作的方法。在这里只是将 counter 的值加 1（也就是递增计数），并将值显示出来。

doShutdown 方法则是进行终止处理的方法。这个方法里应该要进行终止线程时必要的各项处理，在这里就单纯显示出当前 counter 的值。

List 10-1 CountupThread 类（CountupThread.java）

```
public class CountupThread extends Thread {
    // 计数器的值
    private long counter = 0;

    // 已经送出终止请求则为 true
    private volatile boolean shutdownRequested = false;

    // 终止请求
    public void shutdownRequest() {
        shutdownRequested = true;
        interrupt();
    }

    // 判断终止请求是否已经送出
    public boolean isShutdownRequested() {
        return shutdownRequested;
```

注 1：因为不会出现同时有线程要求设置为 true，又有线程要求设置为 false 的 race condition 现象。

```
    }

    // 动作
    public final void run() {
        try {
            while (!shutdownRequested) {
                doWork();
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 操作
    private void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        Thread.sleep(500);
    }

    // 终止处理
    private void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}
```

Main 类

Main 类（List 10-2）会启动 CountupThread 线程，并在约 10 秒后将它结束掉。

启动线程时，如往常一样，使用 start 方法。

对线程送出终止请求，则是使用刚刚在 CountupThread 类里声明的 shutdownRequest 方法。这个方法调用后会马上退出。

Thread 类的 join 方法，是用来等待到线程结束用的方法。在指定的线程结束前，不会从 join 方法中退出。

List 10-2 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
```

```
System.out.println("main: BEGIN");
try {
    // 启动线程
    CountupThread t = new CountupThread();
    t.start();

    // 稍微空出一段时间
    Thread.sleep(10000);

    // 对线程送出终止请求
    System.out.println("main: shutdownRequest");
    t.shutdownRequest();

    System.out.println("main: join");

    // 等待线程结束
    t.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
    System.out.println("main: END");
  }
}
```

执行范例如图 10-4 所示。我们可以看到的确在送出终止请求后，doShutdown 方法会被调用，之后线程就结束了。

```
main: BEGIN
doWork: counter = 1          ← 约 500 毫秒递增一次
doWork: counter = 2
doWork: counter = 3
doWork: counter = 4
doWork: counter = 5
doWork: counter = 6
doWork: counter = 7
doWork: counter = 8
doWork: counter = 9
doWork: counter = 10
doWork: counter = 11
doWork: counter = 12
```

（续上图）

```
doWork: counter = 13
doWork: counter = 14
doWork: counter = 15
doWork: counter = 16
doWork: counter = 17
doWork: counter = 18
doWork: counter = 19
doWork: counter = 20
main: shutdownRequest          ← 主线程调用 shutdownRequest 方法
doShutdown: counter = 20       ← doShutdown 方法进行终止处理
main: join                     ← 主线程等待 CountupThread 线程结束
main: END                      ← 结束
```

图 10-4　执行范例

Two-Phase Termination Pattern 的所有参与者

Two-Phase Termination Pattern 的所有参与者如下：

◆ *TerminationRequester（送出终止请求的人）参与者*

TerminationRequester 参与者对 Terminator 参与者送出终止请求。范例程序中，TerminationRequester 参与者是 Main 类。

◆ *Terminator（终止的人）参与者*

Terminator 参与者会接受终止请求，实际进行终止处理。Terminator 参与者提供有用来提出终止请求的 shutdownRequest 方法。我们没有必要对 shutdownRequest 方法使用 Single Threaded Execution Pattern（第 1 章）。

当 shutdownRequest 方法被调用，Terminator 参与者就会在考虑到安全性的情况下，自己进入【终止处理中】状态。接着在终止处理执行完毕后，线程正式结束。

Terminator 参与者拥有一个标识（门闩）表示自己是否已经收到终止请求。并在可以安全地开始进行终止处理的地方，检查这个标识。如果检查标识的操作很频繁，那从收到终止请求，直到真正进入【终止处理中】状态所间隔的时间，就可以缩短。

范例程序中，Terminator 参与者是 CountupThread 类。

图 10-5 的类图，是假定可以定义 Terminator 参与者的子类，故将 doWork 与 doShutdown 设置成 protected 方法。UML 的类图中，protected 是以井号（#）表示的。

本章末的练习问题 10-4，就要尝试建立覆盖（override）doWork 与 doShutdown 的类。

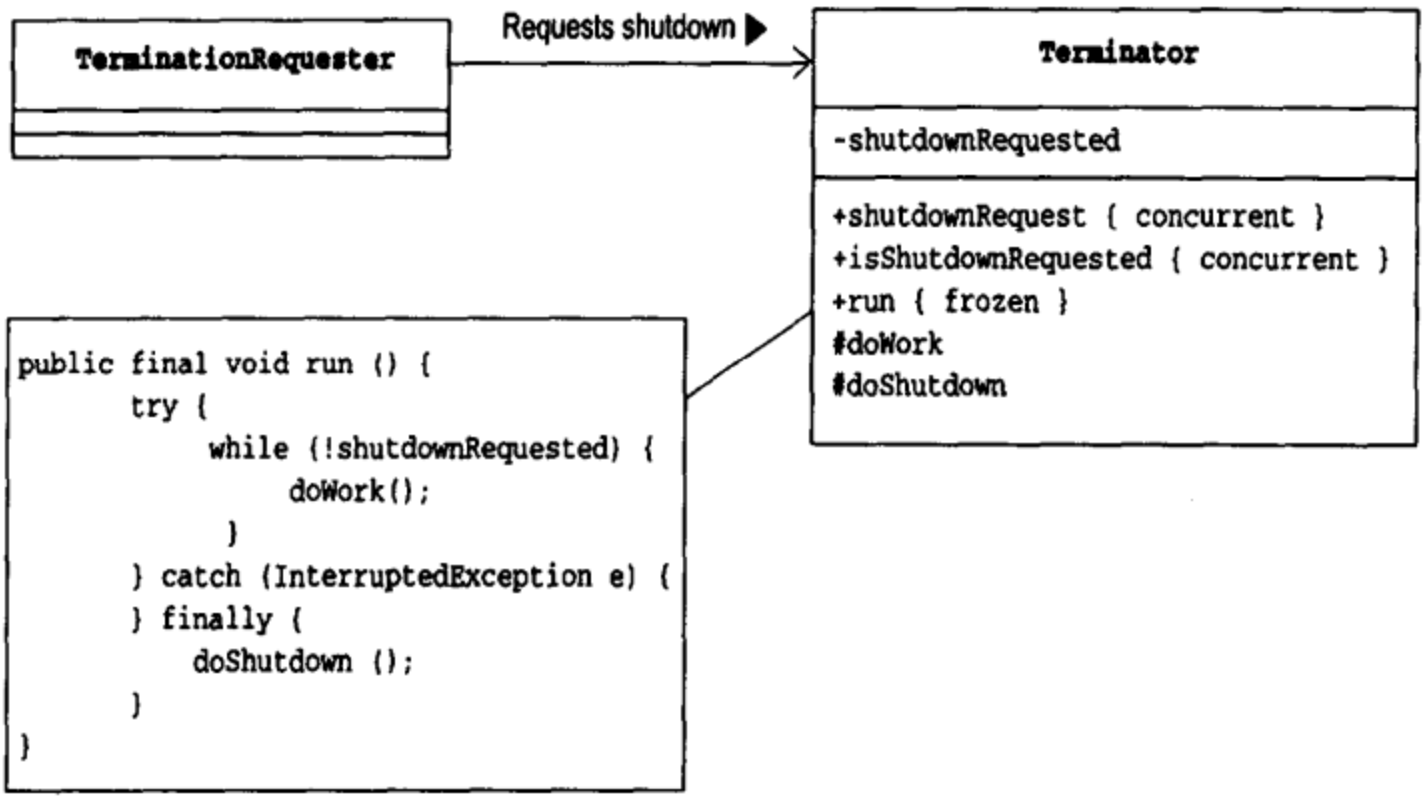


图 10-5　Two-Phase Termination Pattern 的类图

扩展思考方向的提示

不可以使用 Thread 类的 stop 方法

java.lang.Thread 类有一个用来强制结束掉线程的 stop 方法。但是 stop 现在已经是不建议使用（deprecated）的方法，我们不应该使用。

原因是 stop 方法会使实例丧失安全性的保障。使用 stop 方法时，线程会抛出 java.lang.ThreadDeath 异常而马上结束。即使该线程现在正在执行临界区间（例如 synchronized 方法的中间），也会马上结束。例如下面的例子：

```
class Position {
    private int x;
    private int y;
    public synchronized void setXY(int newX, int newY) {
        x = newX;
        y = newY;
    }
    ...
}
```

setXY 是 synchronized 方法，所以这个方法同时只能有一个线程执行。于是，使用这个方法时，即使有多条线程在执行，x 与 y 的值会安全地一起被指定。不会发生只有 x 的值指定了，但 y 的值没有指定的问题。

但是，使用 stop 方法强制停止线程，就可能丧失安全性。因为说不定刚好在线程刚进行

```
x = newX;
```

的指定操作，还没执行到

```
y = newY;
```

的时候，调用到 stop 方法。

于是，stop 方法是不应该使用的。

只检查标识是不够周全的

接下来要思考的是，shutdownRequest 方法为什么需要调用 interrupt 方法。也就是说，为什么只把 shutdownRequested 标识设置成 true 是不够的呢？

理由很简单，当想要结束掉线程的时候，说不定线程现在正在 sleep。如果线程正在 sleep，就算将 shutdownRequested 标识设置成 true，线程也不会开始进行终止处理。或许过了 sleep 的时间后，就会开始执行终止处理，但这样程序的响应性就变差了。所以如果使用 interrupt 方法，就可以把 sleep 中断掉。

线程或许也可能正在 wait。如果线程正在 wait，就算将 shutdownRequested 标识设置成 true，线程也不会退出等待区。所以我们必须使用 interrput 方法，要求线程“不要 wait 了”。

只测试中断状态也是不够的

或许细心的读者已经想到下面的问题。

“调用 interrupt 方法后，如果线程正在 sleep 或 wait，则会抛出 InterruptedException 异常，就算没有抛出异常，线程也会变成中断状态。既然如此，那又何必特地做一个新的 shutdownRequested 标识呢。只要 catch InterruptedException 异常，并使用 isInterrupted 方法检查中断状态，不就好了吗？”

提出这样的问题是很合理的。像范例程序，程序员如果可以掌握整个线程时，就不需要用到 shutdownRequested 标识了。只要 catch InterruptedException，并使用 isInterrupted 方法，就可以正确地进行终止处理。

但是，如果线程会执行到任何一个会忽视 InterruptedException 的地方，这样做就有待商榷了。“会忽视 InterruptedException 的地方”，像下面就是：

```
try {
    Thread.sleep(100);
} catch (InterruptedException e) {
    // 忽略 InterruptedException
}
```

当 wait、sleep、join 一抛出 InterruptedException 后，线程就不是中断状态了（参图 5-4）。也就是说，不使用 shutdownRequested 标识，而用上述的做法，可能无论怎么调用 shutdownRequest 方法，记录还是被抹失了。shutdownRequested 标识是用来记录终止请求的标识。本章末的练习问题 10-2 会具体的以程序来讨论这个问题。

进行繁重的处理前，先检查终止请求

为了使收到终止请求后，能尽快开始终止处理，故每当要开始繁重的处理前，应该先检查 shutdownRequested 标识，或是调用 isShutdownRequested 方法。这样一来，可使程序的响应性提高。本章末的练习问题 10-7，就会以具体的程序讨论这个问题。

▶▶ 补充说明：NIO 与多线程

J2SE 1.4 NIO 中实现 java.nio.channels.Channel 接口的类群，设计时已将多线程考虑进去了。

例如，当线程对 Channel 阻挡住 I/O 时，其他线程可以把这个 Channel 给 close 掉。这时，这个阻挡 I/O 的线程，会收到 AsynchronousCloseException 异常。

另外，线程对 Channel 阻挡住 I/O 时，其他线程也可以 interrupt 掉这个线程。这时，这个阻挡 I/O 的线程，会收到 ClosedByInterruptException 异常。

join 方法与 isAlive 方法

等待指定的线程结束时，如范例程序 java.lang.Thread 所示，要使用 join 方法。

另外，检查指定的线程现在是否结束了，可以使用 java.lang.Thread 的 isAlive 方法。若返回值是 true，该线程还活着。如果是 false，表示线程已经结束了。

程序的结束与 addShutdownHook 方法

虽然与 Two-Phase Termination Pattern 没有直接关系，在这里顺带介绍有关终止处理的一个方法。就是 java.lang.Runtime 的实例方法 addShutdownHook。这个方法会在 Java 执行环境全部结束时（调用 System.exit 方法时，或所有非 Daemon 线程都结束时），调用指定 Thread 的 start 方法（这时候的 Thread 称为 shutdown hook）。使用这个方法，可以编写整个程序的终止处理。

简单的范例程序如下所示。在下面的程序中，设置 shutdown hook 以后，sleep 约 3 秒钟，以 System.exit 强制结束程序。结果如图 10-6 所示。

List 10-3 使用 shutdown hook 的 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("main:BEGIN");

         // 设置 shutdown hook
        Runtime.getRuntime().addShutdownHook(
            new Thread() {
                public void run() {
                    System.out.println("******");
                    System.out.println(Thread.currentThread().
                      getName() + ": SHUTDOWN HOOK!");
                    System.out.println("******");
                }
            }
        );

        System.out.println("main:SLEEP...");

        // 约 3 秒后强制结束程序
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }

        System.out.println("main:EXIT");

        // 在这里强制结束
        System.exit(0);

        // 不会执行到这里
        System.out.println("main:END");
    }
}
```

```
main:BEGIN
main:SLEEP...
main:EXIT                    ← 因为执行 exit 了，所以不会显示 main:END
*****                        ← 执行 shutdown hook
Thread-0: SHUTDOWN HOOK!
*****
```

图 10-6　执行结果

优雅终止的线程

在英文里，使用 graceful（优雅的；得体的）来形容确实执行完终止处理的样子。表示不是把做到一半的事情搁着，就草率地结束掉，而是仔细地将需要刷新的事宜处理好，再真正结束。Two-Phase Termination Pattern 能够将线程 graceful 地结束掉。接下来我们要看的是，Two-Phase Termination Pattern 如何表现提及的 3 个关键（安全性、生命性、响应性）。

◆ 安全地终止（安全性）

即使收到终止请求，也不马上结束线程。而是竖立表示收到终止请求的 shutdownRequested 标识。以对象不会被破坏为前提，进行终止处理。

就像妈妈说了“去睡觉吧”，也不该慌张地整理玩具，而把玩具弄坏一样。

◆ 一定会进行终止处理（生命性）

当收到终止请求时，会中断掉可以中断的 wait，前往终止处理。所以在 shutdownRequest 方法里，调用了 interrupt 方法。

另外，为了在抛出异常结束的时候，也会执行到终止处理，故使用了 try...finally 块。

就像不可以把玩具散落一地，就跑去睡觉了的意思一样。

◆ 送出终止请求后，要尽快进入终止处理（响应性）

当收到终止请求后，会中断掉可以中断的 sleep，尽快前进到终止处理。于是在 shutdownRequest 方法中调用了 interrupt 方法。

另外，在繁重的处理前，应该检查 shutdownRequested 标识。

就像妈妈说了“快把玩具收拾好”后，应该尽快开始整理一样。

像这样，优雅地结束掉线程，就像管教调皮的孩子一样。

相关 Pattern

Before/After Pattern（参考附录 E[Lea]）

Two-Phase Termination Pattern 为了确实进行终止处理，使用了 finally 块。这是使用了 Before/After Pattern。

Multiphase Cancellation Pattern（参考附录 E[Lea]）

Multiphase Cancellation Pattern 会在要求线程终止后，如果执行一定时间后还没有结束，则会不断送出更强的终止请求。

Multi-Phase Startup Pattern [注2]

Two-Phase Termination Pattern 收到终止请求后，不会马上结束，而会先经过【终止处理中】的阶段，安全地结束线程。

Multi-Phase Startup Pattern 则是在有多个子系统存在时，经过多阶段启动整个系统的 Pattern。系统拥有一个整数的 run level 值，表示现在正在启动的 run level。

Java 的 Applet 就使用了退化到 3 阶段的 Multi-Phase Startup Pattern（产生实例→调用 init 方法→调用 start 方法等 3 阶段）。

Balking Pattern（第 4 章）

在 Two-Phase Termination Pattern 中，为了防止终止处理进行的时候，进行不恰当的操作，可使用 Balking Pattern。

进阶说明：

中断状态与 InterruptedException 异常的相互转换

调用 interrupt 方法后，可以中断掉线程。这里所说中断掉线程，是指下面其中一种结果。

（1）线程变成“中断状态”对“状态”的反应；

（2）抛出“异常 InterruptedException” 对“控制”的反应。

通常会是（1）。只有在线程是 sleep、wait、join 时会是（2）（这个时候不会变成中断状态）。

然而，状态（1）、（2）是可以互相转换的。也就是说，可以把（1）变为（2），也可以把（2）变为（1）。配合程序的需要——大致上就是为了让程序不至于忘记被中断的事实——而加以变化。

下面是具体的说明：

中断状态→InterruptedException 异常的转换

“若线程是中断状态，就抛出 InterruptedException 异常”，可以像下面这样写。其中 interrupted 方法，是 java.lang.Thread 类的类方法。

注 2：Multi-Phase Startup（多阶段启动）（平锅建儿）
http://objectclub.esm.co.jp/jplop/startup-2.0.html

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

在花时间的处理前，先加上这个 if 语句，可提高程序对中断的响应性。可以避免不知道自己已经被中断，还开始进行花时间的处理。

反倒是，这个 if 语句乍看之下很简单，要完全了解倒比想象中复杂许多。

◆ 哪个线程来检查 interrupted 方法

Thread.interrupted 方法，会检查 Thread.currentThread()的中断状态。也就是说，上面的 if 语句无论写在哪个类的哪个方法，都是检查执行 if 语句的线程的中断状态。

◆ 不想清除中断状态的时候

调用 Thread.interrputed 方法后，线程就不是中断状态了。也就是说，只要调用一次 Thread.interrputed 方法后，中断状态就会被清除。

如果不想清除中断状态，而要检查现在线程是否被中断，要使用 inInterrupted 实例方法。使用方式如下：

```
if (Thread.currentThread().isInterrupted()) {
    // 若为中断状态时需要进行的处理（中断状态不会清除）
}
```

InterruptedException 异常→转换为中断状态

想要让线程只有在指定的时间才停止时，可以使用 Thread.sleep 方法。因为 Thread.sleep 会抛出 InterruptedException 异常，所以可能我们经常会这样写：

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
}
```

不过，这样写的话，抛出的 InterruptedException 异常会被忽略。当 sleep 被其他线程 interrupt 时，“被中断”这个事实会消失。

若当我们不希望被中断这个事实消失时，就像下面这样，自己再重新中断自己一次。

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
```

```
        Thread.currentThread().interrupt();
    }
```

这样可以将收到的 InterruptedException，转换为中断状态的形式。

InterruptedException 异常→转换为 InterruptedException 异常

收到的 InterruptedException 异常，也可以不马上抛出去，而留下来晚点再抛。如下：

```
InterruptedException savedException = null;
...
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    savedException = e;
}
...
if (savedException != null) {
    throw savedException;
}
```

在这里，先将收到的 InterruptedException 存在变量 savedException 里，留到后面才 throw 出去。

重点回顾

这一章我们学习了 Two-Phase Termination Pattern。

假设现在有一个想要结束掉的线程。在结束前，又必须进行特定的终止处理。并且进入终止处理时，对象也不能丧失安全性。

这种时候，就要准备一个用来“送出终止请求”的方法。在方法里将用来表示收到终止请求的标识设置为 true。而线程要在可以安全地进行终止处理的地方检查这个标识。当收到终止请求时，就开始进行终止处理。是否要前往终止处理，是线程（Terminator 参与者）自己判断的。

在 Java 语言中，为了要中断掉正在 wait、sleep、join 的线程，所以又使用了 interrupt 方法。因为调用 Thread.interrupted 方法后，中断状态会被清除，所以必须检查终止请求。

为了使程序执行时，就算发生异常，也会确实进行终止处理，所以使用了 finally。

这就是 Two-Phase Termination Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 10-1（基础知识的确认）

Java 请就您对范例程序的了解，仔细阅读下列语句，正确的请回答○，不正确的请回答×。

（1）调用出 shutdownRequest 方法的是主线程。

（2）doWork 方法只会被调用一次。

（3）doWork 方法将异常 InterruptedException 抛出时，doShutdown 方法会被调用出来。

（4）shutdownRequest 方法中的 interrupt()也可以写成 Thread.currentThread().interrupt()，意思一样。

问题 10-2（中断状态的变化）

Java List 10-4 从范例程序的 CountupThread 类删除了 shutdownRequested 标识，将程序简单化。这个程序如同我们预期的，正常地结束了（图 10-7）。但是，这样的简化方式，碰到某些 doWork 的内容，可能会无法正常结束。

接下来是问题，请只改写 doWork 方法，使这个程序无法正常结束。

List 10-4 CountupThread 类（CountupThread.java）

```
public class CountupThread extends Thread {
    // 计数器的值
    private long counter = 0;

    // 终止请求
    public void shutdownRequest() {
        interrupt();
    }

    // 动作
    public void run() {
        try {
            while (!isInterrupted()) {
                doWork();
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
```

```
        }
    }

    // 操作
    private void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        Thread.sleep(500);
    }

    // 终止处理
    private void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}
```

```
main: BEGIN
doWork: counter = 1
doWork: counter = 2
doWork: counter = 3
doWork: counter = 4
doWork: counter = 5
doWork: counter = 6
doWork: counter = 7
doWork: counter = 8
doWork: counter = 9
doWork: counter = 10
doWork: counter = 11
doWork: counter = 12
doWork: counter = 13
doWork: counter = 14
doWork: counter = 15
doWork: counter = 16
doWork: counter = 17
doWork: counter = 18
doWork: counter = 19
doWork: counter = 20
main: shutdownRequest
doShutdown: counter = 20
main: join
main: END                          ← 如预期般结束
```

图 10-7　执行范例

问题 10-3（编写终止处理）

Java 请改写范例程序 CountupThread 类（List 10-1）定义的 doShutdown 方法，将调用 doShutdown 方法的时刻时，counter 字段的值存到文件 counter.txt 里。

问题 10-4（Template Method Pattern）

Java List 10-5 是优雅地进行终止处理的 GracefulThread 类。请改写范例程序的 CountupThread 类（List 10-1），作为 GracefulThread 类的子类。

List 10-5 GracefulThread 类（GracefulThread.java）

```
public class GracefulThread extends Thread {
    // 已经送出终止请求则为 true
    private volatile boolean shutdownRequested = false;

    // 终止请求
    public final void shutdownRequest() {
        shutdownRequested = true;
        interrupt();
    }

    // 判断终止请求是否已经送出
    public final boolean isShutdownRequested() {
        return shutdownRequested;
    }

    // 动作
    public final void run() {
        try {
            while (!shutdownRequested) {
                doWork();
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 操作
    protected void doWork() throws InterruptedException {
    }

    // 终止处理
```

```
    protected void doShutdown() {
    }
}
```

问题 10-5（GUI 上的应用）

Java 现在希望写一支 GUI 程序，在单击[Execute]按钮后，会慢慢地显示出 50 个句点（.），而单击[Cancel]按钮可以中途停止显示。

在这里建立了 Main 类（List 10-6）、MyFrame 类（List 10-7），并想将实际的处理交给 Service 类（List 10-8）进行。这里已经写好 Service 类的 service 方法，但 cancel 方法不知道该怎么写。

我们尝试在还没实现 cancel 方法的情况下执行了程序。启动后的画面如图 10-8 所示，在这里，我们单击[Execute]按钮后，直到控制台（console）显示出 done.前（图 10-9），按钮都一直是按下的状态。所以，在 done.前，[Cancel]按钮根本没有办法单击（图 10-10）。

请改写 List 10-8 的 Service 类，并自行增加其他需要的类，使[Cancel]按钮能够提供终止服务的功能。

List 10-6 Main 类（Main.java）下去

```
public class Main {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

List 10-7 MyFrame 类（MyFrame.java）

```
import java.io.IOException;
import java.awt.FlowLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;

public class MyFrame extends JFrame implements ActionListener {
    private final JButton executeButton = new JButton("Execute");
    private final JButton cancelButton = new JButton("Cancel");
    public MyFrame() {
        super("MyFrame");
```

```
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(new JLabel("Two-Phase Termination Sample"));
        getContentPane().add(executeButton);
        getContentPane().add(cancelButton);
        executeButton.addActionListener(this);
        cancelButton.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == executeButton) {
            // 开始执行服务
            Service.service();
        } else if (e.getSource() == cancelButton) {
            // 终止服务
            Service.cancel();
        }
    }
}
```

List 10-8 写到一半的 Service 类（Service.java）

```
public class Service {
    // 开始执行服务
    public static void service() {
        System.out.print("service");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("done.");
    }

    // 终止服务
    public static void cancel() {
        // 尚未实现
    }
}
```

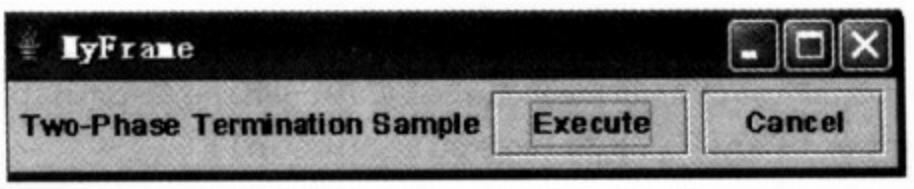


图 10-8　程序启动后的画面

```
service.....................................................done.
```

图 10-9　控制台显示的画面

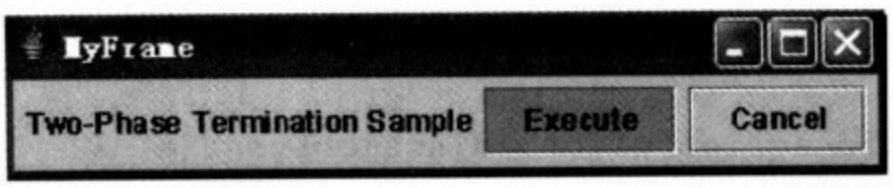


图 10-10　单击[Execute]按钮后，在显示出 done.之前，[Cancel]按钮按不下去

问题 10-6（volatile 的意义）

Java 范例程序中，shutdownRequested 字段声明成 volatile，请问这是为什么？

提示：与 Java 的内存模型有关。

问题 10-7（改善响应性）

Java P.339 的 Main 类（List 10-9）与 HanoiThread 类（List 10-10），是一支解答“汉诺塔（Towers of Hanoi）”的程序。所谓“汉诺塔”，是 Edouard Lucas（1842-1891）所提出的益智问题。基本规定如下：

- 这里有 3 支木棒（A、B、C）
- 其中有一支木棒堆放着多个圆盘（图 10-11 中是 3 个）
- 圆盘的大小都不同
- 我们可以把一支棒子最上面的圆盘，移动到其他棒子的最上面
- 小的圆盘可以放在大的圆盘上面，但反过来就不行

而益智问题的目的是将堆放在棒子的圆盘，全部移到另一支棒子上。

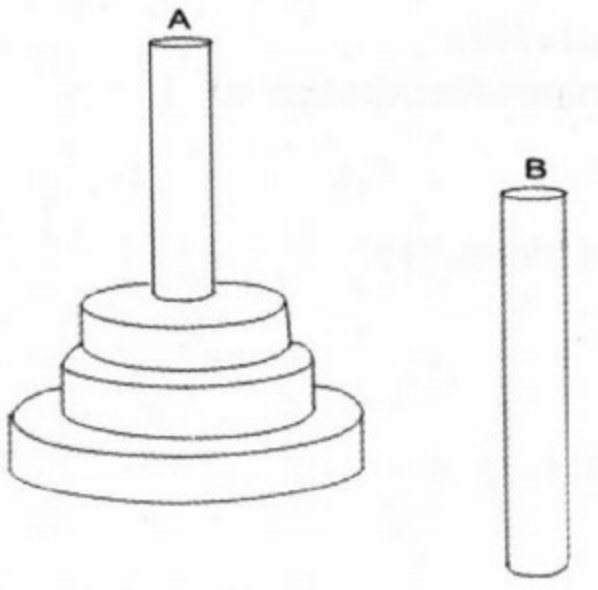


图 10-11　汉诺塔

执行这个程序，只有调用一次 doWork，就可以解开汉诺塔问题。HanoiThread 类（List 10-10）中所写的 doWork 的参数，表示圆盘的个数。执行范例见图 10-12）中下面这样的文字：

```
A -> B
```

表示将 A 棒子最上面的圆盘移到 B 棒子的最上面。

Main 类会先启动 HanoiThread，开始一步一步解答汉诺塔问题。并在约 10 秒后送出终止请求，如同执行范例的最后面所示。但这里在送出终止请求（调用 shutdownRequest 方法）后，直到进入实际的终止处理（调用 doShutdown 方法），竟然花了约 7681 毫秒之久。

那么，接下来就是问题了。请修改程序，缩短收到终止请求直到真正进入终止处理所延迟的时间（latency）。

List 10-9 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("main: BEGIN");
        try {
            // 启动线程
            HanoiThread t = new HanoiThread();
            t.start();

            // 等待一段时间
            Thread.sleep(10000);

            // 对线程送出终止请求
            System.out.println("main: shutdownRequest");
            t.shutdownRequest();

            // 等待线程集结束
            System.out.println("main: join");
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("main: END");
    }
}
```

List 10-10 HanoiThread 类（HanoiThread.java）

```
public class HanoiThread extends Thread {
    // 已经送出终止请求则为 true
    private volatile boolean shutdownRequested = false;
    // 送出终止请求的时刻
    private volatile long requestedTimeMillis = 0;

    // 终止请求
    public void shutdownRequest() {
        requestedTimeMillis = System.currentTimeMillis();
        shutdownRequested = true;
        interrupt();
    }

    // 判断终止请求是否已经送出
    public boolean isShutdownRequested() {
        return shutdownRequested;
    }

    // 动作
    public void run() {
        try {
            for (int level = 0; !shutdownRequested; level++) {
                System.out.println("==== Level " + level + " ====");
                doWork(level, 'A', 'B', 'C');
                System.out.println("");
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 操作
    private void doWork(int level, char posA, char posB, char posC)
throws InterruptedException {
        if (level > 0) {
            doWork(level - 1, posA, posC, posB);
```

```
            System.out.print(posA + "->" + posB + " ");
            doWork(level - 1, posC, posB, posA);
        }
    }

    // 终止处理
    private void doShutdown() {
        long time = System.currentTimeMillis() - requestedTimeMillis;
        System.out.println("doShutdown: Latency = " + time + "
msec.");
    }
}
```

```
main: BEGIN
==== Level 0 ====

==== Level 1 ====
A->B
==== Level 2 ====
A->C A->B C->B
==== Level 3 ====
A->B A->C B->C A->B C->A C->B A->B
==== Level 4 ====
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B
A->C A->B C->B
==== Level 5 ====
A->B A->C B->C A->B C->A C->B A->B A->C B->C B->A C->A B->C
A->B A->C B->C A->B
C->A C->B A->B C->A B->C B->A C->A C->B A->B A->C B->C A->B
C->A C->B A->B
==== Level 6 ====
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B
A->C A->B C->B A->C
B->A B->C A->C B->A C->B C->A B->A B->C A->C A->B C->B A->C
B->A B->C A->C A->B
C->B C->A B->A C->B A->C A->B C->B C->A B->A B->C A->C B->A
C->B C->A B->A C->B
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B
```

（续上图）

```
A->C A->B C->B
(中略)
main: shutdownRequest
(中略)
B->C A->B C->A C->B A->B A->C B->C B->A C->A B->C A->B A->C
B->C A->B C->A C->B
A->B C->A B->C B->A C->A C->B A->B A->C B->C A->B C->A C->B
A->B
doShutdown:Latency=7681 msec.←送出终止请求直到开始终止处理间隔的时间
main: END
```

图 10-12　执行范例

第 11 章

Thread-Specific Storage——每个线程的保管箱

Thread-Specific Storage Pattern

这里是保管箱间，在这里，有许多投币保管箱都排在一起。有个人拿着自己的钥匙进来了保管箱间，当他退出的时候，手上拿着自己的行李。又有一个人拿着自己的钥匙进来了，虽然进到同一个保管箱间，但一定是打开自己保管箱。每个用户，都会从各自的保管箱中拿出自己的行李。

这一章要学习的是 Thread-Specific Storage Pattern。

specific 有“独有的”的意义，而 storage 则有“储藏库”、“存储设备”的意义。所以 Thread-Specific Storage 就是“线程独有的储藏库”、“针对每个线程提供的内存空间”的意义。

Thread-Specific Storage Pattern 是只有一个入口，但内部会对每个线程提供特有存储空间的 Pattern。

Thread-Specific Storage 还有

- Per-Thread Attribute（每个线程的内容）
  （参考 java.lang.InheritableThreadLocal 的 API 引用文件）
- Thread-Specific Data（线程独有的数据）
  （参考附录 E [Lea][POSA2][Lweis00]）
- Thread-Specific Field（线程独有的字段）
  （参考附录 E [Lea]）
- Thread-Local Stroage（线程的区域内存空间）
  （参考附录 E [POSA2]）

等各种称呼。

使用 Java 标准链接库时，可以用 java.lang.ThreadLocal 类加以实现。

关于 java.lang.ThreadLocal 类

在正式进入这一章之前，在此要简单地对 java.lang.ThreadLocal 类作简要的解说。

java.lang.ThreadLocal 是保管箱间

java.lang.ThreadLocal 的实例可以想象成一种集合架构（collection）或许会比较好理解。也就是说，ThreadLocal 的实例只有一个，管理多个对象。

如果要套入前面的譬喻，那 java.lang.ThreadLocal 的实例也可以想象成保管箱间。关键在于不只有一个保管箱，而是一个集合有大量保管箱的房间。

因为 ThreadLocal 的实例管理了多个对象，所以 ThreadLocal 拥有“存放（set）”与“取得（get）”方法。依次说明如下：

◆ *public void set()*

ThreadLocal 类的 set 方法，可以将参数所指定的实例，存放到调用 set 方法的线程（现在的线程）所对应的存储空间。这里所存放的实例，可以使用 get 方法取得。请注意 set 方法并没有用来传入表示线程的参数。在 set 方法里，会检查当前线程（也就是 Thread.currentThread()的值），自动以这个值作为键值存放实例。

调用 set 时，就像把自己的行李放进保管箱一样。

◆ *public Object get()*

ThreadLocal 类的 get 方法，可以取得调用 get 方法的线程所对应的实例（现在线程对应的实例）。之前 set 的实例，就是现在 get 的返回值。如果从来没有 set 过，就会返回 null。

与 set 方法一样，get 方法也没有表示线程的参数。因为程序会在 get 里自己去检查现在的线程。也就是说自己本身就是键值。

调用 get 时，就像是从自己的保管箱拿出行李一样。

范例程序 1：
没有使用 Thread-Specific Storage Pattern 的范例

要了解 Thread-Specific Storage Pattern 前，在此我们先来看一个没有使用这个 Pattern 的范例程序。在脑海里有着范例程序 1 的印象后，再来读下一段的范例程序 2，会有助于对 Pattern 的理解。

范例程序 1 是单线程的程序，使用 Log 类的类方法，在文件留下 log（操作记录）数据。类说明如表 11-1 所示。

表 11-1　范例程序 1 的类一览表

名　　称	说　　明
Log	产生 log 用的类
Main	操作测试用的类

类图如图 11-1 所示。

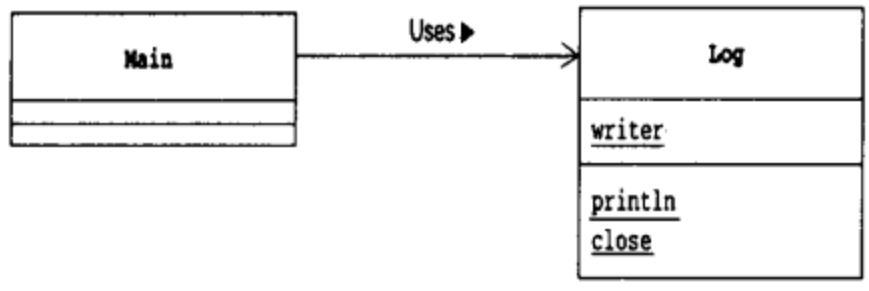


图 11-1　范例程序 1 的类图（没有使用 Thread-Specific Storage Pattern）

Log 类

Log 类（List 11-1）用来将程序的 log 记录存放在 log.txt 里。

为了对文件写入，所以在 writer 字段里存放有 java.io.PrintWriter 的实例。static { ... }块用来对 writer 字段初始化，建立出名为 log.txt 的 log 文件。这个操作只会在初始化 Log 时执行一次，之所以不直接初始化 writer，是因为我们需要捕捉 IOException 异常。

println 方法用来将传入的字符串存到文件里。

close 方法用来关闭 log 文件。

println 与 close 都将实际的工作委托给 writer 字段。

List 11-1　单线程用的 Log 类（Log.java）

```
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Log {
    private static PrintWriter writer = null;

    // 初始化 writer 字段
    static {
        try {
            writer = new PrintWriter(new FileWriter("log.txt"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // 加入一条 log
    public static void println(String s) {
        writer.println(s);
    }

    // 关闭 log
```

```
    public static void close() {
        writer.println("==== End of log ====");
        writer.close();
    }
}
```

Main 类

Main 类（List 11-2）会使用 Log 文件，慢慢地输出 0～9 的数字到 log 文件（间隔约 100 毫秒）。结果如图 11-2 和图 11-3 所示。

List 11-2　Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        System.out.println("BEGIN");
        for (int i = 0; i < 10; i++) {
            Log.println("main: i = " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        Log.close();
        System.out.println("END");
    }
}
```

```
BEGIN
END      ← 约 100 毫秒 × 10=约 1 秒后显示
```

图 11-2　执行结果

```
main: i = 0
main: i = 1
main: i = 2
main: i = 3
main: i = 4
main: i = 5
main: i = 6
main: i = 7
main: i = 8
main: i = 9
==== End of log ====
```

图 11-3　产生的 log 文件（log.txt）的内容

时序图如图 11-4 所示。

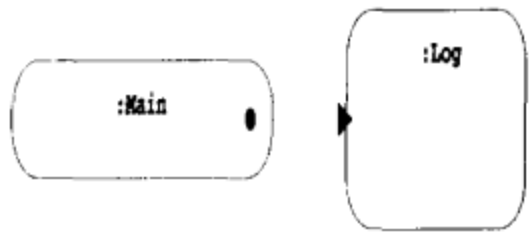


图 11-4　范例程序 1 的时序图（没有使用 Thread-Specific Storage Pattern）

范例程序 2：使用 Thread-Specific Storage Pattern 的范例

前面的范例程序 1，并没有什么特别的内容。只不过调用 println 方法，将字符串输出到 log 文件 log.txt 里而已。

接下来要写的范例程序 2，则是要将不同线程的字符串写到不同的 log 文件里。不过，输出 log 的时候，我们希望与范例文件 1 一样，只写下面的语句：

```
Log.println（输出的字符串）;
```

也就是说，我们并不是建立一个扩充 Thread 的类，加上 PrintWrtier 实例的字段，而是要像范例程序 1 一样，只要机械式地在方法里调用 Log.println 就好了。我们就是要写出程序能够自动区分线程，将字符串写入正确的 log 文件里的机制。类的说明如表 11-2 所示。

表 11-2　范例程序 2 的类一览表

名　　称	说　　明
TSLog	产生 log 的类（实例由每个线程所拥有）
Log	产生 log 的类（进行分配线程的工作）
java.lang.ThreadLocal	用来建立线程特有的空间
ClientThread	调用 Log 的线程
Main	操作测试用的类别

图 11-5 的类中所画的{ sequential }限制，表示“逐一执行（以单线程执行）”的意思。

线程特有的 TSLog 类

首先，在这里范例程序 1 的 Log 类为依据，建立用来输出线程特有的 log 记录的类。名称应该取为 ThreadSpecificLog 类，不过太长了，所以在这里命名为 TSLog 类（List 11-3）。

范例程序 1 的 Log 类（List 11-1）只有类字段和类方法，因为是以“只有一个 log 文件”为前提设计的。

而 TSLog 类则要对每个线程建立一个实例，所以要建立实例字段与实例方法。static { ... }的初始化操作就改到构造器里进行。类图如图 11-5 所示。

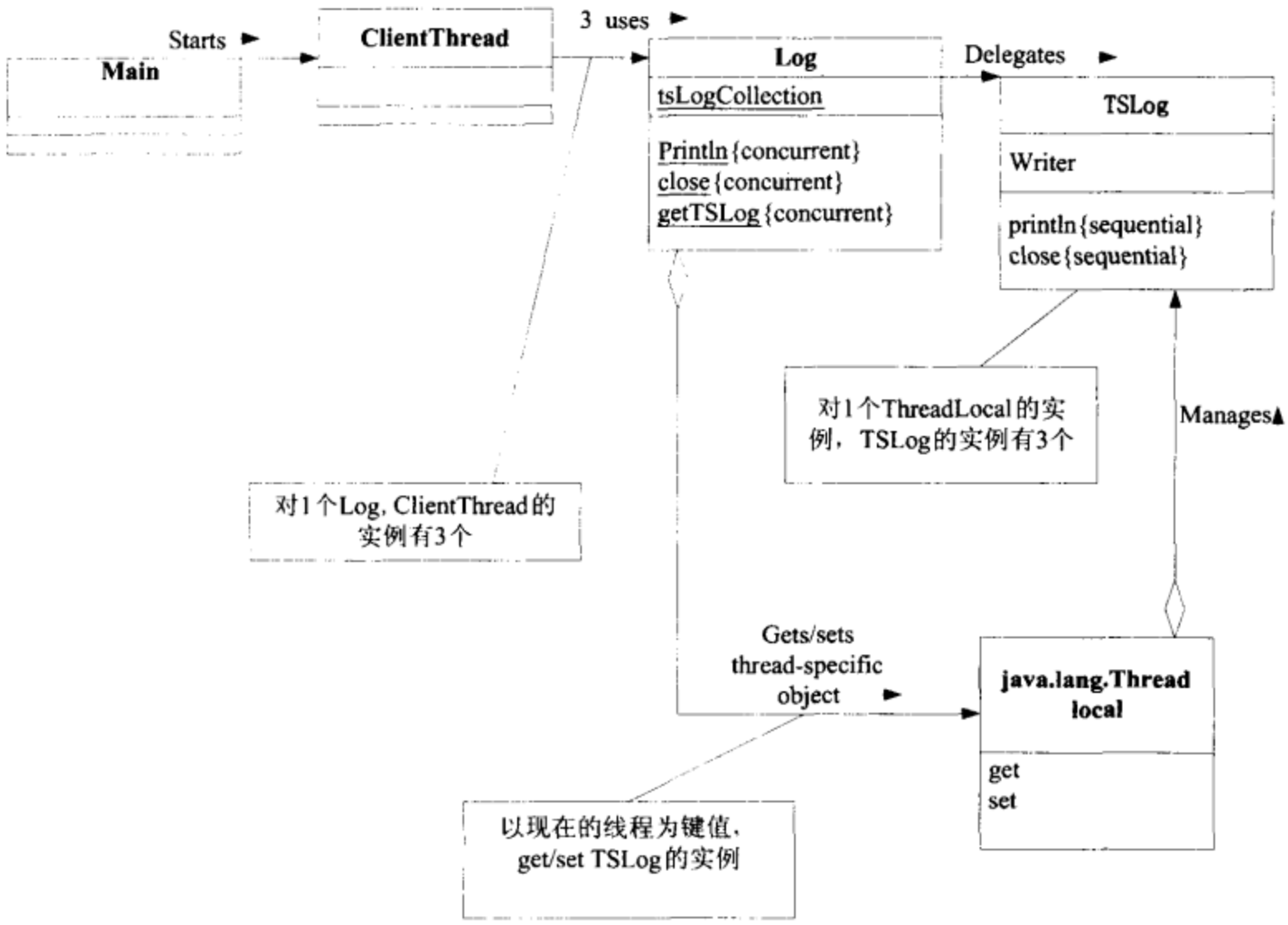


图 11-5　范例程序 2 的类图（使用 Thread-Specific Storage Pattern）

List 11-3　TSLog 类（TSLog.java）

```
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

public class TSLog {
    private PrintWriter writer = null;

    // 初始化 writer 字段
    public TSLog(String filename) {
        try {
            writer = new PrintWriter(new FileWriter(filename));
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }

    // 加入一条 log
    public void println(String s) {
        writer.println(s);
    }

    // 关闭 log
    public void close() {
        writer.println("==== End of log ====");
        writer.close();
    }
}
```

Log 类

请比较 List 11-4 的 Log 类与范例程序 1 的 Log 类（List 11-1）。对外公开的方法（println 和 close）都一样，可是实现却相差很多。

List 11-4 Log 类（Log.java）

```
public class Log {
    private static final ThreadLocal tsLogCollection = new
ThreadLocal();


    // 加入一条 log
    public static void println(String s) {
        getTSLog().println(s);
    }

    // 关闭 log
    public static void close() {
        getTSLog().close();
    }

    // 取得线程特有的 log
    private static TSLog getTSLog() {
        TSLog tsLog = (TSLog)tsLogCollection.get();

        // 如果线程是第一次调用，就建立新文件并注册 log
        if (tsLog == null) {
```

```
            tsLog = new TSLog(Thread.currentThread().getName() +
"-log.txt");
            tsLogCollection.set(tsLog);
        }

        return tsLog;
    }
}
```

范例程序 1 的 Log 类，具有 PrintWriter 类型的类字段（writer）。println 与 close 都将工作委托给这个 writer。这个 writer 对 Log 类只有 1 个，所以范例程序 1 只能支持单线程。

而范例程序 2 的 Log 类，具有 java.lang.ThreadLocal 类型的类字段（tsLogCollection）。这个 tsLogCollection 字段，就像是用来储藏每个线程的 TSLog 实例的保管箱一样。

范例程序 2 中，println 与 close 方法也都把工作委托给 getTSLog()的返回值。

getTSLog 方法，可以取得线程特有的 TSLog 实例。

getTSLog 的返回值，会因为调用这个方法的线程不同而更改。因为会自动判断线程，所以不需要为不同线程定义不同的方法。也就是不需要定义 getTSLogForAlice、getTSLogForBobby、getTSLogForChris 之类的方法。另外，也不需要将线程名称作为参数传入 getTSLog 里。

getTSLog 做的操作，就像是从保管箱里取出自己的行李一样。取出的行李（TSLog 的实例）是自己（线程）所拥有的。

getTSLog 里，使用 if 语句检查 tsLogCollection.get 的返回值是不是 null。这是为了检查现在这条线程是不是第一次调用 getTSLog。如果返回值是 null，就像是线程第一次进来保管箱间，所以打开自己的保管箱，当然是“空”的。空的，就表示目前还没有放入任何东西——也就是第一次使用。

在第一次调用 getTSLog 的时候，程序会产生新的 TSLog 实例，set 给 tsLogCollection。这就像线程第一次把自己的行李放进保管箱里。这时，将 log 文件命名为“线程的名称-log.txt”。

ClientThread 类

ClientThread（List 11-5）是使用 Log.println 与 Log.close 的线程。请与范例 1 的 Main 类（List 11-2）比较。相信读者看的出来，只不过是把与 Main 类的 main 方法所作的事改在 run 方法里。

List 11-5 ClientThread 类（ClientThread.java）

```
public class ClientThread extends Thread {
    public ClientThread(String name) {
```

```
        super(name);
    }
    public void run() {
        System.out.println(getName() + " BEGIN");
        for (int i = 0; i < 10; i++) {
            Log.println("i = " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        Log.close();
        System.out.println(getName() + " END");
    }
}
```

Main 类

范例程序 2 的 Main 类（List 11-6）启动了 3 条 ClientThread 的线程（Alice、Bobby、Chris）。

List 11-6 Main 类（Main.java）

```
public class Main {
    public static void main(String[] args) {
        new ClientThread("Alice").start();
        new ClientThread("Bobby").start();
        new ClientThread("Chris").start();
    }
}
```

执行结果如图 11-6 所示。

