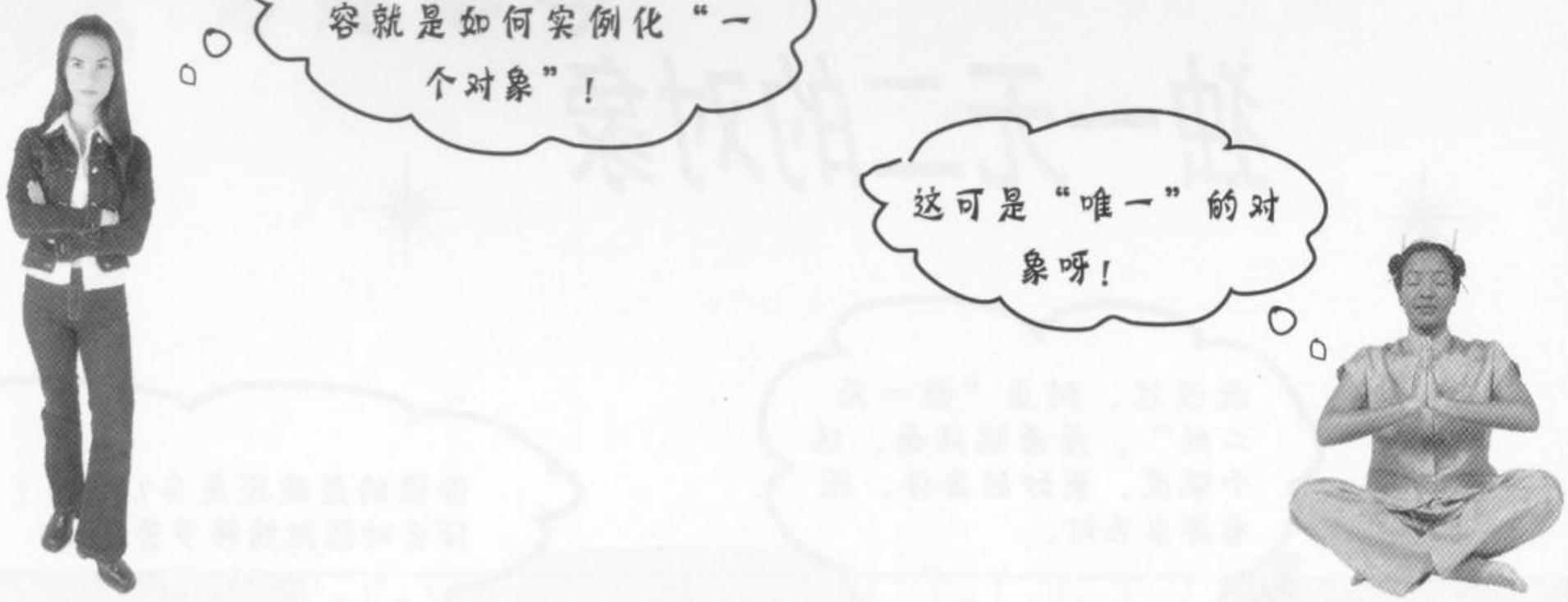
5 单件模式

独一无二的对象

下一站是单件模式（Singleton Pattern）：用来创建独一无二的,只能有一个实例的对象的入场券。告诉你一个好消息，单件模式的类图可以说是所有模式的类图中最简单的，事实上，它的类图上只有一个类！但是，可不要兴奋过头，尽管从类设计的视角来说它很简单，但是实现上还是会遇到相当多的波折。所以，系好安全带，出发了！



开发人员：这有什么用处？

大师：有一些对象其实我们只需要一个，比方说：线程池（threadpool）、缓存（cache）、对话框、处理偏好设置和注册表（registry）的对象、日志对象，充当打印机、显卡等设备的驱动程序的对象。事实上，这类对象只能有一个实例，如果制造出多个实例，就会导致许多问题产生，例如：程序的行为异常、资源使用过量，或者是不一致的结果。