```
Alice BEGIN
Bobby BEGIN
Chris BEGIN
Alice END
Chris END
Bobby END
(建立了 Alice-log.txt、Bobby-log.txt、Chris-log.txt 三个文件)
```

图 11-6 执行结果

图 11-7 表示 Alice、Bobby、Chris 三条 ClientThread 通过 Log 类使用线程特有的 TSLog 对象的样子。虽然 3 条线程都是调用 Log 类的同一个方法，但实际上使用

的是每个线程独自的 TSLog 实例。

请与范例程序 1 的时序图（图 11-4）加以比较。

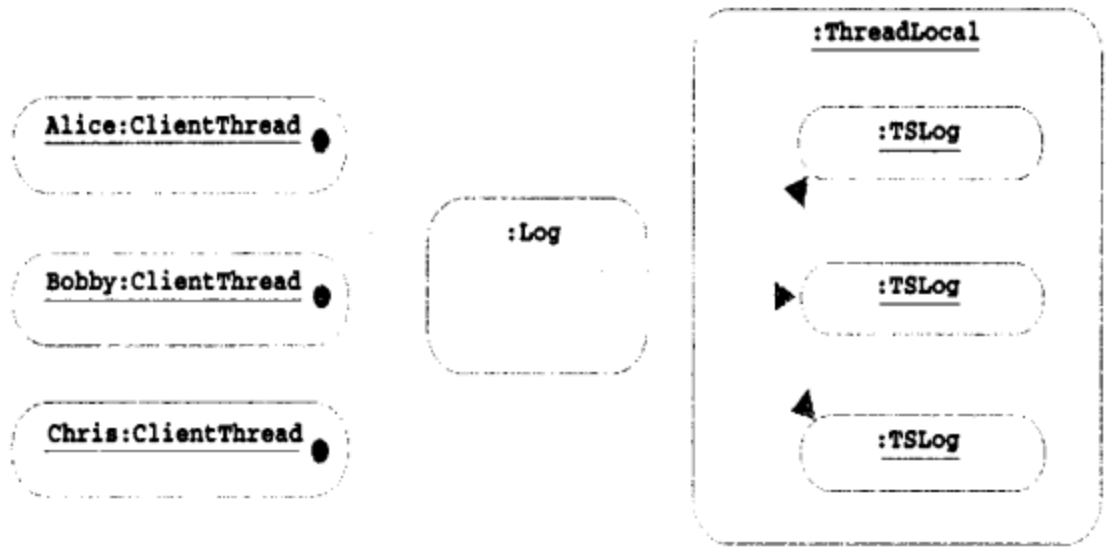


图 11-7　范例程序 2 的时序图（使用 Thread-Specific Storage Pattern）

Thread-Specific Storage Pattern 的所有参与者

Thread-Specific Storage Pattern 的所有参与者如下（由于参与者的名字都一一加上 Thread-Specific 实在太冗长，所以都省略成 TS）：

◆ *Client（委托人）参与者*

Client 参与者会将工作委托给 TSObjectProxy 参与者。一个 TSObjectProxy 参与者可由多个 Client 参与者一起使用。

范例程序 2 中，Client 参与者是 ClientThread 类。

◆ *TSObjectProxy（线程独有对象的代理者）参与者*

TSObjectProxy 参与者会处理多个 Client 参与者委托的工作。

首先，TSObjectProxy 参与者会使用 TSObjectCollection 参与者，取得 Client 参与者所对应的 TSObject 参与者。并将工作委托（delegation）给 TSObject 参与者。

范例程序 2 中，TSObjectProxy 参与者是 Log 类。

◆ *TSObjectCollection（线程独有对象的集合）参与者*

TSObjectCollection 参与者拥有 Client 参与者与 TSObject 参与者的对照表。

当 getTSObject 方法被调用时，就检查对照表，返回 Client 参与者所对应的 TSObject 参与者。而 setTSObject 方法被调用时，则在对照表里设置 Client 参与者与 TSObject 参与者的组合。

范例程序 2 中，TSObjectCollection 参与者是 java.lang.ThreadLocal 类。

◆ *TSObject（线程独有的对象）参与者*

TSObject 参与者存放有线程特有的信息。

TSObject 参与者由 TSObjectCollection 参与者所管理。TSObject 参与者的方法只会由单线程调用。

范例程序 2 中，TSObject 参与者是 TSLog 类。

类图如图 11-8 所示。

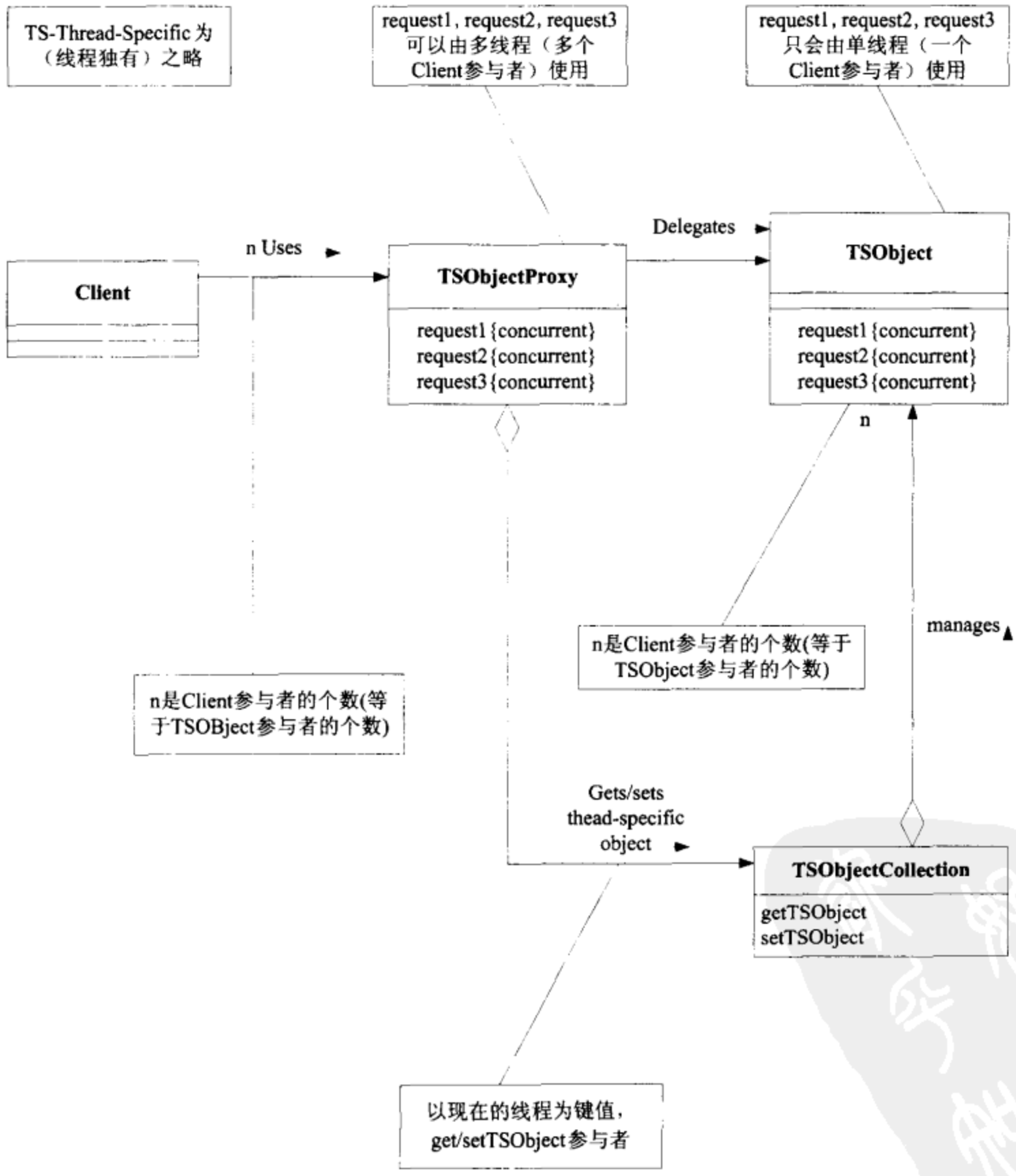


图 11-8　Thread-Specific Storage Pattern 的类图

图 11-9 是第一次调用 TSObjectProxy 参与者时，建立新 TSObject 参与者时的样子。

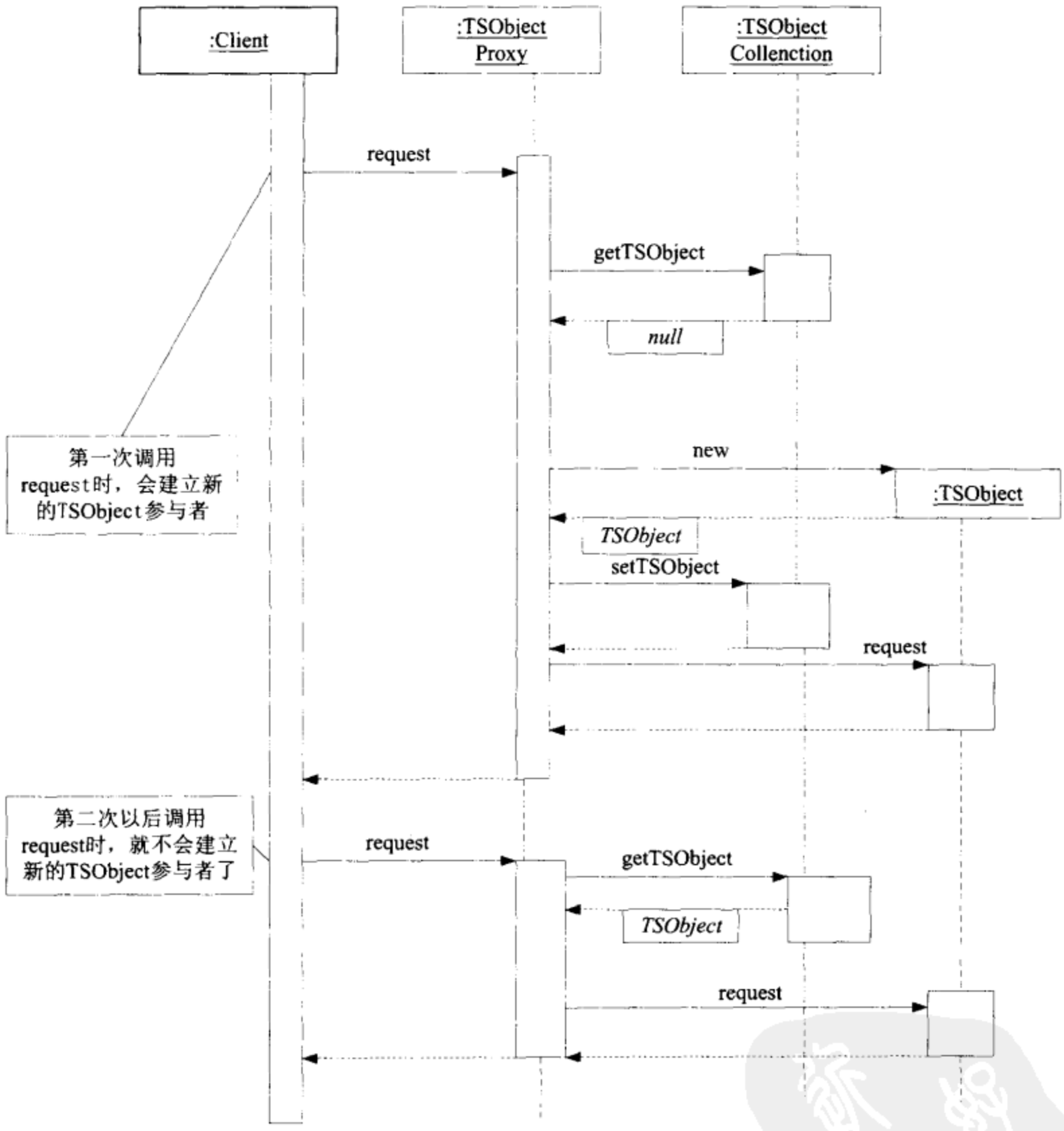


图 11-9　Thread-Specific Storage Pattern 的顺序图（建立新的 TSObject 参与者）

图 11-10 则是多个 Client 参与者使用自己专用的 TSObject 参与者时的样子。

注意：本书所介绍的 Thread-Specific Storage Pattern，与附录 E [POSA2]实现的方式不同。

本书中，TSObjectProxy 参与者拥有 1 个 TSObjectCollection 参与者。而以 java.lang.ThreadLocal 类作为 TSObjectCollection 参与者。

而[POSA2]中，TSObjectProxy 参与者拥有多个 TSObjectSet 参与者，TSObjectSet 参与者就对应到每条线程（本书没有出现 TSObjectSet 参与者）。java.lang.ThreadLocal 类就是 TSObjectProxy 参与者。

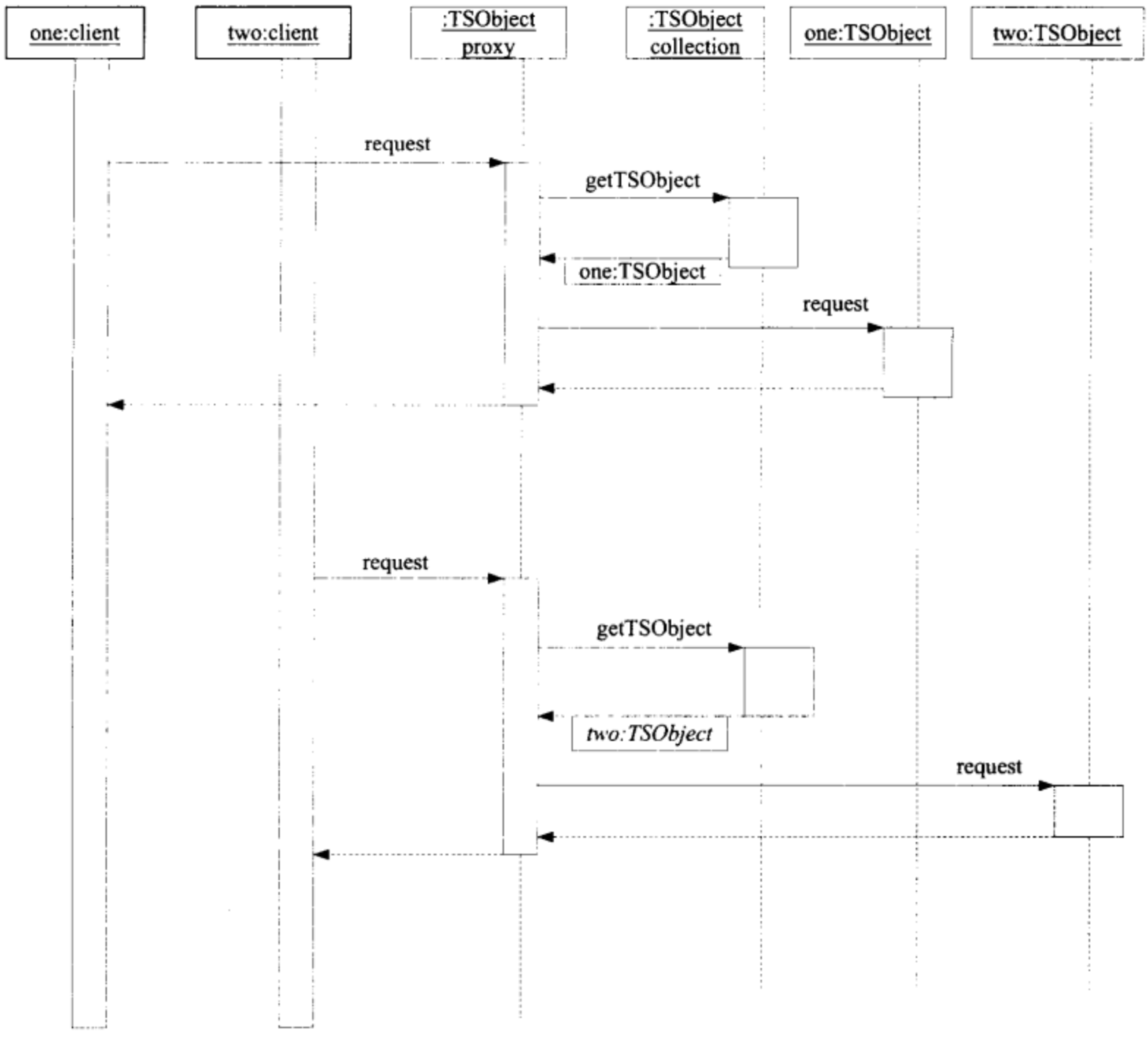


图 11-10　Thread-Specific Storage Pattern 的顺序图
（多个 Client 参与者会访问自己的 TSObject 参与者）

扩展思考方向的提示

局部变量与 java.lang.ThreadLocal 类

线程本来就有其特有的区域，就是存放方法局部变量的堆栈。在方法里分配的局部变量，都是线程所独有，无法由其他线程访问。但是，这些变量一退出方法就

会消失。而 ThreadLocal 则是与方法调用无关，为线程分配特有空间的类。

▶▶ 补充说明：局部变量

请注意上述的说明是单纯化之后的结论。虽然局部变量无法由其他线程访问，但局部变量所存放的实例，就可能由其他线程访问了。因为这个实例可能是以方法参数的形式，从外面传进来的：

```
void method(Object givenobj) {
    Object var1 = givenobj;
    object var2 = new Object;
    ...
}
```

这时，局部变量 var1、var2，都不可能被调用 method 方法以外的线程访问。但是，var1 所存放的 givenobj 就有可能被其他线程所访问了。像这种情况的范例，我们曾经在 Immutable Pattern（第 2 章）的练习问题 2-4 讨论过了。

放置线程特有信息的地方

放置线程特有信息的“地方”，有两种：

（1）线程外（thread-external）

（2）线程内（thread-internal）

下面依次解说：

◆ 将特有信息放在线程外（*thread-external*）

范例程序 2 中所出现的线程特有信息，就是 TSLog 的实例。范例程序 2 中，所有 TSLog 的实例。范例程序 2 中，所有 TSLog 的实例都存放在 Log 类所拥有的 java.lang.ThreadLocal 的实例里。

ThreadLocal 的实例可说是保管箱间。每个线程的保管箱，都集中在保管箱间里。线程不会带着保管箱到处跑。

像这样，将线程特有的信息放在线程外的方式，称为线程外（thread-external）。使用将线程特有的信息放外面的方式时，不需要修改表示线程的现有类（范例程序 2 中是 ClientThread 类），所以可以适用于任何线程。相对地，也可能会有线程类不易阅读的危险性。因为只阅读线程的原始程序代码，并无法得知其他类中也存放有线程的信息。

◆ 将特有信息放在线程内（*thread-internal*）

假设现在建立一个 Thread 类的子类 MyThread。我们声明的 MyThread 的字段，就

是线程特有的信息。这时，我们说是将线程特有的信息放在线程内（thread-internal）。

我们使用“拥有”这个词，可能表示“所有”，也可能表示“带着”。我们说我所“拥有”的钱，并不表示我就“带着”钱（或许会把钱存在名为银行的TSObjectCollection参与者里）。

将特有信息存放在线程外的方式，就像是自己所有，但没有带在身上；而将特有信息存放在线程内的方式，就像是拿在自己手上一样。

不必担心被其他线程访问

thread-specific storage是“线程独有的内存空间”的意思。“线程独有”说起来好像文诌诌的，其实也可以说成是“线程自己的”、“线程每个人各自专属的”。意即，再换个说法，可说是不用担心被其他线程擅自乱动的。

请仔细思考这一点，“其他线程无法擅自乱动”是很重要的性质。虽说多线程程序设计中，共享互斥是很重要的，但要确实现到共享互斥并不容易。我们在这一本书中，学过对线程进行共享互斥，使共享的实例不会损坏，但又知道共享互斥会使执行性能偏低，所以要力求将共享互斥的范围尽可能缩小。

Thread-Specific Storage Pattern提供了以线程为键值，访问线程独有对象的机制。这个对象（具体来说是范例程序2的TSLog的实例）会为每个线程分别分配，而且绝对会被其他线程碰到。但是，Thread-Specific Storage Pattern出现的参与者，没有进行共享互斥的操作（其实可能有，而被隐藏在TSObjectCollection参与者里）。这是非常方便的架构。

在这里我们更进一步思考。仔细一想，需要对线程进行共享互斥，绝对是因为存在有“共享的东西”。因为共享互斥为了保护共享的东西。在Single Threaded Execution Pattern（第1章）中，我们以SharedResource参与者来表示这个共享的东西。Thread-Specific Storage Pattern里没有出现共享互斥，是因为没有共享的东西（SharedResource参与者）。

请再看一次Thread-Specific Storage Pattern的类图（见图11-8）。会被多个线程访问的，是TSObjectProxy参与者。而这个TSObjectProxy参与者会（使用TSObjectCollection参与者）分配TSObject参与者给每个线程。Thread-Specific Storage Pattern中，多个线程所共享的是TSObjectProxy参与者，实际的工作则是在将TSObject参与者分配给线程以后进行。也就是使用TSObjectProxy参与者来分配不该被共享的TSObject参与者。

throughput的提升取决于实现

Thread-Specific Storage Pattern并没有进行共享互斥。可能会有人认为这样程序的throughput会比起使用Single Threaded Execution Pattern（第1章）时来得高。但

是，这并不是绝对的。就像前面所说，互斥控制可能会隐藏 TSObjectCollection 参与者里。另外，通过 TSObjectProxy 参与者调用方法时，每次都要从 TSObjectCollection 参与者取得 TSObject 参与者，也要花一些时间。

使用 Thread-Specific Storage Pattern，与其将着眼点放在 throughput 上，更重要的是程序的复用性：

（1）不需要改变程序的结构；

（2）共享互斥不在表面上出现，可避免犯错。

关于（1），相信读者看过范例程序 1 改成范例程序 2 所需要作的操作，就有所体悟了。而（2）则是避免自己实现 Single Threaded Execution Pattern 时忘记加 synchronized，或是实现 Guarded Suspension Pattern（第 3 章）时，写错警戒条件等等。可避免程序出现死锁的危险性。

隐藏 context 的危险性

Thread-Specific Storage Pattern 中，TSObjectCollection 参与者会自己判断现在的线程。也就是说，线程相关的信息，不需要以自变量传入。这是在程序中加入 context（前后文；配置；前后关系）的概念。context 虽然很方便，但也具有危险性，因为这会使处理时需要的信息躲过程序员的眼睛。

context 的概念，在计算机绘图处理时常常用到。与计算机绘图有关的链接库提供的方法，通常自变量都有数量较多的倾向。这是因为就光要画出一条线，就需要坐标值、坐标系、单位、颜色、线段粗细、画笔类型等，非常多的信息。为了设法减少自变量的量，就会定义 graphics context 或是 device context 等数据结构。而对链接库指示“现在调用的方法，都请在这个 context 来处理”。context 可使程序结构简单化。可是相对的，可能带来识别程序使用到的信息变得暧昧模糊的危险性。context 的危险性，跟全局变量的危险性差不多。

当我们要寻找 bug 发生的原因时，会紧盯信息的流程追踪。例如为了检查传递给方法的数据是否正确时，就会尝试将参数的值进行合法性判断并输出显示出来。但是，如果使用了 context，解决起来就困难多了。因为程序以前的行为，也有可能会使 context 出现异常，而是造成现在的 bug 的真正原因。

相关 Pattern

Singleton Pattern（参考附录 E [GoF][Yuki01]）

Singleton Pattern 可以用来保证只建立出 1 个实例。

Thread-Specific Storage Pattern 可以分配线程独有的空间。也就是可以看作是以

线程为单位的 Singleton（附录 E [Lea]的说法）。

下列的网页也有相关数据：

- Using ThreadLocal to implement a per-thread Singleton
  http://www-106.ibm.com/developerworks/java/library/j-threads3.html

Worker Thread Pattern（第 8 章）

Thread-Specific Storage Pattern 中，执行时所需的信息是线程自己所有的。即使是存放在保管箱里的信息，能够打开保管箱的，也只有线程本身。

Worker Thread Pattern 中，每个工人线程做完一件工作后就重复使用于下一件工作。

所以，Worker Thread Pattern 与 Thread-Specific Storage Pattern 很难同时使用。

Single Threaded Execution Pattern（第 1 章）

Thread-Specific Storage Pattern 给每个线程自己独自拥有的空间。因为能够访问这个空间的，只有一个线程，所以不需要使用 Single Threaded Execution Pattern 进行共享互斥。

Proxy Pattern（参考附录 E [GoF][Yuki01]）

Proxy Pattern 可以不更改对象的接口（API），但更改对象的访问方式。Thread-Specific Storage Pattern 中，TSObjectProxy 参与者担任 TSObject 参与者的代理者的部分，使用了 Proxy 参与者。

进阶说明：Actor-based 与 Task-based

线程与线程之间使用信息的关系，可分为 Actor-based 与 Task-based 两种思考模式。这一章的进阶说明，就要解释这两者的差异。

主体与客体

组装模型玩具时，下列两者都是必要的：

- 组装模型的人
- 一套模型（说明书和零件）

只有组装模型的人，或是只有一套模型，都无法组装出模型。

那么，现在想要使用线程进行某些工作时：

- 用来进行工作的线程
- 进行工作所需的信息

两者都是必要的。

无论是拼装玩具模型，或是线程进行工作，都需要

- 进行某件事所需的“主体”
- 进行某件事所需的“客体”

两者兼具才行。

设计使用到线程的程序时，会因为重点放在“主体”与“客体”的不同，出现两种开发方式：

- Actor-based 注重主体
- Task-based 注重客体

Actor-based 的思维

所谓 Actor-based，用一句话说，就是“偏重于线程”的开发方式。

Actor-based 开发方式中，代表线程的实例，会拥有进行工作所需的信息（context、状态）。这样一来，可降低线程之间需要传递的信息。每个线程会使用其他线程所传来的信息进行处理，改变自己的内部状态。这种线程通常称为 actor（操作者）。

我们可建立 Thread 类的子类，并定义自己的字段，将工作相关的信息都放入字段中。这就是在建立“小型的 actor”。

Actor 长得大概像这样：

```
class Actor extends Thread {
        操作者的内部状态
        public void run() {
            从外部取得任务，改变自己内部状态的循环
        }
}
```

第 12 章所要学的 Active Object Pattern，就是一种操作者。

Task-based 的思维

Task-based，用一句话说，就是“偏重于工作”的开发方式。

Task-based 的开发方式，就是不将信息（context、状态）放到线程里。而是把信息放在线程之间所传递的实例里。并不是只有数据，包括用来执行工作的方法，也放在这个实例里。这种线程间传递的实例称为“消息”、“请求”或“命令”。在这里我们又称它为 task（任务）。任务内含有足够的信息，所以任务可以由任何线程来进行。这个开发方式，可使巨大的任务能在轻巧的线程之间往来。

这种开发方式最典型的就是 Worker Thread Pattern（第 8 章）。

我们可以将工作时需要的信息，定义在 implements Runnable 的类里，再定义 run

方法，使它能够交由线程执行。这就建立出“小型的 task”了。将建立出的任务交给线程后，工作就能够被执行了。任务长得大概像这样：

```
class Task implements Runnable {
        执行任务所需的信息
        public void run() {
            执行任务所需的处理内容
        }
}
```

▶▶ 补充说明：java.util.Timer 类与 java.util.TimerTask 类

java.util.TimerTask 就是一个 Task-based 的类。这个类实现了 java.lang.Runnable 接口，可由 java.util.Timer 类调用。java.util.TimerTask 类中可以记载在一定时间后要作的事，或是定期要执行的操作。

实际上两个开发方式是混用的

这里所说的 Actor-based 与 Task-based 两种开发方式，也不是完全分离的。即使是在一个程序里，两者也可能各占一定比例。一般来说，会是“task 在一群 actor 之间传递”的状况。

另外，Actor-based 与 Task-based 的分类，是 Doug Lea 所提出的。详见附录 E 的[Lea]。

重点回顾

这一章我们学习了 Thread-Specific Storage Pattern。

手边有一个假定单线程作为执行环境的对象。现在我们想将这个对象放在多线程环境下执行。这时，我们又不想修改使用端的线程，也不能改变对象的接口（API）。

但是，要对线程进行共享互斥，以支持多线程，是很困难的。一不小心就可能使对象丧失安全性，或是发生死锁、使线程丧失生命性。

这时就使用 Thread-Specific Storage Pattern。在此将目的对象当作 TSObject 参与者，并建立与 TSObject 参与者具有相同接口（API）的 TSObjectProxy 参与者。另外，为了管理“Client 参与者→TSObject 参与者”的对照表，又加上了 TSObjectCollection 参与者。TSObjectProxy 参与者会使用 TSObjectCollection 参与者，取得现在的线程所对应的 TSObject 参与者。而将工作委托给这个 TSObject 参与者。

像这样，不需要修改使用端的线程，也不需要修改对象的接口（API）。而且，TSObjecy 参与者一定会由特定的线程调用，这个部分也不需要互斥控制。关于多线

程的部分，则隐藏在 TSObjectCollection 参与者里。

Java 语言中，java.lang.ThreadLocal 类就担任了 TSObjectCollection 参与者的工作。

不过，使用 Thread-Specific Storage Pattern，等于是对程序引进 context 概念，可能对程序带来不易阅读的危险。

这就是 Thread-Specific Storage Pattern。

那么，就进入练习问题吧。

练习问题

解答请见附录 A

问题 11-1（基础知识的确认）

Java 请就您对范例程序 2 的了解，仔细阅读下列语句，正确的请回答○，不正确的请回答×。

（1）产生 3 个 ThreadLocal 的实例。

（2）产生 3 个 PrintWriter 的实例。

（3）ThreadLocal 的 set 方法被调用出来 3 次。

（4）ThreadLocal 的 get 方法会被调用 3 次。

（5）产生 3 个 TSLog 的实例。

（6）产生 3 个 Log 的实例。

问题 11-2（不需要 synchronized 的理由）

Java 关于范例程序 2，请回答下列问题：

（1）请说明 TSLog 类的 println 方法与 close 方法不需要 synchronized 的理由。

（2）请说明 Log 类的 println 方法与 close 方法不需要 synchronized 的理由。

问题 11-3（线程的终止处理）

Java 请修改范例程序 2，使线程在结束前可以不需要关闭 log。也就是说，在 ClientThread 类里，可以不需要明确地执行

```
Log.close();
```

线程在结束时，就会自己关闭掉 log 文件。

问题 11-4（构造器）

Java 我们希望把构造器被调用的操作记录在 log 文件里，故将范例程序 2 的 ClientThread 类（List 11-5）修改成 List 11-7 的样子。

然而，实际执行时却发现 Alice-log.txt、Bobby-log.txt、Chris-log.txt 的内容都没有出现"constructor is called."字符串，请问为什么？

List 11-7 修改后的 ClientThread 类（ClientThread.java）

```
public class ClientThread extends Thread {
    public ClientThread(String name) {
        super(name);
        Log.println("constructor is called.");
    }
    public void run() {
        System.out.println(getName() + " BEGIN");
        for (int i = 0; i < 10; i++) {
            Log.println("i = " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        Log.close();
        System.out.println(getName() + " END");
    }
}
```

问题 11-5（thread-internal 信息的范例）

Java 请举例说明 java.lang.Thread 类在线程内（thread-internal）所拥有的线程特有的信息。

第 12 章

Active Object——接受异步消息的主动对象

Active Object Pattern

公司里有很多人在工作。柜台人员、营业人员、生产线的人员、管理人员、负责产品发送、搬运的人员等，各种人员互相协调，组成一个公司。公司整体联在一起，彷佛是有一个人的人格——所以叫做“法人”。

这一章要学习的是 Active Object Pattern。

active 是“主动的”的意思，所以 active object 是“主动对象”的意思。“主动的”一般是指“自己拥有独立的线程”的意思。所以 Java 的 java.lang.Thread 类的实例，也是一种主动对象。

不过，Active Object Pattern 里出现的“主动对象”，并不只是“自己拥有独立的线程”而已。它还具有可从外部接收异步消息，并能配合需要返回处理结果的特征。

使用 Active Object Pattern 组织成“主动对象”，就很像是一群个人集合在一起构成法人一样。虽然使用没有提供异步消息的 Java 语言，但是使用 Active Object Pattern，就可以组织出能够处理异步消息的“主动对象”。

Active Object Pattern 中使用到了 Producer-Consumer Pattern（第 5 章）、Thread-Per-Message Pattern（第 7 章）、Future Pattern（第 9 章）等 Pattern。故在阅读此章前，强烈建议读者先阅读上面说的这几章。

Active Object Pattern，又称为 Actor Pattern（参考附录 E [Lea]）或 Concurrent Object Pattern（参考附录 E [POSA2]）。

范例程序

在这里的范例程序所要建立的是一个具有“建立字符串（makeString）”、“显示字符串（displayString）”两个功能（可处理两种异步消息）的“主动对象”。

类与接口一览如表 12-1 所示。与“主动对象”相关的类与接口，都定义在 activeobject 包里。凡是 activeobject 包里 public 的类与接口，在表格里以粗体字表示。public 的类与接口可以从 activeobject 包的外面参考。

类图如图 12-1 所示。因为图表实在太复杂，图 12-1 中省略了 Main、DisplayClientThread、ActiveObjectFactory 等类。顺序图则如图 12-2 所示。

表 12-1　类与接口一览表（粗体字是 activeobject 包里 public 的东西）

包	名　称	说　明
匿名	Main	操作测试用的线程
匿名	MakerClientThread	请求建立字符串的线程
匿名	DisplayClientThread	请求显示字符串的线程

续上表

包	名　称	说　明
activeobject	**ActiveObject**	规定有“主动对象”接口（API）的接口
activeobject	**ActiveObjectFactory**	产生“主动对象”的类
activeobject	Proxy	将方法调用转换成 MethodRequest 对象（实现 ActiveObject 接口）的类
activeobject	SchedulerThread	execute MethodRequest 对象的类
activeobject	ActivationQueue	MethodRequest 对象可依次存放的类
activeobject	MethodRequest	表示请求的抽象类
activeobject	MakeStringRequest	makeString 方法（建立字符串）所对应的类，MethodRequest 类的子类
activeobject	DisplayStringRequest	displayString 方法（显示字符串）所对应的类，MethodRequest 类的子类
activeobject	**Result**	表示执行结果的抽象类
activeobject	FutureResult	Future Pattern 中用来显示执行结果的抽象类
activeobject	RealResult	表示实际执行结果的类
activeobject	Servant	进行实际处理的类（实现 ActiveObject 接口）

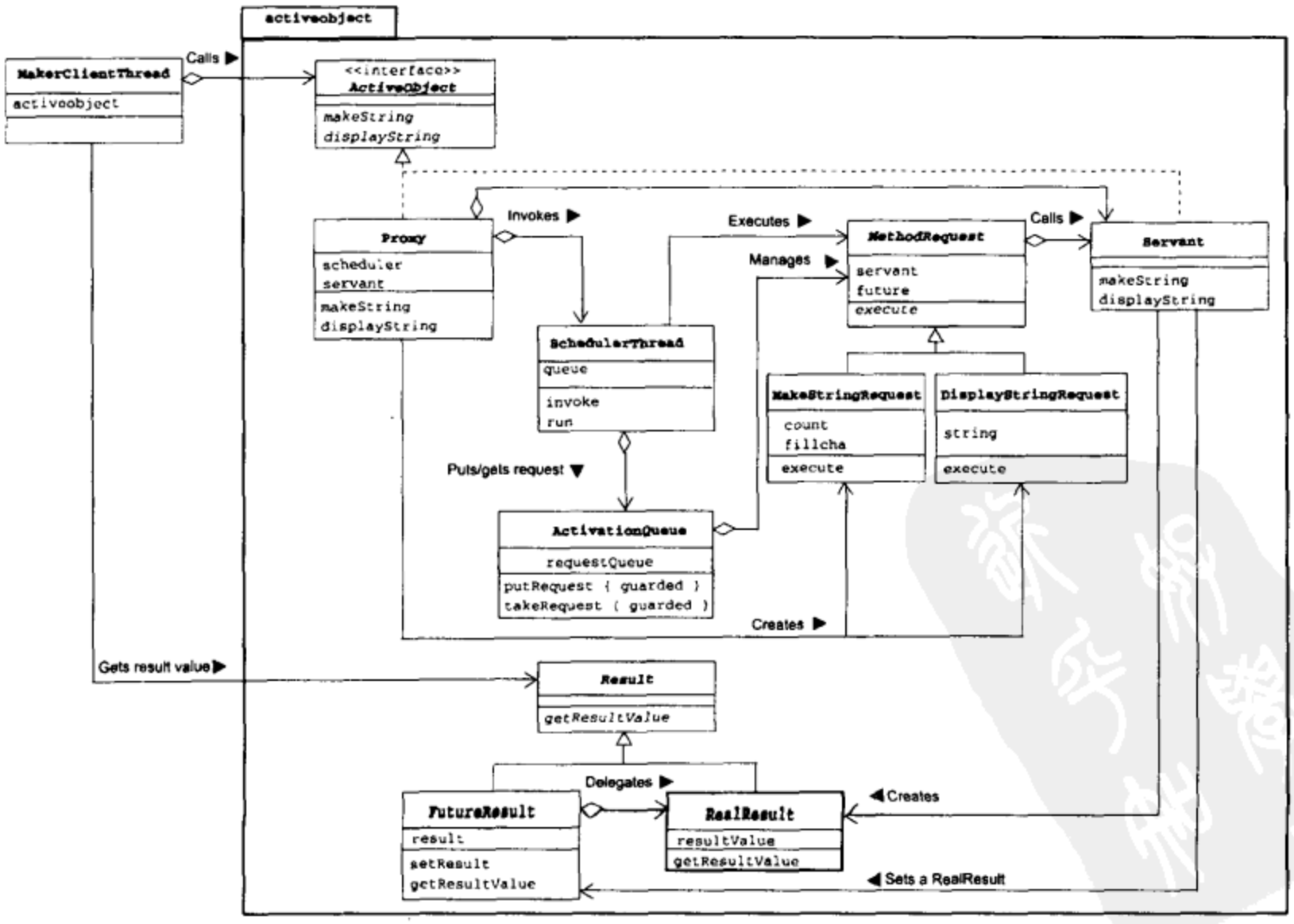


图 12-1　范例程序的类图

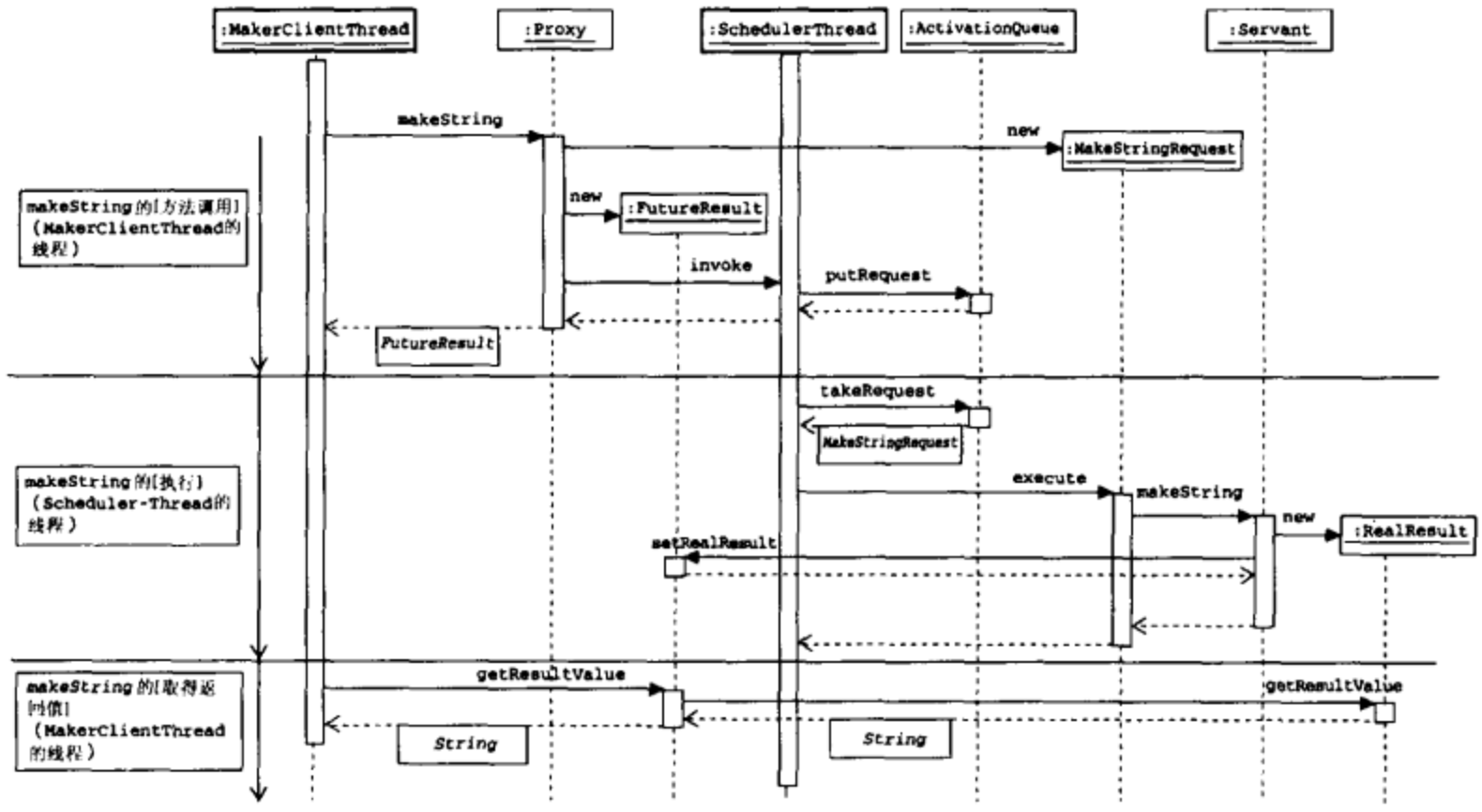


图 12-2　范例程序的类图

使用端：Main 类

Main 类（List 12-1）用来操作作测试。Main 类使用 ActiveObjectFacorty 类的 createActiveObject 方法，建立一个“实现 ActiveObject 接口的类的实例”。并启动 MakerClientThread 与 DisplayClientTherad 线程。

在此建立了两个 MakerClientThread 的实例（Alice 与 Bobby），与一个 DisplayClientTherad 线程的实例（Chris）。

因为“实现 ActiveObject 接口的类的实例”过于冗长不易阅读，后文就称之为“ActiveObject 对象”。

List 12-1　操作测试用的 Main 类（Main.java）

```
import activeobject.ActiveObject;
import activeobject.ActiveObjectFactory;

public class Main {
    public static void main(String[] args) {
        ActiveObject activeObject = ActiveObjectFactory.
createActiveObject();
      new MakerClientThread("Alice", activeObject).start();
      new MakerClientThread("Bobby", activeObject).start();
      new DisplayClientThread("Chris", activeObject).start();
    }
}
```

使用端：MakerClientThread 类

MakerClientThread（List 12-2）类是用来调用“主动对象”的 makeString 方法（建立字符串）的线程。

activeObject 字段，存放着 ActiveObject 对象。MakerClientThread 类，会调用这个对象的 makeString 方法。

调用 makeString 方法以后，会马上回来。这个调用操作等同于对“主动对象”送出“建立字符串”这个异步消息。

makeString 方法的返回值是 Result 类型，但实际上这里使用了 Future Pattern（第 9 章）。实际的返回值要使用 getResultValue 方法获取。

程序在这里虽然使用 Thread.sleep 方法停住约 10 毫秒，这个位置在正式使用时，是拿来执行其他的操作。

List 12-2 MakerClientThread 类（MakerClientThread.java）

```
import activeobject.ActiveObject;
import activeobject.Result;

public class MakerClientThread extends Thread {
    private final ActiveObject activeObject;
    private final char fillchar;
    public MakerClientThread(String name, ActiveObject activeObject) {
        super(name);
        this.activeObject = activeObject;
        this.fillchar = name.charAt(0);
    }
    public void run() {
        try {
            for (int i = 0; true; i++) {
                // 有返回值的调用
    Result result = activeObject.makeString(i, fillchar);
    Thread.sleep(10);
    String value = (String)result.getResultValue();

System.out.println(Thread.currentThread().getName() + ": value =
" + value);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

使用端：DisplayClientThread 类

DisplayClientThread 类（List 12-3）与 MakerClientThread 类一样，是使用 ActiveObject 对象的线程。

DisplayClientThread 线程会调用“主动对象”的 displayString 方法（显示字符串）。

与 MakerClientThread 相同，调用 displayString 后会马上退出。这个调用操作，相当于对“主动对象”送出“显示字符串”的异步消息。这里显示的字符串（string），是在线程的名称后面加上 0、1、2、……的流水号。

displayString 没有返回值。所以不需要使用 Future Pattern（第 9 章）。

List 12-3 DisplayClientThread 类（DisplayClientThread.java）

```
import activeobject.ActiveObject;
import activeobject.Result;

public class DisplayClientThread extends Thread {
   private final ActiveObject activeObject;

    public DisplayClientThread(String name, ActiveObject activeObject){
        super(name);
        this.activeObject = activeObject;
    }
    public void run() {
        try {
            for (int i = 0; true; i++) {
                // 没有返回值的调用
                String string = Thread.currentThread().getName() +
" " + i;
                activeObject.displayString(string);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

“主动对象”端：ActiveObject 接口

以上是使用 ActiveObject 对象的使用端。接下来就是 ActiveObject 的实现端，也就是“主动对象”的程序部分。

首先是 ActiveObject 接口（List 12-4）。这个接口定义了“主动对象”的接口（API）。因为 ActiveObject 接口是 public 的，所以可从 activeobject 包外看见。

List 12-4 ActiveObject 接口（ActiveObject.java）

```
package activeobject;

public interface ActiveObject {
    public abstract Result makeString(int count, char fillchar);
    public abstract void displayString(String string);
}
```

“主动对象”端：ActiveObjectFactory 类

ActiveObjectFactory 类（List 12-5）用来构建 ActiveObject 对象。这个类具有 createActiveObject 类方法。这个方法可用来建立 ActiveObject 对象。

要建立 ActiveObject 对象，必须组合下面 4 个类的实例：[注1]

- Servant 用来进行实际处理的类（实现了 ActiveObject 接口）；
- ActivationQueue 用来依次存放 MethodRequest 对象的类；
- SchedulerThread 用来 execute MethodRequest 对象的类；
- Proxy 将方法调用转换成 MethodRequest 对象的类（实现了 ActiveObject 接口）。

因为 ActiveObjectFactory 类是 public，所以可以从包外看见。

List 12-5 ActiveObjectFactory 类（ActiveObjectFactory.java）

```
package activeobject;

public class ActiveObjectFactory {
    public static ActiveObject createActiveObject() {
        Servant servant = new Servant();
        ActivationQueue queue = new ActivationQueue();
        SchedulerThread scheduler = new SchedulerThread(queue);
        Proxy proxy = new Proxy(scheduler, servant);
        scheduler.start();
        return proxy;
    }
}
```

注 1：本书为了使读者较容易理解 AcriveObject 对象的产生，引进了 ActiveObjectFactory 的概念。但附录 E 的[POSA2]书中所描述的 Active Object Pattern，并没有这个类。

“主动对象”端：Proxy 类

Proxy 类（List 12-6）是 MakerClientThread 与 DisplayClientThread 会调用的类。Proxy 的工作，是将方法调用转换成对象（实例）。

请注意 makeString 方法。在这里，使用了 makeString 的参数，建立出 MakeStringRequest 的实例。并将这个实例传给 scheduler 的 invoke 方法。也就是说将

```
activeObject.makeString( … )
```

这样的方法调用，转换成

```
new MakeStringRequest( … )
```

的实例形式[注2]。

这里进行的操作有何意义呢？其实这就是曾经以前说过的“分离方法的启动（invocation）与执行（execution）”。“启动”与“执行”中，Proxy 类进行相当于“启动”的操作。因为不进行“执行”，所以线程可以马上从 Proxy 的 makeString 方法退出。

makeString 的返回值，是 FutureReselt 的实例。从类名称就可以看出，这里使用了 Future Pattern（第 9 章）。

接着，我们要看的是 displayString。与 makeString 方法相比较之下，displayString 方法单纯得多。这是因为 displayString 是没有返回值的方法。displayString 会建立出 DisplayStringRequest 的实例，并传递给 scheduler 的 invoke 方法。因为没有返回值，所以不需要使用 Future Pattern。

Proxy 类虽然放在 activeobject 包内，但要注意的是 makeString 方法、displayString 方法是由 MakerClientThread、DisplayClientThread 等线程执行的。在多线程程序设计中，千万要随时意识到每个方法是由哪个线程调用的。

因为 Proxy 类不是 public 的，所以无法从 activeobject 包外直接看到。Proxy 实现（implements）了 ActiveObject 接口，在包外部是把它当作 ActiveObject 接口访问。

List 12-6 Proxy 类（Proxy.java）

```
package activeobject;

class Proxy implements ActiveObject {
    private final SchedulerThread scheduler;
    private final Servant servant;
    public Proxy(SchedulerThread scheduler, Servant servant) {
        this.scheduler = scheduler;
```

注 2：因为方法调用转换成实例，所以就可以存放在队列里了。这是 Command Pattern（参考附录 E [GoF][Ykuki01]）的主要概念。

```
        this.servant = servant;
    }
    public Result makeString(int count, char fillchar) {
        FutureResult future = new FutureResult();
        scheduler.invoke(new MakeStringRequest(servant, future,
count, fillchar));
        return future;
    }
    public void displayString(String string) {
        scheduler.invoke(new DisplayStringRequest(servant, string));
    }
}
```

“主动对象”端：SchedulerThread 类

SchedulerThread 类（List 12-7）是 Active Object Pattern 的核心部分。

SchedulerThread 类中有一个队列（queue），用来存储 Proxy 类传来的请求。当 invoke 方法被调用后，queue 里会多出一个请求（request）。

因为 SchedulerThread 是 java.lang.Thread 类的子类，所以具有 run 方法。在 run 方法里，会反复从 queue 里取出一个请求（takeRequest），并执行（execute）之。

在 queue 里放入 request 的部分，相当于“主动对象”的方法的“启动（invocation）”。而执行 request 的 execute 方法的部分，则相当于方法的“执行（execution）”。

比较细心的读者，应该已经注意到下面两者的线程并不相同：

（1）调用 invoke 方法的线程；

（2）调用 execute 方法的线程。

（1）是调用 Proxy 类的 makeString 方法或 displayString 方法的线程。具体地说，就是 MakerClientThread、DisplayClientThread 所对应的线程。

相对地，（2）的则是 SchedulerThread 类所对应的线程。

从本书前面开始顺序渐进看到这里的读者，应该会注意到“啊！这是第 5 章的 Producer-Consumer Pattern 嘛”。的确如此。调用 invoke 方法的线程是 Producer 参与者，将 request putRequest 到 queue 里。而调用 execute 方法的线程则是 Consumer 参与者，从 queue 里 takeRequest 出来执行。使用 Producer-Consumer Pattern，就可以将 MakeStringRequest 的实例与 DisplayStringRequest 的实例安全地从 Producer 参与者传到 Consumer 参与者手里。

List 12-7 SchedulerThread 类（SchedulerThread.java）

```
package activeobject;

class SchedulerThread extends Thread {
```

```
    private final ActivationQueue queue;
    public SchedulerThread(ActivationQueue queue) {
        this.queue = queue;
    }
    public void invoke(MethodRequest request) {
        queue.putRequest(request);
    }
    public void run() {
        while (true) {
            MethodRequest request = queue.takeRequest();
            request.execute();
        }
    }
}
```

“主动对象”端：ActivationQueue 类

ActivationQueue 类（List 12-8）就相当于 Producer-Consumer Pattern（第 5 章）的 Channel 参与者。putRequest 方法可将请求放进队列，而 takeRequest 方法可从队列拿出请求。

在 List 12-8 的实现中，以常数 MAX_METHOD_REQUEST 指定可以存放的最大请求数。超过这个数量，还试图对队列加入请求的线程，会在 putRequest 方法里 wait 住。这是我们所熟悉的 Guarded Suspension Pattern（第 3 章）。相反地，当队列里没有存放任何请求，却尝试获取请求时，会在 takeRequest 方法里 wait 住。这也是 Guarded Suspension Pattern。

因为 putRequest、takeRequest 并没有声明 throws InterruptedException，所以我们可以知道 ActivationQueue 是设计成忽略 interrupt 的。

List 12-8 ActivationQueue 类（ActivationQueue.java）

```
package activeobject;

class ActivationQueue {
    private static final int MAX_METHOD_REQUEST = 100;
    private final MethodRequest[] requestQueue;
    private int tail;  // 下一个 putRequest 的地方
    private int head;  // 下一个 takeRequest 的地方
    private int count; // MethodRequest 的数量

    public ActivationQueue() {
        this.requestQueue = new MethodRequest [MAX_METHOD_ REQUEST];
```

```
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
    public synchronized void putRequest(MethodRequest request) {
        while (count >= requestQueue.length) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        requestQueue[tail] = request;
        tail = (tail + 1) % requestQueue.length;
        count++;
        notifyAll();
    }
    public synchronized MethodRequest takeRequest() {
        while (count <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
            MethodRequest request = requestQueue[head];
            head = (head + 1) % requestQueue.length;
            count--;
            notifyAll();
            return request;
        }
    }
```

“主动对象”端：MethodRequest 类

MethodRequest 类（List 12-9）表示抽象化的请求。

具体化的请求则是 MakeStringRequest 类与 DisplayStringRequest 类，这些都是 MethodRequest 的子类。

MethodRequest 类的 servant 字段，存放有用来进行实际处理的 Servant 的实例。

future 字段则存放有用来设置“返回值”的 FutureRequest 的实例。当请求不需要“返回值”时，就不使用这个字段，而存入 null。

servant 字段与 future 字段，都假定可让子类使用，所以声明成 protected。

execute 方法是抽象方法，并没有定义属性。这表示设计者“要求子类务必要实现 execute 方法”的意图。“实现 execute 方法”，就是定义执行这个请求时，应该处理的具体属性。

List 12-9 MethodRequest 类（MethodRequest.java）

```
package activeobject;

abstract class MethodRequest {
    protected final Servant servant;
    protected final FutureResult future;
    protected MethodRequest(Servant servant, FutureResult future)
{
        this.servant = servant;
        this.future = future;
    }
    public abstract void execute();
}
```

“主动对象”端：MakeStringRequest 类

MakeStringRequest 类（List 12-10）表示“建立字符串”。

MakeStringRequest 对应的是 ActiveObject 接口的 makeString 方法。实际上，MakeStringRequest 的实例，是由 Proxy 类的 makeString 方法建立的。

若您已经了解 Future Pattern（第 9 章），应该不难阅读这个类的程序代码。

MakeStringRequest 里定义有 count、fillchar 字段，这些字段刚好对应到 makeString 方法的参数。

execute 方法会调用 servant 的 makeString 方法。这是请求的“执行”。而对 future 作 setResult，则是请求的“设置返回值”。

MakeStringRequest 类用来表示 ActiveObject 接口的 makeString 方法。请确实了解方法要如何以一个类来表达，在此整理如下：

- makeString 方法的参数，对应到 MakeStringRequest 的实际字段；
- makeString 方法的启动，对应到建立 MakeStringRequest 的实例，并调用 SchedulerThread 的 invoke 方法；
- makeString 方法的执行，对应到调用 servant 的 makeString 方法；
- makeString 方法的返回值，对应到将 result setResult 到 future 里。

List 12-10 MakeStringRequest 类（MakeStringRequest.java）

```
package activeobject;

class MakeStringRequest extends MethodRequest {
```

```
    private final int count;
    private final char fillchar;
    public MakeStringRequest(Servant servant, FutureResult future,
int count, char fillchar) {
        super(servant, future);
        this.count = count;
        this.fillchar = fillchar;
    }
    public void execute() {
        Result result = servant.makeString(count, fillchar);
        future.setResult(result);
    }
}
```

“主动对象”端：DisplayStringRequest 类

DisplayStringRequest 类（List 12-11）表示“显示字符串”。

DisplayStringRequest 类对应的是 ActiveObject 接口的 displayString 方法。实际建立 DisplayStringRequest 实例的，则是 Proxy 类的 displayString 方法。

在 execute 方法里，会调用 servant 的 displayString 方法。与 MakeStringRequest 类（List 12-10）的最大差异，在于 displayString 方法没有“返回值”。因为没有“返回值”，所以不需要使用 Future Pattern（第 9 章）。

List 12-11 DisplayStringRequest 类（DisplayStringRequest.java）

```
package activeobject;

class DisplayStringRequest extends MethodRequest {
    private final String string;
    public DisplayStringRequest(Servant servant, String string)
{
        super(servant, null);
        this.string = string;
    }
    public void execute() {
        servant.displayString(string);
    }
}
```

“主动对象”端：Result 类

Result 类（List 12-12）相当于 Future Pattern（第 9 章）的 VirtualData 参与者。Result 类有两个子类。Future 参与者 FutureResult 类与 RealData 参与者 RealResult 类。

若您已经了解 Future Pattern，上述的说明应该已经足够了。使用 Pattern 的好处就是可以轻松地与知道这个 Pattern 的程序员互相沟通。

Result 的 getResultValue 方法是抽象方法，也就是希望由子类实现。

List 12-12 Result 类（Result.java）

```
package activeobject;

public abstract class Result {
    public abstract Object getResultValue();
}
```

“主动对象”端：FutureResult 类

FutureResult 类（List 12-13）是 Future Pattern（第 9 章）的 Future 参与者，也就是用来管理“返回值”的类。要设置“返回值”时可使用 setResult 方法，要获取“返回值”时则可使用 getResultValue 方法。

getResultValue 方法使用了 Guarded Suspension Pattern（第 3 章），防卫条件是“返回值已经设置了”。

List 12-13 FutureResult 类（FutureResult.java）

```
package activeobject;

class FutureResult extends Result {
private Result result;
    private boolean ready = false;
    public synchronized void setResult(Result result) {
        this.result = result;
        this.ready = true;
        notifyAll();
    }
    public synchronized Object getResultValue() {
        while (!ready) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return result.getResultValue();
    }
}
```

“主动对象”端：RealResult 类

RealResult 类（List 12-14）是 Future Pattern（第 9 章）的 RealData 参与者。这是为了使用 Future Pattern，而将 resultValue 字段包装（wrap）起来的类。为了增加程序通用性，resultValue 的类型设置成 java.lang.Object。就范例程序而言，其实声明成 java.lang.String 也可以。

List 12-14 RealResult 类（RealResult.java）

```
package activeobject;

class RealResult extends Result {
    private final Object resultValue;
    public RealResult(Object resultValue) {
        this.resultValue = resultValue;
    }
    public Object getResultValue() {
        return resultValue;
    }
}
```

“主动对象”端：Servant 类

因为这一章的范例程序中类数量真的很多，或许有些读者已经看到快要打哈欠了。不过，请揉揉眼睛吧，这里是很重要的地方。

Servant 类（List 12-15）是 ActiveObject 端实际进行处理的部分。Servant 类具有 makeString 方法与 displayString 方法，在里面进行实际的处理。

而这里正是精彩的地方，Servant 类实现了 ActiveObject 接口。也就是说，Proxy 类与 Servant 类，都实现了相同的 ActiveObject 接口。

这样一来，

- “主动对象”可以启动的各种方法（由 Proxy 类实现）；
- 实际执行“主动对象”的各种方法（由 Servant 类实现）。

上述两者“各种方法”，可在程序代码上明确表现出都是一致的。

再进一步，请注意对包外公开的部分：

- ActiveObject 接口　　public 的部分；
- Proxy 类　　非 public（默认访问权限）；
- Servant 类　　非 public（默认访问权限）。

“主动对象”的接口（API），以 ActiveObject 接口的形式对包外公开。而 Proxy 类、Servant 类这些实现的部分，是隐藏在 activeobject 包里的。

List 12-15 Servant 类（Servant.java）

```
package activeobject;

class Servant implements ActiveObject {
    public Result makeString(int count, char fillchar) {
        char[] buffer = new char[count];
        for (int i = 0; i < count; i++) {
            buffer[i] = fillchar;
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        return new RealResult(new String(buffer));
    }
    public void displayString(String string) {
        try {
            System.out.println("displayString: " + string);
            Thread.sleep(10);
        } catch (InterruptedException e) {
        }
    }
}
```

执行范例程序

执行范例程序后，结果如图 12-3 所示。Alice 会以 makeString 方法建立连续 A 字母的字符串（空字符串、A、AA、AAA、……），Bobby 会以 makeString 方法建立连续 B 字母的字符串（空字符串、B、BB、BBB、……）。另一边，Chris 会以 displayString 方法重复显示 Chris 0、Chris 1、Chris 2、……字符串。

ActiveObject 对象会在进行 Alice、Bobby 的处理（建立字符串）时，也会接受 Chris 线程传来的显示字符串请求。因为 Alice 与 Bobby 建立的字符串越来越长，可明显看出处理所花的时间也渐渐增加。所以从执行结果可以看出，Chris 的部分（显示字符串）也渐渐连在一起出现。

```
displayString: Chris 0
Alice: value =
Bobby: value =
Alice: value = A
Bobby: value = B
Alice: value = AA
displayString: Chris 1
Bobby: value = BB
```

（续上图）

```
displayString: Chris 2
Alice: value = AAA
displayString: Chris 3
displayString: Chris 4
Bobby: value = BBB
Alice: value = AAAA
displayString: Chris 5
displayString: Chris 6
Bobby: value = BBBB
displayString: Chris 7
displayString: Chris 8
Alice: value = AAAAA
displayString: Chris 9
displayString: Chris 10
Bobby: value = BBBBB
displayString: Chris 11
displayString: Chris 12
Alice: value = AAAAAA
displayString: Chris 13
displayString: Chris 14
displayString: Chris 15
(以下省略。CTRL+C 结束程序)
```

图 12-3　执行范例

Active Object Pattern 的所有参与者

Active Object Pattern 的所有参与者如下所示。因为参与者非常的多，所以请与图 12-4 的类图对照阅读，会比较容易理解。另外，这个 Pattern 融入了一些本书其他章节学过的 Pattern，建议读者在阅读时可以一边确认。

◆ *Client（委托人）参与者*

Client 参与者会调用 ActiveObject 参与者，将工作委托别人。Client 参与者可以使用的 ActiveObject 参与者提供的方法。调用这些方法后（在 ActivationQueue 参与者还没积满的情况下），控制权马上会回来。

Client 参与者只知道 ActiveObject 参与者的存在，不过实际调用到的是 Proxy 参与者。

Client 参与者要获取执行结果时，必须调用 VirtualResult 参与者的 getResultValue 方法。这里使用了 Future Pattern（第 9 章）。

范例程序里，MakerClientThread 类与 DisplayClientThread 类都是 Client 参与者。

◆ *ActiveObject（主动对象）参与者*

ActiveObject 参与者规定了“主动对象”要对 Client 参与者公开的接口（API）。

范例程序中，ActiveObject 参与者是 ActiveObject 接口[注3]。

◆ *Proxy（代理者）参与者*

Proxy 参与者的工作是将方法调用转换为 MethodRequest 参与者的对象形式。转换后的 MethodRequest 参与者就传给 Scheduler 参与者。

Proxy 参与者会实现 ActvieObject 参与者所提供的接口（API）。

会调用 Proxy 参与者的是 Client 参与者。转换成 MethodRequest 参与者、传给 Scheduler 参与者这部分的工作，是由 Client 参与者的线程进行的。

范例程序中，Proxy 参与者是 Proxy 接口。

◆ *Scheduler 参与者*

Scheduler 参与者做两件事。一是将从 Proxy 参与者处传入的 MethodRequest 参与者传给 ActivationQueue 参与者，二是从 ActivationQueue 参与者取出 MethodRequest 参与者来执行。

将 MethodRequest 参与者传给 ActivationQueue 参与者的工作，是由 Client 参与者的线程进行的。

而从 ActivationQueue 参与者取出 MethodRequest 参与者来执行的工作，则是自己的线程在进行的。在 Active Object Pattern 里，只有 Client 参与者与 Scheduler 参与者是线程。

Scheduler 参与者会从 ActivationQueue 参与者里取出任务（MethodRequest 参与者）来执行。所以 Scheduler 参与者可以判断下一个要进行的什么任务。如果要自己实现请求调度算法，就要写在这里。这就是名为 Scheduler 参与者的原因。

范例程序中，Scheduler 参与者是 SchedulerThread 类。SchedulerThread 类里，没有定义特殊的调度方式，只使用了一般的 FIFO。

◆ *MethodRequest 参与者*

MethodRequest 参与者是 Client 参与者的请求所对应的对象形式。规定了用来处理这个请求的 Servant 参与者、用来写入返回值的 Future 参与者，与用来“执行”的方法（execute）[注4]。

注 3：本书中为了明确定义出界面（API），所以加入了 ActiveObject 参与者，附录 E 的[POSA2]没有 ActiveObject 参与者。[POSA2]中，Client 参与者是直接调用 Proxy 参与者。

注 4：附录 E 的[POSA2]书中，本书的 execute 方法使用了 call 方法的名称。

MethodRequest 参与者是“主动对象”的接口（API）对象化后的形式。

范例程序中，MethodRequest 参与者是 MethodReqeust 类。

◆ *ConcreteMethodRequest 参与者*

ConcreteMethodRequest 参与者是实际对应到具体的方法的 MethodRequest 参与者。ActiveObject 参与者里所定出的方法，要逐一声明成 MethodAlphaRequest、MethodBetaRequest 等类。在图 12-4 中，为了使 methodAlpha、methodBeta 等方法所对应的对象一目了然，所以不使用 ConcreteMethodRequest 的名称，而画成 MethodAlphaRequest、MethodBetaRequest。

ConcreteMethodRequest 参与者里定义的字段，就是方法的参数。

范例程序中，MakeStringRequest 类与 DisplayStringRequest 类是 ConcreteMethodRequest 参与者。MakeStringRequest 类对应到 makeString 方法（建立字符串），而 DisplayStringRequest 类对应到 displayString 方法（显示字符串）。

◆ *Servant（仆人）参与者*

Servant 参与者实际负责处理请求。

调用 Servant 参与者的是 Scheduler 参与者。Scheduler 参与者会从 ActivationQueue 参与者里取出 MethodRequest 参与者（实际上是 ConcreteMethodRequest 参与者），并调用 MethodRequest 参与者的 execute 方法。在 execute 方法里调用了 Servant 参与者的方法。

Servant 参与者实现了 ActiveObject 参与者提供的接口（API）。

请求会先由 Proxy 参与者转换成 MethodRequest 参与者，而由 Servant 参与者来实际执行。在 Proxy 参与者与 Servant 参与者之间加上 Scheduler 参与者，管理执行请求的顺序。

范例程序中，Servant 参与者是 Servant 类。

◆ *ActivationQueue（活性化队列）参与者*

ActivationQueue 参与者是用来存放 MethodRequest 参与者的类。

调用 putRequest 的是 Client 参与者的线程，而调用 takeRequest 的则是 Scheduler 参与者的线程。这里使用了 Producer-Consumer Pattern（第 5 章）。

范例程序中，ActivationQueue 参与者是 ActivationQueue 类。

◆ *VirtualResult（虚拟的结果）参与者*

VirtualResult 参与者与 Future 参与者、RealResult 参与者合在一起构成了 Future Pattern（第 9 章）。

Client 参与者要获取处理结果时，要调用 VirtualResult 参与者（实际上是 Future

参与者）的 getResultValue 方法。

范例程序中，VirtualResult 参与者是 Result 类。

◆ *Future（期货）参与者*

Future 参与者是在想要获取处理结果时，实际被调用的参与者。如果处理的结果还没有运算出来，会使用 Guarded Suspension Pattern（第 3 章），要求 Client 参与者的线程等待。

范例程序中，Future 参与者是 FutureResult 类。

◆ *RealResult（实际的结果）参与者*

RealResult 参与者用来表示实际的处理结果。Servant 参与者会建立出 RealResult 参与者，作为实际的处理结果，并设置给 Future 参与者。

范例程序中，RealResult 参与者是 RealResult 类。

Active Object Pattern 的顺序图如图 12-5 和时序图如图 12-6 所示。

扩展思考方向的提示

到底做了什么事呢

这一章读到这里，相信一定有读者觉得“一次用到很多 Pattern 虽然很有趣，但到底是在做什么啊？”。

Active Object Pattern 的类图（图 12-4）真的十分复杂。可是，我们就在此后退一步，来重新看过一次整体轮廓吧。左边是 Client 参与者，而右边是 activeobject 包里的所有参与者。Client 参与者会调用 ActiveObject 参与者的“方法”，并配合需要使用 Future Pattern 获取“返回值”。

与平常不同的是，在这里“方法”是由 Client 参与者以外的线程在执行的。也就是说，Active Object Pattern 作到了“异步的方法”。这也可以说是作到了“异步的消息”。

接下来我们要看的是 activeobject 包的属性。我们可看出，通过 activeobject 包里所有的参与者互相协调，组合一个“主动对象”。这个主动对象，具有下列的特征：

- 具有接口（API）——ActiveObject 参与者里定义了 API；
- 可以获取异步消息——Proxy 参与者会将方法调用转换成 MethodRequest 参与者，存放在 ActivationQueue 参与者里；
- 与 Client 参与者拥有不同的线程——Scheduler 参与者提供线程；
- 可以执行处理——Servant 参与者可以单线程执行处理；
- 可以返回返回值——以 Futrue 参与者作为返回值的提货单。

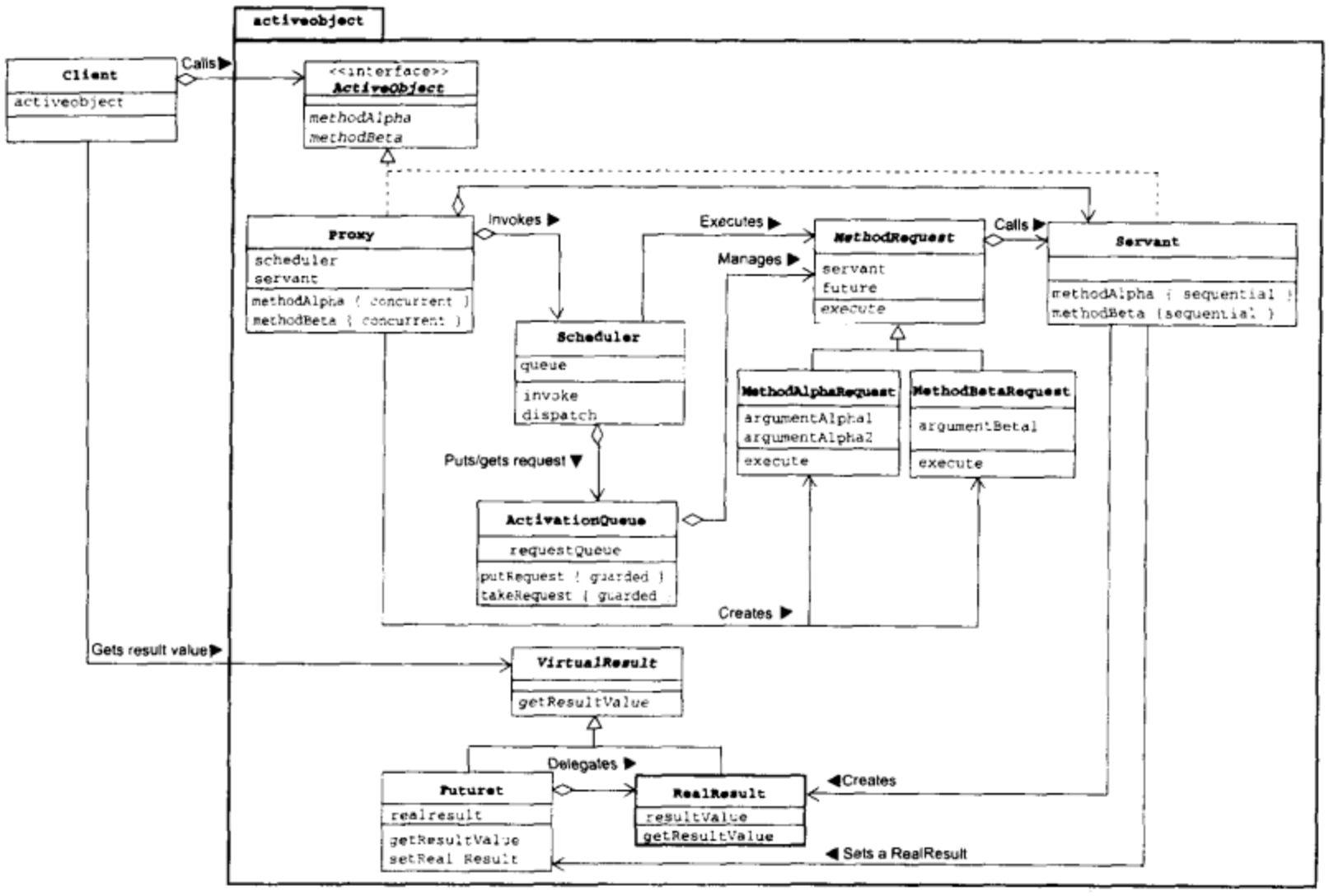


图 12-4　Active Object Pattern 的类图

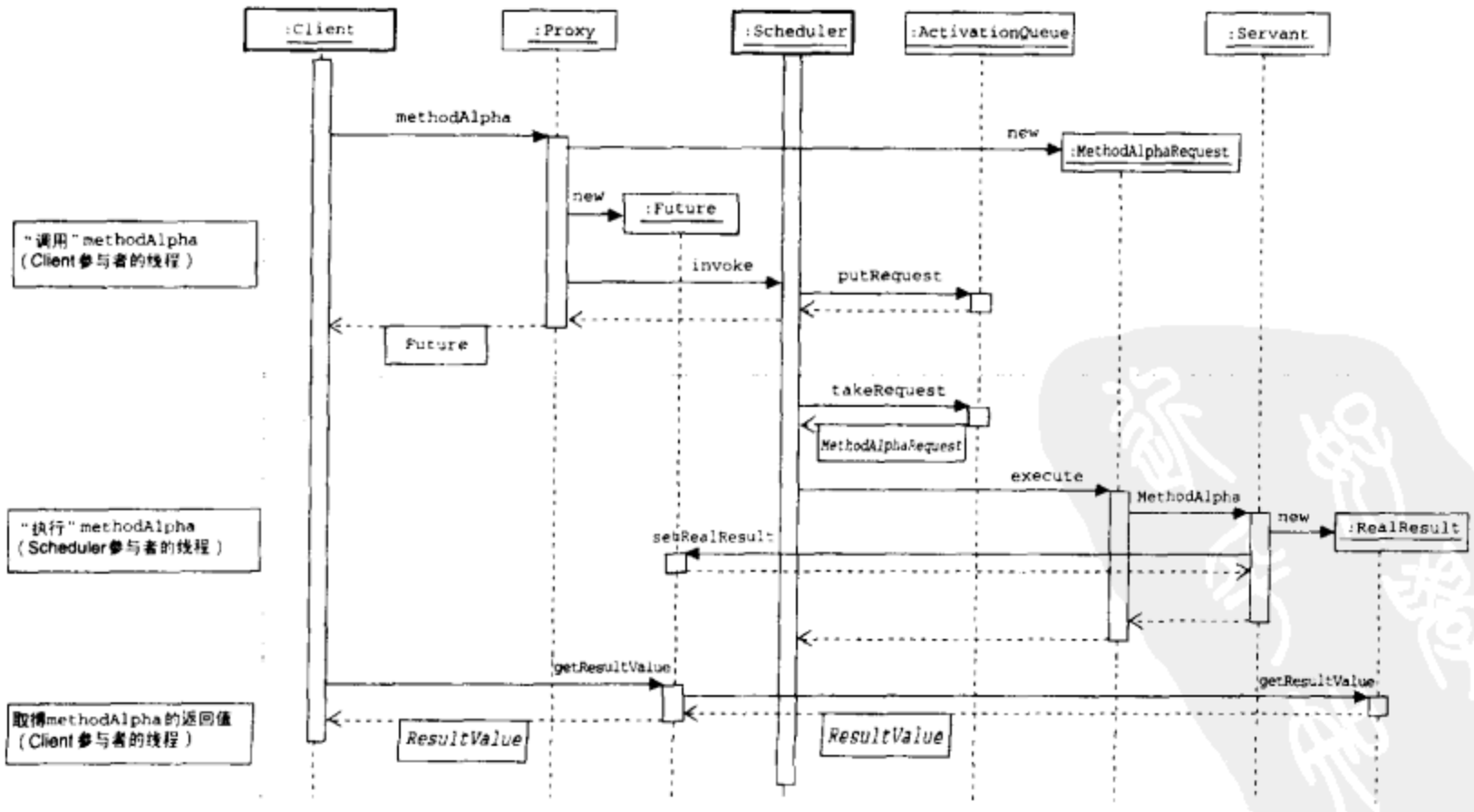


图 12-5　Active Object Pattern 的顺序图

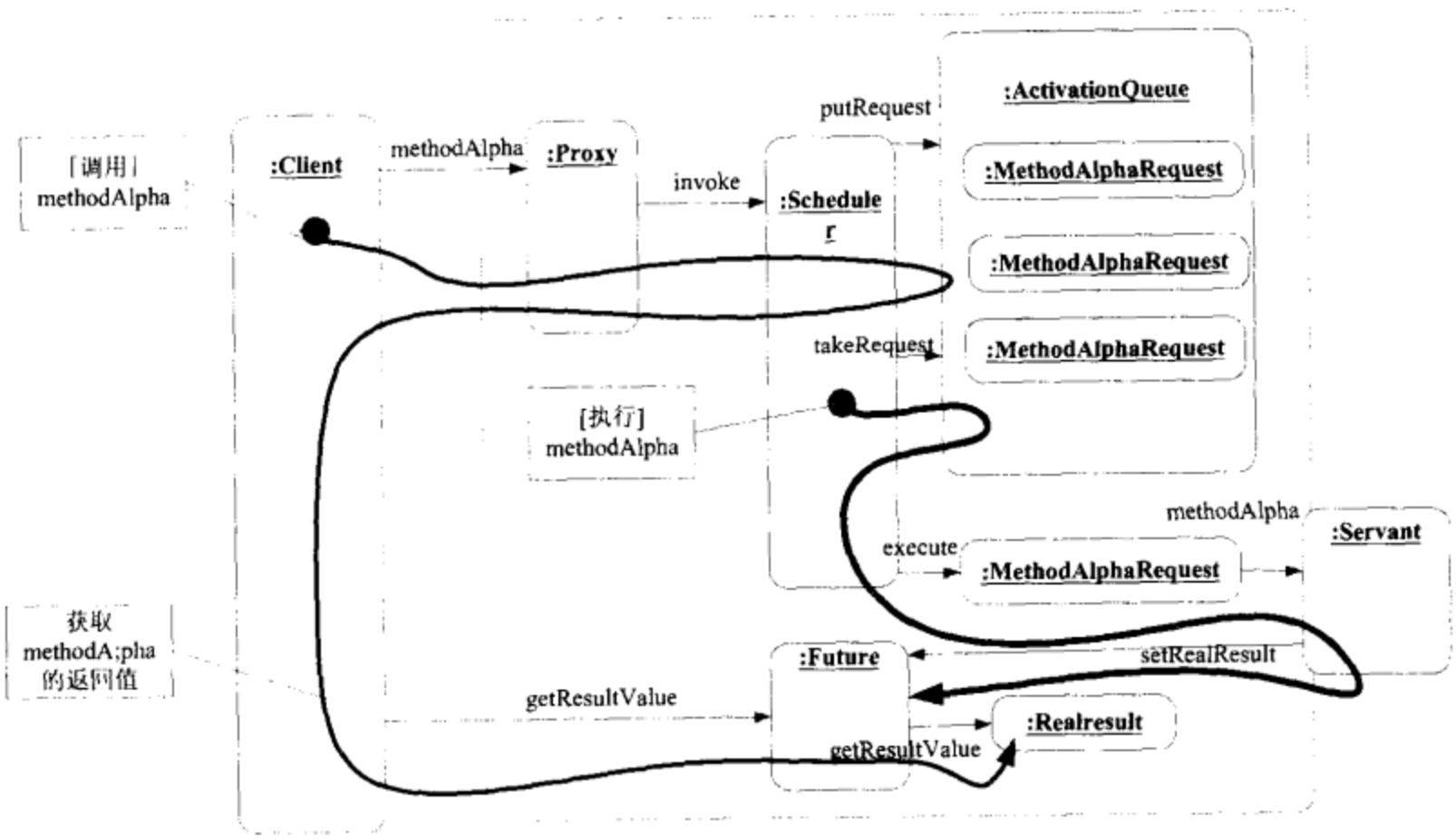


图 12-6　Active Object Pattern 的时绪图

请回想这一章最前面的比喻。许多个人互相协调，可以构成一个“法人”。Active Object Pattern 则是协调许多对象，构成一个“主动对象”。

Pattern 是否适合使用要考虑问题的规模

Active Object Pattern 的组成元素很多，是一个大型的 Pattern。考虑是否要使用这个 Pattern 的时候，必须注意问题的规模。所谓问题的规模，是指问题的大小，为了解决问题所需的处理量，到底有多大。范例程序里用来举例的“建立字符串”、“显示字符串”，问题的规模都太小，其实都不适合使用 Active Object Pattern（当然书中解释时，问题规模缩小有助于促进理解）。

较大型的问题规模，例如从大量网页查找特定字符串的查找服务、给予密码与明文而自动加以编码化的模块、进行大量数据收发的网关（这是附录 E [POSA2]所写的范例）等。

之所以规模太小不适合使用，是因为 Proxy 参与者建立 ConcreteMethodRequest 参与者与 ActivationQueue 参与者进行沟通，这些工作的需要花的时间也不容忽视。当规模太小时，与其使用 ActivationQueue 参与者将处理线程单一化，还不如使用 Guarded Suspension Pattern（第 3 章），程序性能还会比较高。

注意并发性

请观察图 12-4，注意方法所附加的并发性限制。

Proxy 参与者中，下列的方法加有{ concurrent }限制，这是“并发执行”的意思。

methodAlpha { concurrent }

methodBeta { concurrent }

而 Servant 参与者中，下列的方法则加有{ sequential }限制，这是“逐一执行”的意思。

```
methodAlpha { sequential }
methodBeta { sequential }
```

简单的说，就是：

- Proxy 参与者可由多个线程调用（{ concurrent }）
- Servant 参与者只能由单一线程调用（{ sequential }）

也就是说，比起调用 Proxy 参与者的方法，其实调用 Servant 参与者的方法时更需要小心谨慎。不过没关系，因为只有一条 Scheduler 参与者的线程会调用 Servant 参与者的方法。Proxy 参与者可能会被多个线程（Client 参与者）不断调用。但是多线程相关的共享互斥，都是交给 Producer-Consumer Pattern（第 5 章）来管理。Client 参与者对应到 Producer-Consumer Pattern 的 Producer 参与者，ActivationQueue 参与者对应到 Channel 参与者，而 Scheduler 参与者则对应到 Consumer 参与者。

即使 Producer 参与者有很多条，只要 Consumer 参与者只有一条，那 Consumer 参与者所调用的对象就不是多线程环境。所以 Servant 只会有单线程执行，是安全的。

我们可以把 Active Object Pattern 看作是在以单线程为前提的 Servant 参与者外面包上一层外皮，使多线程的 Client 参与者可以使用它。

把处理降到只有单线程在执行的优点，我们已经在 Worker Thread Pattern（第 8 章）的进阶说明“Swing 的 Event-Dispatching Thread”里说明过了。

新增方法

范例程序中，ActiveObject 接口有 makeString 与 displayString 两个方法。这是“主动对象”的接口（API）。

若要在范例程序中加上新的方法时，修改的顺序如下：

（1）在 ActiveObject 接口里定义新的方法；

（2）继承 MethodRequest 类，建立新的类来对应这个方法；

（3）在 Proxy 类里加上新的方法；

（4）在 Servant 类里加上新的方法。

如果方法需要返回值，还要使用 Future Pattern（第 9 章）。建立 Future 参与者是 Proxy 类的工作，建立 RealResult 参与者则是 Servant 类的任务。

SchedulerThread 类不需要修改。SchedulerThread 类并不知道“主动对象”有哪些方法，只负责调用 MethodRequest 参与者的 execute 方法而已。所以要加上新的方法时，并不需要修改 SchedulerThread 类。

本章末的练习问题 12-2，就要练习加上新的方法。

Scheduler 参与者的角色

本章对 Active Object Pattern 的描述中，参考了附录 E 的[POSA2]一书对 Active Object Pattern 的描述。在[POSA2]对 Active Object Pattern 的描述中，Scheduler 参与者担任很多调度相关的工作。

[POSA2]一书的写法如下：首先，在每个 ConcreteMethodRequest 参与者定义了 guard 方法。若 ConcreteMethodRequest 参与者可以执行，则 guard 方法的返回值为 true。Scheduler 参与者只有在 guard 方法的返回值为 true 时，才会调用 ConcreteMethodRequest 参与者的 execute 方法[注 5]。这样一来，就可以在 ConcreteMethodRequest 参与者上对每个方法加上防卫条件。

详细内容请参考附录 E 列出的[POSA2]一书。

“主动对象”之间的对话

本书中以“Client 参与者使用主动对象”的形式描述这个 Pattern。不过，其实定义多个“主动对象”，让它们互相对话，其实也是可以的。也就是说，可以由 Servant 参与者调用其他 ActiveObject 参与者的“方法”。构思一个有多个“主动对象”，互相以异步消息对话的系统，似乎满有趣的喔。

迈向分布处理——将线程的界限移到机器的界限

相信阅读本书到这里的读者，在看到一个方法时，应该会去意识到“这个方法是在哪个线程上运行的”。

Active Object Pattern 中，“方法的调用（invocation）”部分是在 Client 参与者的线程上运行；而“方法的执行（execution）”部分则是在 Scheduler 参与者的线程上运行的。在这里作出了第 5 章所提过的“invocation 与 execution 的分离”，将进行 invocation 的线程（Client 参与者）与进行 execution 的线程（Scheduler 参与者）拆开。

第 5 章也曾经提过，只要将线程拆开，就可能将线程所运行的机器也拆开。也就

注 5：附录 E 的[POSA2]书中，本书的 execute 方法使用了 call 方法的名称。

是分离进行 invocation 与进行 execution 的机器，在中间使用网络连接起来。要说网络传递的东西是什么……，没错，就是 MethodRequest 参与者与 Result 参与者。方法的调用与返回值的设置因为都具有“对象”的形式了，所以可以通过网络来往了。

我们可以说是“将线程的界限移到机器的界限”。

Java 里，提供了 Remote Method Invocation（RMI）这个 Active Object Pattern 的相关技术。RMI 是在自己的计算机调用方法，而可以由网络对面的另一台计算机来实际进行处理的技术。为了让对象在网络上传送，Java 又使用了序列化（serialization）[注 6] 技术。详请见下面的 URL 与 Java 的参考文件。

- Java Remote Method Invocation（RMI）

  `http://java.sun.com/products/jdk/rmi/`

相关 Pattern

Producer-Consumer Pattern（第 5 章）

Producer-Consumer Pattern 用来让 Client 参与者可以安全地将 ConcreteMethodRequest 参与者传到 Scheduler 参与者的手上。Producer-Consumer Pattern 的 Channel 参与者，对应到 Active Object Pattern 的 ActivationQueue 参与者。

Future Pattern（第 9 章）

为了将返回值传给 Client 参与者，使用了 Future Pattern 参与者。

Worker Thread Pattern（第 8 章）

实现 Scheduler 参与者的部分，使用到了 Worker Thread Pattern。

Thread-Specific Storage Pattern（第 11 章）

Thread-Specific Storage Pattern 可让线程持有自己的信息。Active Object Pattern 中，“启动”方法的线程与“执行”方法的线程并不相同，所以套用 Thread-Specific Storage Pattern 并不简单。

重点回顾

这一章我们学习了 Active Object Pattern。

注 6：一般又称为 marshalling。

这里有委托处理的人（Client 参与者）与执行处理的人（Servant 参与者）。因为 Servant 参与者进行处理需要花一些时间，我们宁可延后处理进行的时刻，也不希望对 Client 参与者的执行造成影响。

另外，并不是 Client 参与者单方面对 Servant 参与者调用，Servant 参与者也可能要将执行结果返回给 Client 参与者，也就是可以双向的调用。而且，我们希望处理的委托顺序与处理的执行顺序可以独立。

这个时候，就要使用 Active Object Pattern。

在这里以“调用 Proxy 参与者的方法”表现 Client 参与者送出的请求，而 Proxy 参与者会将请求转换成 1 个 ConcreteMethodRequest 参与者对象，通过 Scheduler 参与者，存放在 ActivationQueue 参与者里。Client 参与者的线程所作的事就到这里为止。实际进行处理的部分，不在 Client 参与者的线程上进行。

从 ActivationQueue 参与者里选出要执行的 ConcreteMethodRequest 参与者，是 Scheduler 参与者的工作。Scheduler 参与者在与 Client 参与者不同的线程上执行，并通过 ConcreteMethodRequest 参与者，将工作委托给 Servant 参与者。

为了做到双向的调用，使用了 Future Pattern（第 9 章）。

这就是 Active Object Pattern。

Active Object Pattern 里，混合使用了本书前面学到的各种 Pattern。Active Object Pattern 组成了具有下列特征的“主动对象”。

- 异步接受外部传来的请求
- 可自由的调度
- 实际的处理由单线程运行
- 可返回执行结果
- 拥有独立的线程

那么，就进入练习问题吧。

练习问题

解答请见附录

问题 12-1（基础知识的确认）

Java 请就您对范例程序的认知，阅读下列语句，正确的请回答○，不正确的请回答×。

（1）Proxy 类与 Servant 类安装了 ActiveObject 接口。

（2）MakerClientThread 类的线程所执行的 makeString 方法，被以 Servant 类安装。

（3）每次调用 displayString 方法时，都会制作出新的线程。

（4）由于数个线程都可以连接上 Servant 类，因此需要进行共享互斥。

（5）ActivationQueue 类的 putRequest 方法会被数个线程调用。
（6）ActivationQueue 类的 takeRequest 方法会被数个线程调用。
（7）MakerClientThread 类的线程在调用出 getResultValue 方法时，如果还没制作出字符串的话，线程会 wait。
（8）Servant 类的 makeString 方法随着所制作的字符串越长而处理时间就越久。
（9）MakerClientThread 的线程调用出 makeString 时，参数值 count 越大则从方法回来的时间越久。

问题 12-2（追加方法）

Java 请在范例程序的“主动对象”中，加上下列 add 方法，用来进行任意精度的整数加法运算。

public Result add(String x, String y)

在这里，参数 x 与参数 y 都是"1234"般的十进制字符串。返回值是 Result 对象。对返回值调用 getResultValue 方法，可以获取表示 x 与 y 两个整数相加后结果的十进制 String 实例。如果发生任何错误，getResultValue 方法就返回 null。执行范例如图 12-7 所示。

提示：任意精度的整数加法运算可使用 java.math.BigInteger 类。

List 12-16 测试用的 Main 类（Main.java）

```
import activeobject.ActiveObject;
import activeobject.ActiveObjectFactory;

public class Main {
    public static void main(String[] args) {
     ActiveObject activeObject = ActiveObjectFactory.
createActiveObject();
        new AddClientThread("Diana", activeObject).start();
    }
}
```

List 12-17 AddClientThread 类（AddClientThread.java）

```
import activeobject.ActiveObject;
import activeobject.Result;

public class AddClientThread extends Thread {
    private final ActiveObject activeObject;
    private String x = "1";
    private String y = "1";
    public AddClientThread(String name, ActiveObject activeObject) {
```

```
            super(name);
            this.activeObject = activeObject;
        }
        public void run() {
            try {
                for (int i = 0; true; i++) {
                    // 有返回值的调用
                    Result result = activeObject.add(x, y);
                    Thread.sleep(100);
                    String z = (String)result.getResultValue();
                    System.out.println(Thread.currentThread().
    getName() + ": " + x + " + " + y + " = " + z);
                    x = y;

                    y = z;
                }


            } catch (InterruptedException e) {
            }
        }
    }
```

```
Diana: 1 + 1 = 2
Diana: 1 + 2 = 3
Diana: 2 + 3 = 5
Diana: 3 + 5 = 8
Diana: 5 + 8 = 13
Diana: 8 + 13 = 21
Diana: 13 + 21 = 34
Diana: 21 + 34 = 55
Diana: 34 + 55 = 89
Diana: 55 + 89 = 144
Diana: 89 + 144 = 233
Diana: 144 + 233 = 377
(中略)
Diana: 2880067194370816120 + 4660046610375530309 = 7540113804746346429
Diana: 4660046610375530309 + 7540113804746346429 = 12200160415121876738
Diana: 7540113804746346429 + 12200160415121876738 = 19740274219868223167
Diana: 12200160415121876738 + 19740274219868223167 = 31940434634990099905
Diana: 19740274219868223167 + 31940434634990099905 = 51680708854858323072
(以下略。CTRL+C 结束程序)
```

图 12-7　执行范例

问题 12-3（GUI 上的应用）

Java 请使用 Active Object Pattern 建立一支程序，可从大量网页中查找特定字符串，列出包含该字符串的网址。执行的画面如图 12-8～图 12-10 所示。

注意：解答中并没有真正在网络上查找，只是搁置约 5 秒后开始随便输出 URL。

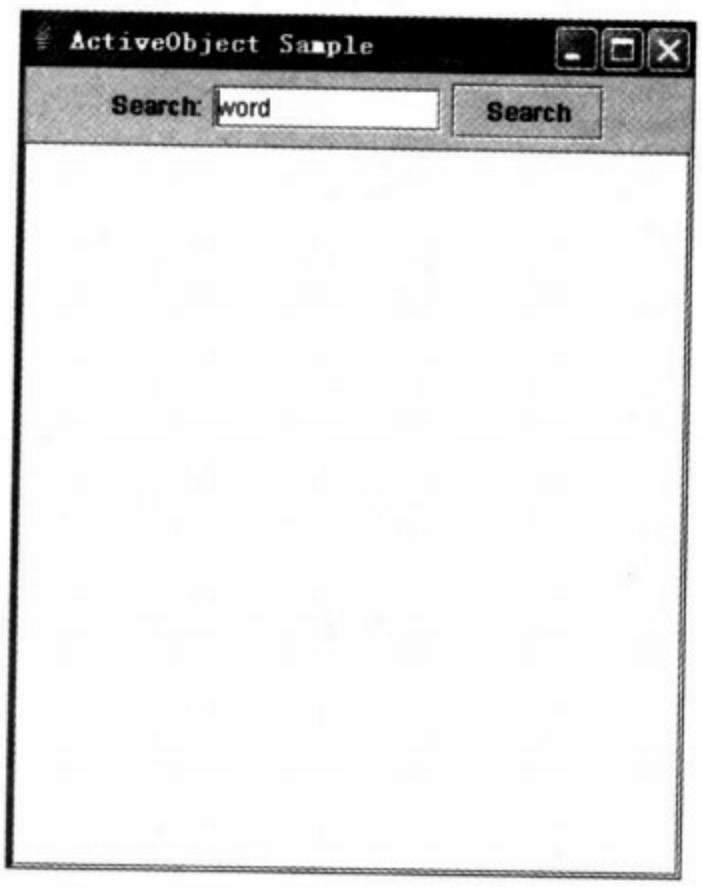


图 12-8　启动程序时

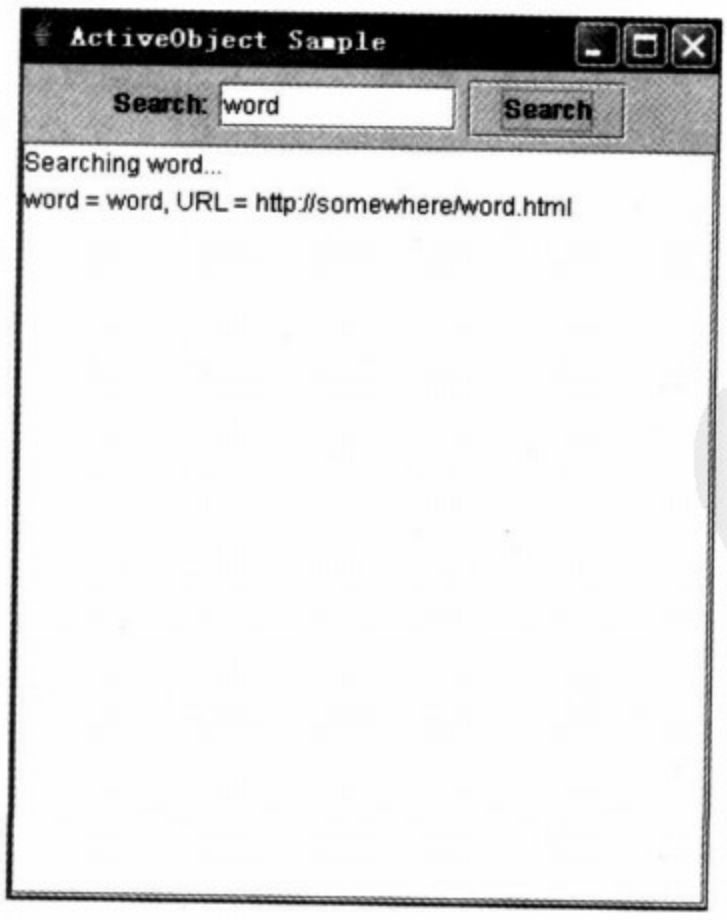


图 12-9　查找单字"word"后，等待约 5 秒会开始出现 URL

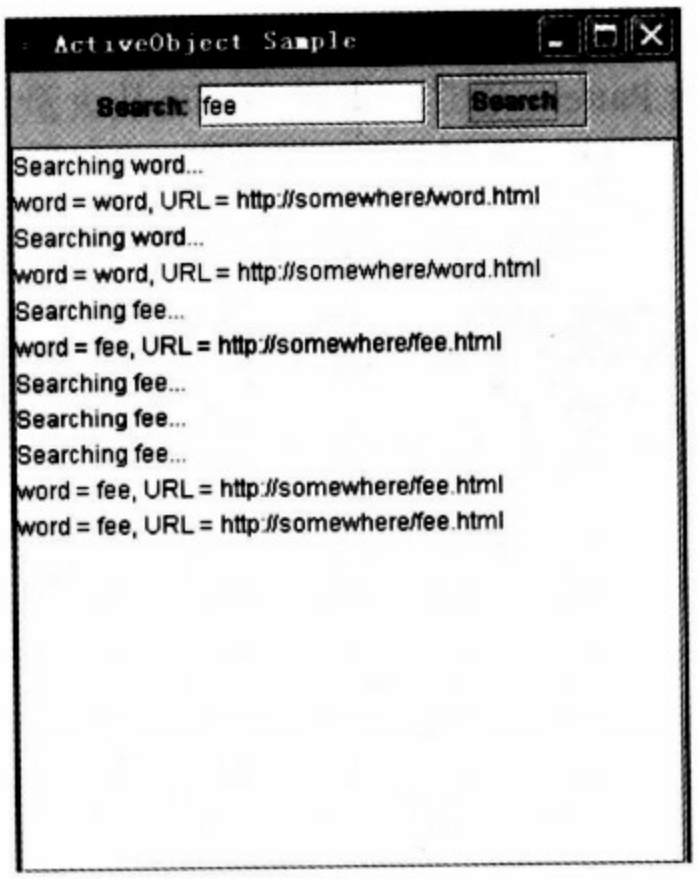


图 12-10　继续输入新的查找关键字，会分别展开查找

总　结

多线程程序设计的模式语言

多线程程序设计的模式语言

本书也进入了最后一章。我们在本书一开始的时候，曾经写过下面这样一个单线程的程序，但是如今看来彷佛已有阔别数年之感。