开发人员：好吧！或许的确有一些类应该只存在一个实例，但这需要花整个章节的篇幅来说明吗？难道不能靠程序员之间的约定或是利用全局变量做到？你知道的，利用Java的静态变量就可以做到。

大师：许多时候，的确通过程序员之间的约定就可以办到。但如果有更好的做法，大家应该都乐意接受。别忘了，就跟其他的模式一样，单件模式是经得起时间考验的方法，可以确保只有一个实例会被创建。单件模式也给了我们一个全局的访问点，和全局变量一样方便，又没有全局变量的缺点。

开发人员：什么缺点？

大师：举例来说：如果将对象赋值给一个全局变量，那么你必须在程序一开始就创建好对象★，对吧？万一这个对象非常耗费资源，而程序在这次的执行过程中又一直没用到它，不就形成浪费了吗？稍后你会看到，利用单件模式，我们可以在需要时才创建对象。

开发人员：我还是觉得这没什么困难的。

大师：利用静态类变量、静态方法和适当的访问修饰符（access modifier），你的确可以做到这一点。但是，不管使用哪一种方法，能够了解单件的运作方式仍然是很有趣的事。单件模式听起来简单，要做得对可不简单。不信问问你自己：要如何保证一个对象只能被实例化一次？答案可不是三言两语就说得完的，是不是？

★这其实和实现有关。有些JVM的实现是：在用到的时候才创建对象。

小小单件

苏格拉底式的诱导问答

如何创建一个对象？

```
new MyObject();
```

万一另一个对象想创建MyObject会怎样？可以再次new MyObject吗？

是的，当然可以。

所以，一旦有一个类，我们是否都能多次地实例化它？

如果是公开的类，就可以。

如果不是的话，会怎样？

如果不是公开类，只有同一个包内的类可以实例化它，但是仍可以实例化它多次。

嗯！有意思！你知道可以这么做吗？

我没想过。但是，这是合法的定义，有一定的道理。

```
public MyClass {

  private MyClass() {}

}
```

怎么说呢？

我认为含有私有的构造器的类不能被实例化。

有可以使用私有的构造器的对象吗？

嗯，我想MyClass内的代码是唯一能调用此构造器的代码。但是这又不太合乎常理。

为什么?

因为必须有MyClass类的实例才能调用MyClass构造器，但是因为没有其他类能够实例化MyClass，所以我们得不到这样的实例。这是“鸡生蛋，蛋生鸡”的问题。我可以在MyClass类型的对象上使用MyClass构造器，但是在这之前，必须有一个MyClass实例。在产生MyCalss实例之前，又必须在MyClass实例内才能调用私有的构造器……

嘿！我有个想法。
你认为这样如何?

MyClass有一个静态方法。我们可以这样调用这个方法：

```
MyClass.getInstance();
```

```
public MyClass {

   public static MyClass getInstance() {
   }
}
```

为何调用的时候用MyClass的类名，
而不是用对象名?

因为getInstance()是一个静态方法，换句话说，是一个“类”方法。引用一个静态方法，你需要使用类名。

有意思。假如把这些合在一起“是否”就可以初始化一个MyClass?

当然可以。

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

好了，你能想出第二种实例化对象的方式吗?

```
MyClass.getInstance();
```

你能够完成代码使MyClass只有一个实例被产生吗?

嗯，大概可以吧……

（下一页有这个代码。）

剖析经典的单件模式实现

注意！

如果你只是很快地翻到这一页，不要盲目地键入代码。在本章后面的部分中，你会看到这个版本有一些问题。

把MyClass改名为Singleton。

利用一个静态变量来记录Singleton类的唯一实例。

```
public class Singleton {
        private  static  Singleton
uniqueInstance;
    // 这里是其他的有用实例化变量
    private Singleton() {}

        public  static  Singleton
getInstance() {
        if (uniqueInstance == null)
{
                uniqueInstance = new
Singleton();
        }
        return uniqueInstance;
    }
    // 这里是其他的有用方法
}
```