```
public class Main {
    public static void main(String[] args[]) {
        for (int i = 0; i < 10000; i++) {
            System.out.println("Good!");
        }
    }
}
```

在这一章中，将以模式语言的形式，回顾一下在本书所学到的 12 个 Pattern。

Pattern 与 Pattern Language

在开始回头看 12 个 Pattern 前，在这里想稍微提一下 Pattern 与模式语言（Pattern Language）。

Pattern（模式）是“针对某个在特定背景下，一直反复发生的问题的解决方案”。Pattern 都会有一个容易了解的名称。

背景（context）是指发生问题的状况与背景、环境。

问题中，还具有称为外力（force）的条件。这在 Introduction2“多线程程序的评价基准”中曾经提过，就是能够解决问题的条件，非跨过不可的栅栏。通常，多项外力还互为不可兼得的关系。在这一章中，外力一并描述在问题的段落里。

模式语言（Pattern Language），用一句话来说，就是 Pattern 的集合。不过不是单纯把 Pattern 集合在一起就好了。集合相关、互补的许多 Pattern，浅显地描述之间的关系，这就是模式语言。

Pattern 可说是对一个问题的一种解决方法，而模式语言则可说是对某个领域的问题群的解决方法的集合。不只是程序设计或软件开发，无论在什么领域，只要能妥善描述出原理、重点、诀窍、技巧，就是模式语言。

阅读模式语言，可以了解该范畴领域中会碰到的各种问题与各种解决方式，使自己再次面对问题的时候，可以选择恰当的 Pattern 加以套用。

下面就要使用本书出现过的 Pattern，来描述多线程的模式语言。

为了使 Pattern 之间的关系一目了然，在此先将本书出现的 Pattern 绘出关系图，如图 13-1 所示。

多线程程序设计的模式语言

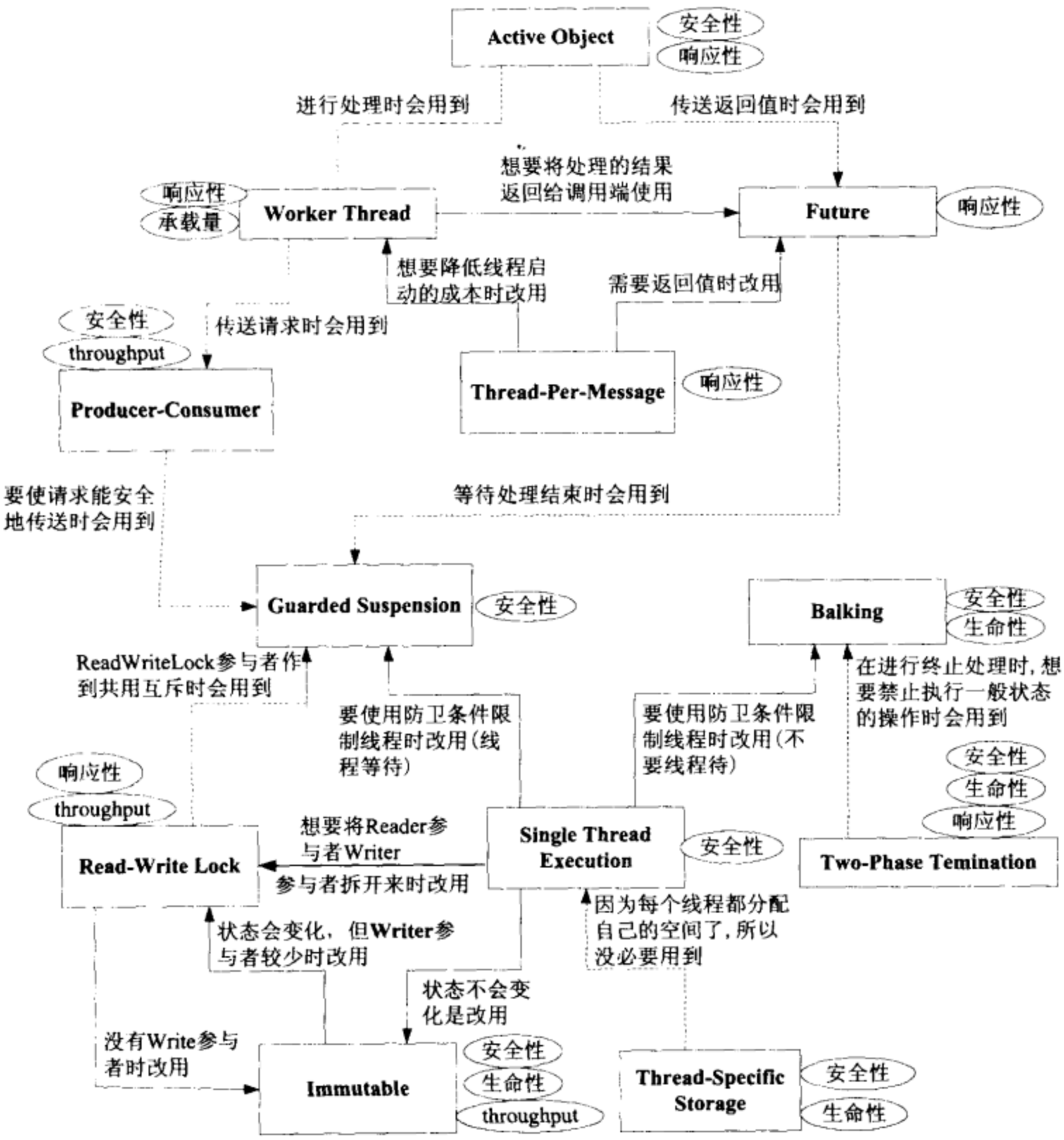


图 13-1　本书出现的 Pattern 关系图

Single Threaded Execution Pattern（第 1 章）

——能通过这座桥的，只有一个人

◆ 别名

- Critical Section
- Critical Region

◆ 背景

多个线程共享一个实例。

◆ 问题

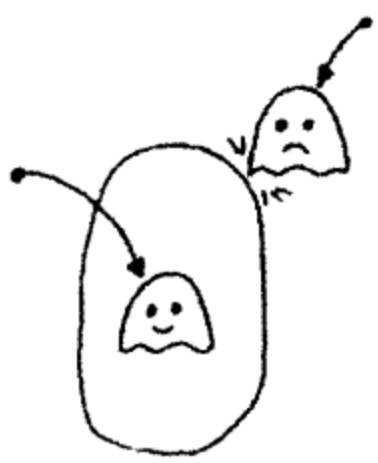
若多个线程都擅自更改实例的状态，实例会丧失安全性。

◆ 解决方式

首先，仔细找出实例状态不稳定的范围（临界区间）。并对临界区间加以防护，使同时执行的线程保持在只有一条的情况。这样一来，就能保护住实例的安全性。

这就是 Single Threaded Execution Pattern。

为了使读者能更直觉地掌握线程的操作，我们以漫画图解如下：



◆ 实现

Java 语言里，可使用 synchronized 来实现出临界区间。

◆ 相关

当实例的状态不会改变时，为了提升 throughput，可使用 Immutable Pattern（第 2 章）。

想要将引用实例状态的线程与改变实例状态的线程拆开，以提高 throughput 时，可使用 Read-Write Lock Pattern（第 6 章）。

Immutable Pattern（第 2 章）
——想破坏它也没办法

◆ 背景

多个线程共享一个实例，实例的状态不会改变。

◆ 问题

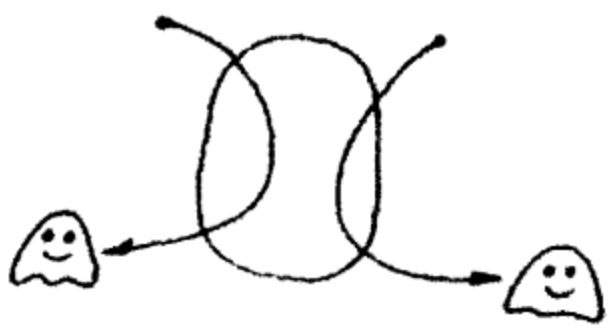
使用 Single Threaded Execution Pattern（第 1 章），会降低 throughput。

◆ 解决方式

当实例建立后状态就不会再变化时，就要停止使用 Single Threaded Execution Pattern（第 1 章）了。

为了避免失误造成更改了实例的状态，故将类写成无法由线程更改。另外，删除实例里所有用来更新状态的方法（setter）。引用实例状态用的方法（getter）就无妨。

这就是 Immutable Pattern。



使用 Immutable Pattern，可提高 thoughput。但是要持续保证不变性（immutablity）并不简单。记得在帮助文件中也注明这是一个 immutable 的类。

◆ 实现

在 Java 中使用 private 来隐藏字段，此外由于无法确保不可更改，因此还要使用 final。

◆ 关联

对多个线程进行共享互斥，可使用 Single Threaded Execution Pattern（第 1 章）。

当修改用的线程数量比用来读取的线程数量多时，可考虑使用 Read-Write Lock Pattern（第 6 章）。

Guarded Suspension Pattern（第 3 章）
——等到我准备好

◆ 别名

- Spin Lock
- Guarded Wait

◆ 背景

多个线程共享一个实例。

◆ 问题

若多个线程都擅自更改实例的状态，实例会丧失安全性。

◆ 解决方式

当实例的状态不恰当时，就要求线程等待到适合的状态时。首先，以“警戒条件”来表示实例的“适当的状态”。并且在进行有安全性疑虑的操作前，都要检查是否警戒条件满足。如果警戒条件不成立，就要求线程等待到成立为止。



这就是 Guarded Suspension Pattern。

使用 Guarded Suspension Pattern，能以警戒条件限制方法的执行。不过，如果警戒条件一直不成立，线程会永远等待下去，会使程序丧失生命性。

◆ 实现

Java 语言中，检验警戒条件时使用 while 语句，而要让线程等待时则使用 wait 方法。并使用 notify/notifyAll 通知警戒条件的改变。检验、修改警戒条件时，会使用到 Single Threaded Execution Pattern（第 1 章）。

◆ 相关

当警戒条件不成立时想要马上退出，就使用 Balking Pattern（第 4 章）。

Guarded Suspension Pattern 中检验、更改警戒条件的部分，会用到 Single Thread Execution（第 1 章）。

Balking Pattern（第 4 章）
——不需要的话，就算了吧

◆ 背景

多个线程共享一个实例。

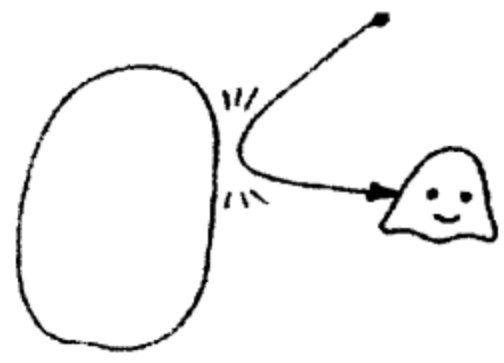
◆ 问题

若多个线程都擅自更改实例的状态，实例会丧失安全性。可以一直等待安全的时机，又会使程序响应性降低。

◆ 解决方式

当实例的状态不适合时，就中断掉处理的进行。首先，以“警戒条件”来表示实例的“适当的状态”，并且在进行有安全性疑虑的操作前，都要检查是否满足警戒条件。只有在警戒条件成立时，才会继续执行；如果警戒条件不成立，就直接中断（balk）执行，马上退出。

这就是 Balking Pattern。



◆ 实现

Java 语言中，检验警戒条件时要使用 if 语句。当要 balk 时，可使用 return 退出方法，或使用 throw 抛出异常。检验、修改警戒条件时，会使用到 Single Threaded Execution Pattern（第 1 章）。

◆ 关联

当想要等到警戒条件成立再执行时，可使用 Guarded Suspension Pattern（第 3 章）。

Balking Pattern 中检验、更改警戒条件的部分，会用到 Single Thread Execution（第 1 章）。

Producer-Consumer Pattern（第 5 章）——你来做，我来用

◆ 背景

当要从某个线程（Producer 参与者）将数据传给其他线程（Consumer 参与者）时。

◆ 问题

当 Producer 参与者与 Consumer 参与者处理的速度不同时，速度慢的会扯速度

快的后腿，而降低程序的 throughput。另外，当 Producer 参与者要写入数据时，Consumer 参与者若同时读取数据，数据会失去安全性。

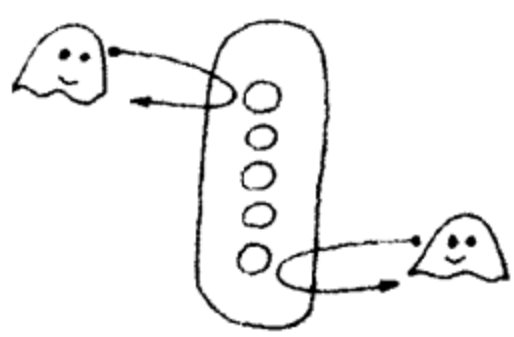
◆ 解决方式

在 Producer 参与者与 Consumer 参与者之间，加上中继用的 Channel 参与者。并让 Channel 参与者存放多条数据。这样一来，就可以缓冲 Producer 参与者与 Consumer 参与者之间处理速度的差异。另外，只要在 Channel 参与者里进行共享互斥，数据就不会丧失安全性。于是 throughput 可以不降低，又可在多个线程之间安全地传送数据。

这就是 Producer-Consumer Pattern。

◆ 相关

Channel 参与者安全传递数据的部分，使用了 Guarded Suspension Pattern（第 3 章）。



Future Pattern（第 9 章）在传递返回值的时候，使用了 Producer-Consumer Pattern。

Worker Pattern（第 8 章）在传递请求的时候，使用了 Producer-Consumer Pattern。

Read-Write Lock Pattern（第 6 章）
——大家想看就看吧，不过看的时候不能写喔[注1]

◆ 别名

- Reader Writer

注 1：更正确的说法是：

- 大家想看就看吧，不过看的时候不能写喔
- 可以写的只有一个人，写的时候就不能看啰

不过实在太长而无法当作标语了。

- Reader/Writer Lock
- Readers/Writer Lock

◆ 背景

多条线程共享一个实例，并有会参考实例状态的线程（Reader 参与者），与会改变实例状态的线程（Writer 参与者）。

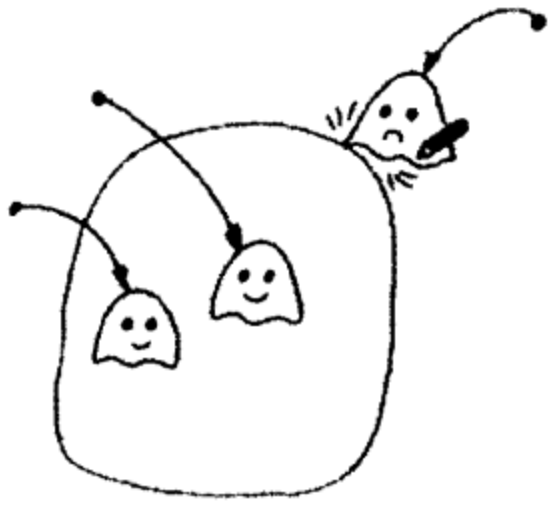
◆ 问题

若线程之间不进行共享互斥，会丧失安全性。但使用 Single Threaded Execution Pattern（第 1 章）会使程序 throughput 降低。

◆ 解决方式

首先，将“控制 Reader 参与者的锁定”与“控制 Writer 参与者的锁定”分开，加入 ReadWriteLock 参与者，以提供两种不同的锁定。ReadWriteLock 参与者会对“Writer 参与者－Writer 参与者”、“Reader 参与者－Writer 参与者”进行互斥控制。因为“Reader 参与者－Reader 参与者”不会发生冲突，故不会影响安全性，于是不进行共享互斥。这样可以在不影响安全性的前提下提高 throughput。

这就是 Read-Write Lock Pattern。



◆ 实现

Java 语言可以使用 finally 块避免忘记解除锁定。

◆ 相关

Read-Write Lock Pattern 中，ReadWriterLock 参与者进行共享互斥的地方，用到了 Guarded Suspension Pattern（第 3 章）。

完全没有 Writer 参与者的时候，可使用 Immutable Pattern（第 2 章）。

Thread-Per-Message Pattern（第7章）——这个工作交给你了

◆ 背景

线程（Client参与者）要调用实例（Host参与者）的方法。

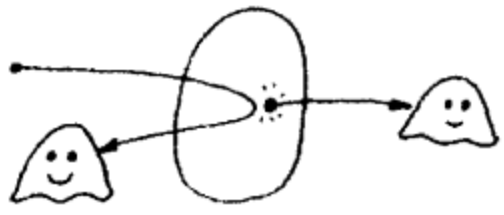
◆ 问题

在方法的属性处理完之前，控制权不会从Host参与者退出。如果方法的处理属性很花费时间，程序的响应性会降低。

◆ 解决方式

在Host参与者里，启动新的线程。并且将方法应该进行的工作，交给这个新的线程。这样Client参与者的线程就可以继续执行下一个操作了。这样做，不用更改Client参与者的程序代码，并能提高程序的响应性。

这就是Thread-Per-Message Pattern。



◆ 实现

Java语言中，为了简化启动线程的程序，可使用匿名内部类。

◆ 相关

想节省启动线程所花费的时间时，可以使用Worker Thread Pattern（第8章）。
想要将处理的结果返回给Client参与者时，可以使用Future Pattern（第9章）。

Worker Thread Pattern（第8章）——等到工作来，来了就工作

◆ 别名

- Thread Pool

- Background Thread

◆ 背景

线程（Client 参与者）要调用实例（Host 参与者）的方法。

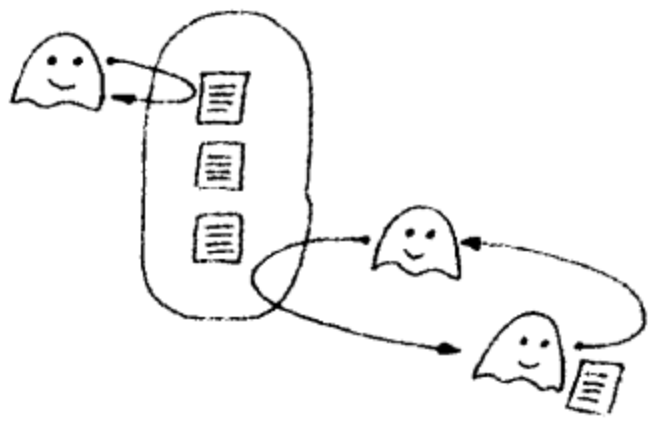
◆ 问题

如果方法的处理属性很花时间，程序的响应性会降低。为了提高响应性，而启动新的线程来处理方法时，启动线程所花的时间又会降低 throughput。另外，当送出的请求太多时，会启动过多的线程，这会使承载量变差。

◆ 解决方式

首先，我们事先启动一些用来进行处理的线程（工人线程）。并将代表请求的实例传给工人线程。这样就不需要每次都重新启动新的线程了。

这就是 Worker Thread Pattern。



◆ 相关

想要获取工人线程的处理结果时，可以使用 Future Pattern（第 9 章）。

想要将代表请求的实例传递给工人线程时，可使用 Producer-Consumer Pattern（第 5 章）。

Future Pattern（第 9 章）
——先给您这张提货单

◆ 背景

线程（Client 参与者）会将工作委托给其他线程，而 Client 参与者希望得到处理的结果。

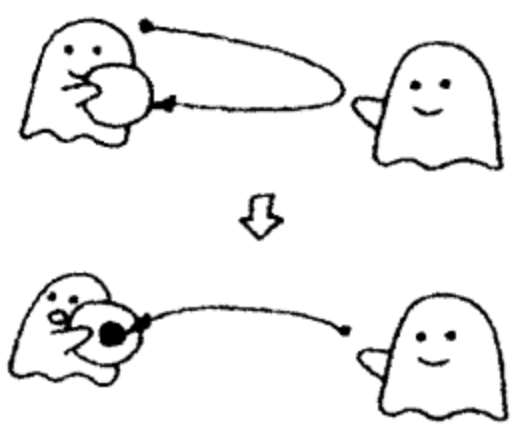
◆ 问题

将工作委托给别人时，如果又等待执行结果，会使响应性降低。

◆ 解决方式

首先，建立一个与处理结果具有相同接口（API）的 Future 参与者。在处理开始时，先把Future参与者当作返回值返回。处理的结果事后再设置给Future参与者。这样Client参与者就可以在适当的时机，通过 Future 参与者，获取（等待）处理的结果。

这就是 Future Pattern。



◆ 相关

等待 Client 参与者的处理结果时，会使用 Guarded Suspension Pattern（第 3 章）。

Future Pattern 可用在 Thread-Per-Message Pattern（第 7 章）想要获取处理结果时。

Future Pattern 可用在 Worker Thread Pattern（第 8 章）想要获取处理结果时。

Two-Phase Termination Pattern（第 10 章）——快把玩具收拾好，去睡觉吧

◆ 背景

想要结束运行中的线程。

◆ 问题

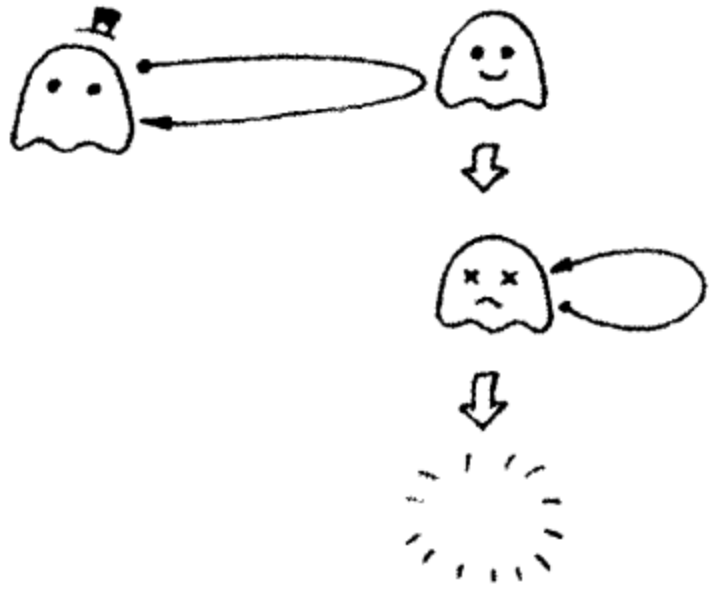
从外部忽然结束掉线程，会丧失安全性。

◆ 解决方式

首先，适合进行终止的时机，还是要交给线程自己判断。所以，定义一个送出

“终止请求”的方法用来结束线程。这个方法事实上只会将标识设置成“收到终止请求”而已，线程要在每个可以开始终止处理的地方自己检查这个标识。如果检查的结果为真，就开始进行终止处理。

这就是 Two-Phase Termination Pattern。



◆ 实现

Java 语言中，不但要设置表示收到终止请求的标识，还要使用 interrupt 方法中断掉 wait、sleep、join 的等待状态。因为线程到 wait、sleep、join 抛出 InterruptedException 以后，就不是中断状态了，所以若是使用 isInterrupted 方法来检查终止请求，必须特别小心。

为了在执行时发生异常也能确实进行终止处理，所以要使用 finally 块。

◆ 相关

进行终止处理中时，为了禁止其他操作，可使用 Balking Pattern（第 4 章）。

为了确实进行终止处理，使用了 Before/After Pattern。

Thread-Specific Storage Pattern（第 11 章）——每个线程的保管箱

◆ 别名

- Per-Thread Attribute
- Thread-Specific Data
- Thread-Specific Field
- Thread-Local Storage

◆ 背景

想要将假定在单线程环境下运行的对象（TSObject 参与者），在多线程的环境下使用。

◆ 问题

想要使用 TSObject 参与者并不简单。要将 TSObject 参与者改写成支持多线程，可能一不小心就丢掉安全性和生命性了。而且，TSObject 参与者可能根本不能改写。而我们也不想改写使用 TSObject 参与者的对象（Client 参与者）的程序代码，所以也不想修改 TSObject 参与者的接口（API）。

◆ 解决方法

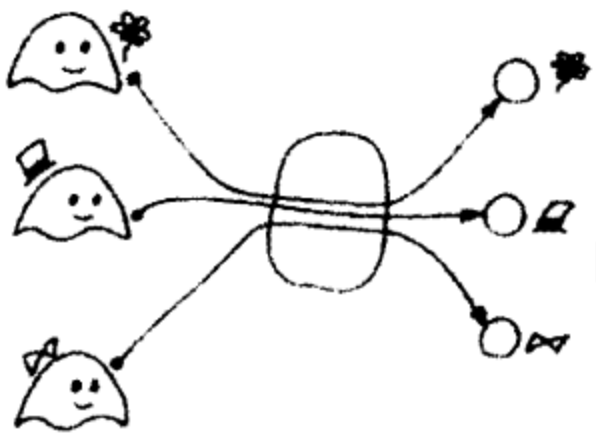
建立线程独有的空间，并管理这些空间与线程之间的对照关系。

首先，建立一个与 TSObject 参与者具有相同接口（API）的 TSObjectProxy 参与者。并建立 TSObjectCollection 参与者，管理“Client 参与者→TSObject 参与者”的对照关系。

TSObjectProxy 参与者会通过 TSObjectCollection 参与者，获取当前线程所对应的 TSObject 参与者，并将工作委托给 TSObject 参与者。Client 参与者会拿 TSObjectProxy 参与者来代替 TSObject 参与者使用。

这样一来，每个 TSObject 参与者一定只会有特定的一个线程调用他，所以 TSObject 参与者不需要进行共享互斥。关于多线程的部分，都隐藏在 TSObjectCollection 参与者里了。另外，TSObject 参与者的接口（API）也不必修改。

这就是 Thread-Specific Storage Pattern。



不过，使用 Thread-Specific Storage Pattern，等于是在程序里加上隐性的 context，有程序的可读性可能变差的危险性。

◆ 实现

Java 语言中，使用 java.lang.ThreadLocal 类担任 TSObjectCollection 参与者。

◈ 相关

要对多数的线程进行共享互斥时，要使用 Single Threaded Execution Pattern（第 1 章）。

Active Object Pattern（第 12 章）——接受异步消息的主动对象

◈ 别名

- Actor
- Concurrent Object

◈ 背景

这里有送出请求的线程（Client 参与者）与实际用来进行处理的对象（Servant 参与者）。可是 Servant 参与者是假定在单线程环境下运行开发出来的。

◈ 问题

想要以多个 Client 参与者使用 Servant 参与者，而 Servant 参与者又不是线程安全。如果 Servant 参与者处理的操作很花费时间，还会拖垮 Client 参与者的响应性。

处理的请求顺序与执行顺序并不一定一致。

处理的结果会返回 Client 参与者。

◈ 解决方式

我们在此建立一个可接受异步消息的主动对象，让他拥有独立于 Client 参与者的线程。

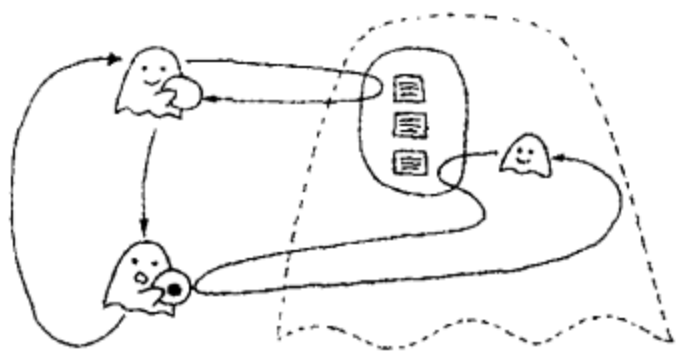
首先，在这里加入一个 Scheduler 参与者。Servant 参与者是由 Scheduler 参与者调用的。这是只有一个工人线程的 Worker Thread Pattern（第 8 章）。这样一来，Servant 参与者就不需要支持多线程，也能处理多个 Client 参与者的请求。

Client 参与者送出请求的操作实现时是调用 Proxy 参与者。Proxy 参与者会将请求转换成一个对象，并使用 Producer-Consumer Pattern（第 5 章），传给 Scheduler 参与者。这样，就算 Servant 参与者进行处理需要花些时间，也不会拖慢 Client 参与者的响应性。

接下来，选择要执行的请求，则是 Scheduler 参与者的工作。执行请求的顺序是由 Scheduler 参与者决定的。

执行结果可使用 Future Pattern（第 9 章）返回给 Client 参与者。

这就是 Active Object Pattern。



◆ 相关

实现 Scheduler 参与者的部分，使用了 Worker Thread Pattern（第 8 章）。

从 Proxy 参与者将请求传给 Scheduler 参与者的部分，使用了 Producer-Consumer Pattern（第 5 章）。

对 Client 参与者返回执行结果时，使用了 Future Pattern（第 9 章）。

尾声

从单线程这一条线开始的故事，编织出了许多的 Pattern，终于走到了这里。而编织未来故事的工作，在这里就交给您了。

如果读者想要更深入学习 Java 的线程，请阅读《Java Thread Programming》（参考附录 E [Lea]）一书。另外，从 Doug Lea 的网页上（参考附录 E [LeaWeb]），还可能获取 util.concurrent 包。这是 Java 多线程程序设计的实用类链接库。在此特别建议读者尝试使用。

感谢读者沿着线程这一条线，阅读本书到了这里。期待有朝一日，我们有缘再次相见。

```
public class Main {
    public static void main(String[] args) {
        new Thread() {
            public void run() {
                System.out.println("Enjoy multithreading!");
            }
        }.start();
    }
}
```



附录A

练习问题的解答

Introduction 1

问题 I 1-1 的解答

○（1）Java 的程序中，最少会执行一个线程。

×（2）Thread 类的 run 方法被调用出来后，就会启动新的线程。
要启动新的线程，必须使用 start 方法而非 run 方法。

×（3）Runnable 接口中声明有 start 方法与 run 方法。
声明在 Runnable 接口中的只有 run 方法。

○（4）有时候多个线程都可以调用出同一个实例的方法，。
正因为如此，所以必须要有线程的共享互斥（exclusive control）。

○（5）有时候多个线程都可以调用出同一个 Thread 类的实例（Instance）的方法。
虽然是 Thread 类的实例，与其他类的实例的方法并无二致，都可以由多个线程来调用出。

×（6）执行 sleep 方法后，在指定的时间内所有的线程都会暂停执行。
暂时执行的只有调用出 sleep 方法的线程（即现在的线程）而已。

×（7）某个线程在执行 synchronized 方法时，其他的所有线程都会停止执行。
会停止执行的，只有要取得相同实例的锁定（lock）的线程而已。

×（8）执行 sleep 方法后，该线程只有在指定的时间内才会进入 wait set。
若是 sleep 方法，线程不会进入到 wait set。只有执行 wait 方法的时候才会进入 wait set。

×（9）调用 wait 的语句一定要写在 sychronized 方法中才行。
调用 wait 方法的语句可以写在 synchronized 方法中、synchronized 段中或是从这两者调用出的别的方法中。只要执行 wait 方法的线程在执行时已取得对象实例的锁定（lock）即可。

○（10）notifyAll 方法就是 java.lang.Object 类的实例方法。

问题 I 1-2 的解答

这是因为在类链接库中进行了线程的共享互斥（exclusive control）的原因。

也就是说，一个线程在显示 Good!这个字符串时，显示 Nice!的其他线程就会在一旁等待。

问题 I 1-3 的解答

因为主线程调用出来的是 run 方法而非 start 方法。

new PrintThread("*").run();

上面的语句会产生 PrintThread 类的实例，并执行该实例的 run 方法。但是，run 方法并不是由新线程执行的原因，是因为由主线程执行的原因。

显示完 1000 个*之后，才会执行以下的语句：

new PrintThread("+").run();

到最后，所有的显示都是由主线程单独执行

也就是说，这个程序是单一线程（Single Thread）的程序。

问题 I 1-4 的解答

确认用的程序如下。确认的时间有时候可能可以更短。

List AI1-1 确认存款是否可能变成负数的 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
    // 开一个 1000 元的银行账户
        Bank bank = new Bank("A Bad Bank", 1000);
        new ClientThread(bank).start();
    }
}
```

List AI1-2 重复提出及存入 1000 元的 ClientThread 类 (ClientThread.java)

```
public class ClientThread extends Thread {
    private Bank bank;
    public ClientThread(Bank bank) {
        this.bank = bank;
    }
    public void run() {
        while (true) {
            boolean ok = bank.withdraw(1000);
            if (ok) {
                bank.deposit(1000);
            }
        }
    }
}
```

结果如图 AI1-1 所示。

```
存款余额为负数！ money = -1000
(CTRL + C 即可结束)
```

图 AI1-1 执行范例

问题 I 1-5 的解答

造成执行 t.sleep(1000);时暂停，并非加上了 t 的线程，而是执行此语句的线程。使用

```
t.sleep(1000);
```

调用出来的并非 t 的实例方法，而是 Thread 的类方法。也就是说，跟执行以下语句意义相同。

```
Thread.sleep(1000);
```

新启动的线程若要予以暂停，就要在 MyThread 类的 run 方法中调用 sleep 方法。

▶▶ 补充一点：通过实例来调用类方法会招致误解

在本问题中，通过实例来调用类方法在写法上并没有错，但是容易让阅读程序的人造成误解，所以写程序时一定要如下这样写喔！

```
t.sleep(1000);          ×容易招致误解的写法
Thread.sleep(1000);     ○最佳写法
```

问题 I 1-6 的解答

○（1）x.iA();
　　与
　　x.iA();
只要是 synchronized 以外的方法，都可以从两个线程执行。

○（2）x.iA();
　　与
　　x.iB();
只要是 synchronized 以外的方法，都可以从两个线程执行。

○（3）x.iA();
　　与
　　x.iSyncA();
只要是 synchronized 以外的方法，都可以从两个线程执行。即使有线程在执行其他 synchronized 方法，也仍能执行。

×（4）x.iSyncA();
　　与
　　x.iSyncA();
相同实例的 synchronized 的实例方法，一次只能由一个线程来执行。

×（5）x.iSyncA();

与

x.iSyncB();

相同实例的 synchronized 的实例方法，一次只能由一个线程来执行。

○（6）x.iSyncA();

与

y.iSyncA();

由于实例不同则锁定（lock）也不同，因此即使是 synchronized 的实例方法也能从两个以上的线程来同时执行。

○（7）x.iSyncA();

与

x.iSyncB();

由于实例不同则锁定（lock）也不同，因此即使是 synchronized 的实例方法也能从两个以上的线程来同时执行。

○（8）x.iSyncA();

与

something.cA();

类方法本来就不是 synchronized 方法，因此可以同时执行。

○（9）x.iSyncA();

与

something.cSyncA();

synchronized 的实例方法与 synchronized 的类方法由于锁定（lock）不同，因此可以从数个线程来同时执行。

×（10）Something.cSyncA();

与

Something cSyncA();

Synchronized 的类方法无法从数个线程同时执行。

×（11）Something.cSyncA();

与

Something cSyncB();

Synchronized 的类方法无法从数个线程同时执行。

×（12）x.cSyncA();

与

y.cSyncB();

x.cSyncA()调用的方法是 Something.cSyncA()，而 y.cSyncB()调用的方法是 Something.cSyncB()，因此都会变成 Synchronized 的类方法，所以无法从数个线程同时执行。

注意：本题的第（12）小题中，为了向读者解说，因此加上了“通过实例来调用类方法”这句话。可是为免导致读者的误解，实际在写程序时，请勿加上这段话。

Introduction2

问题 I2-1 的解答

（1）在方法中一概都加上 synchronized，是不是就万无一失了呢？

【安全性】

加上 synchronized 方法并不会降低安全性，但是安全性也不见得就会有所提升。例如：应该将保护的字段若设置为 public，不管在方法加多少 synchronized，这个程序仍然欠缺安全性。

【生存性】

一概加上 synchronized 的话，一般而言其生存性都会下降。例如，很容易导致死锁（dead lock）的发生。

【性能性】

没有必要却还加上 synchronized 的话，性能（例如 throughput）会下降。这是因为一般而言，synchronized 方法的调用比非 synchronized 方法的调用更花时间的缘故。

（2）synchronized 方法中不是变成了无穷循环了吗？这样的程序是不是不好？

【安全性】

synchronized 方法中即使变成无穷循环，只要循环中所进行的处理是正确的，就不会危及其安全性。

【生存性】

synchronized 方法中要是变成无穷循环，那么想执行其 synchronized 方法的其他线程就会锁住，因而导致生存性下降。

【性能性】

生存性下降，则性能性（例如 throughput）也会下降。

（3）由于程序的错误而陷入无穷循环的线程即使启动 100 个，也由于线程只是不断执行而已，所以应该没有什么关系。

【安全性】

即使有很多只是进行循环的线程，也不会导致安全性下降。

【生存性】

如果只是有很多进行循环的线程，并不会导致生存性下降。但是，线程

使用的内存若是超过系统所能提供的话，某些重要的线程可能会无法启动，因此生存性会下降。

【性能性】

线程要执行时会用到内存，之后又会消耗 CPU 时间这项 resource，因此性能性会下降。

（4）此服务器一次只能连接一个客户端，请将此服务器调整得更好一点。

【性能性】

从这个问题看来，意思好像是说能够同时连接数个客户端的才是好的服务器。所以问题在于服务器的“负荷量”。

（5）这个查找程序有够烂。一旦启动就得要等到全部查找完毕为止才能取消（Cancel）。

【性能性】

这个问题跟程序的“响应能力”有关。

（6）这样的话，类 A 与类 B 双方必须注意线程共享互斥的影响，这样是不是不太好？

【重复使用性】

问题在于类的“重复使用性”。发问者可能担心程序代码必须考虑到类 A 与类 B 两者对线程共享互斥的问题。所要启动的线程的相关程序代码只要在某一方的类中予以关闭，就可以减少类的影响，也可以提高这个零件的重复使用性。

问题 I 2-2 的解答

一般而言，不会变成两倍。可能的原因可举出如下 3 点：

理由 1：有时由于硬件的限制，并不一定 2 倍的线程可以同时执行操作（并列执行操作）。此外，这种情况下也很容易发生线程切换的 overhead。

理由 2：处理的工作不一定能平均分配给所有的线程。

理由 3：即使没有硬件的控制，2 倍的线程同时执行，也会发生用以进行线程间共享互斥的 overhead，这也是原因之一。

第 1 章

问题 1-1 的解答

延长临界区间（Critical Section），就能够提高检出错误的可能性。

例如，我们在 pass 方法中“指定到 name”与“指定到 address”之间加上 sleep

方法的调用。List A1-1 中，就将线程暂停了 1000 毫秒。

List A1-1 尽早检测出没有 Thread Safe 的 Gate 类(Gate.java)

```
public class Gate {
    private int counter = 0;
    private String name = "Nobody";
    private String address = "Nowhere";
    public void pass(String name, String address) {
            this.counter++;
            this.name = name;
            this.address = address;
        try {
            Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            check();
    }
    public String toString() {
    return "No." + counter + ": " + name + ", " + address;
    }
    private void check() {
        if (name.charAt(0) != address.charAt(0)) {
            System.out.println("***** BROKEN ***** " + toString());
        }
    }
}
```

执行范例结果如图 A1-1 所示。此例中，countet 到 3 时，就已经检测出错误了。

```
Testing Gate, hit CTRL+C to exit.
Alice BEGIN
Bobby BEGIN
Chris BEGIN
***** BROKEN ***** No.3: Chiris, Alaska
***** BROKEN ***** No.4: Alice, Brazil
***** BROKEN ***** No.5: Bobby, Canada
***** BROKEN ***** No.6: Chiris, Alaska
***** BROKEN ***** No.7: Alice, Brzil
***** BROKEN ***** No.8: Bobby, Canada
***** BROKEN ***** No.9: Chris, Alaska
***** BROKEN ***** No.10: Alice, Brazil
(以下省略)
```

图 A1-1　执行范例

▶▶ 补充一点：Thread.yield

也可以在临界区间（Critical Section）中，调用出 Thread 类的 yield 方法，来催促进行线程的切换。

问题 1-2 的解答

将字段（field）设为 private 是为了让程序设计人员能更易于确认类的安全性。

有 private 的字段只能从类来进行访问。因此，只要确认该类中声明的方法能安全地访问字段，就能确认字段的安全性。至于此类以外的类则没有确认的必要。

有 protected 的字段也可以从该类的子类或相同包内的类来访问。因此，为了确认安全性，就必须连子类或同包内的类都一起检查确认。

有 public 的字段则可以从任意的类来访问。因此，为了确认安全性，凡所有能访问到这个字段的所有类都必须检查才行。

问题 1-3 的解答

我们假设 List 1-4 的 Gate 类被其他程序使用。

```
public String to String() {
      returen "No." + counter + ": "+ name + "
}
```

我们假设 UserThread 类的线程在通过 pass 方法时，其他线程 X 调用 toString。线程 X 在引用 name 字段的值之后到引用 address 字段的值之间，UserThread 的线程可能会改掉 address 的值，这样一来，toString 方法很可能对线程使用 name 与 address 第一个字母不一致的值来构成字符串。

范例程序的范围中，虽然将 toString 设成 synchronized 方法其实也无损其安全性，一般而言，数个线程共有的字段都必须要设 synchronized（或是 volatile）来予以保护。

由于 check 方法是 private，所以绝对不可能从 Gate 类之外直接调用出。看看 Gate 类，调用 check 方法的都是已设成 synchronized 方法的 pass 方法。因此，不需要再将 check 方法设成 synchronized 方法（不过改成 synchronized 方法也不会降低安全性）。

问题 1-4 的解答

○（1）无法建立 Point 类的子类。

Point 类被声明为 final，因此无法建立子类。

○（2）要指定到 Point 类的 x 字段的语句只能写在 Point 类的构造器（contructor）以及方法中。

x 字段被声明为 private，因此无法以 Point 类以外的构造器或方法来指定。

○（3）关于有 Point 类的实例，可以执行这里读取的 move 方法的是一次一个的线程。

move 方法被声明为 synchronized，因此一次只能执行一个线程。

×（4）Point 类即使由数个线程来使用也很安全。

只使用 move 方法的话很安全。但是，Point 类还没读到的部分可能会有下列指定到个别字段的方法，因此无法断言这样很安全。

```
public synchronized void setX(int x) {
    this.x = x;
    }
public synchronized void setY(int y) {
    this.y = y;
    }
```

×（5）只要将 Point 类剩下的方法全部都读取，就可以判断是否发生死锁。

即使读取 Point 类剩下的方法，也并不一定能够判断是否发生了死锁。这是因为使用 Point 类的其他类的关系，有时会发生死锁之故。

▶▶ 补充一点：程序设计人员应该要注意的事项范围很广

我们先来思考题目中“只读取部分程序代码”的意思。

访问控制（access control）在只读取类的一部份时就能够进行判断。但是，安全性则不能光读一部分的类就判断得知。生存性则是有时候即使读了整个类也不见得能判断得出来。

由此可知，多线程程序设计的困难之处了。程序设计人员要注意的事项范围比一般的程序设计还要广泛了。

问题 1-5 的解答

不，并不安全。为了要保障安全，enter、exit、getCounter 的所有方法都必须要设成 synchronized 方法。

我们来看看下面这个可以递增 counter 字段的值的语句。

```
counter++;
```

这个语句只有一行，线程在执行时，并无法不受其他线程影响。线程会重复执行调查 counter 的值，然后增加 1，再指定到 counter 的多项处理。

线程 A 的 enter 方法与线程 B 的 exit 方法要同时执行时，不使用 Single Threaded Execution 模式的话，就可能会变成如下的顺序。这时，线程 B 执行 exit 方法的效果会消失。

线程 A (enter)	线程 B (exit)	counter 的值
调查 counter 的值		100
增加 1		100
<<<<在此切换成线程 B>>>>		100
	调查 counter 的值	
	减少 1	100
	将 99 指定到 counter	100
<<<<在此切换成线程 A>>>>		99
将 101 指定到 counter		101

图 A1-2　线程 B 的 exit 的效果消失的范例

测试 SecurityGate 类的安性的程序如 List A1-2。执行这个程序很有可能就可以检查出并不安全（有的时候可能会无法检查出来）。以下的 List A1-2 中，建立了 100 个 CrackerThread 类的实例，各自会执行 1000000 次 enter 与 exit。enter 与 exit 因为被调用的次数是相同的，所以所有的线程结束操作之后，getCounter()的值应该会回到 0。但是，有时候也会像执行范例（图 A1-3, A1-4）一样没有变成 0。

以下的程序是使用 join 方法来检查线程是否已执行完了。

List A1-2　检查安全门(Security Gate)的程序(main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing SecurityGate...");
        SecurityGate gate = new SecurityGate();
        CrackerThread[] t = new CrackerThread[100];

        // CrackerThread 启动
        for (int i = 0; i < t.length; i++) {
            t[i] = new CrackerThread(gate);
            t[i].start();
        }

        // 等待 CrackerThread 结束
        for (int i = 0; i < t.length; i++) {
            try {
                t[i].join();
            } catch (InterruptedException e) {
            }
        }

        // 确认
```

```
            if (gate.getCounter() == 0) {
                System.out.println("SecurityGate may be safe.");
            } else {
                System.out.println("SecurityGate is NOT safe!");
                System.out.println("getCounter()      ==      "      +
gate.getCounter());
            }
        }
}
```

List A1-3 表示 enter 与 exit 各执行 1000000 次的线程的类(CrackerThread.java)

```
public class CrackerThread extends Thread {
    private SecurityGate gate;
    public CrackerThread(SecurityGate gate) {
        this.gate = gate;
    }
    public void run() {
        for (int i = 0; i < 1000000; i++) {
            gate.enter();
            gate.exit();
        }
    }
}
```

```
Testing SecurityGate...
SecurityGate is NOT safe!
getCounter() == 7          ←不会变成 0
```

图 A1-3 执行范例 1

```
Testing SecurityGate...
SecurityGate is NOT safe!
getCounter() == -8         ←不会变成 0
```

图 A1-4 执行范例 2

要将 enter 方法与 exit 方法设为 synchronized 的理由我想各位已经了解了。将 getCounter 设为 Synchronized 的理由又为何呢?如果没有将 getCounter 方法设为 synchronized 或未将 counter 字段设为 volatile 时，其他的线程不管怎样更改 counter 的值，都只是对工作内存（Working Memory）进行操作而已，而不见得会反映到主保存器（Main Memory）。关于工作内存与主保存器，请参照附录 B “Java 的内存模式”。

问题 1-6 的解答

如下介绍两个方法（另有其他方法）。

◆ 方法 1: Alice 与 Bobby 是以相同顺序拿餐具的方法

最单纯的方法就是让 Alice 与 Bobby 以相同顺序拿餐具。ListA1-4 的 Main 类中 Alice 与 Bobby 都是先拿汤匙再拿叉子。这样就不会发生 dead lock。这样一来书中所说的“SharedResource 角色为对称”这个条件就不成立了。用这个方法并不需要再修改 Tool 类与 EaterThread 类。

List A1-4 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing EaterThread, hit CTRL+C to
exit.");
        Tool spoon = new Tool("Spoon");
        Tool fork = new Tool("Fork");
        new EaterThread("Alice", spoon, fork).start();
        new EaterThread("Bobby", spoon, fork).start();
    }
}
```

```
(前略)
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right).
Alice is eating now, yam yam!
Alice puts down [ Fork ] (right).
Alice puts down [Spoon ] (left).
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right)
Alice is eating now, yam yam!
Alice puts down [ Fork ] (right).
Alice puts down [Spoon ] (left).
Alice takes up [ Spoon ] (left).
Alice takes up [ Fork ] (right)
(以下省略。按 CTRL+C 结束)
```

图 A1-5　执行范例

◆ 方法 2: 取得“汤匙与叉子”配对的方法

现在我们不要个别取得汤匙或叉子的 lock，而准备代表“汤匙与叉子”配对的另一个对象。另外我们假设我们已通过取得实例的 lock 而取得“汤匙与叉子”的配对了。这样 dead lock 就不会发生。这样在本章中所说的“有数个 SharedResource 角色”这个条件就不完整了。

修改后的程序就如 List A1-5~A1-7 所示。

List A1-5 修改后的 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Testing EaterThread, hit CTRL+C to
exit.");
        Tool spoon = new Tool("Spoon");
        Tool fork = new Tool("Fork");
        Pair pair = new Pair(spoon, fork);
        new EaterThread("Alice", pair).start();

        new EaterThread("Bobby", pair).start();
    }
}
```

List A1-6 修改后的 EaterThread 类(Eater.java)

```
public class EaterThread extends Thread {
    private String name;
    private final Pair pair;
    public EaterThread(String name, Pair pair) {
        this.name = name;
        this.pair = pair;
    }
    public void run() {
        while (true) {
            eat();
        }
    }
    public void eat() {
        synchronized (pair) {
            System.out.println(name + " takes up " + pair + ".");
```

```
            System.out.println(name + " is eating now, yam yam!");

            System.out.println(name + " puts down " + pair + ".");
        }

    }
}
```

List A1-7 修改后的 Pair 类(Pair.java)

```
public class Pair {
    private final Tool lefthand;
    private final Tool righthand;
    public Pair(Tool lefthand, Tool righthand) {
        this.lefthand = lefthand;
        this.righthand = righthand;
    }
    public String toString() {
        return "[" + lefthand + " and " +righthand + " ] ";
        }
    }
```

```
Alice takes up [ [ Spoon ] and [ Fork ] ].
Alice is eating now, yam yam!
Alice puts down [ [ Spoon ] and [ Fork ] ].
Alice takes up [ [ Spoon ] and [ Fork ] ].
Alice is eating now, yam yam!
Alice puts down [ [ Spoon ] and [ Fork ] ].
Bobby takes up [ [ Spoon ] and [ Fork ] ].
Bobby is eating now, yam yam!
Bobby puts down [ [ Spoon ] and [ Frok ] ].
Alice takes up [ [ Spoon] and [ Fork ] ].
Alice is eating now, yam yam!
Alice puts down [ [ Spoon ] and [ Forkk ] ].
(以下省略。按 CTRL+C 结束)
```

图 A1-6 执行范例

问题 1-7 的解答

安装的方法有很多。在此依顺序单纯程度介绍两个范例（越单纯，使用时的限制越大，重复使用性也越低）。

◆ 解答例 1：单纯的 *Mutex* 类

List A1-8 是最单纯的 Mutex 类。在此使用 busy 这个 boolean 类型的字段。busy 若是 true，就表示执行了 lock；如果 busy 是 false，则表示执行了 unlock 方法。

Lock 与 unlock 双方都已是 synchronized 方法保护着 busy 字段。

List A1-8　单纯的 Mutex 类(Mutex.java)

```
public final class Mutex {
    private boolean busy = false;

    public synchronized void lock() {
        while (busy) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        busy = true;
    }
    public synchronized void unlock() {
        busy = false;
        notifyAll();
    }
}
```

List A1-8 的 Mutex 类在问题 1-7 会正确地执行。但是，若使用于其他用途，则会发生如下的问题。这就是对使用 Mutex 类的限制。这意味着 Mutex 类的重复使用性上会有问题。

【问题点 1】

假设有某个线程连续两次调用出 lock 方法。调用后，在第 2 次的调用时，由于 busy 字段已经变成 true 因此会 wait。这就好像自己把自己锁在外面，进不了门的意思是一样的。

【问题点 2】

即使是尚未调用出 lock 方法的线程，也会变成可以调用 unlock 方法。就好比即使不是自己上的锁，自己还是可以将门打开一样。

◆ 解答范例 2：改良后的 *Mutex* 类

List A1-9 是将类似范例 1 的问题予以改良而形成的新的 Mutex 类。在此，现在的 lock 数记录在 locks 字段中。这个 lock 数是从 lock 的调用次数扣掉 unlock 调用次数得出的结果。连调用 lock 方法的线程也记录在 owner 字段上。我们现在用 locks

与 owner 来解决上述的问题点。

List A1-9 改良后的 Mutex 类 (Mutex.java)

```
public final class Mutex {
    private long locks = 0;
    private Thread owner = null;
    public synchronized void lock() {
        Thread me = Thread.currentThread();
        while (locks > 0 && owner != me) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        // locks == 0 || owner == me
        owner = me;
        locks++;
    }
    public synchronized void unlock() {
        Thread me = Thread.currentThread();
        if (locks == 0 || owner != me) {
            return;
        }
        // locks > 0 && owner == me
        locks--;
        if (locks == 0) {
            owner = null;
            notifyAll();
        }
    }
}
```

▶▶ 补充一点:assert

List A1-9 的程序代码中，注释内的算式表示该时间点应该成立的条件。

lock 方法中的

```
// locks == 0 || owner == me
```

其意思是“lock 数为 0，或者自己已 lock 了”。

unlock 方法中的

```
// locks > 0 && owner == me
```

其意思是“lock 数比 0 大，而且自己已锁住了”。

改良后的 Mutex 类的 unlock 方法中，如果 unlock 太多次，或是将没有 lock 住的线程进行 unlock 时，就会马上 return。这个部分应该可以抛出异常，返回错误值。

J2SE 1.4 中，Java 新增了新的关键词 assert。使用这个关键字，就可以撰写此时间点应该成立的条件。

详情请参考如下网站:

- Programming with Assertions
  http://java.sun.com/j2se/1.4/docs/guide/lang/asert.html
- Assertion 功能 (名称为日文解释)
  http://www.shudo.net/java/assert/assert.html

第 2 章

问题 2-1 的解答

○（1）java.lang.String 类是 immutable 的类。
×（2）java.lang.StringBuffer 类是 immutable 的类。
○（3）变成 final 的字段无法被指定两次。
×（4）变成 private 的字段可以从该类与子类直接连接。
可以直接连接上变成 private 的字段的，只有已经声明白该字段的类。
×（5）将方法设为 synchronized 不会导致什么问题，所以应该尽量都加上 synchronized。
可能会降低类的生存性或性能性，因此并不宜全部都加。

问题 2-2 的解答

的确，执行问题中所示的程序后，会显示出 CAT。但是保存了变量 s 的实例的属性并没有因此被置换。Replace 方法会新建立一个替代字符串中包含的文字的实例，将此实例设为返回值[注 1]。

List A2-1 的程序可以确认已经建立了其他的实例。其执行结果则如图 A2-1 所示。

List A2-1 确认以 replace 方法已建立其他实例(Main.java)

```
public class Main {
    public static void main(String[] args) {
        String s = "BAT";
        String t = s.replace('B', 'C'); // 将'B'代换成'C'
```

注 1：但是，代表传递给 replace 方法的置换前的文字与置换后的文字的参数值如果相同（也就是实际上未置换时），就不会建立出新的实例，原来的实例（this）则会成为返回值。

```
            System.out.println("s = " + s); // 执行 replace 后的 s
            System.out.println("t = " + t); // replace 的返回值 t
            if (s == t) {
                System.out.println("s == t");
            } else {
                System.out.println("s != t");
            }
        }
    }
```

```
    s = BAT      ← 执行 replace 之后，s 的值为"BAT"
    t = CAT      ← replace 的返回值为"CAT"
    s != t       ← 双方的实例相异
```

图 A2-1　执行结果

问题 2-3 的解答

在笔者的使用环境下会得出图 A2-2 这样的结果。使用 synchronized 跟使用 Immutable 模式比起来，约要花 2 倍的时间。在此，计算时间时，线程的冲突并没有发生，因此计算的时间为取得实例的 lock 的处理与释放 lock 的处理所花费的时间。

```
NotSynch: BEGIN
NotSynch: END
Elapsed time = 38896msec. ← 没有 synchronized 时所花费的时间
Synch: BEGIN
Synch: BEGIN
Elapsed time = 76400msec. ← 有 synchronized 时所花费的时间
```

图 A2-2　执行结果

但是，请注意在此我们连续调用出了几乎没有任何处理的方法。这个结果很有可能会导致用 synchronized 所占的时间增加比例比一般的程序还大。

Java 编译器的最佳化的不同，或是 Java 执行环境的不同等都会大幅影响时间差的比例，因此本程序的执行范例请作为参考即可。

问题 2-4 的解答

这是因为 getInfo 方法的返回值（StringBuffer 的实例）若改变，info 返回值引用的实例的属性也会改变。

实际动手写写看改变 UserInfo 实例属性的程序吧（ListA2-2）！

List A2-2 改变 UserInfo 状态的 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        //建立实例
        UserInfo userinfo = new UserInfo("Alice", "Alaska");

        //显示
        System.out.println("userinfo = " + userinfo);

        //更改状态
        StringBuffer info = userinfo.getInfo();
        info.replace(12, 17, "Bobby");  // 12..16是"Alice"的位置

        //再度显示
        System.out.println("userinfo = " + userinfo);
    }
}
```

在此，首先制作并显示 Alice（来自 Alaska）的数据。但是，之后就利用 getinfo 方法所得到的 StringBuffer 的实例(info)，将"Alice"改写成"Bobby"。之后再度显示看看，我们可以发现 UserInfo 的实例已经改变了（图 A2-3）。

```
userinfo = [ UserInfo: <info name="Alice" address="Alaska" /> ]
userinfo = [ UserInfo: <info name="Bobby" address="Alaska" /> ]
← 名字已改成 Bobby
```

图 A2-3 执行范例

用 getInfo 方法得到的 info 字段所拥有的实例并非 String 的实例，而是 StringBuffer 的实例。StringBuffer 类与 String 类不同，拥有更改内部状态的方法，因此可以从外部来改写 info 字段的属性。

String 类的 replace 方法不会更改实例本身，而 StringBuffer 类的 replace 方法会更改实例本身。StringBuffer 类是 mutable。

info 字段已被声明为 final，因此 info 字段的值本身（指的是那个实例）不会有所改变。但是 info 字段所指的对象其实例状态则会有所改变。

问题 2-5 的解答

Line 类并非 immutable。

我们先从 Line 类使用的 Point 类看起。Point 类的 x 字段或 y 字段变成 public，而且还不是 final，所以这些字段的值会被自由地改写。因此，Point 类是 mutable 类。

接下来我们来看看 Line 类的第 2 个参数所给的实例会被指定到字段中构造器。

```
Public Line(Point startPoint, Point endPoint) {
    this.starPoint = starPoint;
    this.endPoint = endPoint;
}
```

构造器的参数所给予的实例会被直接指定到字段。这是什么意思呢？参数所给予的 Point 实例是在构造器的外部所建立的，因此调用出这个构造器者很有可能就拥有这个实例的引用。

```
this.startPoint
```

是 final，这个字段的值本身就不会再变化了。但是，这个字段所指的 Point 的实例的内部状态却有可能改变。

我们在 List A2-3 可以确认这件事情。

List A2-3 改变 Line 状态的 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
        // 建立实例
        Point p1 = new Point(0, 0);
        Point p2 = new Point(100, 0);
        Line line = new Line(p1, p2);

        // 显示
        System.out.println("line = " + line);

        // 改变状态
        p1.x = 150;
        p2.x = 150;
        p2.y = 250;

        // 再次显示

        System.out.println("line = " + line);
    }
}
```

```
line = [ Line: (0,1)-(100,0) ]
line = [ Line: (150,0)-(150,250) ]     ← 直线移动了
```

图 A2-4

我们可以看到，虽然只是随便随便替换一下 p1、p2 这两个 Point 的实例，可是图 A2-4 的 Line 的实例属性的确已经改变了。

在 Point 这个非 immutable 的类(mutable 的类)的影响下，连 Line 类都变成 mutable。

◆ 将 Line 类设置成 immutable 的诀窍

要如何才能将 Line 类设为 immutable 呢？当然，将 Point 类设为 immutable 的话，Line 类也会变成 immutable。例如，Point 的 x 字段与 y 字段都设为 public final，就可以做到。

可是，Point 不管是 immutable 还是 mutable，都可以将 Line 设置成 immutable。方法如 List A2-4。

List A2-4 设置为 immutabe 的 Line 类(Line.java)

```
public class Line {
    private final Point startPoint;
    private final Point endPoint;
    public Line(int startx, int starty, int endx, int endy) {
        this.startPoint = new Point(startx, starty);
        this.endPoint = new Point(endx, endy);
    }
    public Line(Point startPoint, Point endPoint) {
      this.startPoint = new Point(startPoint.x, startPoint.y);
      this.endPoint = new Point(endPoint.x, endPoint.y);
    }
    public int getStartX() { return startPoint.getX(); }
    public int getStartY() { return startPoint.getY(); }
    public int getEndX() { return endPoint.getX(); }
    public int getEndY() { return endPoint.getY(); }
    public String toString() {
        return "[ Line: " + startPoint + "-" + endPoint + " ]";
    }
}
```

List A2-4 中，我们要建立一个拥有与所给予的实例相同属性的新实例，将此新实例指定到字段中。

设置为 immutable 的 List A2-4 的 Line 类，如果用刚才 List A2-3 的方式予以移动，就会出现图 A2-5 的结果。可以看到 Line 的确并没有受到外界的影响。

```
line = [ Line: (0,0) - (100,0) ]
line = [ Line: (0,0) - (100,0) ]        ←没有变化
```

图 A2-5 执行结果

List A2-4 中，使用 new 拷贝了 Point 类的实例，但是应该也可以使用 Java 的 clone 方法。不过，clone 方法会执行 shallow copy，这点要请多加注意。

问题 2-6 的解答

ImmutablePerson 类中有错误。

List 2-10 的 ImmutablePerson 类欠缺安全性可以从 List A2-5 的程序检测得知。

在此，我们将 MutablePerson 的实例连续以主线程来改写。为了便于确认，我们将 name 字段与 address 字段设为相同字符串(数字字符串)。

而在 CrackerThread 类中，从被给予的 MutablePerson 也生出一个个新的 ImmutablePerson 的实例。我们来看看这样产生的 ImmutablePerson 的实例的 name 与 address 的值是否永远相等。

List A2-5 检测 ImmutablePerson 的确欠缺安全性的 Main 类(Main.java)

```
import person.MutablePerson;
import person.ImmutablePerson;
public class Main {
    public static void main(String[] args) {
        MutablePerson mutable = new MutablePerson("start", "start");
        new CrackerThread(mutable).start();
        new CrackerThread(mutable).start();
        for (int i = 0; true; i++) {
            mutable.setPerson("" + i, "" + i);
        }
    }
}

class CrackerThread extends Thread {
    private final MutablePerson mutable;
    public CrackerThread(MutablePerson mutable) {
        this.mutable = mutable;
    }
    public void run() {
        while (true) {
            ImmutablePerson immutable = new ImmutablePerson(mutable);
            if
(!immutable.getName().equals(immutable.getAddress())) {
                System.out.println(currentThread().getName() + "
```

```
***** BROKEN ***** " + immutable);
            }
        }
    }
}
```

这样一来，就会如图 A2-6 这样出现 BROKEN 的显示。我们就可了解安全性的确有问题。

```
Thread-1 ***** BROKEN ***** [ ImmutablePerson: 213444, 214742 ]
Thread-2 ***** BROKEN ***** [ ImmutablePerson: 386237, 387624 ]
Thread-1 ***** BROKEN ***** [ ImmutablePerson: 618964, 619060 ]
Thread-2 ***** BROKEN ***** [ ImmutablePerson: 918612, 919875 ]
(以下省略。按 CTRL+C 结束)
```

图 A2-6　执行范例

以下就来指出错误所在，并进行修改。

ImmutablePerson 类的构造器（Constructor）中，有的方法会将 MutablePerson 当作参数。在这当中如下调用出 MutablePerson 类的 getName 方法与 getAddress 方法。

```
this.name = person.getName();
this.address = person.getAddress();
```

但是，这 2 个调用必须放在临界区间（Critical Section）当中。

原因在于，在第一次的 getName 调用之后，到下一个 getAddress 的调用之间，可能会有别的线程使用 MutablePerson 的 setPerson 方法而将 address 字段改写掉。

在此，由于是构造器（Constructor），所以就要使用 synchronized 段，这时应该取得 lock 的实例，也就是以参数所给予的 MutablePerson，这点请注意。

修改后的 Immutable Person 类如 List A2-6 所示。

List A2-6　具备安全性的 ImmutablePerson 类(ImmutablePerson)

```
package person;

public final class ImmutablePerson {
    private final String name;
    private final String address;
    public ImmutablePerson(String name, String address) {

        this.name = name;
        this.address = address;
    }
```

```
    public ImmutablePerson(MutablePerson person) {
        synchronized (person) {
            this.name = person.getName();
            this.address = person.getAddress();
        }
    }
    public MutablePerson getMutablePerson() {
        return new MutablePerson(this);
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public String toString() {
        return "[ ImmutablePerson: " + name + ", " + address + " ]";
    }
}
```

从刚才的 Main 类来执行之，也不会出现 BROKEN（图 A2-7）。

```
(没有出现任何消息。按 CTRL+C 结束)
```

图 A2-7　执行范例

第 3 章

问题 3-1 的解答

○（1）getRequest 与 putRequest 分别由不同的线程来调用。

getRequest 由 ServerThread 来调用，putRequest 则是由 ClientThread 来调用。

×（2）RequestQueue 的实例可以做出两个来。

只会由 Main 类做出一个，而这个体就由 ClientThread 与 ServerThread 共享。

○（3）从 getRequest 当中调用 removeFirst 方法时，queue.size() > 0 的值一定是 true。

queue.size() > 0 是 Guard 条件。Guarded Suspension 模式（Pattern）能够保证在执行所要进行的处理时，能满足 Guard 条件。

○（4）从 getRequest 当中调用 wait 方法时，queue.size() > 0 的值一定为 false。

在未满足 Guard 条件时，才会不易调用出 wait 方法。

× （5）ClientThread 的线程在执行 putRequest 时，ServerThread 的线程不会进行任何处理。

ServerThread 的线程有可能会执行处理也有可能不执行处理。例如，如果要执行 getRequest，反而无法取得 lock 而会 block 起来。而要执行 wait 的话，则会在 wait set 中停住。而执行 sleep 的话，则会在指定的时间内停止。但是除此之外，ServerThread 的线程都可以自由执行。

× （6）从 getRequest 调用出 wait 方法的线程，会释放 lock，进入 queue 的 waitset。

会进入的是 this(Request Queue 类的实例)的 waitset，而非 queue(LinkedList 类的实例)的 waitset。

× （7）putRequest 方法中的 notifyAll();这个语句跟写成 queue.notifyAll();的意思一样，只是后者比较冗长。

写成 queue.notifyAll();意思就截然不同了。写成 this.notifyAll();则与 notifyAll();意思相同。

问题 3-2 的解答

即使先执行 notifyAll()，RequestQueue 类也会安全执行。

在执行 notifyAll 时，参数 request 还没有被添加到队列。但是，执行 notifyAll 的线程由于有 this 的锁定，因此为了 notifyAll，来自 waitset 的其他线程都要取得锁定，全部都要阻挡起来。所以，其他线程的处理实质上并不会有什么进展（Guard 条件也不测试）。

另一方面，执行 notifyAll 的线程会在执行过 addLast 之后从 putRequest 回来。这时，this 的锁定也会被解放。之后，被阻挡住的其他线程（其中一个）会取得 this 的锁定而继续进行处理（Guard 条件不易检测）。

不过，由于将 notifayAll 写在最后比较容易理解，因此建议在写程序时可以比照这样的写法。

问题 3-3 的解答

首先，我们先具体思考看看什么是“理当如此”的处理操作。

- 在调用 getRequest 时，如果是 queue.size() <= 0，线程 Bobby 就会 wait。
- 在 putRequest 之后，线程 Alice 会进行 notifyAll。
- notifyAll 之后，原本 wait 的线程 Bobby 会结束 wait。

在这里，我们将调试输出（Debug Print）加到 wait()的前后与 notifyAll()的执行部分（List A3-1）。

List A3-1 加入调试输出(Debug Print)的 RequestQueue 类 (RequestQueue.java)

```
import java.util.LinkedList;

public class RequestQueue {
    private final LinkedList queue = new LinkedList();
    public synchronized Request getRequest() {
        while (queue.size() <= 0) {
            try {

System.out.println(Thread.currentThread().getName() + ": wait()
begins, queue = " + queue);
                wait();

System.out.println(Thread.currentThread().getName() + ": wait()
ends,   queue = " + queue);
            } catch (InterruptedException e) {
            }
        }
        return (Request)queue.removeFirst();
    }
    public synchronized void putRequest(Request request) {
        queue.addLast(request);
        System.out.println(Thread.currentThread().getName() + ":
notifyAll() begins, queue = " + queue);
        notifyAll();
        System.out.println(Thread.currentThread().getName() + ":
notifyAll() ends, queue = " + queue);
    }
}
```

```
Alice request [ Request No.0 ]    ←Alice 对 No.0 进行 request
Alice: notifyAll() begins, queue = [[Request No.0 ]]
                                  ←Alice 进行 notifayAll(谁也没有在等)
Alice: notifyAll() ends, queue = [[ Request No.0 ]]
Bobby handles [ Request No.0 ]    ←Bobby 处理 No.0(不等待)
Bobby: wait() begins, queue = []  ←Bobby 进行 wait(进入等待)
Alice requests [ Request No.1 ]   ←Alice 对 No.1 进行 request
Alice: notifyAll() begins, queue = [[ Request No.1 ]]
                                  ←Alice 进行 notifayAll(Bobby 在等待)
Alice: notifyAll() ends, queue = [[ Request No.1 ]]
Bobby: wait() ends, queue = [[ Request No.1 ]]
                                  ←Bobby 结束 wait。No.1 进入了 queue
Bobby handls [ Request No.1 ]     ←Bobby 处理 No.1(在等待之后)
Alice requests [ Request No.2 ]   ←Alice 对 No.2 进行 request
```

（续上图）

```
Alice: notifyAll() begins, queue = [[ Request No.2 ]]
                                    ←Alice 进行 notifayAll(谁也没有在等)
Alice: notifyAll() ends, queue = [[ Request No.2 ]]
Bobby handles [ Request No.2 ]    ←Bobby 处理 No.2(不等待)
Alice requests [ Request No.3 ]   ←Alice 对 No.3 进行 request
Alice: notifyAll() begins, queue = [[ Request No.3 ]]
                                    ←Alice 进行 notifayAll(谁也没有在等)
Alice: notifyAll() ends, queue= [[ Request No.3 ]]
Alice requests [ Request No.4 ]   ←Alice 对 No.4 进行 request
Alice: notifyAll() begins, queue = [[ Request No.3 ], [ Request No.4]]
                                    ←Alice 进行 notifayAll(谁也没有在等)
Bobby handles [ Request No.3 ]    ←Bobby 处理 No.3(不等待)
Bobby handles [ Request No.4 ]    ←Bobby 处理 No.4(不等待)
(以下省略。按 CTRL+C 结束)
```

图 A3-1　执行范例

问题 3-4 的解答

在（1）~（4）之中还有部分问题。将各项说明如下。

◆ *(1) 将 while 改成 if 时(List 3-7)*

在此范例文件中的话是没有问题。但是一般情况的话就没有这么单纯了。

我们假设有好几个线程都在 wait 时，RequestQueue 的实例被执行了 notifyAll。这样一来，好几个线程就会全部开始运行。如果这时 queue.size()的值是 1，那么开始运行的第一个线程在调用出 queue.removeFirst()后，queue.size()的值会变成 0。结果，即使 queue.size()的值是 0，开始运行的第二个线程还是会调用出 queue.removeFirst()。使用这个类恐怕有安全上的问题。

保持 wait 的线程在开始开始运行前一定要再次检查 Guard 条件。这时我们就要使用 while 而非使用 if。

如果不用 notifyAll 而用 notify 的话，是不是就可以使用 if 了呢?如果是这个范例程序的话，的确没有问题。但是，如果在更大型的程序中使用的话，就会发生问题。因为 RequestQueue 的实例很可能被某个线程给 notify 或 notifyAll 的缘故。不用 notifyAll 而使用 notify，while 不用而使用 if 来安装 Guarded Susspension 模式，则这个类的重复使用性会降低。

要再强调一次，Guarded Suspension 模式要使用 while，来“以条件来保护(Guard)”。notify/notifyAll 不过是去检查条件的一个关键点。

让我们想象一下，假设我们在开车，碰到红灯将车停下后正在发呆。这时候如果同车的人喊我们一下，我们也可以慌慌张张踩下油门就这样开走。一定要待我们

确定已经变成绿灯，安全没有问题才能踩油门。Guarded Suspension 模式的道理也是一样，notify/notifyAll 不过只是同车的人喊我们一下的这个操作，要继续进行处理之前，一定要重新再确实检查 Guarded 条件才行。

◆ *(2) synchronized 的范围只在 wait 的话 (List 3-8)*

就范例程序的情况而言是没有问题，但是一般的程序就不是如此了。

这时，下列的处理会跑出 synchronized 段之外。

- 条件的测试
- 调用出 removeFirst

queue.size()的值为 1 时，像图 A3-2 这样两个线程都执行的话，线程 1 会收到异常 NoSuchElementException。

而且，原本 queue 字段的 LinkedList 类就不是 ThreadSafe。

使用这个类会降低安全性。

线程 1	线程 2
条件的测试	
	条件的测试
	removeFirst 的调用
removeFirst 的调用	

图 A3-2　两个线程的处理交错

◆ *(3) 将 try..catch 写在 while 之外时(List 3-9)*

如果是范例程序这样的程序是没有问题，如果是一般程序的话就不行了。

线程在 wait 时，如果其他线程调用 interrupt 方法，即使 Guard 条件没有被满足，这个线程还是会从 while 语句跳到 catch 节，而调用出 removeFirst。也就是说，没有办法等到 Guard 条件被满足。

使用这个类会导致安全性的问题。练习 3-6 中，我们会来建立一个能正确处理异常 InterruptedException 的程序。

◆ *(4) 使用 Thread.sleep 来代替 wait 时(List 3-10)*

即使是在范例程序也有问题。

如果您以为这是“因为每隔 100 毫秒才测试 Guard 条件，因此性能会降低。”，这您就错了。问题不在于性能，而是在于生存性。wait 与 Thread.sleep 不同，执行 wait 的线程会释放实例的锁定，但是 Thread.sleep 则不会。因此，在 getRequest 这个 synchronized 方法中执行 Thread.sleep 的话，其他的线程也无法进入到 putRequest 方法或 getRequest 方法中（会自己 block 起来）。因为无法进入 putRequest 方法，因此 queue.size()的值永远都是 0，Guard 条件永远都不会满足。而原本 sleep 的线程每

100 毫秒醒来一次来对 Guard 条件做测试。但是 Guard 条件为假，所以线程会再沉睡下去。线程就一直重复“醒来→测试→再睡下去”，而在此当中一直想执行 putRequest 或 getRequest 的其他线程则会全部都 block 住。

因此，使用这个类就会发生生存性不佳的问题。在本例中，约 100 毫秒就测试一次 Guard 条件的线程并不会停止处理。不过，也由于只是不断定期测试 Guard 条件而已，处理无法有所进展。像这样，虽然在运行，但是实际上只是在原地踏步的情况，我们就称之为 livelock。

本例也可说是练习问题 I2-1 中出现的“在 synchronized 方法中不断进行无穷循环”的情况之一。

问题 3-5 的解答

之所以不动了，是因为 Alice 与 Bobby 的两个线程发生死锁的缘故。

仔细看 TalkThread 类的 run 方法，会发现一开始就是从 getRequest 方法开始的。

- Alice 在 getRequest 中等待 Bobby 的 Request 而 wait。
- Bobby 在 getRequest 中等待 Alice 的 Request 而 wait。

这样一来，Alice 与 Bobby 两个就卡在一起而动弹不了（死锁）。这就好像模仿对方的两只鹦鹉你瞪我、我瞪你一样，

解决方法之一如 List A3-2 所示。List A3-2 中，一开始先在 requestQueue1 将名为"Hello"的请求予以 putRequest。这就是所谓的“种”，Alice 会在第一个 getRequest 就得到这个请求。

这样的话，两个线程之间传递的数据中，Hello 之后会跟上很多个惊叹号，像图 A3-3 这样（另有其他解决方法）。

顺便一提，Alice 与 Bobby 在陷入死锁时，2 人都没有 requestQueue1 或 requestQueue2 的锁定。因为正在 wait，并正在释放锁定之故。请与练习问题 1-6 中介绍的死锁相互比较。

List A3-2 一开始就将“种”加进去以解决它(Main.java)

```
public class Main {
    public static void main(String[] args) {
        RequestQueue requestQueue1 = new RequestQueue();
        RequestQueue requestQueue2 = new RequestQueue();
        requestQueue1.putRequest(new Request("Hello"));
        new TalkThread(requestQueue1, requestQueue2, "Alice").start();
        new TalkThread(requestQueue2, requestQueue1, "Bobby").start();
    }
}
```

```
Alice:BEGIN
Alice gets [ Request Hello ]       ←一开始 Alice 就取了“种”
Alice puts [ Request Hello! ]      ←加上一个惊叹号再返回
Bobby:BEGIN
Bobby gets [ Request Hello! ]      ←Bobby 接受它
Bobby puts [ Request Hello!! ]     ←再加上一个惊叹号再返回(共 2 个)
Alice gets [ Request Hello!! ]     ←Alice 接受它
Alice puts [ Request Hello!!! ]    ←再加上一个惊叹号再返回(共 3 个)
(中间部分省略)
Alice gets [ Request Hello!!!
```

List A3-4 没有顾虑到异常 InterruptedException 的 ClientThread 类(ClientThread.java)

```
import java.util.Random;

public class ClientThread extends Thread {
    private Random random;
    private RequestQueue requestQueue;
    public ClientThread(RequestQueue requestQueue, String name,
long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                Request request = new Request("No." + i);
                System.out.println(Thread.currentThread().
getName() + " requests " + request);
                requestQueue.putRequest(request);
                Thread.sleep(random.nextInt(1000));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

List A3-5 没有顾虑到异常 InterruptedException 的 ServerThread 类(ServerThread.java)

```
import java.util.Random;

public class ServerThread extends Thread {
    private Random random;
    private RequestQueue requestQueue;
    public ServerThread(RequestQueue requestQueue, String name,
long seed) {
        super(name);
        this.requestQueue = requestQueue;
        this.random = new Random(seed);
    }
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                Request request = requestQueue.getRequest();
                System.out.println(Thread.currentThread().
```

```
getName() + " handles  " + request);
                Thread.sleep(random.nextInt(1000));
            }

        } catch (InterruptedException e) {
        }
    }
}
```

```
(前略)
Alice requests [ Request No.14 ]
Bobby handles [ Request No.14 ]
Alice requests [ Request No.15 ]
Bobby handles [ Request No.15 ]
Alice requests [ Request No.16 ]
Bobby handles [ Request No.16 ]
Alice requests [ Request No.17 ]
Bobby handles [ Request No.17 ]
***** calling interrupt *****          ←程序在此结束
```

图 A3-4　执行范例

▶▶ 补充一点：如果忘了修改 RequestQueue 类会怎么样呢？

让我们来想想如果忘了修改 RequestQueue 类会怎么样？

假设我们没有修改 RequestQueue 类。这样的话，如果 interrupt 方法被调用时线程正在 sleep，程序就可以正常地结束。但是，如果 interrupt 被调用时线程正在 wait，程序就不会结束。这是因为没有顾虑到 InterruptedException 的缘故。也就是说，这个程序每执行几次就会有一次（可能几十次或几百次才发生一次）无法正常结束。

关于如何确实地、优雅地让线程结束，我们在第 10 章的 Two-Phase Termination 模式已介绍。

第 4 章

问题 4-1 的解答

×（1）只有 SaveThread 能调用出 save 方法。

SaveThread 与 ChangerThread 都可以调用出 save 方法。

×（2）将 synchronized 全部删除，将 changed 字段改成 volatile，还是可以执行相同的操作。

change 方法、save 方法都不会变成 Single Thread Execution 模式(第 1 章)，所以不会变成相同操作。即使设为 volatile，也不会进行线程的共享互斥。详情请参考附录 B “Java 的内存模式”。

×（3）change 方法的话，一定要先指定到 content 字段，再将 changed 设为 true。即使先将 changed 设为 true，还是可以正常执行。

○（4）doSave 方法不是 synchronized 方法。

但是，由于 doSave 方法只能从 synchronized 方法来调用，因此来到 doSave 方法的线程一定有 this 的 lock。

○（5）doSave 方法不会同时被两个线程所调用。

save 方法是 sychronized，private 的 doSave 方法只会被 save 方法调用。因此，doSave 方法不会同时被两个线程所调用。

问题 4-2 的解答

将修改过（1）与（2）的 Data 类分别显示在 List A4-1 与 List A4-2。

◆ *(1) 附加上调试输出(Debug Print)*

save 方法当中 return 的部分是在 balk，因此我们将调试输出置于 balk 之前。

List A4-1 加上调试输出的 Data 类 (Data.java)

```
import java.io.IOException;
import java.io.FileWriter;
import java.io.Writer;

public class Data {
    private String filename;      // 保存的文件名称
    private String content;       // 数据属性
    private boolean changed;      // 如果变动属性没有被保存就是 true

    public Data(String filename, String content) {
        this.filename = filename;
        this.content = content;
        this.changed = true;
    }

    // 改写数据属性
    public synchronized void change(String newContent) {
        content = newContent;
        changed = true;
    }

    // 如果数据属性变动过，就保存到文件中
```

```
    public void save() throws IOException {   // not synchronized
        if (!changed) {
            System.out.println(Thread.currentThread().getName() + "balks");
            return;
        }
        doSave();
        changed = false;

    }

    // 数据的属性实例保存到文件中
    private void doSave() throws IOException {
        System.out.println(Thread.currentThread().getName() + "
calls doSave, content = " + content);
        Writer writer = new FileWriter(filename);
        writer.write(content);
        writer.close();
    }
}
```

```
SaverThread calls doSave, content = No.0
ChangerThread balks
ChangerThread calls doSave, content = No.1
ChangerThread calls doSave, content = No.2
SaverThread calls doSave, content = No.3
ChangerThread balks
ChangerThread calls doSave, content = No.4
ChangerThread balks
ChangerThread calls doSave, content = No.5
SaverThread calls doSave, content = No.6
ChangerThread balks
(中略)
ChangerThread balks
SaverThread calls doSave, content = No.78
ChangerThread balks
ChangerThread calls doSave, content = No.79
SaverThread calls doSave, content = No.80
ChangerThread balks
SaverThread balks
ChangerThread calls doSave, content = No.81
SaverThread calls doSave, content = No.82
ChangerThread balks
(以下省略。按CTRL+C结束)
```

图 A4-1　执行范例

◆ (2) synchronized 的删除

删除 synchronized，在将 changed 标识设为 false 之前，先 sleep 约 100 毫秒。这样的话，就会如执行范例（图 A4-2）一般被重复写入。

List A4-2 加上侦错打印，将 synchronized 删除的 Data 类 (Data.java)

```
import java.io.IOException;
import java.io.FileWriter;
import java.io.Writer;


public class Data {
    private String filename;     // 保存文件的名称
    private String content;      // 数据的属性
    private boolean changed;     // 如果更改的属性没有被保存的话就是 true

    public Data(String filename, String content) {
        this.filename = filename;
        this.content = content;
        this.changed = true;
    }

    // 改写数据属性
    public synchronized void change(String newContent) {
        content = newContent;
        changed = true;
    }

    // 数据一旦更改就要保存起来
    public synchronized void save() throws IOException {  // not
synchronized
        if (!changed) {

            System.out.println(Thread.currentThread().getName()
+ " balks");
            return;
        }
        doSave();
         try {
             Thread.sleep.sleep(100);
         } catch (InterruptedException e) {
         }
```

```
            changed = false;
        }

        // 将数据属性实例保存到文件中
        private void doSave() throws IOException {
            System.out.println(Thread.currentThread().getName() + "
    calls doSave, content = " + content);
            Writer writer = new FileWriter(filename);
            writer.write(content);
            writer.close();
        }
    }
```

```
SaverThread calls doSave, content = No.0
ChangerThread balks
SaverThread calls doSave, content = No.1
ChangerThread balks
SaverThread calls doSave, content = No.2
ChangerThread calls doSave, content = No.2        ←将No.2写入两次
ChangerThread calls doSave, content = No.3
SaverThread calls doSave, content = No.4
ChangerThread balks
ChangerThread calls doSave, content = No.5
SaverThread calls doSave, content = No.6
ChangerThread balks
ChangerThread calls doSave, content = No.7
SaverThread calls doSave, content = No.8
ChangerThread balks
SaverThread calls doSave, content = No.9
ChangerThread calls doSave, content = No.9        ←将No.9写入两次
ChangerThread calls doSave, content = No.10
SaverThread calls doSave, content = No.10         ←将No.10写入两次
ChangerThread balks
SaverThread calls doSave, content = No.12
(以下省略。按CTRL+C结束)
```

图 A4-2　执行范例

问题 4-3 的解答

如图 A4-3 所示。在此，我们将 changed 字段变成 true 的范围以灰底显示。Changed

方法在执行途中就开始出现灰底，doSave 方法执行后灰底的部分也结束了。

从图中我们可以看到，changed 字段为 true 的这段范围（灰底部分）中，如果 save 方法被调用，doSave 也会被调用。false 的这段范围（白色部分）中，调用 save 并不会让 doSave 方法也连带被调用。

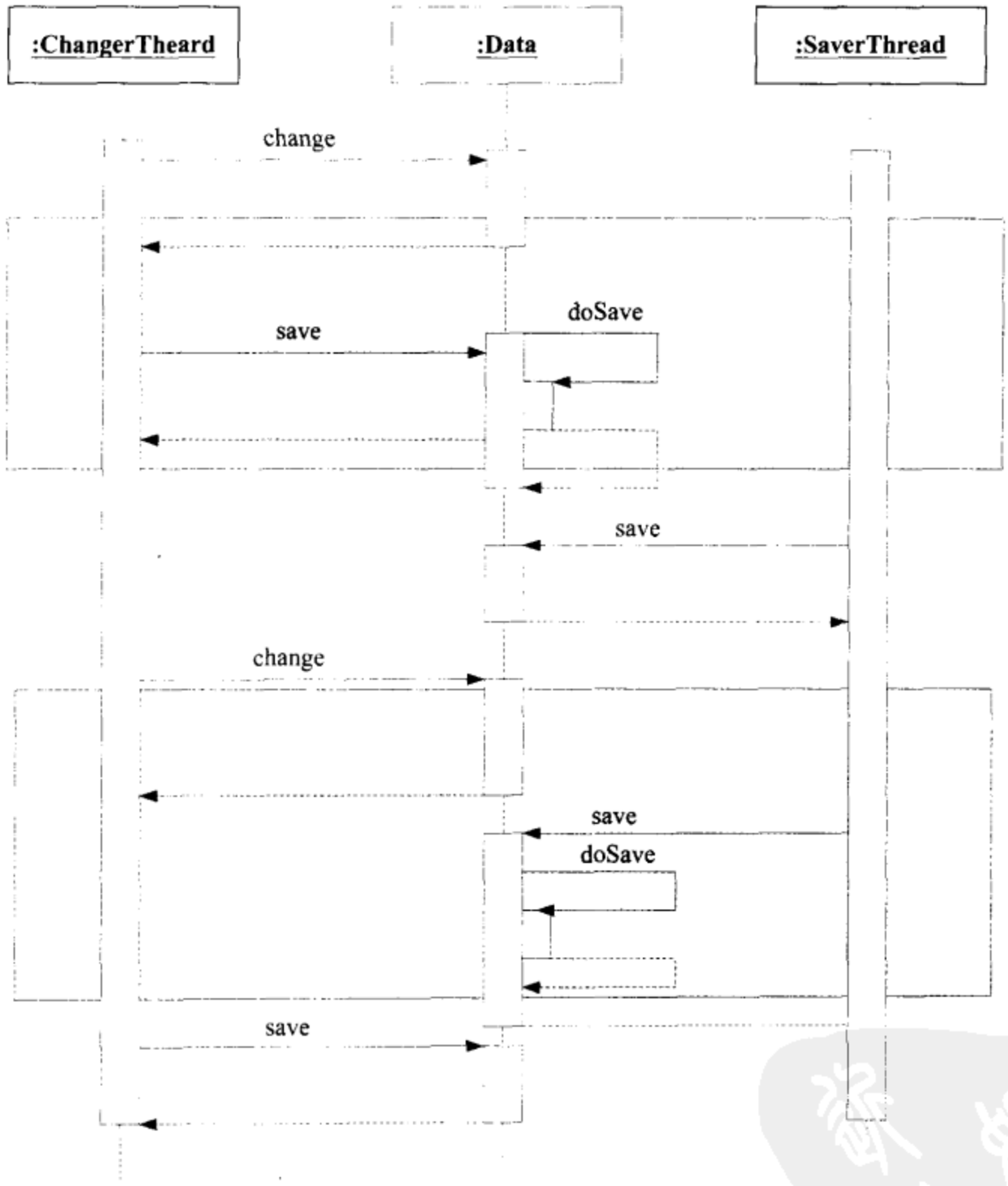


图 A4-3　显示 changed 字段为 true 的范围的流程图

问题 4-4 的解答

修改过的 RequestQueue 类就如 List A4-3 一样。

在这里，我们的程序代码是要在进入 while 循环后“到 Guard 条件被满足为止的休息时间（timeout）约为 30 秒”，而不是要让 while 循环内“每次 wait 的休息时

间（timeout）约为 30 秒”。

约 30 秒，将异常 LivenessExceptionm（见 List 4-9）给 throw 出去。

List A4-3 RequestQueue 类 (RequestQueue.java)

```
import java.util.LinkedList;

public class RequestQueue {
    private static final long TIMEOUT = 30000;
    private final LinkedList queue = new LinkedList();
    public synchronized Request getRequest() {
        long start = System.currentTimeMillis();// 开始时间
        while (queue.size() <= 0) {
            long now = System.currentTimeMillis();// 现在时间
            long rest = TIMEOUT - (now - start); // 剩下的等待时间

            if (rest <= 0) {
                throw new LivenessException("thrown by " +
Thread.currentThread().getName());
            }
            try {
                wait(rest);
            } catch (InterruptedException e) {
            }
        }
        return (Request)queue.removeFirst();
    }
    public synchronized void putRequest(Request request) {
        queue.addLast(request);
        notifyAll();
    }
}
```

如果我们将发生死锁的 Guard Suspension 模式的 RequestQueue 类改成上述的版本，再执行操作试试看，就会得到如图 A4-4 所示的结果。

执行后马上出现

Alice:BEGIN

Bobby:BEGIN

然后就停住了，就这样等了约 30 秒后，Alice 与 Bobby 会分别抛出 LivenessException。看来 timeout 的检查运行得十分顺利。

```
Alice:BEGIN
Bobby:BEGIN                                    ←在显示这段话之后，等待约 30 秒
LivenessException: thrown by Alice    ←Alice 丢出的 LivenessException
        at RequestQueue.getRequest(RequestQueue.java:12)
        at TalkThread.run(TalkThread.java:13)
LivenessException: thrown by Bobby    ←Bobby 丢出的 LivenessException
        at RequestQueue.getRequest(RequestQueue.java:12)
        at TalkThread.run(TalkThread.java:13)
```

图 A4-4　Dead Lock 的检查范例

问题 4-5 的解答

这是因为在 Main 类的主线程中，对 TestThread 调用多次 start 方法的缘故。对一个实例，start 方法只会调用一次。

Thread 类（以及其子类）的实例，一被调用 start 方法，就会变成“结束 start”状态。

start 方法被再次调用出来的时候，就会进行 balk 让线程的启动不会再次执行，并且抛出异常 IllegalThreadStateException。

Thread 类的 start 方法就会变成如同本章 P.125 所说的“不能执行两次以上”。也就是说，Thread 类的 start 方法中使用了 Balking 模式（Pattern）。

在执行范例（图 4-6）中，只出现一次的 BEGIN...END，是由一开始 start 时启动的线程来显示。

第 5 章

问题 5-1 的解答

○（1）MakerThread 类的构造器中所写的 super(name)调用出 Thread 类的构造器。

×（2）MakerThread 类的 nextId 方法变成 Synchronized，是由于 EaterThread 类也在调用之故。

MakerThread 类的 nextId 方法变成 Synchronized，是由于有两个以上的 MakerThread 同时都在调用它。而 nextID 并没有从 EaterThread 类则进行调用。

×（3）在桌上没有放蛋糕时调用 take，线程就会去取得 Table 的实例的 lock，而进行 block。

线程并不会“取得 Table 的实例的 lock，而进行 block”，而是在 Guard 条件未满足前保持 wait。

○（4）如果桌上没有任何蛋糕时，count 字段的值就是 0。

×（5）桌上放满了蛋糕，再也放不下时，count 字段的值等于 buffer.length-1。桌上放满蛋糕时，count 的值为 buffer.length，而不是 buffer.length-1。

○（6）head 字段的值不会超过 buffer.length。

问题 5-2 的解答

MakerThread 的 Alice 与 EaterThread 的 Bobby 收到与 Table 类相异的实例。也就是说，这个线程中并没有共享 table。

Alice 做了 3 个蛋糕（No.0, No.1, No.2）放在桌上，正在 wait 放上第 4 个（No.3）蛋糕。但是，没有任何线程从 Alice 放置蛋糕的桌子取走蛋糕。因此，Alice 就永远 wait 下去。我们另外假设，Bobby 从另一个桌子拿蛋糕。但是那个桌上什么都没有，所以 Bobby 也在 wait。但是也没有任何线程放蛋糕到 Bobby 要拿蛋糕的这个桌子上，所以 Bobby 也会一直 wait 下去。

而这样就是图 5-5 的执行结果。

问题 5-3 的解答

非多线程的程序中，以 take 方法为例，只要持续“一有蛋糕就拿走”的处理方式，就没有问题。但是，多线程的程序中，很有可能有两个以上的线程会来调用 take 方法，所以就没这么简单了。

take 方法所采取的“一有蛋糕就拿走”的处理方式可分成两个阶段：

（1）判断“有蛋糕”

（2）执行“拿走”的处理

假设我们没有将 take 方法设为 synchronized，这样的话，在线程 A 执行了（1）后尚未执行（2）之前，很有可能会有其他的线程 B 会插进来执行（1）或（2）。也就是说，在判断“有蛋糕”到“拿走”之间，很有可能被别的线程抢先执行了“一有蛋糕就拿走”的处理。如果桌上的蛋糕已经是最后一个了，线程 A 就无法取得原本认为应能取得的蛋糕了。

take 方法设为 synchronized 的话，能够执行 take 的线程一次只有一个，就不会发生如上其他线程介入的情况。

因此才会将 take 方法设为 synchronized。将 put 方法设为 synchronized 也是基于相同的理由。

问题 5-4 的解答

如 List A5-1 所示。

List A5-1 增加侦错显示的 Table 类(Table.java)

```
public class Table {
    private final String[] buffer;
    private int tail;   // 下一个要 put 的位置
    private int head;   // 下一个要 take 的位置
    private int count;  // buffer 内的蛋糕数
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
    public synchronized void put(String cake) throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + "
puts " + cake);
        while (count >= buffer.length) {
            System.out.println(Thread.currentThread().getName()
+ " wait BEGIN");
            wait();
            System.out.println(Thread.currentThread().getName()
+ " wait END");
        }
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notifyAll();
    }
    public synchronized String take() throws InterruptedException
{
        while (count <= 0) {
            System.out.println(Thread.currentThread().getName()
+ " wait BEGIN");
            wait();
            System.out.println(Thread.currentThread().getName()
+ " wait END");
        }
        String cake = buffer[head];
        head = (head + 1) % buffer.length;
        count--;
        notifyAll();
        System.out.println(Thread.currentThread().getName() + "
```

```
takes " + cake);
        return cake;
    }
}
```

执行范例如图 A5-1 所示。

问题 5-5 的解答

例如会变成 List A5-2。

Head、tail、count 的各个字段值设为 0，进行 notifyAll。

忘了 notifyAll 时，put 方法中如果有线程在等待“桌子是空的”，也就是说蛋糕都拿光了，这个线程就会一直等 wait 下去。

List A5-2 中，buffer 属性保持原状，但是也可以如下将 null 指定进去。

```
for (int i = 0; i < buffer.length; i++) {
  buffer[i] = null;
}
```

我们建立 ClearThread 类（List A5-3）以作为使用 create 方法的范例。ClearThread 约每隔一秒就调用 clear。

```
(前略)
MakerThread-1 puts [ Cake No.8 by MakerThread-1 ]
MakerThread-2 puts [ Cake No.8 by MakerThread-2 ]
EaterThread-3 Takes [ Cake No.8 by MakerThread-1 ]
EaterThread-2 Takes [ Cake No.9 by MakerThread-2 ]
EaterThread-3 wait BEGIN   ←桌子是空的，因此 EaterThread-3 在等待
EaterThread-1 wait BEGIN   ←桌子是空的，因此 EaterThread-1 也在等待
MakerThread-3 puts [ Cake No.10 by MakerThread-3 ]
                           ←MakerThread-3 将蛋糕放在这里(进行 notifyAll)
EaterThread-3 wait END     ←EaterThread-3 取得 lock 再往前进
EaterThread-3 take3 [ Cake No.10 by MakerThread-3 ]
                           ←取得蛋糕
EaterThread-1 wait END     ←EaterThread-1 也取得 lock 再往前进
EaterThread-1 wait BEGIN   ←但是桌子是空的，因此再次等待
MakerThread-1 puts [ Cake No.11 by MakerThread-1 ]
                           ←MakerThread-1 将蛋糕放在这里(进行 notifyAll)
EaterThread-1 wait END ←EaterThread-1 取得 lock 再往前进
EaterThread-1 takes [ Cake No.11 by MakerThread-1     ]
                           ←取得蛋糕
EaterThread-2 wait BEGIN   ←桌子是空的，因此 EaterThread-2 在等待
EaterThread-1 wait BEGIN   ←桌子是空的，因此 EaterThread-1 在等待
```

（续上图）

```
MakerThread-2 puts [ Cakes No.12 by MakerThread-2 ]
                              ←MakerThread-2 将蛋糕放在这里(进行 notifyAll)
EaterThread-2 wait END        ←EaterThread-2 取得 lock 再往前进
EaterThread-2 takes [ Cake No.12 by MakerThread-2 ]
                              ←取得蛋糕
(按 CTRL+C 结束)
```

图 A5-1　执行范例

List A5-2　新增了 clear 方法的 Table 类(Table.java)

```
public class Table {
    private final String[] buffer;
    private int tail;   //下一次要 put 的位置
    private int head;   //下一次要 take 的位置
    private int count;  //buffer 内的蛋糕数
    public Table(int count) {
        this.buffer = new String[count];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
    }
    public synchronized void clear()  {
        //这段 while 语句只是为了表示被取走的蛋糕，不写也没关系
        while (count > 0) {
            String cake = buffer[head];
            System.out.println(Thread.currentThread().getName()
+ " clears " + cake);
            head = (head + 1) % buffer.length;
            count--;
        }
        head = 0;
        tail = 0;
        count = 0;
        notifyAll();
    }
    public   synchronized   void   put(String   cake)   throws
InterruptedException {
        System.out.println(Thread.currentThread().getName() + "
puts " + cake);
        while (count >= buffer.length) {
            wait();
        }
```

```
        buffer[tail] = cake;
        tail = (tail + 1) % buffer.length;
        count++;
        notifyAll();
    }
    public synchronized String take() throws InterruptedException {
        while (count <= 0) {
            wait();
        }
        String cake = buffer[head];
        head = (head + 1) % buffer.length;
        count--;
        notifyAll();
        System.out.println(Thread.currentThread().getName() + "
takes " + cake);
        return cake;
    }
}
```

List A5-3 约每隔 1 秒就执行 clear 的清扫人员，ClearThread 类(ClearThread.java)

```
public class ClearThread extends Thread {
    private final Table table;
    public ClearThread(String name, Table table) {
        super(name);
        this.table = table;

    }
    public void run() {
        try {
            while (true) {
                Thread.sleep(1000);
                System.out.println("===== " + getName() + " clears =====");
                table.clear();
            }
        } catch (InterruptedException e) {
        }
    }
}
```

List A5-4 厨师、客人与清扫人员的 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);   //建立一个能放三个蛋糕的桌子
        new MakerThread("MakerThread-1", table, 31415).start();
        new MakerThread("MakerThread-2", table, 92653).start();
        new MakerThread("MakerThread-3", table, 58979).start();
        new EaterThread("EaterThread-1", table, 32384).start();

        new EaterThread("EaterThread-2", table, 62643).start();
        new EaterThread("EaterThread-3", table, 38327).start();
        new ClearThread("ClearThread-0", table).start();
    }
}
```

```
(前略)
EaterThread-1 takes [ Cake No.95 by MakerThread-3 ]
MakerThread-2 puts [ Cake No.97 by MakerThread-2 ]
EaterThread-1 takes [ Cake No.96 by MakerThread-1 ]
MakerThread-3 puts [ Cake No.98 by MakerThread-3 ]
EaterThread-2 takes [ Cake No.97 by MakerThread-2 ]
=====ClearThread-0 clears =====   ←在这里将桌子清掉(之前放了蛋糕No.98)
ClearThread-0 clears [ Cake No.98 by MakerThread-3 ]
MakerThread-3 puts [ Cake No.99 by MakerThread-3 ]
EaterThread-3 takes [ Cake No.99 by MakerThread-3 ]
MakerThread-2 puts [ Cake No.100 by MakerThread-2 ]
MakerThread-1 puts [ Cake No.101 by MakerThread-1 ]
EaterThread-1 takes [ Cake No.100 by MakerThread-2 ]
MakerThread-2 puts [ Cake No.102 by MakerThread-2 ]
MakerThread-3 puts [ Cake No.103 by MakerThread-3 ]
EaterThread-3 takes [ Cake No.101 by MakerThread-1 ]
EaterThread-2 takes [ Cake No.102 by MakerThread-2 ]
EaterThread-2 takes [ Cake No.103 by MakerThread-3 ]
=====ClearThread-0 clears =====   ←在这里虽然想将桌子清掉，但是桌上没有放蛋糕
MakerThread-3 puts [ Cake No.104 by MakerThread-3 ]
EaterThread-1 takes [ Cake No.104 by MakerThread-3 ]
MakerThread-1 puts [ Cake No.105 by MakerThread-1 ]
MakerThread-3 puts [ Cake No.106 by MakerThread-3 ]
EaterThread-1 takes [ Cake No.105 by MakerThread-1 ]
MakerThread-3 puts [ Cake No.107 by MakerThread-3 ]
```

（续上图）

```
EaterThread-2 takes [ Cake No.106 by MakerThread-3 ]
MakerThread-1 puts [ Cake No.108 by MakerThread-1 ]
EaterThread-3 takes [ Cake No.107 by MakerThread-3 ]
MakerThread-2 puts [ Cake No.109 by MakerThread-2 ]
MakerThread-2 puts [ Cake No.110 by MakerThread-2 ]
EaterThread-2 takes [ Cake No.108 by MakerThread-1 ]
MakerThread-2 puts [ Cake No.111 by MakerThread-2 ]
=====ClearThread-0 clears =====   ← 在这里将桌子清掉(之前放有蛋糕 No.109,
   110,111)
ClearThread-0 Clears [ Cake No.109 by MakerThread-2 ]
ClearThread-0 Clears [ Cake No.110 by MakerThread-2 ]
ClearThread-0 Clears [ Cake No.111 by MakerThread-2 ]
MakerThread-3 puts [ Cake No.112 by MakerThread-3 ]
MakerThread-3 puts [ Cake No.113 by MakerThread-3 ]
EaterThread-1 takes [ Cake No.112 by MakerThread-3 ]
EaterThread-3 takes [ Cake No.113 by MakerThread-3 ]
MakerThread-1 puts [ Cake No.114 by MakerThread-1 ]
MakerThread-1 takes [ Cake No.114 by MakerThread-1 ]
(按 CTRL+C 结束)
```

图 A5-2　执行范例

问题 5-6 的解答

如 List A5-5 所示。这样会在使用 sleep 停顿约 10 秒后，对各个线程调用 interrupt 方法。

List A5-5　在约 10 秒后结束线程的 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3);   // 建立一个只能放三个蛋糕的桌子
        Thread[] threads = {
            new MakerThread("MakerThread-1", table, 31415),
            new MakerThread("MakerThread-2", table, 92653),
            new MakerThread("MakerThread-3", table, 58979),
            new EaterThread("EaterThread-1", table, 32384),
            new EaterThread("EaterThread-2", table, 62643),
            new EaterThread("EaterThread-3", table, 38327),
        };

        // 启动线程
```

```
        for (int i = 0; i < threads.length; i++) {
            threads[i].start();
        }

        // 约休息 10 秒
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
        }

        System.out.println("***** interrupt *****");

        // interrupt
        for (int i = 0; i < threads.length; i++) {
            threads[i].interrupt();
        }
    }
}
```

问题 5-7 的解答

我们加上 throws 这段，让 execute 方法可以抛出异常 InterruptedException，并在可以删除的地方加上

```
if (Thread.interrupted()) {
  throw new InterruptedException();
}
```

这样一来，现在正在执行操作的线程如果是 interrupt 状态，就会被抛出 InterruptedException。由于正在执行 Thread.interrupted，线程会变成非 interrupt 状态。

List A5-6 中的程序会开始执行 100 次大型的处理，并在大约 15 秒后取消。执行范例请看图 A5-3。只有确认以 Thread.interrupted 检查 interrupted 状态的时机，才会抛出 InterruptedException。并非一出现***** interrupt *****就会马上抛出 InterruptedException。

List A5-6 main 处理(Main.java)

```
public class Main {
    public static void main(String[] args) {
        // 执行 Host 的大型处理的线程
        Thread executor = new Thread() {
            public void run() {
                System.out.println("Host.execute BEGIN");
                try {
```

```
                Host.execute(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Host.execute END");
        }
    };

    // 启动
    executor.start();

    // 休息约 15 秒
    try {
        Thread.sleep(15000);
    } catch (InterruptedException e) {
    }

    // 执行取消
    System.out.println("***** interrupt *****");
    executor.interrupt();
  }
}
```