把构造器声明为私有的，只有自Singleton类内才可以调用构造器。

用getInstance()方法实例化对象，并返回这个实例。

当然，Singleton是一个正常的类，具有一些其他用途的实例变量和方法。

再靠近一点

uniqueInstance拥有“一个”实例，别忘了，它是个静态变量。

如果uniqueInstance是空的，表示还没有创建实例……

……而如果它不存在，我们就利用私有的构造器产生一个Singleton实例并把它赋值到uniqueInstance静态变量中。请注意，如果我们不需要这个实例，它就永远不会产生。这就是“延迟实例化”（lazy instantiaze）。

```
if (uniqueInstance == null) {
    uniqueInstance = new MyClass();
}
return uniqueInstance;
```

如果uniqueInstance不是null，就表示之前已经创建过对象。我们就直接跳到return语句。

当执行到这个return，就表示我们已经有了实例，并将uniqueInstance当返回值。

模式告白

本周访问：
单件的告白

HeadFirst：今天我们很高兴专访单件对象。一开始，不妨先介绍一下你自己。

单件：关于我，我只能说我很独特，我是独一无二的。

HeadFirst：独一无二？

单件：是的，独一无二。我是利用单件模式构造出来的，这个模式让我在任何时刻都只有一个对象。

HeadFirst：这样不会有点浪费吗？毕竟有人花了这么多时间写了类的代码，而这个类竟然只产生一个对象。

单件：不，一点儿也不浪费！“一个”的威力很强大呢！比方说，如果有一个注册表设置（registry setting）的对象，你不希望这样的对象有多个拷贝吧？那会把设置搞得一团乱。利用像我这样的单件对象，你可以确保程序中使用的全局资源只有一份。

HeadFirst：请继续……

单件：嗯！我擅长许多事。有时候独身是有些好处的。我常常被用来管理共享的资源，例如数据库连接或者线程池。

HeadFirst：但我还是觉得，一个人好像有一点孤单。

单件：因为只有我一个人，所以通常很忙，但还是希望更多开发人员能认识我。许多开发人员因为产生了太多同一类的对象而使他们的代码出现了bug，但他们却浑然不觉。

HeadFirst：那么，请允许我这么问，你怎么能确定只有一个你？说不定别人也会利用new产生多个你呢。

单件：不可能，我是独一无二的。

HeadFirst：该不会要每个开发人员都发毒誓绝对不会实例化多个你吧？

单件：当然不是，真相是…… 唉呀！这牵扯到个人隐私…… 其实…… 我没有公开的构造器。

HeadFirst：没有公开的构造器！！噢！抱歉！我太激动了。没有公开的构造器？

单件：是的，我的构造器是声明为私有的。

HeadFirst：这怎么行得通？你“究竟”是怎样被实例化的？

单件：外人为了要取得我的实例，他们必须“请求”得到一个实例，而不是自行实例化得到一个实例。我的类有一个静态方法，叫做getInstance()。调用这个方法，我就立刻现身，随时可以工作。事实上，我可能是在这次调用的时候被创建出来的，也可能是以前早就被创建出来了。