List A5-7 大型处理(Host.java)

```
public class Host {
    public static void execute(int count) throws InterruptedException {
        for (int i = 0; i < count; i++) {
            if (Thread.interrupted()) {
                throw new InterruptedException();
            }
            doHeavyJob();
        }
    }
    private static void doHeavyJob() {
        // 以下为
        // “不可取消的大型处理”的代用
        // (进行约 10 秒的循环)
        System.out.println("doHeavyJob BEGIN");
        long start = System.currentTimeMillis();
        while (start + 10000 > System.currentTimeMillis()) {
            // busy loop
        }
        System.out.println("doHeavyJob END");
    }
}
```

```
Host.execute BEGIN          ←开始执行 execute
doHeavyJob END              ←开始执行第一次大型处理 doHeavyJob
doHeavryJob END             ←在执行第一次之后约 10 秒即结束执行
doHeavryJob BEGIN           ←开始执行第二次大型处理 doHeavyJob
***** interrupt *****       ←在此主线程执行 interrupt
doHeavyJob END              ←在第二次执行之后约 10 秒即结束执行
java.lang.InterruptedException    ←在 catch 中显示调用堆栈(call stack)
      at.Host.execute(Host.java:5)
      at Main$1.run(Main.java:8)
Host.execute END            ←execute 执行结束
```

图 A5-3　执行范例

问题 5-8 的解答

notify 方法只会从在 waitset 中等待的线程中调用一个。因此，无关的线程进入 waitset 时若正好碰上 notify 引发该线程的话，notify 所要进行的“通知”就会不见了。

在下面介绍的程序就可以证明这一点。

List A5-8 的 LazyThread 类虽然在 Table 的实例上进行 wait，但是实际上却是什么都不做的线程。List A5-9 的 Main 类内混进 LazyThread 的线程并予以执行的话，就会如 A5-3 的执行范例（图 5-4）一样，程序执行到一半会停下来。

这是由于为了调用 LazyThread，而随便滥用 notify 方法的调用的原因。Table 类中，如果我们使用的是 notifyAll 而非 notify 方法的话，即使其中参杂了 LazyThread 的线程，程序也不会停止而会继续执行操作。

List A5-8　什么都不做的线程(LazyThread.java)

```
public class LazyThread extends Thread {
    private final Table table;
    public LazyThread(String name, Table table) {
        super(name);
        this.table = table;
    }
    public void run() {
        while (true) {
            try {
                synchronized (table) {
                    table.wait();
                }
```

```
                System.out.println(getName() + " is notified!");
            } catch (InterruptedException e) {
            }
        }
    }
}
```

List A5-9 将什么都不做的线程也混进来，移动厨师与客人的 Main 类(Main.java)

```
public class Main {
    public static void main(String[] args) {
        Table table = new Table(3); // 建立一个只能放 3 个蛋糕的桌子
        new MakerThread("MakerThread-1", table, 31415).start();
        new MakerThread("MakerThread-2", table, 92653).start();
        new MakerThread("MakerThread-3", table, 58979).start();
        new EaterThread("EaterThread-1", table, 32384).start();
        new EaterThread("EaterThread-2", table, 62643).start();
        new EaterThread("EaterThread-3", table, 38327).start();
        new LazyThread("LazyThread-1", table).start();
        new LazyThread("LazyThread-2", table).start();
        new LazyThread("LazyThread-3", table).start();
        new LazyThread("LazyThread-4", table).start();
        new LazyThread("LazyThread-5", table).start();
        new LazyThread("LazyThread-6", table).start();
        new LazyThread("LazyThread-7", table).start();
    }
}
```

```
(前略)
LazyThread-5 is notified!
EaterThread-3 takes [ Cake No.25 by MakerThread-2 ]
LazyThread-6 is notified!
MakerThread-2 puts [ Cake No.26 by MakerThread-2 ]
LazyThread-7 is notified!
EaterThread-3 takes [ Cake No.26 MakerThread-2 ]
LazyThread-1 is notified!
MakerThread-2 puts [ Cake No.27 by MakerThread-2 ]
LazyThread-2 is notified!
MakerThread-2 puts [ Cake No.28 by MakerThread-2 ]
MakerThread-2 puts [ Cake No.29 by MakerThread-2 ]
MakerThread-3 puts [ Cake No.30 by MakerThread-3 ]
                                        ←在进行到一半时会停下来
(按 CTRL+C 结束)
```

图 A5-4 执行范例

问题 5-9 的解答

something.method(long)与 Thread.sleep(long)相等，也就是说，暂停的时间等于参数所指定的时间。

以参数 0 调用 sleep 时，会暂时停止 0 毫秒，而以参数 0 调用出 wait 后，又会变成无限长的休息时间(Time Out)，因此用 if 语句将 0 的状况排除在外。

此外，为了 wait 不会被 notify 或 notifyAll 给打断，我们在方法中建立取得 lock 的实例。由于没有从外部获得这个实例的方法，因此没有办法 notify 或 notifyAll。

List A5-10 中介绍的程序会使用 Something.method 休息约 3 秒。执行结果如图 A5-5 所示。

此问题参考了附录 E 的“Lea”。

List A5-10 试着动动看(Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("BEGIN");
        try {
            Something.method(3000);
        } catch (InterruptedException e) {
        }
        System.out.println("END");
    }
}
```

```
BEGIN                ← 马上显示
END                  ← 约三秒后显示
```

图 A5-5 执行结果

第 6 章

问题 6-1 的解答

○（1）doWrite 方法不会同时被数个线程来执行。

×（2）doRead 方法不会同时被数个线程来执行。

doRead 方法有时会由数个 ReaderThread 的线程同时执行。

○（3）某个线程正在执行 doWrite 方法时，readingReaders 字段的值一定是 0。在撰写时，不会有线程同时正在进行读取。

○（4）某个线程正在执行 doRead 方法时，writingWriters 字段的值一定是 0。当在读取处理时，不会有线程同时正在进行撰写。

问题 6-2 的解答

ReaderThread 的线程也好，WriterThread 的线程也好，完全都不会进行共享互斥。因此，就会像图 A6-1 这样，还在撰写文字时就读取，或是读取到数个 WriterThread 正在撰写的状态。前者是 read-write conflict，后者是 write write conflict 与 read-write conflict 两方。

Data 参与者的安全性已然失去。

```
(前略)
Thread-1 reads aaaa*****   ←读取正在写入 a 的状态(read-write
conflict)
Thread-2 reads aaaa*****
Thread-0 reads aaaa*****
Thread-3 reads aaaa*****
Thread-4 reads aaaa*****
Thread-5 reads aaaa*****
Thread-0 reads aaaa*****
Thread-1 reads aaaa*****
(中略)
Thread-0 reads hhhhFFgggg  ←在撰写 F 的时候，读取撰写 h 的状态
Thread-3 reads hhhhFFgggg    (write-write 与 read-write conflict)
Thread-4 reads hhhhFFgggg
Thread-5 reads hhhhFFgggg
Thread-2 reads hhhhFFgggg
(以下省略。按 CTRL+C 结束)
```

图 A6-1　执行范例

问题 6-3 的解答

为了比较 List 6-2 与 List 6-7 中所花的时间，我们将 ReaderThread 类改成如 List A6-1 这样。ReaderThread 的线程在执行 20 次 read 的调用的前后，取得现在的时间。

执行范例各自如图 A6-2、图 A6-3 所示。我们可以发现，使用 ReadWriteLock 类的话，只要大约两秒执行就可以结束，但是使用 sychronized 却要大约 9～10

秒。但是，这些值会受到在 doRead 或 doWrite 所花的时间，以及 ReaderThread 与 WriterThread 的个数所影响而变动。此外，跟 Java 执行处理系统的安装也有关，这点请注意。

List A6-1 经修改以进行计时的 ReaderThread 类 (ReaderThread.java)

```
public class ReaderThread extends Thread {
    private final Data data;
    public ReaderThread(Data data) {
        this.data = data;
    }
    public void run() {
        try {
            long begin = System.currentTimeMillis();
            for (int i = 0; i < 20; i++) {
                char[] readbuf = data.read();
                System.out.println(Thread.currentThread().getName() +
" reads " + String.valueOf(readbuf));
            }
            long time = System.currentTimeMillis() - begin;
            System.out.println(Thread.currentThread().getName()
+ ": time = " + time);
        } catch (InterruptedException e) {
        }
    }
}
```

```
(前略)
Thread-0 reads aaaaaaaaaa
Thread-4 reads aaaaaaaaaa
Thread-2 reads aaaaaaaaaa
Thread-3 reads aaaaaaaaaa
Thread-3: time = 2043
Thread-1 reads aaaaaaaaaa
Thread-1: time = 2053
Thread-5 reads aaaaaaaaaa
Thread-5: time = 2053
Thread-0 reads aaaaaaaaaa
Thread-0: time = 2063
Thread-4 reads aaaaaaaaaa
Thread-2 reads aaaaaaaaaa
Thread-2: time = 2063
Thread-4: time = 2063          ←在中途终止
(按 CTRL+C 结束)
```

图 A6-2　执行范例（使用 ReadWriteLock 类时）

```
(前略)
Thread-4 reads ccccccccccc
Thread-5 reads ccccccccccc
Thread-0 reads ccccccccccc
Thread-1 reads ccccccccccc
Thread-2 reads ddddddddddd
Thread-3 reads ddddddddddd
Thread-4 reads ddddddddddd
Thread-5 reads ddddddddddd
Thread-0 reads ddddddddddd
Thread-0: time = 9423
Thread-1 reads ddddddddddd
Thread-1: time = 9473
Thread-2 reads DDDDDDDDDD
Thread-2: time = 10024
Thread-3 reads DDDDDDDDDD
Thread-3: time = 10074
Thread-4 reads DDDDDDDDDD
Thread-4: time = 10134
Thread-5 reads DDDDDDDDDD
Thread-5: time = 10184          ←在中途终止
(按 CTRL+C 结束)
```

图 A6-3　执行范例（使用 sychronized 而不使用 ReadWriteLock 类时）

问题 6-4 的解答

以下只是方法之一。例如，在 Doug Lea 的 util.concurrent 包中，使用 Template Method Pattern 以提高重复使用性（参照附录 E[LeaWeb]的 WriterPreferenceReadWriteLock 类）。

修改后的程序如 List A6-2~ List A6-7。图 A6-4 是类的图示。

为了解决（1）与（2），我们使用 GoF 的 Strategy Pattern（请参考附录 E[GoF][Yuki01]）。

关于读写处理的 policy 我们以 ReadWriteStrategy 表示，Guard 条件相关的 policy 则以 GuardStrategy 接口表示。之后，在建立 Data 类的实例时，设置表示各自 policy 的 ConcreteStrategy 参与者，在执行时将处理交给对方处理。如果 policy 未被设置，则默认的 policy 如 DefaultReadWriteStrategy 类、DefaultGuardStategy 类会成为 Data 类的内部类（inner class）。在 ListA6-4 中已将 Default Policy 加入范例程序当中了。

为了解决（3），我们将 ReadWriteLock 类当成 Data 类的内部类。

我们还将 ReadWriteStrategy 接口与 GuardStragy 接口与 Data 类集成到 readwritelock 包中。

只要先走这一步，就可以切换读写处理与 Guard 条件的 policy。但是，这个方法也会有使得类或接口增加，导致管理困难，用户自己都难以了解其中结构的缺点。

自己设计的程序要有多优秀的重复使用性？以及可以接受因而增加的类或接口的数量是多少？设计者必须要自己权衡轻重得失。

List A6-2 GuardStrategy 接口(GuardStrategy.java)

```
package readwritelock;

public interface GuardStrategy {
    public abstract void beforeReadWait();
    public abstract boolean readGuard();
    public abstract void afterReadWait();
    public abstract void beforeDoRead();
    public abstract void afterDoRead();

    public abstract void beforeWriteWait();
    public abstract boolean writeGuard();
    public abstract void afterWriteWait();
    public abstract void beforeDoWrite();
    public abstract void afterDoWrite();
}
```

List A6-3 ReadWriteStrategy 接口(ReadWriteStrategy.java)

```
package readwritelock;

public interface ReadWriteStrategy {
    public abstract Object doRead() throws InterruptedException;
    public abstract void doWrite(Object arg) throws
InterruptedException;
}
```

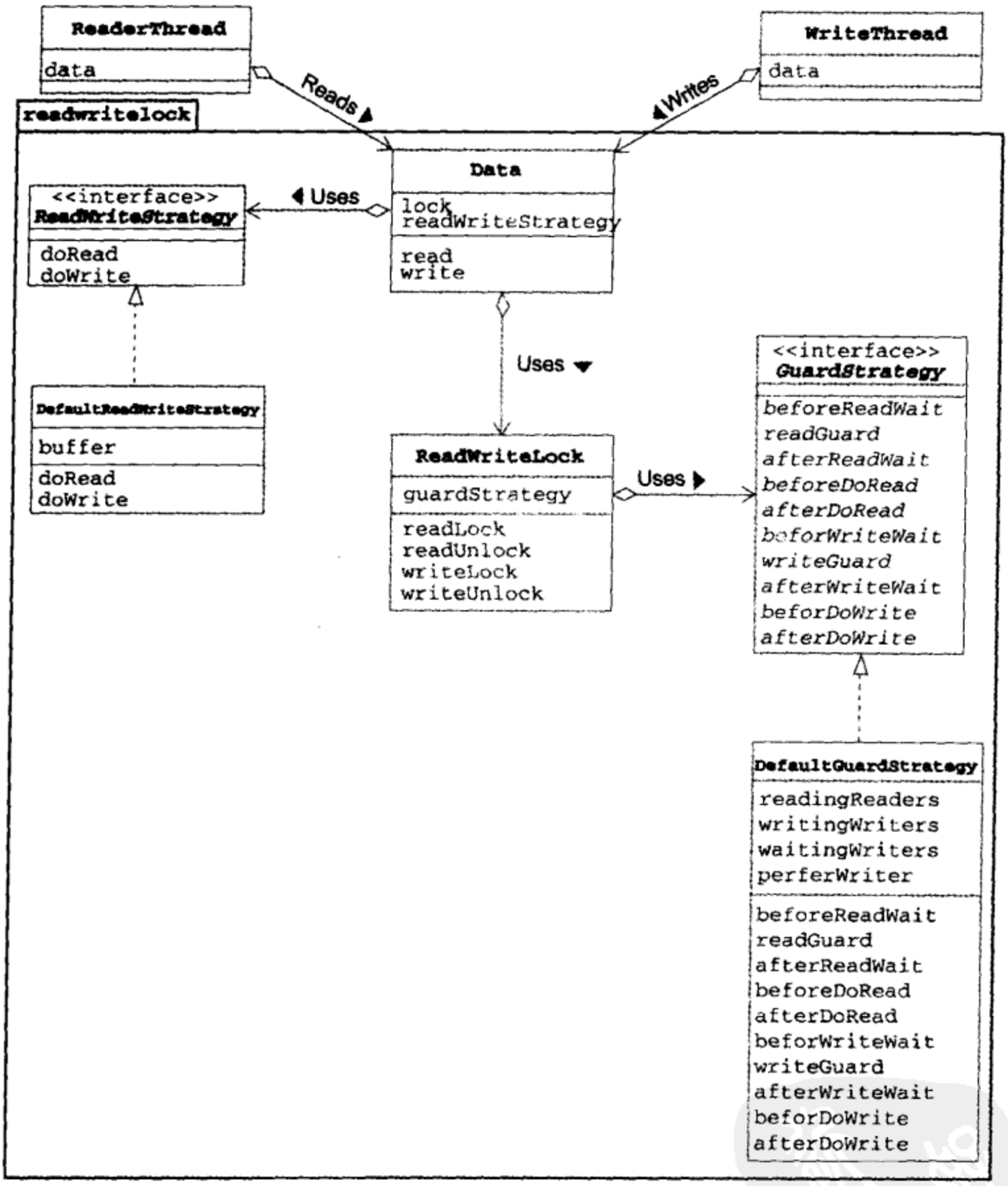


图 A6-4　类的图示

List A6-4　Data 类(Data.java)

```
package readwritelock;

public class Data {
    private final ReadWriteLock lock;
    private final ReadWriteStrategy readWriteStrategy;
```

```
    public Data() {
        this.lock = new ReadWriteLock(new DefaultGuardStrategy());
        this.readWriteStrategy = new DefaultReadWriteStrategy();
    }

    public Data(GuardStrategy guardStrategy) {
        this.lock = new ReadWriteLock(guardStrategy);
        this.readWriteStrategy = new DefaultReadWriteStrategy();
    }
    public Data(ReadWriteStrategy readWriteStrategy) {
        this.lock = new ReadWriteLock(new DefaultGuardStrategy());
        this.readWriteStrategy = readWriteStrategy;
    }
    public Data(GuardStrategy guardStrategy, ReadWriteStrategy
readWriteStrategy) {
        this.lock = new ReadWriteLock(guardStrategy);
        this.readWriteStrategy = readWriteStrategy;
    }
    public Object read() throws InterruptedException {
        lock.readLock();
        try {
            return readWriteStrategy.doRead();
        } finally {
            lock.readUnlock();
        }
    }
    public void write(Object arg) throws InterruptedException {
        lock.writeLock();
        try {
            readWriteStrategy.doWrite(arg);
        } finally {
            lock.writeUnlock();
        }
    }

    // Inner class

    private class ReadWriteLock {
        private final GuardStrategy guardStrategy;
        public ReadWriteLock(GuardStrategy guardStrategy) {
            this.guardStrategy = guardStrategy;
```

```
}
public synchronized void readLock() throws InterruptedException {
    guardStrategy.beforeReadWait();
    try {
        while (!guardStrategy.readGuard()) {
            wait();
        }
    } finally {
        guardStrategy.afterReadWait();
    }
    guardStrategy.beforeDoRead();
}
public synchronized void readUnlock() {
    guardStrategy.afterDoRead();
    notifyAll();
}
public synchronized void writeLock() throws InterruptedException {
    guardStrategy.beforeWriteWait();
    try {
        while (!guardStrategy.writeGuard()) {
            wait();
        }
    } finally {
        guardStrategy.afterWriteWait();
    }
    guardStrategy.beforeDoWrite();
}
public synchronized void writeUnlock() {
    guardStrategy.afterDoWrite();
    notifyAll();
}
}

// Inner class

private class DefaultGuardStrategy implements GuardStrategy
{
    private int readingReaders = 0; // 实际上读取的线程数目
    private int writingWriters = 0; // 实际上撰写中的线程数目
    private int waitingWriters = 0; // 正在等待撰写的线程数目
    private boolean preferWriter = true; // 如果优先撰写，就是 true
    public void beforeReadWait() {
```

```
            // no operation
        }
        public boolean readGuard() {
            return !(writingWriters > 0 || (preferWriter &&
waitingWriters > 0));
        }
        public void afterReadWait() {
            // no operation
        }
        public void beforeDoRead() {
            readingReaders++;
        }
        public void afterDoRead() {
            readingReaders--;
            preferWriter = true;
        }
        public void beforeWriteWait() {
            waitingWriters++;
        }
        public boolean writeGuard() {
            return !(readingReaders > 0 || writingWriters > 0);
        }
        public void afterWriteWait() {
            waitingWriters--;
        }
        public void beforeDoWrite() {
            writingWriters++;
        }
        public void afterDoWrite() {
            writingWriters--;
            preferWriter = false;
        }
}

// Inner class

private class DefaultReadWriteStrategy implements ReadWriteStrategy{
    private final char[] buffer;
    public DefaultReadWriteStrategy() {
        this(10);
```

```
}
public DefaultReadWriteStrategy(int size) {
    this.buffer = new char[size];
    for (int i = 0; i < buffer.length; i++) {
        buffer[i] = '*';
    }
}
public Object doRead() throws InterruptedException {
    char[] newbuf = new char[buffer.length];
    for (int i = 0; i < buffer.length; i++) {
        newbuf[i] = buffer[i];
    }
    slowly();
    return newbuf;
}
public void doWrite(Object arg) throws InterruptedException {
    char c = ((Character)arg).charValue();
    for (int i = 0; i < buffer.length; i++) {
        buffer[i] = c;
        slowly();
    }
}
    private void slowly() throws InterruptedException {
        Thread.sleep(50);
    }
}
```

List A6-5 ReaderThread 接口(ReaderThread.java)

```
import readwritelock.Data;

public class ReaderThread extends Thread {
    private final Data data;
    public ReaderThread(Data data) {
        this.data = data;
    }
    public void run() {
        try {
            while (true) {
                char[] readbuf = (char[])data.read();
                System.out.println(Thread.currentThread().getName()
```

```
+ " reads " + String.valueOf(readbuf));
            }
        } catch (InterruptedException e) {
        }
    }
}
```

List A6-6 WriteThread 接口(WriteThread.java)

```
import readwritelock.Data;
import java.util.Random;

public class WriterThread extends Thread {
    private static final Random random = new Random();
    private final Data data;
    private final String filler;
    private int index = 0;
    public WriterThread(Data data, String filler) {
        this.data = data;
        this.filler = filler;
    }
    public void run() {
        try {
            while (true) {
                Character c = nextchar();
                data.write(c);
                Thread.sleep(random.nextInt(3000));
            }
        } catch (InterruptedException e) {
        }
    }
    private Character nextchar() {
        char c = filler.charAt(index);
        index++;
        if (index >= filler.length()) {
            index = 0;
        }
        return new Character(c);
    }
}
```

List A6-7 Main 类(Main.java)

```
import readwritelock.Data;

public class Main {
    public static void main(String[] args) {
        Data data = new Data();
        new ReaderThread(data).start();

        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new ReaderThread(data).start();
        new WriterThread(data, "ABCDEFGHIJKLMNOPQRSTUVWXYZ").start();
        new WriterThread(data, "abcdefghijklmnopqrstuvwxyz").start();
    }
}
```

问题 6-5 的解答

如果是在范例程序中的话，可以正确执行无误。

但是，如果线程被 interrupt，很可能会调用过多次 readUnlock 或 writeUnlock。

现在，假设线程在 lock.readLock()中正在 wait。这个线程被 interrupt 的话，异常 InterruptException 就会被抛出，而从 readLock 方法跳脱出来。这时，readingReaders 字段并不会增加。

从 readLock 跳脱出来的线程会跳到 finally 那段，执行 lock.readUnlock()。在这当中，之前没有增加的 readingReaders 会被减少，readingReaders 字段的值会变得太小。

处理 InterruptedException 是为了让程序的响应更好而必要进行的处理。但是稍微的修改恐怕会让程序的安全性发生问题，因此需要特别注意。

我们再来看一次题目中所示 Before/After pattern 的架构。

```
before();
try {
    execute();
} finally {
    after();
}
```

在此，before 在 try 之外，表示“如果在 before 的执行过程中发生异常，就不执

行 execute 与 after”。例如，从 before 中抛出 InterruptedException，就可以想成是“before 的执行中断”。如果 before 放在 try 这段程序代码块中，即使中断 before 的执行，也会调用出 after。

问题 6-6 的解答

这个现象是因为 ReaderThread 的线程比 WriterThread 的线程数目还多所造成的。

即使有一个 ReaderThread 在进行 doRead，WriterThread 也会无法执行 doWrite。但是 ReaderThread 不会被设置为共享互斥，因此会一个个执行 doRead。到最后，WriterThread 就越来越无法执行 doWrite。这就好像马路上的车子川流不息，而无法穿越马路的意思是一样的。

范例程序的 ReadWriteLock 类（List 6-5）的 waitingWriter 字段会保持 wait 中的 WriterThread 的数目。当 waitingWriters > 0 成立时，通过让 ReaderThread 的线程 wait，就可以避免 WriterThread 无法开始执行的情况。这就像当有人等待过马路时，红绿灯自己就变成红灯让车辆停下来，行人可以安全通过一样。

但是，光是考虑 waitingWriters 的情况的话，下次可能变成 ReaderThread 无法开始执行 doRead。那种情况就是 1 个 WriterThread 在结束执行 doWrite 之前，其他的 WriterThread 已经到达了 write。只要有一个 WriterThread 正在 wait，ReaderThread 就无法开始执行 doRead，因此到最后，ReaderThread 就无法开始执行 doRead。这就好像一直有人要过马路，自按式交通信号灯一直有人按着，两旁的车子就一直等待红灯而无法前进一样。

范例程序的 ReadWriteLock 类（List 6-5）的 preferWriter 字段是决定 ReaderThread 与 WriterThread 何者优先的标识（Flag）。preferWriter 只要是 true，就开始要考虑 Guard 条件的 waitingWriters。然后，在 readUnlock 中（也就是 doRead 结束时）将 preferWriter 设置为 true。此外，在 writeUnlock 中（也就是 doWrite 结束时），将 preferWriter 设置为 false。结果，read 处理结束后优先进行 write 处理，在 write 处理结束后就优先处理 read。preferWriter 字段就是用来让 ReaderThread 与 WriterThread 轮流优先执行的。这就好比行人的信号量灯与行车的信号量灯轮流变成红灯是一样的。

问题 6-7 的解答

（1）错。

有时候 ReaderThread 与 WriterThread 两方的线程会在 this 上 wait。

（2）错。

有时候 ReaderThread 与 WriterThread 两方的线程会在 this 上 wait。例如，preferWriter && waitingWriters > 0 为 true 时，后面的 ReaderThread 的线程就必须等待。

第 7 章

问题 7-1 的解答

○（1）每次调用出 request 方法时，都会启动新的线程。
request 方法中，每次都会启动新的线程。

×（2）每次调用出 request 方法时，都会生成 Helper 类的实例。
Helper 类的实例只有在 Host 类的实例产生时才生成。

×（3）如果没有从 request 方法回来，就不会调用出 handle 方法。
request 方法中启动新的线程，该线程则又会调用 handle 方法。这跟一开始的线程是否从 request 方法返回无关。

×（4）调用出 handle 方法，显示出文字的是主线程。
调用出 handle 方法的，是在 request 方法中重新启动的线程，而非主线程。

×（5）slowly 方法中，延长 sleep 时间的话，从 request 方法回来的时间也会延长。
slowly 方法中，即使延长 sleep 时间，从 request 方法回来的时间也不变。执行 request 方法的线程并不会调用 slowly 方法。

问题 7-2 的解答

将 Host 类改成如 List A7-1 一样。我们设置为在 request 方法中会直接调用 handle 方法。Main 类、Helper 类并没有变更的必要。

List A7-1 更改后的 Host 类 (Host.java)

```
public class Host {
    private final Helper helper = new Helper();
    public void request(final int count, final char c) {
        System.out.println("    request(" + count + ", " + c + ")
BEGIN");
        helper.handle(count, c);
        System.out.println("    request(" + count + ", " + c + ")
END");
    }
}
```

执行结果如图 A7-1。此执行结果并不会因为时间不同而变化。不过，本文中所介绍的范例程序的执行范例（见图 7-2）则会随时间不同而变化。

我们可以看到由于一个 request 结束之后，下一个 request 才会开始，因此 A、B、C 的文字不会混在一起，以及所有的 request 结束之后，主线程才会结束。

```
main BEGIN
      request(10, A) BEGIN
          handle(10, A) BEGIN
AAAAAAAAAA
          handle(10, A) BEGIN
      request(10, A) END
      request(20, B) BEGIN ←一个 request 结束之后，下一个 request 才会开始
          handle(20, B) BEGIN
BBBBBBBBBBBBBBBBBBBB
          handle(20, B) END
      request(20, B) END
      request(30, C) BEGIN ←一个 request 结束之后，下一个 request 才会开始
          handle(30, C) BEGIN
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
          handle(30, C) END
      request(30, C) END
main END                    ←所有的 request 结束之后，主线程才会结束
```

图 A7-1　执行结果

问题 7-3 的解答

执行结果如图 A7-2 所示。与上一题用单线程得出的执行结果（见图 A7-1）一样。之所以如此，是因为在线程启动的部分调用出的是 run 方法，而非 start 方法。即使调用出 run 方法，新的线程也不会启动，因此反而变成只由主线程来执行匿名内部类（inner class）。这样的话，响应性是不会提升的。

```
main BEGIN
    request(10, A) BEGIN
        handle(10, A) BEGIN
AAAAAAAAAA
        handle(10, A) END
    request(10, A) END
    request(20, B) BEGIN
        handle(20, B) BEGIN
BBBBBBBBBBBBBBBBBBBB
        handle(20, B) END
    request(20, B) END
    request(30, C) BEGIN
        handle(30, C) BEGIN
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
        handle(30, C) END
    request(30, C) END
main END
```

图 A7-2　执行结果

问题 7-4 的解答

介绍两个解答范例。

◆ 解答 1：将 *HelperThread* 类声明为 *Top Level* 类

将 Host 类、HelperThread 类分别写成如 List A7-2、List A7-3 所示。Main 类、Helper 类维持范例程序的属性即可。

跟使用匿名内部类的情况相比，将被包含在要求中的参数传递给线程，在写程序代码时稍微比较麻烦，因为必须要一个个整理成字段的形式。不过，比起使用匿名内部类，程序代码的可读性也许更高也不一定。

List A7-2 Host 类(Host.java)

```
public class Host {
    private Helper helper = new Helper();
    public void request(int count, char c) {
        System.out.println("   request(" + count + ", " + c + ") BEGIN");
        new HelperThread(helper, count, c).start();
        System.out.println("   request(" + count + ", " + c + ") END");
    }
}
```

List A7-3 HelperThread 类(HelperThread.java)

```
public class HelperThread extends Thread {
    private final Helper helper;
    private final int count;
    private final char c;
    public HelperThread(Helper helper, int count, char c) {
        this.helper = helper;
        this.count = count;
        this.c = c;
    }
    public void run() {
        helper.handle(count, c);
    }
}
```

◆ 解答 2：将 *HelperThread* 声明为非匿名的内部类

写成如 List A7-4 这样的话，就可以在 Helper 类与 HelperThread 类在 Host 类中

进行声明。这样一来的话，Helper 与 HelperThread 的两个类可以表现出与 Host 类的确关系密切。但是，Host 类的声明变长了变得比较不好看，这是其缺点。

Main 类保持范例文件的原样。

List A7-4 Host 类 (Host.java)

```
public class Host {
    private final Helper helper = new Helper();
    public void request(int count, char c) {
        System.out.println("   request(" + count + ", " + c + ") BEGIN");
        new HelperThread(helper, count, c).start();
        System.out.println("   request(" + count + ", " + c + ") END");
    }

    // Inner class
    private class Helper {
        public void handle(int count, char c) {
            System.out.println("       handle(" + count + ", " + c
+ ") BEGIN");
            for (int i = 0; i < count; i++) {
                slowly();
                System.out.print(c);
            }

            System.out.println("");
            System.out.println("       handle(" + count + ", " + c
+ ") END");
        }
        private void slowly() {

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }

    // Inner class
    private class HelperThread extends Thread {
        private final Helper helper;
        private final int count;
        private final char c;
```

```
        public HelperThread(Helper helper, int count, char c) {
            this.helper = helper;
            this.count = count;
            this.c = c;
        }
        public void run() {
            helper.handle(count, c);
        }
    }
}
```

问题 7-5 的解答

这里提出 4 个解答范例。这 4 个解答的不同之处在于用户连续按下按键时的处理方式。

◆ 解答 1：使用 *Thread-Per-Message* 模式

service 方法中启动新的线程，该新的线程则会调用 doService 这个方法。然后以 doService 进行实际的处理。这样一来，就可以从 service 方法马上回来。

这个解答在用户连续按下按键时，会同时执行好几个线程 doService（见图 A7-3）。

List A7-5 解答 1：连续按下按键，每次按下都会执行 doService 的 Service 类(Service.java)

```
public class Service {
    public static void service() {
        new Thread() {
            public void run() {
                doService();
            }
        }.start();
    }
    private static void doService() {
        System.out.print("service");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
```

```
        }
        System.out.println("done.");
    }
}
```

```
service.......service.......service.....................
.................................................done.
.............done.
......done.
```

图 A7-3　解答 1：按键按 3 次的执行范例（3 个线程的输出会混在一起）

◆ 解答 2：使用 *Thread-Per-Message* 模式与 *Single Threaded Execution* 模式

另一个解答则如 List A7-6。在此程序中，线程从 service 方法立即回来。而且保证用户即使连续按下按键多次，同时执行 doService 方法的线程也只有一个。

此项解答在用户连续按下按键多次时，按几次就会执行 doService 几次，输出结果不会混在一起。

List A7-6　解答 2：即使连续按下按键多次，输出结果也不会混在一起（Service.java）

```
public class Service {
    public static void service() {
        new Thread() {
            public void run() {
                doService();
            }
        }.start();
    }
    private static synchronized void doService() {
        System.out.print("service");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("done.");
    }
}
```

```
service.....................................done.
service.....................................done.
service.....................................done.
```

图 A7-4　解答 2：按键按 3 次的执行范例（3 个线程会按顺序输出）

◆ 解答 3：使用 *Thread-Per-Message* 模式与 *Balking* 模式

List A7-7 会从 service 方法马上回来，而且在用户连续按键多次时，而且保证执行 doService 方法的只有第一个线程。在此，使用 Balking 模式（第 4 章），将同时想要执行 doService 的线程予以 balk 起来（图 A7-5）。

List A7-7　解答 3：连续按键多次时就予以 balk 的类(Service.java)

```
public class Service {
    private static volatile boolean working = false;

    public static synchronized void service() {
        System.out.print("service");
        if (working) {
            System.out.println(" is balked.");
            return;
        }
        working = true;
        new Thread() {
            public void run() {
                doService();
            }
        }.start();
    }
    private static void doService() {
        try {
            for (int i = 0; i < 50; i++) {
                System.out.print(".");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
            System.out.println("done.");
        } finally {

            working = false;
```

```
        }
    }
}
```

```
service.......service is balked.
.........service is balked.
..............................done.
```

图 A7-5　解答 3：连续按键 3 次的执行范例（第 2 次与第 3 次进行 balk）

◆ 解答 4：连续按键多次时就将执行中的处理予以取消

List 7-8 会从 service 方法马上回来，而且用户连续按下按键多次时，会取消 doService 方法。

在这里，使用 interrupt 取消处理。关于线程的结束，请一并参照 Two-Phase Termination 模式（第 10 章）。

List A7-8　解答 4：连续按下按键多次时就将执行中的处理予以取消的 Service 类(Service.java)

```
public class Service {
    private static Thread worker = null;
    public static synchronized void service() {
    // 如果有执行中的处理，就使用 interrupt 取消
     if (worker != null && worker.isAlive()) {
        worker.interrupt();
        try {
            worker.join();
        } catch (InterruptedException e) {
        }
        worker = null;
    }
    System.out.print("service");
    worker = new Thread() {
        public void run() {
            doService();
        }
    };
    worker.start();
}
private static void doService() {
    try {
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
```

```
                Thread.sleep(100);
            }
            System.out.println("done.");
        } catch (InterruptedException e) {
            System.out.println("cancelled.");
        }
    }
}
```

```
service..............cancelled.
service......cancelled.
service.....................................done.
```

图 A7-6　解答 4：连续按按键 3 次的执行范例（第 1 次与第 2 次被取消了）

问题 7-6 的解答

将 List 7-9 的 MiniServer 类改成 List A7-9。Main 类、Service 类可以不必修改。

List A7-9　MiniServer 类 (MiniServer.java)

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.IOException;

public class MiniServer {
    private final int portnumber;
    public MiniServer(int portnumber) {
        this.portnumber = portnumber;
    }
    public void execute() throws IOException {
        ServerSocket serverSocket = new ServerSocket(portnumber);
        System.out.println("Listening on " + serverSocket);
        try {
            while (true) {
                System.out.println("Accepting...");
                final Socket clientSocket = serverSocket.accept();
                System.out.println("Connected to " + clientSocket);
                new Thread() {
                    public void run() {
                        try {
                            Service.service(clientSocket);
                        } catch (IOException e) {
```

```
                        e.printStackTrace();
                    }
                }.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            serverSocket.close();
        }
    }
}
```

问题 7-7 的解答

解答如 List A7-10（应该也有其他方法）。

我们将 List A7-10 的思考过程依次解说。

看看期待的执行结果（图 7-9），就可以了解到：

[1] 因为显示了 Step 2，magic 方法就不会丢出异常。

[2] Step3 或 End 不会被显示出来，因此线程不会从 enter 方法回来。

[3] Step 3 没有被显示出来是因为线程没有取得 obj 的锁定就阻挡起来。

目前为止都已经能了解的话，就会发现 magic 方法的工作就是下面的[4]。

[4] magic 方法的工作中，就是取得参数 obj。

但是，若想要取得锁定，即使单纯地使用 synchronized 也是不行的。在从 magic 方法回来的时候，就从 synchronized 跳出，是因为锁定被解开的缘故。我们可以知道，在这样的情况下，我们只要进行如下[5]、[6]的处理即可。

[5] magic 方法中，启动新的线程，让这个线程取得 obj 的锁定。

[6] 新的线程永远保持取得 obj 的锁定的状态。

如果想说到这个阶段就可以了，那你很可能会忽略了下面的[7]。

[7] 新的线程启动，取得 obj 的锁定为止，原来的线程不能从 magic 方法回来。

List A7-10 就是以如上[1]~[7]为基础为写成的程序。为了将参数 obj 传递给内部类（inner class）的方法 run，我们在此将参数 obj 声明为 final。

此外，为了满足[7]，我们使用第 3 章的 Guarded Suspension 模式（Pattern）。原来的线程会在 thread 上一直 wait，直到 thread 的名字在""之外。另一方面，新的线程在取得 obj 的锁定之后，将名称改成"Locked"，将在 thread（此与 this 等值）上 wait 的原线程进行 notifyAll。

在此我们使用 notifyAll，不过在 thread 上 wait 的线程一定只有一个，所以就算用 notify 也是没问题的。

List A7-10 完成的 Blackhole 类 (之 1) (Blackhole.java)

```
public class Blackhole {
    public static void enter(Object obj) {
        System.out.println("Step 1");
        magic(obj);
        System.out.println("Step 2");
        synchronized (obj) {
            System.out.println("Step 3 (never reached here)"); //
不会到这里来
        }
    }
    public static void magic(final Object obj) {
        // thread会取得obj的lock，变成无穷循环的线程
        // 将thread的名称当作Guard条件来使用
        Thread thread = new Thread() {   // inner class
            public void run() {
                synchronized (obj) {     //在此取得obj的lock
                    synchronized (this) {
                        this.setName("Locked"); // Guard 条件的变化
                        this.notifyAll();   // 通知已经取得obj的lock
                    }
                    while (true) {
                        // 无穷循环
                    }
                }
            };
            synchronized (thread) {
                thread.setName("");
                thread.start();// 线程的启动
                // Guarded Suspension模式
                while (thread.getName().equals("")) {
                    try {
                        thread.wait();// 等待新的线程取得obj的lock
                    } catch (InterruptedException e) {
                    }
                }
            }
        }
}
```

也可以写成 List A7-11。这里也是一样，不用 notifyAll 而用 notify 也可以。

List A7-11 完成的 Blackhole 类(之 2) (Blackhole.java)

```
public class Blackhole {
    public static void enter(Object obj) {
```

```
        System.out.println("Step 1");
        magic(obj);
        System.out.println("Step 2");
        synchronized (obj) {
          System.out.println("Step 3(never reached here)");//不会到这里来
        }
    }
    public static void magic(final Object obj) {
        // thread会取得obj的lock，变成等待自己结束的线程
        Thread thread = new Thread() {
            public void run() {
                synchronized (obj) { // 在这里取得obj的锁定
                    synchronized (this) {

                        this.notifyAll(); // 通知已经取得obj的锁定
                    }
                    try {
                        this.join();// 永远等待
                    } catch (InterruptedException e) {
                    }
                }
            }
        };
        synchronized (thread) {
            thread.start();// 线程的启动
            try {
                thread.wait();// 等待新的线程取得obj的锁定
            } catch (InterruptedException e) {
            }
        }
    }
}
```

第8章

问题8-1的解答

×（1）没有任何请求的时候，WorkerThread的线程会一直sleep下去。

没有锁定时，WorkerThread的线程在Channel的实例上wait。

×（2）来自某个ClientThread的请求在被执行（execute）时，来自同一个ClientThread的请求就不会被执行（execute）。

在一个execute结束之前，别的WorderThread的线程也可以执行（execute）来自相同ClientThread的请求。范例程序中常常发生。

○（3）会调用 putRequest 方法的只有 ClientThread。
○（4）会调用 takeRequest 方法的只有 WorkerThread。
○（5）没有必要将 execute 方法设成 synchronized。
范例程序中，对一个 Request 的实例进行 execute 是一个线程，因此没有必要设为 synchronized。

问题 8-2 的解答

如 List A8-1。在 putRequest 方法中启动线程。这就是 Thread-Per-Message 模式（Pattern）（第 7 章）。执行范例请参考图 A8-1。

List A8-1 每次都启动新的线程 Channel 类(Channel.java)

```
public final class Channel {
    public Channel(int threads) {
    }
    public void startWorkers() {
    }
    public void putRequest(final Request request) {
        new Thread() {
            public void run() {
                request.execute();
            }
        }.start();
    }
}
```

```
Thread-1 executes [ Request from Alice No.0 ]
Thread-2 executes [ Request from Bobby No.0 ]
Thread-3 executes [ Request from Chris No.0 ]
Thread-4 executes [ Request from Chris No.1 ]
Thread-5 executes [ Request from Alice No.1 ]
Thread-6 executes [ Request from Alice No.2 ]
Thread-7 executes [ Request from Bobby No.1 ]
Thread-8 executes [ Request from Bobby No.2 ]
Thread-9 executes [ Request from Chris No.2 ]
Thread-10 executes [ Request from Alice No.3 ]
Thread-11 executes [ Request from Bobby No.3 ]
Thread-12 executes [ Request from Alice No.4 ]
Thread-13 executes [ Request from Chris No.3 ]
Thread-14 executes [ Request from Chris No.4 ]
Thread-15 executes [ Request from Bobby No.4 ]
Thread-16 executes [ Request from Chris No.5 ]
Thread-17 executes [ Request from Alice No.5 ]
(以下省略。按 CTRL+C 结束)
```

图 A8-1　执行范例

Thread-0, Thread-1, Thread-2,……会一直增加上去，因此每次执行新的线程时都可以知道。

“Thread-数字”这样的线程名称是由 java.lang.Thread 类自动获取。

问题 8-3 的解答

我们改写范例的 Main 类，设置成在执行开始后约 30 秒就进行强制结束（List A8-2）。通过比较请求的编号增加到多少，来比较两支程序的（Throughput）。为了能识别其差异性，我们将线程的等待时间取消（List A8-3, List A8-4）。

List A8-2 修改成能强制终了的 Main 类（Main.java)

```
public class Main {
    public static void main(String[] args) {
        Channel channel = new Channel(5); // WorkerThread的个数
        channel.startWorkers();
        new ClientThread("Alice", channel).start();
        new ClientThread("Bobby", channel).start();
        new ClientThread("Chris", channel).start();

        try {
            Thread.sleep(30000);
        } catch (InterruptedException e) {
        }
        System.exit(0);
    }
}
```

List A8-3 取消等待时间的 ClientThread 类（ClientThread.java)

```
public class ClientThread extends Thread {
    private final Channel channel;
    public ClientThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        for (int i = 0; true; i++) {
            Request request = new Request(getName(), i);
            channel.putRequest(request);
        }
    }
}
```

List A8-4 取消等待时间的 Request 类 (Request.java)

```
public class Request {
    private final String name;
    private final int number;
    public Request(String name, int number) {
        this.name = name;
        this.number = number;
    }
    public void execute() {
        System.out.println(Thread.currentThread().getName() + "
executes " + this);
    }
    public String toString() {
        return "[ Request from " + name + " No." + number + " ]";
    }
}
```

图 A8-2 是使用了 Worker Thread 模式的执行范例，而图 A8-3 则是没有使用 Worker Thread 的执行范例。使用了 Worker Thread 模式的话，会处理 409683(=135956+136879+136848) 个 request，没有的话会处理时 37799 个 (=12186+13002+12611)个 request。这个执行范例中，throughput 提升了约 10 倍以上。

注意：数值会随执行环境而有很大变化。

```
(前略)
Worker-0 executes [ Request from Alice No.136876 ]
Worker-4 executes [ Request from Chris No.136845 ]
Worker-2 executes [ Request from Bobby No.135954 ]
Worker-0 executes [ Request from Alice No.136877 ]
Worker-4 executes [ Request from Chris No.136846 ]
Worker-2 executes [ Request from Bobby No.135955]
←Bobby 的 request 被处理的个数有 0~135955，共 135956 个
Worker-3 executes [ Request from Alice No.136878 ]
←Alice 的 request 被处理的个数有 0~136878，共 136879 个
Worker-0 executes [ Request from Chris No.136847 ]
←Chris 的 request 被处理的个数有 0~136847，共 136848 个
```

图 A8-2 执行范例 (使用 Worker Thread 模式)

```
(前略)
Thread-37705 executes [ Request from Alice No.12183 ]
Thread-37706 executes [ Request from Alice No.12184 ]
```

（续上图）

```
Thread-37707 executes [ Request from Alice No.12185 ]
←Alice 的 request 被处理的个数有 0~112185，共 12186 个

（中略）
Thread-37783 executes [ Request from Alice No.12999 ]
Thread-37784 executes [ Request from Alice No.13000 ]
Thread-37785 executes [ Request from Alice No.13001 ]
←Chris 的 request 被处理的个数有 0~13001，共 13002 个

（中略）
Thread-37797 executes [ Request from Alice No.12608 ]
Thread-37798 executes [ Request from Alice No.12609 ]
Thread-37799 executes [ Request from Alice No.13610 ]
←Bobby 的 request 被处理的个数有 0~12610，共 12611 个
```

图 A8-3　执行范例（不使用 Worker Thread 模式，每次都启动新的线程）

问题 8-4 解答

因为会失去生存性。

如果调用 invokeAndWait 方法，在处理过事件队列（Event Queue）中的所有事件后，参数的 Runnable 被执行，控制才会回来。但是，从 Event Dispatching Thread 调用出 invokeAndWait，就等于是调用出事件队列中的一个事件 invokeAndWait。从 invokeAndWait 回来时，这个事件的处理才告结束。但是，为了从 invokeAndWait 回来，首先必须要事件队列中的所有事件都先处理完毕。这样到最后，就变成无法从 invokeAndWait 回来了。Event Dispatching Thread 就动弹不了了，GUI 应用程序的事件处理就失去其功能了。

问题 8-5 解答

会变成问题中所述那样的结果，是因为 Swing 的 Event Dispatching Thread 为了进行处理而花了约 10 秒在进行循环的缘故。会进行画面的显示的，就是 Event Dispatching Thread。Event Dispatching Thread 如果没有从 actionPerformed 方法回来，画面就不会更新。

修改的 MyFrame 如 List A8-5 所示，修改后的画面如图 A8-4~图 A8-8 所示。执行结果如图 A8-9 所示。

为了解决这个问题，我们可进行修改如下。

◆ 为了提升响应性，将工作委托其他线程

countUp 方法中，启动了 invokerThread 这个线程。这个线程执行实际的累加

（countUp）操作。调用出 countUp 的线程（Event Dispatching Thread）启动 invoderThread，马上就从 countUp 方法回来。这就是第 7 章学到的 Thread-Per-Message 模式。

◈ 为了将工作委托 Event Dispatching Thread，必须使用 invokeLater 方法

invoketThread 当中，会从 0,1,2,.....,9 进行累加，再以 sleep 方法休息约 1 秒。invokerThread 并不会直接对累加过的数目进行 setText。原因在于，invokerThread 并非 Event Dispatching Thread。为了让 Event Dispatching Thread 执行“将数字 setText”，就要使用下面的语句。

SwingUtilities.invokeLater(exercutor);

使用参数的 exector，可以用 Runnable 对象将下述的语句让 Event Dispatching Thread 来执行。

Label.setText(string);

看起来好像非常复杂，但是一步一步仔细地思考，就能豁然开朗。在图 A8-10 中，我们可以看到整个流程图。

List A8-5 修改后的 MyFrame 类(MyFrame.java)

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.SwingUtilities;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;


public class MyFrame extends JFrame implements ActionListener {
    private final JLabel label = new JLabel("Event Dispatching
Thread Sample");
    private final JButton button = new JButton("countUp");
    public MyFrame() {
        super("MyFrame");
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(label);
        getContentPane().add(button);
        button.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }
```

```
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button) {
            countUp();
        }
    }
    private void countUp() {
        System.out.println(Thread.currentThread().getName()+
":countUp:BEGIN");

        // invokerThread在指定时间内sleep之后,调用SwingUtilities.invokeLater
        new Thread("invokerThread") {
            public void run() {
                System.out.println(Thread.currentThread().getName() +
":invokerThread:BEGIN");
                for (int i = 0; i < 10; i++) {
                    final String string = "" + i;
                    try {
                        // executor会被从Event Dispatching Thread调用
                        final Runnable executor = new Runnable() {
                            public void run() {

System.out.println(Thread.currentThread().getName() + ":executor:BEGIN:
string = " + string);
                                label.setText(string);
System.out.println(Thread.currentThread().getName() + ":executor:END");
                            }
                        };

                        // 让Event Dispatching Thread调用出exector
                        SwingUtilities.invokeLater(executor);

                        Thread.sleep(1000);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(Thread.currentThread().getName()
+ ":invokerThread:END");
            }
        }.start();

        System.out.println(Thread.currentThread().getName()  +
":countUp:END");
    }
}
```



图 A8-4　启动后的画面

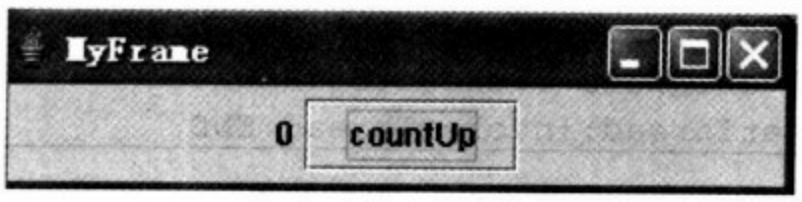


图 A8-5　按下[countUp]后的画面

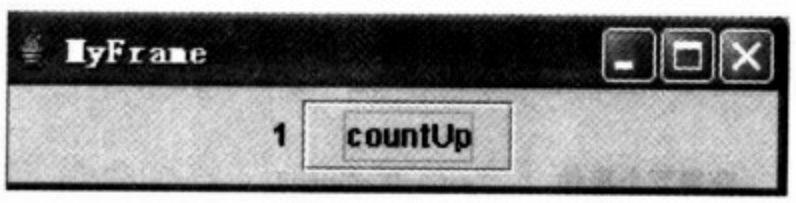


图 A8-6　约 1 秒后的画面

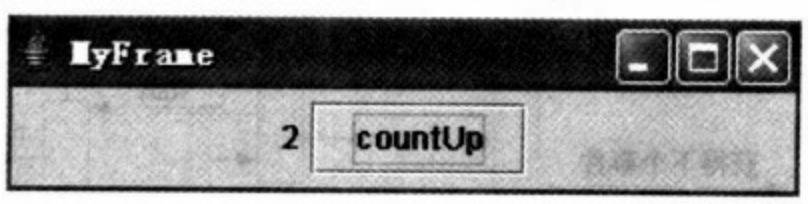


图 A8-7　约 2 秒后的画面

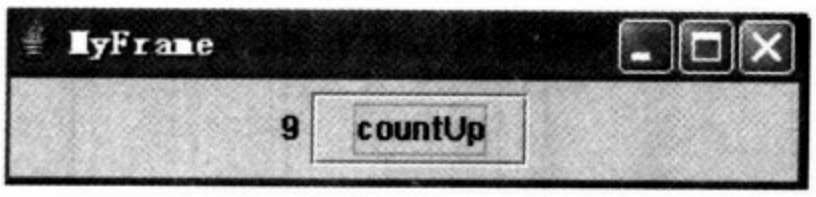


图 A8-8　最后的画面

```
main:BEGIN
main:END
AWT-EventQueue-0:countUp:BEGIN
AWT-EventQueue-0:countUP:END
invokerThread:invokerThread:BEGIN
AWT-EventQueue-0:executor:BEGIN:string = 0
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 1
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 2
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 3
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 4
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 5
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 6
AWT-EventQueue-0:executor:END
```

（续上图）

```
AWT-EventQueue-0:executor:BEGIN:string = 7
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 8
AWT-EventQueue-0:executor:END
AWT-EventQueue-0:executor:BEGIN:string = 9
AWT-EventQueue-0:executor:END
invokerThread:invokerThread:END
```

图 A8-9　执行结果

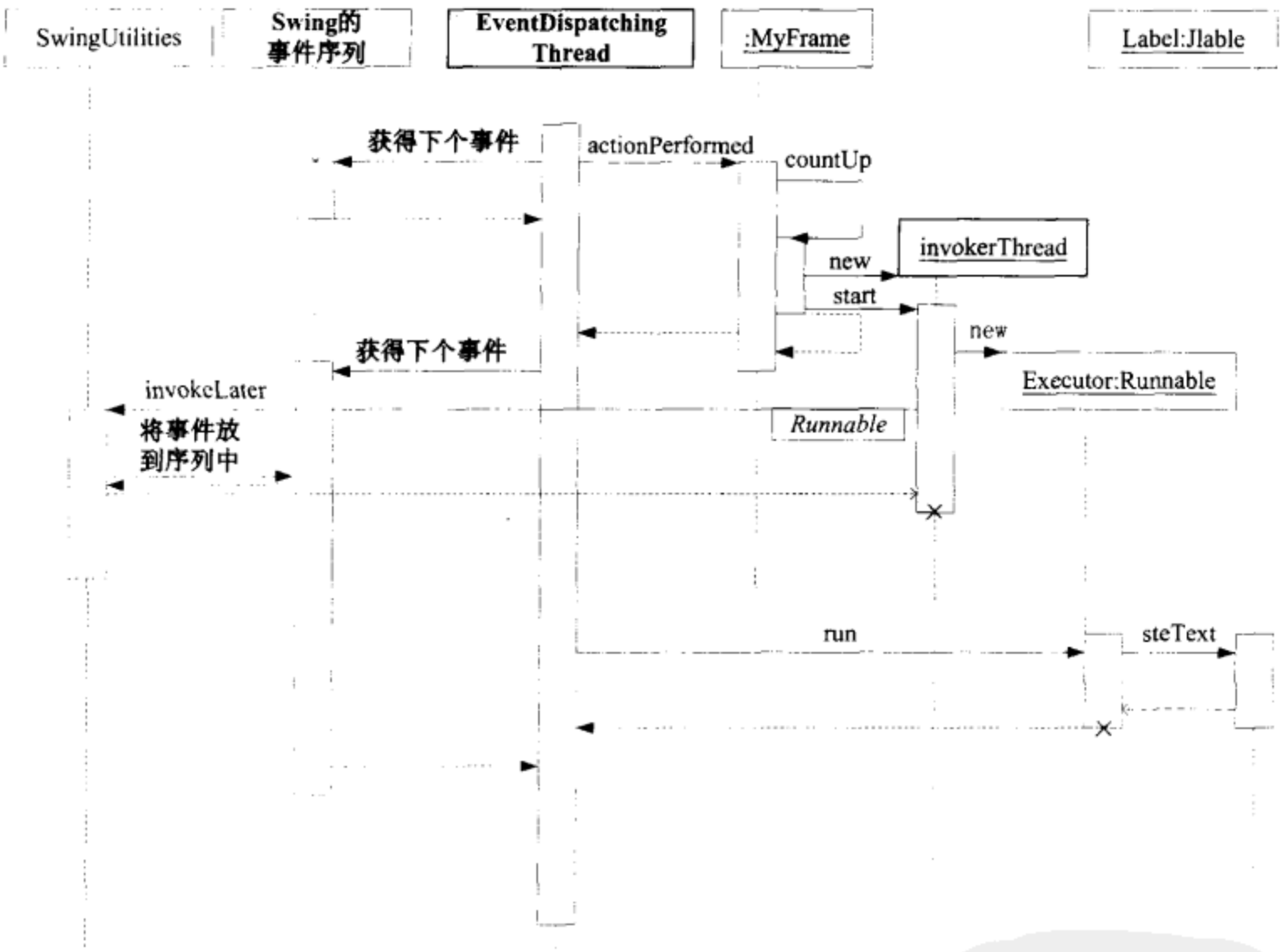


图 A8-10　问题 8-5 的流程图

问题 8-6 的解答

如 List A8-6~List A8-8。执行结果如图 A8-11 所示。

首先，我们不用 Thread 类的 stop 方法。这是因为 stop 方法即使是对锁定的线程，也能让它结束，所以较不安全。

Chanel 类的 putRequest 方法与 takeRequest 方法变成会抛出异常 InterruptedException。然后，stopAllWorkers 方法会进行安装，以调用自己所保存的 WorkerThread 类的

stopThread。

各个线程（ClientThread, WorkerThread）的 stopThread 中，将 terminated 字段设为 true，调用出 interrupt 方法。terminated 字段会显示是否已经进行过结束处理，而 interrupt 方法会对指定的线程（在此指其本身）执行中断。关于此方法，我们在 Two Phase Termination Pattern（第 10 章）中将详细介绍。

List A8-6 修改后的 Channel 类(Channel.java)

```
public final class Channel {
    private static final int MAX_REQUEST = 100;
    private final Request[] requestQueue;
    private int tail;   //下次要 putRequest 的位置
    private int head;   //下次要 takeRequest 的位置
    private int count;  // Request 的次数

    private final WorkerThread[] threadPool;

    public Channel(int threads) {
        this.requestQueue = new Request[MAX_REQUEST];
        this.head = 0;
        this.tail = 0;
        this.count = 0;
        threadPool = new WorkerThread[threads];
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i] = new WorkerThread("Worker-" + i, this);
        }
    }
    public void startWorkers() {
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i].start();
        }
    }
    public void stopAllWorkers() {
        for (int i = 0; i < threadPool.length; i++) {
            threadPool[i].stopThread();
        }
    }
    public synchronized void putRequest(Request request) throws
InterruptedException {
        while (count >= requestQueue.length) {
            wait();
```

```
        }
        requestQueue[tail] = request;
        tail = (tail + 1) % requestQueue.length;
        count++;
        notifyAll();
    }
    public synchronized Request takeRequest() throws
InterruptedException {
        while (count <= 0) {
            wait();
        }
        Request request = requestQueue[head];
        head = (head + 1) % requestQueue.length;
        count--;
        notifyAll();
        return request;
    }

}
```

List A8-7 修改后的 WorkerThread 类(WorkerThread.java)

```
zpublic class WorkerThread extends Thread {
    private final Channel channel;
    private volatile boolean terminated = false;
    public WorkerThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        try {
            while (!terminated) {
                try {
                    Request request = channel.takeRequest();
                    request.execute();
                } catch (InterruptedException e) {
                    terminated = true;
                }
            }
        } finally {
            System.out.println(Thread.currentThread().getName()
+ " is terminated.");
```

```
        }
    }
    public void stopThread() {
        terminated = true;
        interrupt();
    }
}
```

List A8-8 修改后的 ClientThread 类(ClientThread.java)

```
import java.util.Random;

public class ClientThread extends Thread {
    private final Channel channel;
    private static final Random random = new Random();
    private volatile boolean terminated = false;
    public ClientThread(String name, Channel channel) {
        super(name);
        this.channel = channel;
    }
    public void run() {
        try {
            for (int i = 0; !terminated; i++) {
                try {
                    Request request = new Request(getName(), i);
                    channel.putRequest(request);
                    Thread.sleep(random.nextInt(1000));
                } catch (InterruptedException e) {
                    terminated = true;
                }
            }
        } finally {
            System.out.println(Thread.currentThread().getName()
+ " is terminated.");
        }
    }
    public void stopThread() {
        terminated = true;
        interrupt();
    }
}
```

```
（前略）
Worker-4 executes [ Request from Alice No.10 ]
Worker-0 executes [ Request from Alice No.11 ]
Worker-1 executes [ Request from Alice No.12 ]
Worker-4 executes [ Request from Alice No.9 ]
Worker-0 executes [ Request from Chris No.8 ]
Worker-2 executes [ Request from Bobby No.10 ]
Worker-3 executes [ Request from Chris No.9 ]
Worker-4 executes [ Request from Alice No.13 ]
Worker-0 executes [ Request from Chris No.10 ]
Worker-1 executes [ Request from Alice No.14 ]
Worker-3 executes [ Request from Bobby No.11 ]
Worker-0 is terminated.
Worker-1 is terminated.
Worker-2 is terminated.
Worker-3 is terminated.
Worker-4 is terminated.
Alice is terminated.
Bobby is terminated.
Chris is terminated.
```

图 A8-11　执行范例

第 9 章

问题 9-1 的解答

○（1）调用出 request 方法后，一个新的线程就会启动。

○（2）request 方法的返回值类型虽是 Data 接口，实际的返回值是 FutureData 的实例。

request 方法中，对 FutureData 类的实例执行 new。

×（3）会调用 SetRealData 方法的是主线程。

会调用 SetRealData 方法的是在 request 方法中启动的线程。

○（4）执行 RealData 类的 getContent 方法的，是主线程。

×（5）如果 request 方法被从数个线程调用的话，request 方法必须设为 synchronized。

request 方法不会对数个线程所共有的字段进行任何处理，因此不需要设为 synchronized。request 的参数（count, c）或是局部变量（future），只有调用出这个 request 的线程才会读取得到。

问题 9-2 的解答

除了主线程之外，又建立出了 3 个线程，各自进行建立 RealData 的实例的工作。在此定义如下，

- 进行 new RealData(10, 'A')的线程是 A 线程
- 进行 new RealData(20, 'B')的线程是 B 线程
- 进行 new RealData(30, 'C')的线程是 C 线程

则范例程序的执行范例（图 9-3）会各自由图 A9-1 中的线程来进行显示。

线程的主类	显示属性
主线程	main BEGIN
主线程	request(10, A) BEGIN
主线程	request(10, A) END
主线程	request(20, B) BEGIN
主线程	request(20, B) END
主线程	request(30, C) BEGIN
线程 A	making RealData(10, A) BEGIN
线程 B	making RealData(20, B) BEGIN
主线程	request(30, C) END
主线程	main otherJob BEGIN
线程 C	making RealData(30, C) BEGIN
线程 A	making RealData(10, A) END
线程 B	making RealData(20, B) END
主线程	main otherJob END
主线程	data1 = AAAAAAAAAA
主线程	data2 = BBBBBBBBBBBBBBBBBBBB
线程 C	making RealData(30, C) END
主线程	data3 = CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
主线程	main END

图 A9-1　范例程序的输出（图 9-3）是哪个线程在执行的呢?

问题 9-3 的解答

首先，我们修改 Retriever 类，使能返回 AsynContentImpl 类的实例（List A9-1）。之后，如 List A9-2 建立 AsyncContentImpl 类。AsyncContentImpl 类是 Future 参与者，SyncContentImpl 类是 RealData 参与者。

注意：连接上 Web 网页的时间视环境而异，因此这里的测量时间仅作参考。

List A9-1　会返回 AsyncContentImpl 的实例的 Retriever 类(Retriever.java)

```
package content;

public class Retriever {
```

```
    public static Content retrieve(final String urlstr) {
        final AsyncContentImpl future = new AsyncContentImpl();

        new Thread() {
            public void run() {
                future.setContent(new SyncContentImpl(urlstr));
            }
        }.start();

        return future;
    }
}
```

List A9-2 AsyncContentImpl 类(AsyncContentImpl.java)

```
package content;

class AsyncContentImpl implements Content {
    private SyncContentImpl synccontent;
    private boolean ready = false;
    public synchronized void setContent(SyncContentImpl
synccontent) {
        this.synccontent = synccontent;
        this.ready = true;

        notifyAll();
    }
    public synchronized byte[] getBytes() {
        while (!ready) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        return synccontent.getBytes();
    }
}
```

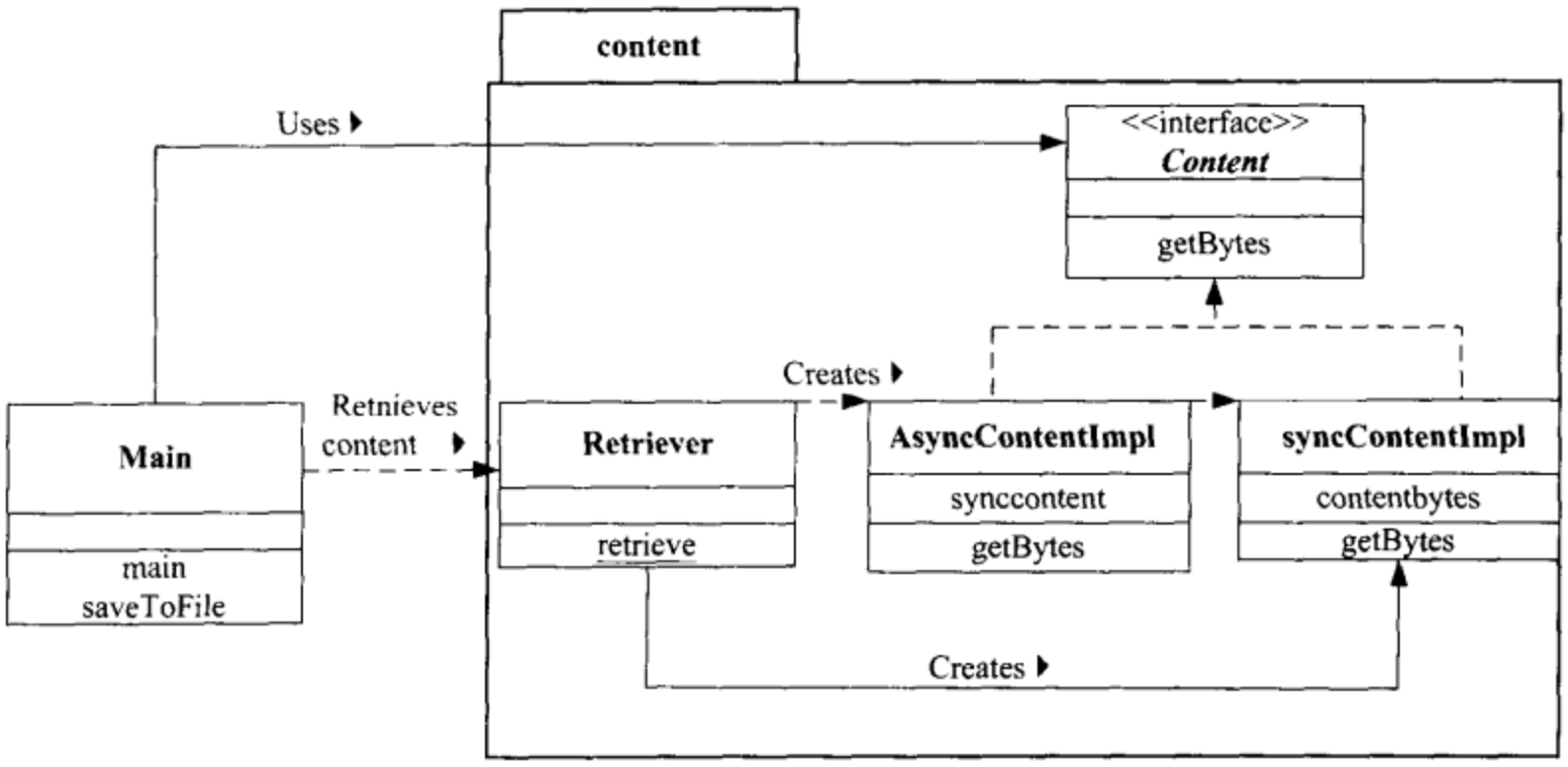


图 A9-2　问题 9-3 的类图（多线程版）

```
Thread-0: Getting http://www.yahoo.com/
Thread-1: Getting http://www.google.com/
Thread-2: Getting http://www.hyuki.com/
main: Saving to yahoo.html
main: Saving to google.html
main: Saving to hyuki.html
Elapsed time = 1062msec.
```

图 A9-3　执行范例

问题 9-4 的解答

其实方法有很多，在此我们使用在附录 E 的[Lea]中所介绍的方法（异常的 wraping）。具体的步骤如下：

- 修改 Host 类，让 RealData 的实例在制作时如果发生异常，会将该异常设置到 FutureData 类。
- 将设置（set）异常的方法 setException 新增到 FutureData 类中（List A9-5）。
- 为了将发生的异常“包起来”，我们使用异常 java.lang.reflect.InvocationTargetException（List A9-5）。
- 还要让 getContent 方法抛出 InvocationTargetException（List A9-4, List A9-5）。

List A9-3　修改后的 Host 类(Host.java)

```
public class Host {
    public Data request(final int count, final char c) {
```

```
        System.out.println("    request(" + count + ", " + c + ")
BEGIN");

        // (1) 制作 FutureData 的实例
        final FutureData future = new FutureData();

        // (2) 启动用以建立 RealData 的新线程
        new Thread() {
            public void run() {
                try {
                    RealData realdata = new RealData(count, c);
                    future.setRealData(realdata);
                } catch (Exception e) {
                    future.setException(e);
                }
            }
        }.start();

        System.out.println("  request(" + count + ", " + c + ")
END");

        // (3) 将 FutureData 的实例当成返回值返回
        return future;
    }
}
```

List A9-4 修改后的 Data 类(Data.java)

```
import java.lang.reflect.InvocationTargetException;

public interface Data {
    public abstract String getContent() throws InvocationTargetException;
}
```

FutureData 类（List A9-5）的 setException 方法与 setRealData 方法大致相同，但要指定到的地方并非 realdata 字段，而是 exception 字段。要指定到 exception 的是 InvocationTargetException 的实例。

List A9-5 修改后的 FutureData 类(FutureData.java)

```
import java.lang.reflect.InvocationTargetException;
public class FutureData implements Data {
    private RealData realdata = null;
```

```
private InvocationTargetException exception = null;
private boolean ready = false;
public synchronized void setRealData(RealData realdata) {
    if (ready) {
        return;
    }
    this.realdata = realdata;
    this.ready = true;
    notifyAll();
}
public synchronized void setException(Throwable throwable) {
    if (ready) {
        return;
    }
    this.exception = new InvocationTargetException(throwable);
    this.ready = true;
    notifyAll();
}
public synchronized String getContent() throws InvocationTargetException{
    while (!ready) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    if (exception != null) {
        throw exception;
    }
    return realdata.getContent();
  }
}
```

```
main BEGIN
        request(-1, N) BEGIN
        request(-1, N) END
            making RealData(-1, N) BEGIN
java.lang.reflect.InvocationTargetException:
java.lang.NegativeArraySizeException
            at RealData.<init>(RealData.java:5)
            at Host$1.run(Host.java:12)← 在这里程序就会结束
```

图 A9-4　执行范例

换句话说，会让异常延迟发生，直到要从 Future 参与者获得必要的值。这有点像傍晚拿着蛋糕提货单到蛋糕店去，蛋糕店却由于烤箱故障而告诉你今天无法做蛋糕是一样的意思。

▶▶ 补充一点：异常 java.lang.reflect.InvocationTargetException

InvocationTargetException 是 java.lang.reflect.Method 类的 invoke 方法与 java.lang.reflect.Constructor 类的 newInstance 方法所抛出的异常。

由于用 InvocationTargetException 包住了，因此与实际发生的异常的类型不会互相影响，可以撰写 getContent 的 throws 这段。练习问题 9-4 中，我们使用了这个 InvocationTargetException，但是其实也可以不用。

关于如何包住异常，请参考如下的网页。

- Chanined Exception Facility
  http://java.sun.com/j2se/1.4/docs/guide/lang/chained-exceptions.html

第 10 章

问题 10-1 的解答

○（1）调用出 shutdownRequest 方法的是主线程。

×（2）doWork 方法只会被调用一次。

doWork 方法是在 while 语句的地方被调用的，因此通常会被调用很多次。

○（3）doWork 方法将异常 InterruptedException 抛出时，doShutdown 方法会被调用出来。

异常 InterruptedException 被抛出时，finally 就会被执行。

×（4）shutdownRequest 方法中的 interrupt()也可以写成 Thread.currentThread().interrupt()，意思一样。

shutdownRequest 方法内的 interrupt()与 this.interrupt()是一样的。不管哪个线程调用出 shutdownRequest，对 CountupThread 的线程都进行中断。

此外 shutdownRequest 方法中，如果撰写了 Thread.currentThread().interrupt()会变成如何呢？在语句 Thread.currentThread 的值会等于这项方法所调用出的线程所对应的对象。因此，在 shutdownRequest 方法中，如果撰写成 Thread.currentThread().interrupt()，对于 shutdownRequest 所调用出的线程会加上中断（interrupt）。举例来说，在范例程序中，主线程就加上了中断。

问题 10-2 的解答

如 List A10-1。在此我们将 sleep 方法的调用用 try...catch 包住，不管异常 InterruptedException。

List A10-1 CountupThread 类 (CountupThread.java)

```
public class CountupThread extends Thread {
    //计数器的值
    private long counter = 0;

    //终止请求
    public void shutdownRequest() {
        interrupt();
    }

    //操作
    public void run() {
        try {
            while (!isInterrupted()) {
                doWork();
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    //操作
    private void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
    }

    //终止处理
    private void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}
```

```
main: BEGIN
doWork: counter = 1
doWork: counter = 2
doWork: counter = 3
doWork: counter = 4
doWork: counter = 5
doWork: counter = 6
doWork: counter = 7
doWork: counter = 8
doWork: counter = 9
doWork: counter = 10
doWork: counter = 11
doWork: counter = 12
doWork: counter = 13
doWork: counter = 14
doWork: counter = 15
doWork: counter = 16
doWork: counter = 17
doWork: counter = 18
doWork: counter = 19
doWork: counter = 20
main: shutdownRequest  ←终止请求
doWork: counter = 21   ←不移动到终止处理
main: join
doWork: counter = 22
doWork: counter = 23
doWork: counter = 24
doWork: counter = 25
(以下省略。按 CTRL+C 结束)
```

图 A10-1　执行范例（不予以结束）

问题 10-3 的解答

如 List A10-2 一样。执行范例则如图 A10-2 所示。

List A10-2　保存文件的 CountupThread 类 (CountupThread.java)

```
import java.io.IOException;
import java.io.FileWriter;

public class CountupThread extends Thread {
    // 计数器的值
```

```
private long counter = 0;

// 已经送出终止请求则为 true
private volatile boolean shutdownRequested = false;

// 终止请求
public void shutdownRequest() {
    shutdownRequested = true;
    interrupt();
}

// 判断终止请求是否已经送出
public boolean isShutdownRequested() {
    return shutdownRequested;
}

// 操作
public void run() {
    try {
        while (!shutdownRequested) {
            doWork();
        }
    } catch (InterruptedException e) {
    } finally {
        doShutdown();
    }
}

// 操作
private void doWork() throws InterruptedException {
    counter++;
    System.out.println("doWork: counter = " + counter);
    Thread.sleep(500);
}

// 终止处理
private void doShutdown() {
    System.out.println("doShutdown: counter = " + counter);
    System.out.println("doShutdown: Save BEGIN");
    try {
        FileWriter writer = new FileWriter("counter.txt");
```

```
            writer.write("counter = " + counter);
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("doShutdown: Save END");
    }
}
```

```
main: BEGIN
doWork: counter = 1
doWork: counter = 2
doWork: counter = 3
(中略)
doWork: counter = 19
doWork: counter = 20
main: shutdownRequest
doWork: counter = 20
main: join
doShutdown: Save BEGIN          ←开始保存
doShutdown: Save END            ←结束保存
main: END
```

图 A10-2　执行范例

问题 10-4 的解答

扩充了 GracefulThread 的 CountupThread 类即如同 List A10-3。在此我们采取如下的做法。

- 扩充(extends) GracefulThread 类，声明 CountupThread 类。
- 将 GracefulThread 类的方法 doWork 与 doShutdown 给重载。

CountupThread 的 doWork 与 doShutdown，会从父类（Super Class）的 run 方法根据适当的顺序、时间被调用出。

像这样，用父类的方法来组成处理的架构，将该方法调用出来的方法用子类来安装（实现），再来执行具体的处理的这种 Pattern，我们称之为 Template Method 模式（请参照附录 E[GoF] [Yuki01]）。此类图请参照图 A10-3。

我们称以父类构成处理的框架的方法为 Template Method。在 GracefulThread 中，run 就会变成 Template Method（因此，我们将 run 设为 final 的方法）。

此外，我们称以子类进行处理的方法为 Hook Method（hook 为“挂上”的意思）。这个问题中，run 变成 Template Method，而 doWork 与 doShutdown 变成 Hook Method。

List A10-3 扩充了 GracefulThread 的 CountupThread 类 (CountupThread.java)

```
public class CountupThread extends GracefulThread {
    // 计数器的值
    private long counter = 0;

    // 操作
    protected void doWork() throws InterruptedException {
        counter++;
        System.out.println("doWork: counter = " + counter);
        Thread.sleep(500);
    }

    // 终止处理
    protected void doShutdown() {
        System.out.println("doShutdown: counter = " + counter);
    }
}
```

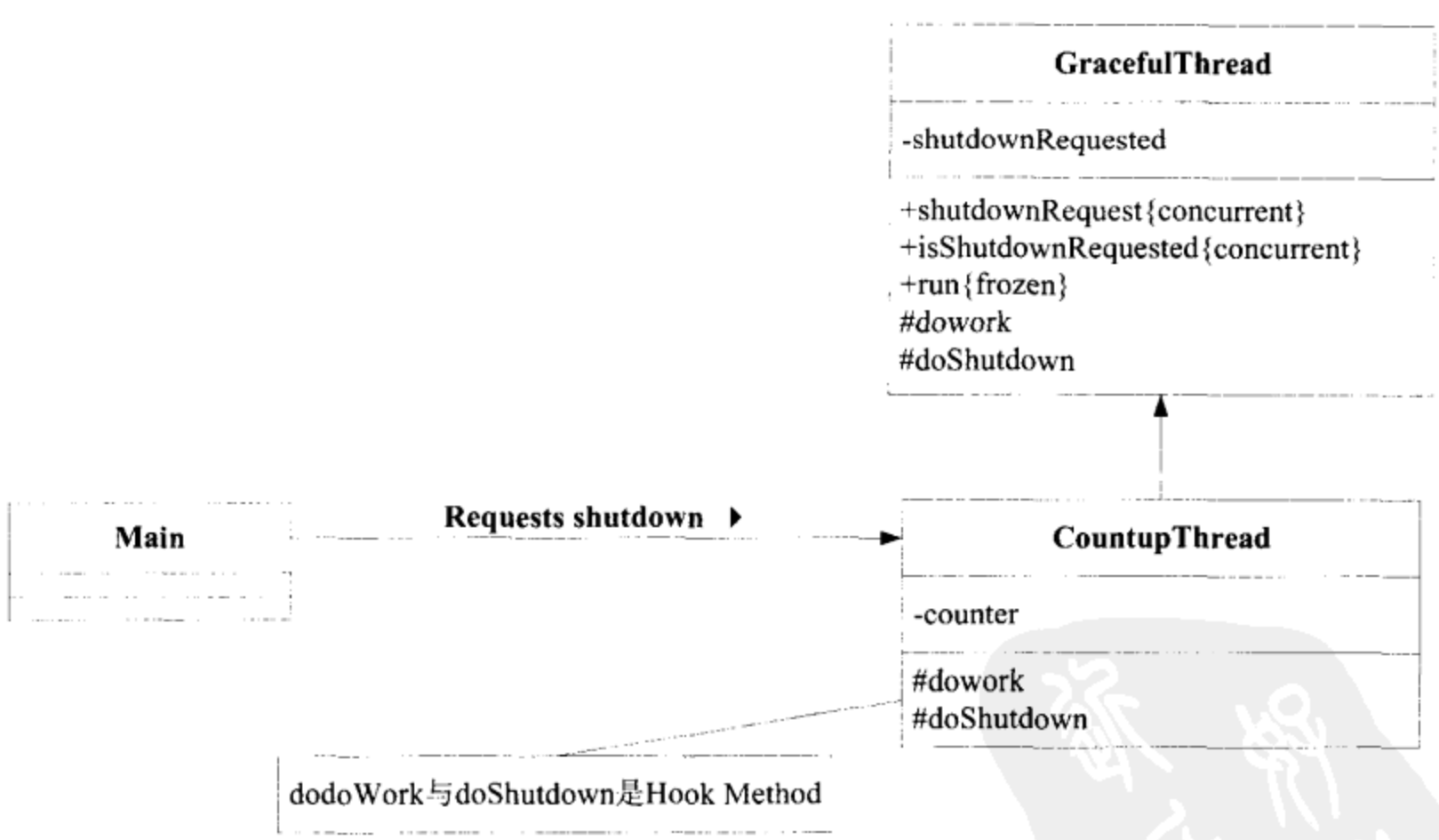


图 A10-3　使用了 Template Method 模式的 CountupThread 的类图

问题 10-5 的解答

解答如 List A10-4 与 A10-5。我们继承问题 10-4 中使用的 GracefulThread 类（List 10-5），建立 ServiceThread 类，从 Service 类启动 ServiceThread 类。这

就是 Thread-Per-Message Pattern（第 7 章）。图 A10-4 就是我们取消执行时的样子。

在连续单击[Execute]按钮时，我们使用 Balking Pattern（第 4 章）在进行 balk。图 A10-5 就是我们进行 balk 时的样子。

List A10-4 Service 类(Service.java)

```
public class Service {
    private static GracefulThread thread = null;

    //开始进行服务(如果是执行中的话就 balk)
    public synchronized static void service() {
        System.out.print("service");
        if (thread != null && thread.isAlive()) {
            // Balking
            System.out.println(" is balked.");
            return;
        }
        // Thread-Per-Message
        thread = new ServiceThread();
        thread.start();
    }

    //暂停服务
    public synchronized static void cancel() {
        if (thread != null) {
            System.out.println("cancel.");
            thread.shutdownRequest();
        }
    }
}
```

List A10-5 ServiceThread 类(ServiceThread.java)

```
public class ServiceThread extends GracefulThread {
    private int count = 0;

    // 执行中
    protected void doWork() throws InterruptedException {
        System.out.print(".");
        Thread.sleep(100);
        count++;
        if (count >= 50) {
```

```
            shutdownRequest();  //   自己结束
        }
    }

    // 结束处理
    protected void doShutdown() {
        System.out.println("done.");
    }
}
```

```
service..............cancel.
done.
```

图 A10-4　在执行服务时单击 1 次[Cancel]按钮

```
serivce.........service is balked.
.........service is balked.
..service is balked.
.............................done.
```

图 A10-5　在执行服务时单击 3 次[Execute]按钮

问题 10-6 的解答

Java 的记忆模式（Memory Model）中，由数个线程共享的字段必须要由 synchronized 来保护，或声明为 volatile。不然的话，一个线程对字段进行的修改，其他线程可能就无法进行引用。

shutdownRequested 字段为数个线程所共有，这数个线程都会连接上这个字段，因此必须以 synchronized 来保护，或是将字段声明成 volatile。线程的同步以及内存的同步请参照附录 B。

在范例文件中，并不需要线程的同步，但是需要内存的同步。因此将 shutdownRequest 字段声明为 volatile。

线程的同步以及内存的同步请参阅附录 B。

问题 10-7 的解答

解答如 List A10-6。在 HanoiThread 类的 doWork 方法中确认 ShutdownRequest 字段，如果出现结束处理的要求，马上就会当场抛出异常 InterruptedException。

执行结果如图 A10-6 所示。从提出结束处理的要求到进入结束处理，所花的时间几乎是 0 秒。

List A10-6 HanoiThread 类 (HanoiThread.java)

```
public class HanoiThread extends Thread {
    // 提出终止处理的请求的话为 true
    private volatile boolean shutdownRequested = false;
    // 提出终止处理的时刻
    private volatile long requestedTimeMillis = 0;

    // 终止请求
    public void shutdownRequest() {
        requestedTimeMillis = System.currentTimeMillis();
        shutdownRequested = true;
        interrupt();
    }

    // 判断终止请求是否已经送出
    public boolean isShutdownRequested() {
        return shutdownRequested;
    }

    // 操作
    public void run() {
        try {
            for (int level = 0; !shutdownRequested; level++) {
                System.out.println("==== Level " + level + " ====");
                doWork(level, 'A', 'B', 'C');
                System.out.println("");
            }
        } catch (InterruptedException e) {
        } finally {
            doShutdown();
        }
    }

    // 操作
    private void doWork(int level, char posA, char posB, char posC)
throws InterruptedException {
        if (level > 0) {
            if (shutdownRequested) {
                throw new InterruptedException();
            }
            doWork(level - 1, posA, posC, posB);
```

```
            System.out.print(posA + "->" + posB + " ");
            doWork(level - 1, posC, posB, posA);
        }
    }

    // 终止处理
    private void doShutdown() {
        long time = System.currentTimeMillis() - requestedTimeMillis;
        System.out.println("doShutdown: Latency = " + time +
" msec.");
    }
}
```

```
main: BEGIN
==== Level 0 ====

==== Level 1 ====
A->B
==== Level 2 ====
A->C A->B C->B
==== Level 3 ====
A->B A->C B->C A->B C->A C->B A->B
==== Level 4 ====
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B A->C
   A->B  C->B
==== Level 5 ====
A->B A->C B->C A->B C->A C->B A->B A->C B->C B->A C->A C->A B->C
   C->A  B->C A->B A->C B->C A->B
C->A C->B A->B C->A B->C B->A C->A C->B A->B  A-> C B->C A->B C->A
   C->B A->B
==== Level 6 ====
A->C A->B C->B A->C B->A B->C A->C A->B C->B C->A B->A C->B A->C
   A->B  C->B A->C
B->A B->C A->C B->A C->B C->A B->A B->C A->C A->B C->B A->C  B->A
   B->C A->C A->B
C->B C->A B->A C->B A->C A->B C->B C->A B->A B->C  A->C B->A C->B
   C->A B->A C->B
A->C A->B C->B A->C B->A B->C A->C A->B  C->B C->A B->A C->B A->C
   A->B C->B
```

（续上图）

```
（中略）
C->A C->B A->B C->A B->C B->A C->A C->B A->B A->C B->C A->B C->A
   C->B  A->B A->C
B->C B->A C->A B->C A->B A->C B->C B->A C->A C->B A->B C->A  B->C
   B->A C->A B->C
A->B A->C B->C A->B C->A C->B A->B A->C B->C B->A  C->A B->C A->B
   A->C B->C B->A
C->A C->B A->B C->A B->C B->A C->A C->B  A->B A->C B->C A->B C->A
   C->B A->B C->A
main: shutdownRequest
main: join
doShutdown: Latency = 0 msec. ←从开始提出结束处理的要求到结束处理开始
   的时间
main: END
```

图 A10-6　执行范例

第 11 章

问题 11-1 的解答

×（1）产生 3 个 ThreadLocal 的实例。

只产生 1 个 ThreadLocal 的实例。其中一个实例被指定到 Log 的类字段（class field）tsLogCollection 中。

○（2）产生 3 个 PrintWriter 的实例。

PrintWriter 的实例被指定到 TSLog 类的实例字段 writer 中。TSLog 的实例与 ClientThread 的数目（3 个）相等，因此 PrintWriter 的实例也是 3 个。

○（3）ThreadLocal 的 set 方法被调用出来 3 次。

ThreadLocal 的 set 方法已在 ClientThread 的线程初次调用出 getTSLog 方法的时候被调用出了。因此，被调用的次数会等同于 ClientThread 的实例数（3 个）。

×（4）ThreadLocal 的 get 方法会被调用 3 次。

ThreadLocal 的 get 方法会在线程调用 println 方法或 close 方法时被调用。println 会被每个线程都调用 10 次，close 会每个线程都被调用 1 次，因此不是 3 次，而是 11 次×3（线程的个数）=33 次。

○（5）产生 3 个 TSLog 的实例。

产生的数目会与 ClientThread 的实例数目（3 个）相同。

×（6）产生 3 个 Log 的实例。

不会产生任何 Log 的实例。

问题 11-2 的解答

（1）的解答：

这是因为 TSLog 类的 println 方法与 close 方法不会被一个以上的线程调用。TSLog 的实例会被 ThreadLocal 类管理，并且当成线程的固有数据来处理。这样的话，某个线程使用的 TSLog 的实例就会固定了。一个线程无法使用因应支持其他线程的 TSLog 的实例。

（2）的解答：

Log 类的 println 方法与 close 方法会被一个以上的线程调用。但是，Log 类并没有不能被数个线程连接的字段。也就是说，并没有应该要保护的部分。因此，没有必要设为 synchronized。

tsLogCollection 字段虽会被数个线程连接，但是 java.lang.ThreadLocal 类是支持多线程的环境（Thread Safe）的类，因此安全性没有问题。

▶▶ 补充一点：java.lang.ThreadLocal 类的安装

ThreadLocal 类是支持多线程的环境（Thread Safe）的类，但并不一定要用 synchronized 来进行共享互斥。是否用 synchronized 进行共享互斥，要视类库（class library）而定。

也有可能不用 synchronized，而安装支持多线程的环境（Thread Safe）的类 ThreadLocal。例如，让 java.lang.Thread 类的内部中保存有 Thread-Specific Storage，而让 ThreadLocal 能连接上该数据。详情请参照如下的网站：

- Exploting ThreadLocal to enhance scalability
  http://www-106.ibm.com/developerworks/java/library/j-threads3.html

问题 11-3 的解答

以下所介绍的解答中，我们采用另外准备监视线程结束的其他线程。

修改过的 ClientThread 类与 Log 类各自是 List A11-1 与 List A11-2。

Log 类中，在新产生出 TSLog 的实例时，就会启动新的线程。Watcher 会使用 join 来监视将 Log 输出到 TSLog 的线程的结束。target 结束后，watcher 的线程会从 join 回来，而在此进行 close。

这样的话，TSLog 的实例好像会被一个以上的线程（target 与 watcher）给连接上。但是，当 watcher 调用 close 方法的时候，target 线程已经结束了，因此不必担心会被数个线程连接上。

执行范例如图 A11-1 所示。

List A11-1 修改后的 ClientThread 类(ClientThread.java)

```
public class ClientThread extends Thread {
    public ClientThread(String name) {
        super(name);
    }
    public void run() {
        System.out.println(getName() + " BEGIN");
        for (int i = 0; i < 10; i++) {
            Log.println("i = " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        // Log.close()  已经不需要 Log.close()
        System.out.println(getName() + " END");
    }
}
```

List A11-2 修改后的 Log 类(Log.java)

```
public class Log {
    private static final ThreadLocal tsLogCollection = new
ThreadLocal();

    // 写 log
    public static void println(String s) {
        getTSLog().println(s);
    }

    // 关闭 log
    public static void close() {
        getTSLog().close();
    }

    // 获得线程固有的 log
    private static TSLog getTSLog() {
        TSLog tsLog = (TSLog)tsLogCollection.get();

        // 如果该线程是第一次调用，就重建注册
        if (tsLog == null) {
            tsLog = new TSLog(Thread.currentThread().getName() +
"-log.txt");
```

```
            tsLogCollection.set(tsLog);
             starWatcher(tsLog);
        }

        return tsLog;
    }
}
// 启动等待线程结束的线程
private static void starWatcher(final TSLog tsLog) {
    // 被监视的线程
    final Thread target = Thread.currentThread();
    // 进行监视的线程
    final Thread watcher = new Thread() {
        public void run() }
            System.out.println("startWatcher for " + target.getName() +
" BEGIN");
            try {
                target.join();
            } catch(InterruptedException e) {
            }
            tsLog.close();
            System.out.println("startWatcher for " + target.getName() +
" END");
        }
    };
    //开始监视
    watcher.start();
 }
}
```

```
Alice BEGIN
Bobby BEGIN
Chris BEGIN
startWatcher for Alice BEGIN   ←Alice 用的 watcher 线程开始
startWatcher for Bobby BEGIN   ←Bobby 用的 watcher 线程开始
startWatcher for Chris BEGIN   ←Chris 用的 watcher 线程开始
Bobby END
startWatcher for Bobby END     ←Bobby 用的 watcher 线程结束
Alice END
startWatcher for Alice END     ←Alice 用的 watcher 线程结束
Chris END
startWatcher for Chris END     ←Chris 用的 watcher 线程结束
```

图 A11-1　执行范例

问题 11-4 的解答

因为在此程序中，正在执行构造器的线程是主线程。

正在执行 ClientThread 的 run 方法的线程，是由主线程启动的新线程，与主线程不同。

这个问题是用来让读者确认自己是否已了解“执行 ClientThread 的构造器的线程与执行 run 方法的线程相异”。

注意：执行题目中的程序，就会建立出 main-log.txt 的文件。在文件名称加上 main，是因为主线程的名称是 main。但是，主线程并没有调用 close，因此，很有可能"constructor is called"这个字符串并没有被保存下来。

问题 11-5 的解答

最常使用的是“线程的名称”。线程的名称写在 java.lang.Thread 的实例内，可以用 getName 方法取得。

现在线程的名称就可以用 Thread.currentThread().getName()来取得。

线程的名称之外，线程的固有数据还包括以下各项。括号中是为取得该项数据的方法。

- 线程的优先级（getPriority）
- 线程组（getThreadGroup）
- 中断的状况（isInterrupted）
- 是否为 Daemon 线程（isDaemon）
- 是否活着（isAlive）

第 12 章

问题 12-1 的解答

○（1）Proxy 类与 Servant 类安装了 ActiveObject 接口。

×（2）MakerClientThread 类的线程所执行的 makeString 方法，被以 Servant 类安装。

MakerClientThread 类的线程所执行的 makeString 方法，被以 Proxy 类安装，而非被以 Servant 类安装。

×（3）每次调用 displayString 方法时，都会制作出新的线程。

×（4）由于数个线程都可以连接上 Servant 类，因此需要进行共享互斥。

可以连接上 Servant 类的线程只有 SchedulerThread 类的线程一个，因此无须

进行共享互斥。

○（5）ActivationQueue 类的 putRequest 方法会被数个线程调用。

putRequest 方法会被 MakerClientThread 类的线程与 DisplayClientThread 类的线程所调用。

×（6）ActivationQueue 类的 takeRequest 方法会被数个线程调用。

takeRequest 方法只会被 SchedulerThread 类的单线程所调用。

○（7）MakerClientThread 类的线程在调用出 getResultValue 方法时，如果还没制作出字符串的话，线程会 wait。

○（8）Servant 类的 makeString 方法随着所制作的字符串越长而处理时间就越久。

×（9）MakerClientThread 的线程调用出 makeString 时，参数值 count 越大则从方法回来的时间越久。

MakerClientThread 的线程调用出 makeString 方法时，并不会实际上制作字符串，而只是制作 MakeStringRequest 类，再置入队列中而已，因此花费的时间不受参数的 count 的值影响。

问题 12-2 的解答

解答即如 List A12-2~List A12-4。方式即按照本章的“新增方法”的步骤进行操作。提示，java.math.BigInteger 是 immutable 的类（请参考第 2 章）。

List A12-1 修改后的 ActiveObject 界面(ActiveObject.java)

```
package activeobject;

public interface ActiveObject {
    public abstract Result makeString(int count, char fillchar);
    public abstract void displayString(String string);
    public abstract Result add(String x, String y);
}
```

List A12-2 修改后的 AddRequest 类(AddRequest.java)

```
package activeobject;

class AddRequest extends MethodRequest {
    private final String x;
    private final String y;
    public  AddRequest(Servant  servant,  FutureResult  future,
String x, String y) {
        super(servant, future);
        this.x = x;
```

```
        this.y = y;
    }
    public void execute() {
        Result result = servant.add(x, y);
        future.setResult(result);
    }
}
```

List A12-3 修改后的 Proxy 类(Proxy.java)

```
package activeobject;

class Proxy implements ActiveObject {
    private final SchedulerThread scheduler;
    private final Servant servant;
    public Proxy(SchedulerThread scheduler, Servant servant) {
        this.scheduler = scheduler;
        this.servant = servant;
    }
    public Result makeString(int count, char fillchar) {
        FutureResult future = new FutureResult();
        scheduler.invoke(new MakeStringRequest(servant, future,
count, fillchar));
        return future;
    }
    public void displayString(String string) {
        scheduler.invoke(new DisplayStringRequest(servant,
string));
    }
    public Result add(String x, String y) {
        FutureResult future = new FutureResult();
        scheduler.invoke(new AddRequest(servant, future, x, y));
        return future;
    }
}
```

List A12-4 修改后的 Servant 类 (Servant.java)

```
package activeobject;

import java.math.BigInteger;

class Servant implements ActiveObject {
```

```
public Result makeString(int count, char fillchar) {
    char[] buffer = new char[count];
    for (int i = 0; i < count; i++) {
        buffer[i] = fillchar;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
    return new RealResult(new String(buffer));
}
public void displayString(String string) {
    try {
        System.out.println("displayString: " + string);
        Thread.sleep(10);
    } catch (InterruptedException e) {
    }
}
public Result add(String x, String y) {
    String retvalue = null;
    try {
        BigInteger bigX = new BigInteger(x);
        BigInteger bigY = new BigInteger(y);
        BigInteger bigZ = bigX.add(bigY);
        retvalue = bigZ.toString();
    } catch (NumberFormatException e) {
        retvalue = null;
    }
    return new RealResult(retvalue);
  }
}
```

问题 12-3

解说以下两个解答。

解答 1 中，会将查找结果的 URL 以 Future pattern（第 9 章）返回。但是，这个解答使用了新的线程，以让调用方（MyFrame）能等待，直到值被设置到 Future（MyFrame 类的 searchWord 方法）。

解答 2 中，我们用 MyFrame 类的 display 方法通知查找结果。这里的解答利用了 Swing 框架工作（Frame Work）的 SwingUtilites.invokeLater，正好扮演了 Active

Object 模式（第 12 章）的 Scheduler 参与者与 ActivationQueue 参与者的工作这一点。MyFrame 类就变成了对应于 Swing 框架工作的 Proxy 参与者了。

解答 2 中出现了两个“主动的对象”。一个在 active object 包中，提供单字查找的服务.....（A）。另一个则由 Swing 框架工作与 MyFrame 类所构成，让 MyFrame 的 TextArea 类中显示字符串.....（B）。

（A）与（B）的两个“主动的对象”彼此调用出相当于异步消息的方法。这个所表示的意思就是解答 2 中“主动的对象”互相交换异步的消息。

◆ 解答 1：用 *Future* 模式返回查找的 *URL*

ActiveObjectFactory、SchedulerThread、ActivationQueue、MethodRequest、Result、FutureResult、RealResult 的各类跟范例程序的是一样的。

List A12-5 修改后的 Main 界面(Main.java)

```
public class Main {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

List A12-6 修改后的 MyFrame 类(MyFrame.java)

```
import java.io.IOException;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.SwingUtilities;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JPanel;

import activeobject.ActiveObjectFactory;
import activeobject.ActiveObject;
import activeobject.Result;

public class MyFrame extends JFrame implements ActionListener {
```

```
    private final JTextField textfield = new JTextField("word", 10);
    private final JButton button = new JButton("Search");
    private final JTextArea textarea = new JTextArea(20, 30);
    private final ActiveObject activeObject = ActiveObjectFactory.
createActiveObject();
    private final static String NEWLINE = System.getProperty
("line.separator");

    public MyFrame() {
        super("ActiveObject Sample");
        getContentPane().setLayout(new BorderLayout());

        // North
        JPanel north = new JPanel();
        north.add(new JLabel("Search:"));
        north.add(textfield);
        north.add(button);
        button.addActionListener(this);

        // Center
        JScrollPane center = new JScrollPane(textarea);

        // Layout
        getContentPane().add(north, BorderLayout.NORTH);
        getContentPane().add(center, BorderLayout.CENTER);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    // [Search]按钮被按着时
    public void actionPerformed(ActionEvent e) {
        searchWord(textfield.getText());
    }

    // 显示
    private void println(String line) {
        textarea.append(line + NEWLINE);
    }

    // 查找
```

```
    private void searchWord(final String word) {
        // 查找的调用
        final Result result = activeObject.search(word);
        println("Searching " + word + "...");
        // 等待查找结果的线程
        new Thread() {
            public void run() {
                // 等待结果
                final String url = (String)result.getResultValue();
                // 获得结果了，请 Event Dispatching Thread 予以显示
                SwingUtilities.invokeLater(
                    new Runnable() {
                        public void run() {
                          MyFrame.this.println("word = " + word + ",
URL = " + url);
                        }
                    }
                );
            }
        }.start();
    }
```

List A12-7 修改后的 ActiveObject 界面(ActiveObject.java)

```
package activeobject;

public interface ActiveObject {
    public abstract Result search(String word);
}
```

List A12-8 修改后的 Proxy 类(Proxy.java)

```
package activeobject;

class Proxy implements ActiveObject {
    private final SchedulerThread scheduler;
    private final Servant servant;
    public Proxy(SchedulerThread scheduler, Servant servant) {
        this.scheduler = scheduler;
        this.servant = servant;
    }
```

```
    public Result search(String word) {
        FutureResult future = new FutureResult();
        scheduler.invoke(new    SearchRequest(servant,    future,
word));
        return future;
    }
}
```

List A12-9 修改后的 SearchRequest 类(SearchRequest.java)

```
package activeobject;

class SearchRequest extends MethodRequest {
    private final String word;
    public SearchRequest(Servant servant, FutureResult future,
String word) {
        super(servant, future);
        this.word = word;
    }
    public void execute() {
        Result result = servant.search(word);
        future.setResult(result);
    }
}
```

List A12-10 修改后的 Servant 类 (Servant.java)

```
package activeobject;

public class Servant implements ActiveObject {
    public Result search(String word) {
        System.out.print("search(" + word + ")");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {

                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("found.");
        String url = "http://somewhere/" + word + ".html"; // dummy
URL
```

```
            return new RealResult(url);
        }
    }
```

◆ 解答2：将查找到的URL返回作为对MyFrame类的调用(叫调用 MyFrame)

ActiveObjectFactory、SchedulerThread、ActivationQueue、MethodRequest、Result、FutureResult、RealResult 的各类与范例程序的是一样的。

List A12-11 修改后的 Main 界面(Main.java)

```
public class Main {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

List A12-12 修改后的 MyFrame 类(MyFrame.java)

```
import java.io.IOException;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.SwingUtilities;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import javax.swing.JScrollPane;
import javax.swing.JPanel;

import activeobject.ActiveObjectFactory;
import activeobject.ActiveObject;
import activeobject.Result;
import activeobject.Display;

public class MyFrame extends JFrame implements Display,
ActionListener {
    private final JTextField textfield = new JTextField("word",
10);
    private final JButton button = new JButton("Search");
```

```
    private final JTextArea textarea = new JTextArea(20, 30);
    private final ActiveObject activeObject = ActiveObjectFactory.
createActiveObject();
    private final static String NEWLINE = System.getProperty
("line.separator");

    public MyFrame() {
        super("ActiveObject Sample");
        getContentPane().setLayout(new BorderLayout());

        // North
        JPanel north = new JPanel();
        north.add(new JLabel("Search:"));
        north.add(textfield);
        north.add(button);
        button.addActionListener(this);

        // Center
        JScrollPane center = new JScrollPane(textarea);

        // Layout
        getContentPane().add(north, BorderLayout.NORTH);
        getContentPane().add(center, BorderLayout.CENTER);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
    }

    // [Search范例程序的执行步骤范例程序的执行步骤]按钮被按着时
    public void actionPerformed(ActionEvent e) {
        searchWord(textfield.getText());
    }

    // 显示
    private void println(String line) {
        textarea.append(line + NEWLINE);
    }

    // 查找
    private void searchWord(String word) {
        // 查找的调用
```

```
            activeObject.search(word, this);
            println("Searching " + word + "...");
        }

        // 显示
        public void display(final String line) {
            // 请 Event Dispatching Thread 予以显示
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        MyFrame.this.println(line);
                    }
                }
            );
        }
}
```

List A12-13 修改后的 Display 界面(Display.java)

```
package activeobject;

public interface Display {
    public abstract void display(String line);
}
```

List A12-14 修改后的 ActiveObject 界面(ActiveObject.java)

```
package activeobject;

public interface ActiveObject {
    public abstract void search(String word, Display display);
}
```

List A12-15 修改后的 Proxy 类(Proxy.java)

```
package activeobject;

class Proxy implements ActiveObject {
    private final SchedulerThread scheduler;
    private final Servant servant;
    public Proxy(SchedulerThread scheduler, Servant servant) {
        this.scheduler = scheduler;
        this.servant = servant;
```

```
    }

    public void search(String word, Display display) {
        scheduler.invoke(new SearchRequest(servant, word, display));
    }
}
```

List A12-16 修改后的 SearchRequest 类(SearchRequest.java)

```
package activeobject;

class SearchRequest extends MethodRequest {
    private final String word;
    private final Display display;
    public SearchRequest(Servant servant, String word, Display display) {
        super(servant, null);
        this.word = word;
        this.display = display;
    }
    public void execute() {
        servant.search(word, display);
    }
}
```

List A12-17 修改后的 Servant 类(Servant.java)

```
package activeobject;

public class Servant implements ActiveObject {
    public void search(String word, Display display) {
        System.out.print("search(" + word + ")");
        for (int i = 0; i < 50; i++) {
            System.out.print(".");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
        System.out.println("found.");
        String url = "http://somewhere/" + word + ".html"; // dummy URL
        // 将结果 display 出来
        display.display("word = " + word + ", URL = " + url);
    }
}
```

PDG

附录B

Java的内存模型

Java 的内存模型

在附录 B，我们要来解说 Java 的内存模型。

Java 的内存模型，亦即 Java 执行处理系统所设置的内存模型，规定于“Java 语言规格书第 2 版”（参照附录 E[JLS]）的第 17 章“线程（thread）与锁定（lock）”，但这个规格在解读上十分困难。

因此，本书将在下面针对以下重点进行详细解说。

- 主存储器（main memory）与工作存储器（working memory）
- synchronized 的两项功能
- volatile 的两项功能
- Double Checked Locking Pattern 的危险性

▶▶ 补充一点：阅读前的建议

在您开始阅读附录 B 之前，应该先熟记下列多线程的基本规则。

- 数个线程所共有的字段，由 synchronized 或 volatile 来保护
- synchronized 负责线程的共享互斥
- volatile 不负责线程的共享互斥

附录 B 仅说明此项规定的理由。

主存储器与工作存储器

Java 的内存模型（memory model）分为主存储器（mail memory）与工作存储器（working memory）两种，各自担负以下的功能。

主存储器就是实例位置所在的区域，所有的实例都存在于主存储器内。尤其，实例所拥有的字段即位于主存储器内的区域。主存储器为所有的线程所共有。

工作存储器为各个线程所拥有的作业区，所有的线程都有其专用的工作存储器。在工作存储器内，存在有主存储器中必要部分的拷贝，称之为工作拷贝（working copy）。

Java 内存模型的概念图请参见图 B-1。

▶▶ 补充说明：与硬件的主存储器无关

Java 内存模型的主存储器，与计算机硬件上所讲的主存储器无关。我们所谈的主存储器与工作存储器，都只是为了解说 Java 内存模型所假设出来的概念。

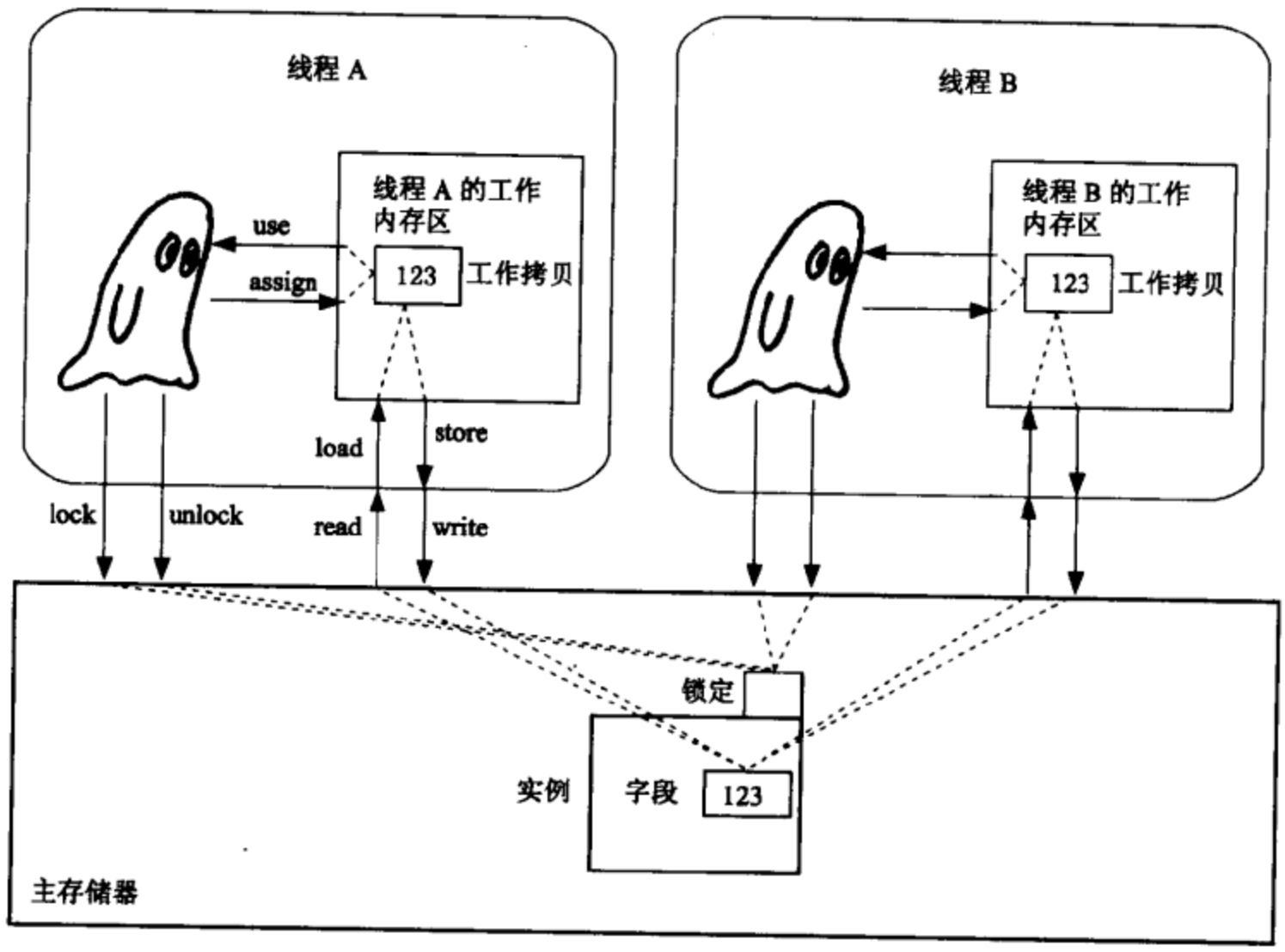


图 B-1　Java 内存模型的概念图

字段（field）的引用

线程无法对主存储器直接进行操作，因此它也无法直接引用字段的值。

当线程欲引用字段的值时，会一次将值从主存储器拷贝到工作存储器上（相当于图 B-1 中的 read→load）。通过此项拷贝所得到的值，会成为工作拷贝（working copy）。拷贝完成后，线程就会引用工作拷贝（在图 B-1，相当于 read→load）。

当同一线程再度引用同一字段的值时，会发生什么事呢？线程或许会引用刚才所制作的工作拷贝（限 use）、也可能会先重新从主存储器拷贝到工作存储器，然后才引用工作拷贝（read→load→use）。会出现哪一种状况，是由 Java 执行处理系统来决定。

指定字段

线程无法直接对主存储器进行操作，因此它也无法将值直接指定给字段。

当线程欲将值指定给字段时，会一次将值指定给位于工作存储器上的工作拷贝（assign）。指定完成后，工作拷贝的内容便会映像到主存储器（store→write）。至

于何时会映像到主存储器，则是由 Java 执行处理系统来决定（各项规定记载于附录 E[JLS]的第 17 章）。

当同一线程反复指定至同一字段时，会发生什么事？在反复进行指定的期间，线程有可能只会对工作拷贝进行指定（限 assign）。此时，只有指定的最后结果会拷贝到主存储器。另外，线程也有可能在每次进行指定时，进行拷贝到主存储器的操作（assign→store→write）。会出现哪一种状况，还是由 Java 执行处理系统来决定。

主存储器好比是大家都看得到的黑板；而工作存储器则像是每个人的笔记本

为了帮助大家的理解，我们来打个比方。主存储器就相当于教室里面的“黑板”；而工作存储器则好比是学生（线程）们各自拥有的“笔记本”。

学生（线程）无法直接在黑板（主存储器）上计算，只能将黑板上的内容（字段的值）抄写在自己所拥有的笔记本（工作存储器）上。学生（线程）利用抄写下来的内容（工作拷贝）来进行计算，并将结果写在笔记本（工作存储器）上。然后在适当的时机，再写在黑板（主存储器）上。

学生所能够看到的部分，只有抄写了黑板上内容的自己的笔记本而已，没有办法窥到其他同学的笔记本。如果想要把自己的计算结果拿给其他同学看，就不能只写在自己的笔记本上，而必须写在黑板上才行。

另外，当自己想要利用其他同学的计算结果时，必须将自己笔记本的内容更新成最新状态。因为自己的笔记本上所记载的内容，可能还是之前黑板上的内容。

为何要把这么麻烦的东西规定在 Java 语言规格书呢？这是因为要预留 Java 执行处理系统的升级空间。如果规定线程所计算出来的结果都必须映像至主存储器，就没有可以再把方法中的计算顺序、指定顺序、不需要的指定等予以最佳化的空间了。

▶▶ 补充一点：自变量与局部变量（local variable）位于何处？

实例的字段在主存储器上，那么方法（method）的自变量与局部变量又位于何处？由于方法的自变量与局部变量只由一个线程进行访问，因此无论是位于主存储器或是工作存储器上，都不会发生问题。不过我们把它想成是位于工作存储器上，可能比较容易了解。

六个操作（action）

图 B-1 所出现的 read、write、use、assign、lock、unlock 等方法，在附录 E 的[JLS]里称为“操作（action）”。这 6 个 action 都为不可分割（indivisible）的最小单

位（atomic）。在[JLS]的第 17 章中，有规定这 6 个 action 应以哪种顺序来执行。Java 的执行处理系统，会在[JLS]所规定的范围内，将这些 action 费心加以排序，以便予以最佳化。

- read/write　负责主存储器与工作存储器间的 atomic action。
  - －read　负责从主存储器拷贝到工作存储器
  - －write　与上述相反，负责从工作存储器拷贝到主存储器
- use/assign　负责线程的执行引擎与工作存储器间的 atomic action。
  - －use　表示线程引用工作存储器的值
  - －assign　意指线程将值指定给工作存储器
- lock/unlock　对于伴随着实例而来的锁定，进行 atomic action。
  - －lock　表示线程取得锁定
  - －unlock　意指线程解除锁定

synchronized 的两项功能

在了解主存储器与工作存储器所扮演的角色后，我们再针对 synchronized 来思考一下。Synchronized 原意为“同时取得”，这里的“同步”分为“线程的同步”与“内存的同步”两种。

线程的同步在本书中曾数次出现。使用 synchronized 来制作临界区间（critical section）时，该区间仅能提供一个线程进行操作。在一个线程作业期间，其他的线程必须在 critical section 的入口处等待。由 Synchronized 所规定的范围，来控制线程的操作。此为线程的同步。

至于内存的同步，在本书中很少提到，这是因为在未了解主存储器与工作存储器前，没有办法解说什么是内存的同步，因此下面才要针对它们来进行解说。

另外，以下所陈述的内容，可适用于 synchronized 方法与 synchronized block 两者。因此便不一一写出“synchronized 方法与 synchronized block”，而仅写成 synchronized。例如“进入 synchronized”，指的就是“进入 synchronized 方法或 synchronized block”的意思。

欲进入 synchronized 时

当线程要进入 synchronized 时，该线程所拥有的工作存储器的内容会被“清除”。亦即将会进行下一个处理。

(A)“强制写入主存储器”

欲进入 synchronized 时，如果工作存储器在有未映像到主存储器的工作拷贝，

该内容就会被强制写入主存储器（store→write）。因此，之前的计算结果会被全部写入主存储器内，成为其他线程可以看得见（visible）的状态。

(B)“工作存储器的释放”

欲进入 synchronized 时，工作存储器上的工作拷贝会被全部丢弃。之后，欲引用主存储器上的值的线程，必定会从主存储器将值拷贝到工作拷贝（read→load）。总归一句话，工作存储器的内容与主存储器的内容会予以同步。

▶▶ 补充一点：强制写入可以吗？

或许有些读者会担心：“如果工作拷贝的内容会强制写入主存储器，那么其他线程所写入的内容岂不是就被覆盖掉了？”。有这一层考虑是无可厚非，但情况还不到需要担心的程度。

数个线程所共有的字段，都会进入到 synchronized 里面。此时，指定字段的操作不可能会在 synchronized 的外面执行。因此，当进入 synchronized 时，共有字段的值根本不可能被写入主存储器。

当然，如果对于共有字段的指定让 synchronized 保护得忘了，那么就会出现之前所担心的状况。这个相当于我们在第 1 章所谈过的 race condition。

欲退出 synchronized 时

当线程欲退出 synchronized 时，会执行相同于进入 synchronized 时，（A）“强制写入主存储器”之处理作业。通过这项处理，所有在 synchronized 中执行的指定结果，都会映像在主存储器上。

不过在退出 synchronized 时，则不会执行（B）“工作存储器的释放”（意思就是说，已经退出 synchronized 的线程可能会继续使用自己的工作存储器）。

何时会进行内存的同步，相信也是大家关注的问题。内存的同步，会在线程“欲进入”与“欲退出”synchronized 时进行。如果是“正在 synchronized 内”或“正在 synchronized 外”时，便不保证一定会进行工作存储器与主存储器之间的同步。执行处理系统或许会根据这个来进行最佳化。

在 Java 的内存模型上，已被写入工作存储器的内容何时会映像到主存储器？关于这个问题，通常（如果没有 synchronized、volatile 的话）是没有办法保证的。有可能立刻映像，也有可能暂时不作映像（至少在程序语言规格上是如此）。

但是如果有 synchronized 的话，情况就不同了。当线程欲进出 synchronized 时，便会将自己工作存储器的内容完全映像到主存储器。

▶▶ **补充一点：单线程，可不必担心内存的同步问题**

线程会经常性地通过工作存储器上的工作拷贝来操作主存储器，因此在单线程运行期间，线程所盯住的目标，不会发生要区别它是主存储器、还是工作存储器上的工作拷贝之问题。单线程运行期间，完全没有必要使用 synchronized 来进行内存的同步。因此在单线程运行时，我们不需要担心“现在这项计算已经映像到主存储器了没有？”。

volatile 的二项功能

在了解主存储器与工作存储器所扮演的角色后，接下来要谈是 volatile。volatile 具有以下两种功能。

- 进行内存的同步
- 以 atomic 的方式来进行 long、double 的指定

volatile 仅进行内存的同步

内存的同步操作会经常性地对于写着 volatile 的字段来进行。volatile 在前面曾经说明过的两种同步当中，会自动执行内存的同步工作。不过进行的只有内存的同步，而不会做线程的同步。

当线程欲引用 volatile 字段的值时，通常都会发生从主存储器到工作存储器的拷贝操作。而相反地，将值指定给写着 volatile 的字段后，工作存储器的内容通常便会映像到主存储器。

volatile 会以 atomic 的方式来进行 long、double 的指定

我们在第 1 章中曾经提过，指定给 long 型与 double 型的操作不保证会以 atomic 的方式来进行。但如果是 long 型或 double 型的字段，只要有写着 volatile，就可以用 atomic 的方式来进行指定工作。

▶▶ **补充一点：atomic 与内存的同步不同**

若欲指定至 long 型、double 型“以外”的字段，不会以 atomic 的方式来进行。举例来说：

```
int field;
```

假设有上述这样一个 int 型的字段存在，当要指定至此字段时，便会以 atomic 的方式来进行。

这里请勿将 atomic 与内存的同步混淆。int 型的指定字段虽然是采用 atomic

方式，但是所指定的值并不保证会立刻映像到主存储器。如果字段中没有写着volatile或指定的线程并未进出synchronized，就不会进行内存的同步。

因此，被数个线程进行变更/引用的字段，即使是int型，也必须以synchronized或volatile来保护。

Double Checked Locking pattern 的危险性

Double Checked Locking Pattern是改善Single Threaded Execution Pattern（第1章）操作（performance）的方法之一，也可以称为test-and-test-and-set（参照附录E[lea]）。

不过，如果要在Jave中使用Double Checked Locking Pattern，必须注意安全性的问题。本节将解说Double Checked Locking这种模式的危险性，同时确认Java内存的模型。

范例程序

以下提出3种方法来安装名为MySystem的类。此类具有以下属性：

- MySystem类的实例是惟一的[注1]
- MySystem类的实例可通过类方法getInstance来取得
- MySystem类的实例，会将java.util.Date类的实例保存在字段（date）。该值为实例被制作出来的时间
- 字段date的值可通过MySystem类的实例方法getDate来取得

（1）利用Single Threaded Execution Pattern来安装

List B-1是使用Single Threaded Execution Pattern（第1章）来安装MySystem类。

在类方法getInstance中，会检查类字段instance的值是否为null。如果是的话，就会以new来产生MySystem的实例。

考虑到同时被数个线程调用的状况，getInstance方法会成为synchronized。因此，即使有数个线程在调用getInstance，也可以保证MySystem的实例是惟一的。

List B-1如默认般来操作，但因getInstance已经成为synchronized方法，在performance上便不为人所喜爱。

注1：这个会变成Singleton Pattern（参照附录E[GoF][Yuki01]。

List B-1 使用 Single Threaded Execution Pattern 的 MySystem 类（MySystem.java）

```
import java.util.Date;

public class MySystem{
    private static MySystem instance = null;
    private Date date = new Date ( );
    private MySystem ( ) {
    }
    public Date getDate ( ) {
    }
        return date;
    }
    public static synchronized MySystem getInstance ( ) {
        if (instance = = null ) {
            instance = new MySystem( );
        }
        return instance;
    }
}
```

程序收录在下载文件夹 AppendixB/Others/SingleThreadedExecution 内

List B-2 Main 类（Main.java）

```
public class Main extends Thread {
    public static void main(String[ ] args) {
        new Main( ).start( );
        new Main( ).start( );
    }
    public void run( ) {

System.out.println(Thread.currentThread( ).getName( )+":"+
MySystem.getInstance( ).getDate( ));
    }
}
```

程序收录在下载文件夹 AppendixB/Others/SingleThreadedExecution 内

（2）Double Checked Locking Pattern

我们在前面已经谈过，Double Checked Locking Pattern 可以用来改善

performance 的问题。在 Double Checked Locking Pattern 下安装 MySystem 类的情况表示于 List B-3。

getInstance 方法并未成为 synchronized 方法，取而代之的是，在 if 语句中含有 synchronized 块。

我们来读一下程序代码。在(a)之 if 语句的条件检查下(第一次 test)，当 instance 等于 null 时，便会进入（b）的 synchronized 块。取得锁定（lock）的对象为 MySystem.class，亦即 MySystem calss 的类对象。

（a）的条件检查是位于临界区间之外，因此会在（c）重新执行条件检查（第二次 test)。当 instance 确实相等于 null 时，便会指定(d)中 MySystem 的实例(set)。(d)的实例是在(b)～(c)的临界区间中产生，因此不会产生两个以上的 MySystem 实例。

只有在（a）的条件测试下 instance 等于 null 时，才会进入（b）的 synchronized 块。因此，getInstance 第二次以后的调用，几乎都不会进入 synchronized 块。故无须担心 performance 不佳的问题。

以上是 Double Checked Locking Pattern 的梗概。在目前所语句的范围内并没有错误。可是在 Java 的规格上，List B-3 的程序有可能无法正确地操作。

问题的所在一言以蔽之，就是“当使用 getInstance 的返回值来调用 getData 方法时，date 字段有可能不会被初始化”。

List B-3 虽使用 Double Checked Locking Pattern，但不保证操作正确的 MySystem 类（MySystem.java）

```
//不保证操作正确
import java.util.Date;

public class MySystem{
    private static MySystem instance = null;
    private Date date = new Date( );
    private MySystem( ) {
    }
    public Date getDate( ) {
        return date;
    }
    public static Mysystem getInstance( ) {
        if (instance = = null) {                    //(a)第一次 test
            synchronized (MySystem.class) { //(b)进入 synchronized 块
                if (instance = = null) {            //(c)第二次 test
                    instance = new MySystem( );  //(d)set
                }
```

```
            }                          //(e)退出 synchronized 块
        }
        return instance;               //(f)
    }
}
```

程序收录在下载文件夹 AppendixB/OthersDoubleCheckedLocking 内

我们以两个线程欲同时执行 getInstance 的情况为例，针对无法顺利操作的案例进行解说（图 B-2）。

线程 A	线程 B
(A-1) 在 (a) 判断 instance = = null	
(A-2) 在 (b) 进入 synchronized block	
(A-3) 在 (c) 判断 instance = = null	
(A-4) 在 (d) 制作 MySystem 的实例，指定给 instance 字段	
<<<<<线程在此更新>>>>>	(B-1) 在 (a) 判断 instance = = null
	(B-2) 在 (f) 将 instance 的值设为 getInstance 的返回值
	(B-3) 调用 getInstance 返回值之 getDate 方法

图 B-2　List B-3 无法正确操作的案例

仔细看图 B-2 中的 (A-4)。这里写着“在 (d) 制作 MySystem 的实例，指定给 instance 字段”，即

```
instance = new MySystem( );的部分。
```

所以知道就是在此处制作 MySystem 的实例。在制作 MySystem 的实例时，new Date()的值会指定给实例字段 date。字段内会有值代入。这是因为线程 A 实例上会对于工作存储器上的工作拷贝进行指定工作（assign）。若线程 A 退出 synchronized block，date 字段的值便会映像到主存储器（store→write）。但在退出前，date 字段的值不保证会映像到主存储器（有可能会映像、也可能不会）。

我们先设置线程 B 在 (B-1) 为 instance != null。如此，线程 B 便不会进入 synchronized block，而会立刻将 instance 的值当作返回值予以返回（return）。然后，线程 B 会在 (B-3) 调用 getInstance 返回值的 getDate 方法。GetDate 的返回值为 date 字段的值。线程 B 会引用 date 字段的值。这是因为线程 B 会引用（use）自己工作存储器的值。如果工作拷贝不存在，便会从主存储器将值拷贝到工作存储器后再引用（rean→load→use）。

不过读者可能已经察觉到，线程 A 欲指定至 date 字段的 new Date( ) 的值，有可能并未映像到主存储器。也就是说，线程 B 会使用未完成初始化它 MySystem 的实例。

▶▶ 补充一点：为何看得见 instance 字段？

细心的读者，对于在 (B-1) 线程 B 正确引用 instance 字段一事，内心可能抱持着这样的怀疑：date 字段的值明明没有映像到主存储器，instance 字段的值却还会映像至主存储器，这不是太奇怪了吗？然而这在 Java 的规格上一点也不奇怪。在没有 synchronized、volatile 时，何时从工作存储器映像到主存储器，是由执行处理系统来决定。当存在两个字段时，可能会根据指定（assign）给它们的顺序、也可能采用映像到主存储器（store→write）的顺序。

▶▶ 补充一点：真的会发生吗？

这里所语句的状况是否会真的发生，是由 Java 的执行处理系统来决定。总之这在 Java 规格上，就算发生也不是什么新鲜事。

正确的解决方法

解决 Single Threaded Execution Patten（第 1 章）performance 不佳、以及 Double Checked Locking Pattern 会造成危险的方法，表示于 List B-4。

此处系将 new MySystem ( ) 当作类字段 instance 的初期值来予以指定。这在 Java 规格上，系由 thread safe 来执行（参照附录 E[JLS]的 §12.4.2 'Detailed Initilalization Procedure'）。

类方法 getInstance 变成不是 synchronized，而一直简化了。

List B-4 正确操作的 MySysten 类（MySystem.java）

```
import java.util.Date;

public class MySystem {
    private static Mystem instance = new MySystem( );
    private Date date = new Date( );
    private MySystem( ) {
    }
    public Date getDate( ) {
        return date;
    public static MySystem getInstance( ) {
        return instance;
    }

}
```

程序收录在下载文件夹 AppendixB/Others/Correct 内

有关 Double Checked Locking Pattern 的危险性，请参考以下说明。

- Double-checked locking: Clever, but broken

  http://www.javaworld.com/jw-02-02001/jw-0209-double.html

- Warning! Threading in a multiprocessor world

  http://www.javaworld.com/jw-02-2001/jw-0209-toolbox.html

PDG

附录C

Java线程的优先级

Java 线程的优先级

Java 线程有优先级（priority）。这个附录 C 就要来谈谈优先级。

Java 的优先级必须在特定的执行处理系统上才会有效。

前面也已经说过，Java 的规格中几乎没有提及任何关于优先级的规定（参照附录 E [JLS]）。因此，只要没有特别指定特定 Java 的执行处理系统（只要没有特别指定 Java VM 的安装与版本以及 OS），优先级就没有任何效用。

优先级的值真的会对线程的功能有所影响吗？会有影响的话又是怎么样的影响呢？这些在规格中都没有规定。虽然有提到“以有高优先级的线程为优先”，但是对“优先”的意思没有明确定义，所以还是没有实质的意义。

就某些执行处理系统而言，有高优先级的线程会比低优先级的线程分配到更长的 CPU 时间。但是就其他的执行处理系统而言，可能是优先度最高的线程使用掉全部的 CPU 时间。这个时候，只要优先度高的线程没有被 block 起来，优先度低的线程就没有办法执行。还有另一些执行处理系统是不管优先度如何，都不会影响到线程的执行。线程的 scheduling 的算法也没有规范于其中。

各位当中一定有人想说“把想要优先执行的线程设置为高优先度，一定没错。”不过，这样想可就错了。因为有的执行处理系统中，只要有某个线程的优先级稍微高一点，这个线程就很有可能会一直执行下去。这个情况下，若将一个线程的优先级提高，只有这个线程会开始执行，其他的线程都会停下来。也就是说，会变成近似单线程（Single Thread）的状态。

如果无论如何都要使用优先级的话，先特别指定 Java 的执行处理系统，并确认该处理系统对优先级如何定义与处理。但是不要期待这个程序在其他的 Java 执行处理系统上也会执行一样的处理。

以下介绍与优先级有关的 Java 的字段与方法。

表示优先级的类字段

Java 的线程的优先级是整数（int）。被声明为表示优先级的类字段（class field）有以下 3 项：

- Thread.MIN_PRIORITY　　表示优先级最低
- Thread.NORM_PRIORITY　　表示默认的优先级
- Thread.MAX_PRIORITY　　表示优先级最高

这 3 项的值都是固定的，不过并不表示 OS 中的优先级。

设置优先级的方法

setPriority 方法，是用来设置优先级的方法。setPriority 方法是 Thread 类的实例方法（instance method）。

获取优先级的方法

getPriority 方法，是获取优先级的方法。getPriority 方法是 Thread 类的实例方法（instance method）。

PDG

附录D

线程相关的主要API



线程相关的主要 API

附录 D 中要介绍的是线程相关的主要 API 的概要。

这里介绍的 API 并非所有线程的 API，关于各 API，并未将规格书的所有内容都详述于此，请读者仅作为参考之用。

在实际设计或写程序时，请务必参照所使用的 Java 处理系统的 API Reference。

▶▶ 补充一点：“这个线程”与“现在的线程”

在这里，“这个线程”与“现在的线程”所指的线程是不一样的。

“这个线程”在 java.lang.Thread 的实例方法中有说明，表示 this（以及相对应于 this 的线程）的意思。

“现在的线程”的意思是调用其他线程的线程。

Java.lang.Object 类

- public final void notify()
  从在这个对象上等待的线程中选择其中一个启动。
  现在的线程如果没有这个对象的锁定时（没有监视器 monitor 时），执行时会被抛出异常 java.lang.IllegalMonitorStateException。
- public final void notify()
  启动在这个对象上所有等待的线程。
  现在的线程如果没有等待这个对象的锁定时（没有监视器 monitor 时），执行时会被抛出异常 java.lang.IllegalMonitorStateException。
- public final void wait() throws InterruptedException
  让现在的线程（调用出 wait 方法的线程）全部 wait。不会终止休息（timeout）。不同于 sleep 方法，现在的线程的所有的锁定会被释放。
  现在的线程如果没有这个对象的锁定时（没有监视器 monitor 时），执行时异常 java.lang.IllegalMonitorStateException 会被抛出。
  其他的线程中断现在的线程时，java.lang.InterruptedException 异常会被抛出，而清除中断状态。
  关于终止休息（Time-Out）请参照第 4 章。
- public final void wait(long millis) throws InterruptedException
  让现在的线程（调用出 wait 方法的线程）全部 wait。millis 是到终止休息（timeout）之前的时间（以毫秒为单位）。wait(0)的话则不会终止休息。

不同于 sleep 方法，现在的线程的所有的锁定会被释放。
现在的线程如果没有这个对象的锁定时（没有监视器 monitor 时），执行时异常 java.lang.IllegalMonitorStateException 会被抛出。
在等待中其他的线程中断现在的线程时，java.lang.InterruptedException 异常会被抛出，而清除中断状态。
关于终止休息（Time-Out）请参照第 4 章。

- public final void wait(long millis, int nanos) throws InterruptedException
  让现在的线程（调用出 wait 方法的线程）全部 wait。1000000 * millis + nanos 即为到终止休息（timeout）之前的时间（以十亿分之一秒为单位）。wait(0, 0) 的话则不会终止休息。
  不同于 sleep 方法，现在的线程所有的锁定会被释放。
  millis 若为负，或者 nanos 不是在 0 以上 999999 以下的话，执行时异常 java.lang.IllegalArgumentException 会被抛出。
  现在的线程如果没有这个对象的锁定时（没有监视器 monitor 时），执行时异常 java.lang.IllegalMonitorStateException 会被抛出。
  在 wait 中其他的线程中断现在的线程时，异常 java.lang.InterruptedException 会被抛出，而清除中断状态。
  关于终止休息（Time-Out）请参照第 4 章。

java.lang.Runnable 接口

- public void run()
  在建立 java.lang.Thread 的实例时，如果使用拥有 Runnable 的接口的类的实例，在线程启动后，run 方法就会被调用出。

java.lang.Thread 类（implements Runnable）

- public Thread()
  产生 java.lang.Thread 的实例。
- public Thread(Runnable target)
  指定安装了 java.lang.Runnable 接口的类的实例，产生 java.lang.Thread 的实例。
- public Thread(Runnable target, String name)
  指定安装了 java.lang.Runnable 接口的类的实例与名称，产生 java.lang.Thread。
- public Thread(String name)
  指定名称，产生 java.lang.Thread 的实例。

- public Thread(ThreadGroup group, Runnable target)
  指定线程群，产生 java.lang.Thread 的实例。
- public Thread(ThreadGroup group, Runnable target, String name)
  指定线程群、安装了 java.lang.Runnable 接口的类的实例以及名称，产生 java.lang.Thread 的实例。
- public Thread(ThreadGroup group, String name)
  指定线程群与名称，产生 java.lang.Thread 的实例。
- public static final int MIN_PRIORITY
  代表线程能获取的最低优先级的数字。
- public static final int NORM_PRIORITY
  默认的线程优先级。
- public static final int MAX_PRIORITY
  代表线程能获取的最高优先级的数字。
- public static Thread currentThread()
  获得对应于现在的线程（调用这个方法的线程）的 java.lang.Thread 的实例。
- public final String getName()
  获得这个线程（this）的名称。
- public final int getPriority()
  获得这个线程（this）的优先级。
- public void interrupt()
  中断这个线程（this）。
  会中断对应于 this 的线程，但不一定中断对应于 Thread.currentThread()的线程，这点请注意。
- public static boolean interrupted()
  检查现在的线程（调用 interrupted 方法的线程）是否处于中断（interrupt）的状态。调用这个方法时，现在的线程就不再是中断状态了（第 5 章）。
- public final boolean isAlive()
  检查这个线程（this）是否活着。
- public final boolean isDaemon()
  检查这个线程（this）是否为 daemon 线程。
- public boolean isInterrupted()
  检查这个线程（this）是为中断的状态。即使调用这个方法，这个线程（this）的中断状态也不会改变（第 5 章）。
- public final void join() throws InterruptedException
  现在的线程（调用 join 方法的线程）会等待这个线程（this）的结束。
  其他的线程中断现在的线程（调用出 join 的线程）时，java.lang.InterruptedException

异常会被抛出，而清除现在的线程的中断状态（第 5 章）。

- public final void join(long millis) throws InterruptedException
  等待这个线程（this）结束。
  millis 是到终止休息（timeout）之前的时间（以毫秒为单位）。join(0)时，不会终止休息。
  millis 为负时，线程 java.lang.InterruptedException 异常会被抛出。
  在等待中其他的线程中断现在的线程时，java.lang.InterruptedException 异常会被抛出，而清除中断状态（第 5 章）。
- public final void join（long millis, int nanos） throws InterruptedException
  等待这个线程（this）结束。
  1000000 * millis + nanos 即为到终止休息（timeout）之前的时间（以十亿分之一秒为单位）。join(0, 0)的话则不会终止休息（timeout）。
  millis 若为负，或是 nanos 不是在 0 以上 999999 以下的话，执行时异常 java.lang.IllegalArgumentException 会被抛出。
  在等待中，其他的线程中断现在的线程时，异常 java.lang.InterruptedException 会被抛出，而清除中断状态（第 5 章）。
- public void resume()
  不建议使用。要与 suspend 方法（也不建议使用此项）配套使用。
- public void run()
  这个线程(this)启动，就会调用出这个方法（method）。
  这个方法通常会被 java.lang.Thread 的子类 override。java.lang.Thread 类的 run 方法什么都不做就会直接结束。
- public final void setDaemon(boolean on)
  on 如果为 true，就将这个线程（this）作为 daemon 线程。
  这个线程已经启动的话，异常 java.lang.IllegalThreadStateException 会被抛出。
  现在的线程如果不能改变这个线程（this），异常 java.lang.SecurityException 会被抛出。
- public final void setName（String name)
  改变这个线程（this）的名称。
  现在的线程如果不能改变这个线程（this），异常 java.lang.SecurityException 会被抛出。
- public final void setPriority（int newPriority）
  设定优先级。
  优先级若不在 MIN_PRIORITY 与 MAX_PRIORITY 的范围内，执行时异常 java.lang.IllegalArgumentException 会被抛出。

现在的线程如果不能改变这个线程（this），异常 java.lang.SecurityException 会被抛出。

- public static void sleep（long millis）throws InterruptedException
  将当前的线程（调用出 sleep 的线程）按指定的时间（毫秒为单位）暂停。
  与 wait 方法不同，现在的线程的锁定没有被释放。
  millis 为负时，执行时异常 java.lang.IllegalArgumentException 会被抛出。停止时，其他的线程中断到现在的线程时，异常 java.lang.InterruptedException 会被抛出，而清除中断状态。
- public static void sleep(long millis, int nanos) throws InterruptedException
  将当前的线程（调用出 sleep 的线程）依指定的时间（以十亿分之一秒为单位）暂停。
  与 wait 方法不同，现在的线程的锁定没有被释放。
  millis 为负时，或是 nanos 不是在 0 以上 999999 以下的话，执行时异常 java.lang.IllegalArgumentException 会被抛出。
  停止时，其他的线程中断到现在的线程时，异常 java.lang.InterruptedException 会被抛出，而清除中断状态（第 5 章）。
  详情请参照第 5 章。
- public void start()
  启动线程。新启动的线程会调用出 this 的 run 方法。
  如果这个方法（this）已经启动了，执行时异常 java.lang.IllegalThreadStateException 会被抛出。
- public final void stop()
  不建议使用。使用这个方法可能导致安全性的问题。详情请参照第 5 章和第 10 章。
- public final void stop(Throwable obj)
  不建议使用。使用这个方法可能导致安全性的问题。
- public final void suspend()
  不建议使用。使用这个方法可能导致安全性的问题。
- public String toString()
  可以从这个线程（this）的名称、优先级、线程群（Thread Group）中取得字符串。
- public static void yield()
  将当前的线程（调用出这个方法的线程）暂停，启动其他线程。

Java.lang.ThreadLocal 类（参照第 11 章）

- public object get()
  取得对应于当前的线程（调用这个方法的线程）的实际。如果一次也没有用 set 保存过的话，null 就会成为返回值。
- public void set（Object obj）
  设置实际，使能对应到当前的线程（调用出这个方法的线程）。

附录E

参考文献

本书在执笔时，参考了以下的文献数据。若文献中记载有本书中解说的模式，会将该模式名称表示出来。文献中使用的表记方式会显示在[]中。

本书相关的 Web 网页如下所示。

http://www.hyuki.com/dp/dp2.html

Java 多线程程序的深入学习

[Lea] "Concurrent Programming in Java, Design Principles and Patterns, Second Edition" by Doug Lea

《Java 执行绪程式撰写 并列式物件导向程式设计原理》

松野良藏 监译 / 株式会社翔泳社 / ISBN4-88135-918-5

完整介绍 Java 的线程程序设计，内容详实而正确。

介绍的 Pattern

- Active Object (Actor)
- Balking
- Future(Futures)
- Guarded Suspension (Guarded Waits)
- Immutable
- Producer-Consumer
- Read-Write Lock(Readers and Writers)
- Thread-Per-Message
- Thread-Specific Storage(Thread-Specific Data)
- Worker Thread (Woker-Threads)

获得类链接库 util.concurrent 的方法

[LeaWeb] Doug Lea's Home Page

http://g.oswego.edu/dl/

我们可以获取多线程的类链接库 util.concurrent。

要学 Java 的语言规格

[JLS] "The Java Language Specification, Second Edition" by Bill Joy, Guy Steele, James Gosling, Gilad Bracha

《Java 语言规格 第 2 版》

村上 雅章 译 / 株式会社 Pierson Education / ISBN4-89471-306-3

http://java.sun.com/docs/books/jls/

要学 Java 虚拟机

[JVMS] "The Java Virtual Machine Specification, Second Edition" by Tim Lindholm, Frank Yellin

《Java 虚拟机器规格 第 2 版》

村上 雅章 译 / 株式会社 Pierson Education / ISBN4-89471-356-X

http://java.sun.com/docs/books/vmspec/

[Venners99] "Inside the Java 2 Virtual Machine, Second Edition" by Bill Venners / Osborne McGraw-Hill / ISBN0-07-135093-4

要学 GoF 的 Design Pattern

[GoF] "Design Pattern: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

《物件导向程式中重复使用的 Design Pattern 修订版》

本位田真一、吉田和树监译/Softbank Publishing 株式会社/ISBN4-7973-1112-6

http://hillside.net/patterns/DPBook/DPBook.html

本书为 Design Pattern 的圣经。其中介绍 23 个最有名的 Design Pattern。繁体本则为台湾培生出版。

要学 Java 中 GoF 的设计模式

[Yuki01]Design Patterns 于 Java 语言上的实习应用

结城 浩著/ 株式会社 Softbank Publishing/ ISBN4-7973-1646-2

http://www.hyuki.com/dp/

Design Pattern 的入门书之一，使用 Java 的范例程序来解说 GoF 中的 23 个模式。繁体本为博硕文化出版。简体中文本为中铁道出版社《设计模式 --Java 语言中的应用》（书名再确定）。

要学 GoF 以外的设计模式

[Grand] "Patterns in Java Volume1, A Catalog of Reusable Design Patterns Illustrated wuth UML" by Mark Grand

《使用了 UML 的 Java 设计模式—可重复使用的程序设计集》

原洁、宫本道夫、濑尾明志译/CUTT System Development Laboratory/身 ISBN4-87783-013-8

本书讲述到许多种 Pattern。虽然也有介绍到许多关于多线程的 Pattern,但是由于省略了相当多的说明部分,因此阅读时需要注意这点。

介绍的 Pattern

- Balking
- Guarded Suspension
- Immutable
- Producer-Consumer
- Read-Write Lock (Read/Write Lock)
- Single Threaded Execution
- Two-Phase Termination

要学 GoF 以外的古典的模式

[POSA1] "Pattern-Qriented Software Architecture, Volume1: A System of Patterns"by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal

《软件架构》

金泽 典子、水野 贵之、樱井 麻里、关 富登志、千叶 宽之 译/近代科学社/ISBN4-7649-0283-4

[POSA2] "Pattern-Qriented Software Architecture, Volume2: Pattern for Concurrent and Networked Object" by Douglas Schmidt, Michael Stal,Hans Rohnert,Frank Buschmann/John Wiley & Sons/ISBN0-47-160695-2

介绍的 Pattern

- Active Object
- Thread-Specific Storage

要学习 Java 的多线程

[Holub00] " Taming Java Threads"
by Allen I.Holub/ Apress/ ISBN1-89-311510-0

关于 Java 多线程的解说书籍。

介绍的 Pattern

- Active Object
- Balking
- Guarded Suspension (Spin Lock)
- Immutable

- Read-Write Lock (Read/Write Lock)
- Single Thread ed Execution (Critical Section)
- Thread-Per-Message (Thread-per-Method)
- Worker Thread (Thread Pool)

[Thomas00] "High-Performance Java Platform Computing: Multithreaded and Networked Programming" by Thomas Christopher, George K. Thiruvathukal/Prentice Hall/ISBN0-13-016164-0

[Lewis00] "Multithreaded Programming with Java Technology" by Bil Lewis, Daniel J.Berg/Prentice Hall/ISBN0-13-017007-0

要学习从 Java 的基本到多线程

[Tomatsu02]《Java 程式设计 第 3 版》

户松 丰和 著/ 株式会社 Softbank Publishing/isbn4-7973-1923-2

本书汇整了 Java 的基本、Pattern 以及多线程的精要。繁体本为博硕文化出版。

介绍的 Pattern

- Single Thread ed Execution（临界区间）
- Guarded Suspension（同步条件、Custom Lock）
- Future（Future Object）
- Producer-Consumer（生产者/消费者）
- Immutable
- Active Object（主动对象）

要学习 Java 的基础

[Yuki99]《Java 语言程式设计讲座》(上下 2 册)

结城 浩 著/株式会社 Softbank Publishing/ ISBN4-7973-0803-6（上册）ISBN4-7973-1010-3（下册）

http://www.hyuki.com/jb/

本书为 Java 的入门书，上册撰写内容着重于 Java 的基础，下册则是针对面象对象设计和多线程。

介绍的 Pattern

- *Single Thread ed Execution*
- Guarded Suspension
- Producer-Consumer

要学习 Java 的技巧

[Bloch] "Effective Java Programming Language Guide"
by Joshua Bloch

《Effective Java 程序语言手册》
柴田 芳树 译/株式会社 Pearson Education Japan/ISBN4-89471-436-1
在第 9 章中记载了关于线程需要注意的地方。

参考网页

- 本书相关网页
  http://www.hyuki.com/dp/dp2.html
- Pattern s Home Page
  http://www.hillside.net/patterns/
- Design Pattern Mailing List
  http://www.hyuki.com/dp/dpml.html
- Object Management Group
  http://www.omg.org/
- DesignPatterns in Wiki
  http://c2.com/cgi/wiki?DesignPatterns
- Why Are Thread.stop, Thread.susoend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?
  http://java.sun.com/products/jdk/1.3/docs/guide/misc/threadPrimitiveDeprecation .html
- 不推荐 Thread.stop、Thread-suspend、Thread-resume 以及 Runtime-runFinalizerOnExit 的理由（上述日文解释）
  http://java.sun .com/products/jdk/1.3/ja/docs/ja/guide/misc/threadPrimitiveDeprecation.html
- Observer 模式以及多线程
  http://objectclub.esm.co.jp/observer-multithread/
- Immutable Object
  http://www.asahi-net.or.jp/~dp8t-asm/java/articles/notes/01/article.html
- Threads and Swing
  http://java.sun .com/products/jfc/swingdoc-archive/threads.html

参考文献

- Threads and Swing

  http://java.sun .com/products/jfc/tsc/articles/threads/threads1.html

- Using a Swing WorkerThread—New Ways to Perform Background Tasks

  http://java.sun .COIn/products/j fc/tsc/articles/threads/threads2.html

- JFC/Swing Event Handling (Tutorial)

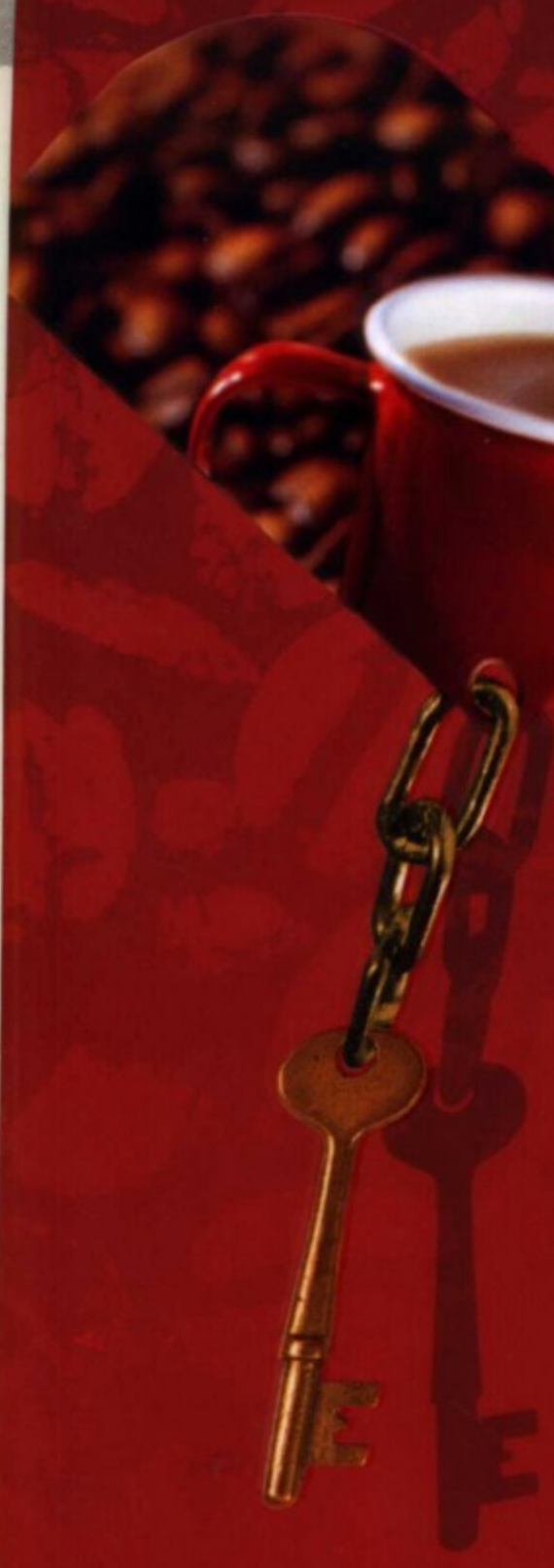
  http://java.sun .com/docs/books/tutorial/uiswing/overview/event.html

- If I were king:A proposal for fixng the Java programming language's threading problems

  http://www-106.ibm.com/developerworks/java/library/j -king .html

- 如果我是神的话：Java 程序语言线程不正常问题解决的方案（上述日文直译）

  http://www-6.ibm.com/jp/developerworks/java/010302j_j-king.html



JAVA 多线程设计模式

多线程与并发处理是程序设计好坏优劣的重要课题，本书通过浅显易懂的文字与实例来介绍Java线程相关的设计模式概念，并且通过实际的Java程序范例和UML图示来一一解说，书中在代码的重要部分加上标注使读者更加容易解读，再配合众多的说明图解，无论对于初学者还是程序设计高手来说，这都是一本学习和认识设计模式非常难得的好书。

书中包含Java线程的介绍导读、12个重要的线程设计模式和全书总结以及丰富的附录内容。每一章相关线程设计模式的介绍，都举一反三使读者学习更有效率。最后附上练习问题，让读者可以温故而知新，能快速地吸收书中的精华，书中最后附上练习问题解答，方便读者学习验证。

【责任编辑】 苏茜　翟玉峰　王占清
【封面制作】 白雪

ISBN 7-113-06402-7

9 787113 064020 >

ISBN 7-113-06402-7/TP·1433
定价：49.00 元

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

中国铁道出版社·计算机图书批销部
地址：北京市宣武区右安门西街8号
邮编：100054
网址：http://www.tqbooks.net
销售服务电话：(010)83550290 (010)63560043
读者热线电话：(010)63583215 (010)51873629