HeadFirst：单件先生，你的内在比外表更加深奥。谢谢你如此坦白，希望能很快再与你见面。

巧克力工厂

大家都知道，现代化的巧克力工厂具备计算机控制的巧克力锅炉。锅炉做的事，就是把巧克力和牛奶融在一起，然后送到下一个阶段，以制造成巧克力棒。

这里有一个Choc-O-Holic公司的工业强度巧克力锅炉控制器。看看它的代码，你会发现代码写得相当小心，他们在努力防止不好的事情发生。例如：排出500加仑的未煮沸的混合物，或者锅炉已经满了还继续放原料，或者锅炉内还没放原料就开始空烧。

```
public class ChocolateBoiler {
   private boolean empty;
   private boolean boiled;

   public ChocolateBoiler() {
       empty = true;
       boiled = false;
   }

   public void fill() {
       if (isEmpty()) {
           empty = false;
           boiled = false;
           // 在锅炉内填满巧克力和牛奶的混合物
       }
   }

   public void drain() {
       if (!isEmpty() && isBoiled()) {
           // 排出煮沸的巧克力和牛奶
           empty = true;
       }
   }

   public void boil() {
       if (!isEmpty() && !isBoiled()) {
           // 将炉内物煮沸
           boiled = true;
       }
   }

   public boolean isEmpty() {
       return empty;
   }

   public boolean isBoiled() {
       return boiled;
   }
}
```

代码开始时，锅炉是空的。

在锅炉内填入原料时，锅炉必须是空的。一旦填入原料，就把empty和boiled标志设置好。

锅炉排出时，必须是满的（不可以是空的）而且是煮过的。排出完毕后，把empty标志设回true。

煮混合物时，锅炉必须是满的，并且是没有煮过的。一旦煮沸后，就把boiled标志设为true。

Choc-O-Holic公司在有意识地防止不好的事情发生，你不这么认为吗？你可能会担心，如果同时存在两个ChocolateBoiler（巧克力锅炉）实例，可能将发生很糟糕的事情。

万一同时有多于一个的ChocolateBoiler（巧克力锅炉）实例存在，可能发生哪些很糟糕的事呢？

请帮Choc-O-Holic改进ChocolateBoiler类，把这个类设计成单件。

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    [                                                    ]

    [          ] ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    [                                                    ]
    [                                                    ]
    [                                                    ]
    [                                                    ]

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // 在锅炉内填充巧克力和牛奶的混合物
        }
    }
    // 其他的部分省略不列出来
}
```

定义单件模式

现在你脑海中已经有了单件的经典实现，该是坐下来享受一条巧克力棒，并细细品味单件模式的时候了。

先看看单件模式的简要定义：

> **单件模式**确保一个类只有一个实例，并提供一个全局访问点。

这定义一点儿都不让人吃惊，但是让我们更深入一点儿：

- 到底怎么回事？我们正在把某个类设计成自己管理的一个单独实例，同时也避免其他类再自行产生实例。要想取得单件实例，通过单件类是唯一的途径。
- 我们也提供对这个实例的全局访问点：当你需要实例时，向类查询，它会返回单个实例。前面的例子利用延迟实例化的方式创建单件，这种做法对资源敏感的对象特别重要。

好吧！来看看类图：

~~Houston~~ Hershey，我们遇到麻烦了……

看起来巧克力锅炉要让我们失望了，尽管我们利用经典的单件来改进代码，但是ChocolateBoiler的fill()方法竟然允许在加热的过程中继续加入原料。这可是会溢出五百加仑的原料（牛奶和巧克力）呀！怎么会这样！？

不知道这是怎么了！新的单件代码原本是一切顺利的。我们唯一能想到的原因就是刚刚使用多线程对ChocolateBoiler进行了优化。

多加线程，就会造成这样吗？不是只要为ChocolateBolier的单件设置好uniqueInstance变量，所有的getInstance()调用都会取得相同的实例吗？对不对？

化身为JVM

这里有两个线程都要执行这段代码。你的工作是扮演JVM角色并判断出两个线程是否可能抓住不同的锅炉对象而扰乱这段代码。提示：你只需要检查getInstance()方法内的操作次序和uniqueInstance的值，看它们是否互相重叠。

用代码帖来帮你研究这段代码为什么可能产生两个锅炉对象。

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

```
public static ChocolateBoiler
    getInstance() {
  if (uniqueInstance == null) {
    uniqueInstance =
        new ChocolateBoiler();
  }
  return uniqueInstance;
}
```

记得在翻页前，先看看188页的答案。

线程一	线程二	uniqueInstance的值

处理多线程

只要把getInstance()变成同步（synchronized）方法，多线程灾难几乎就可以轻易地解决了：

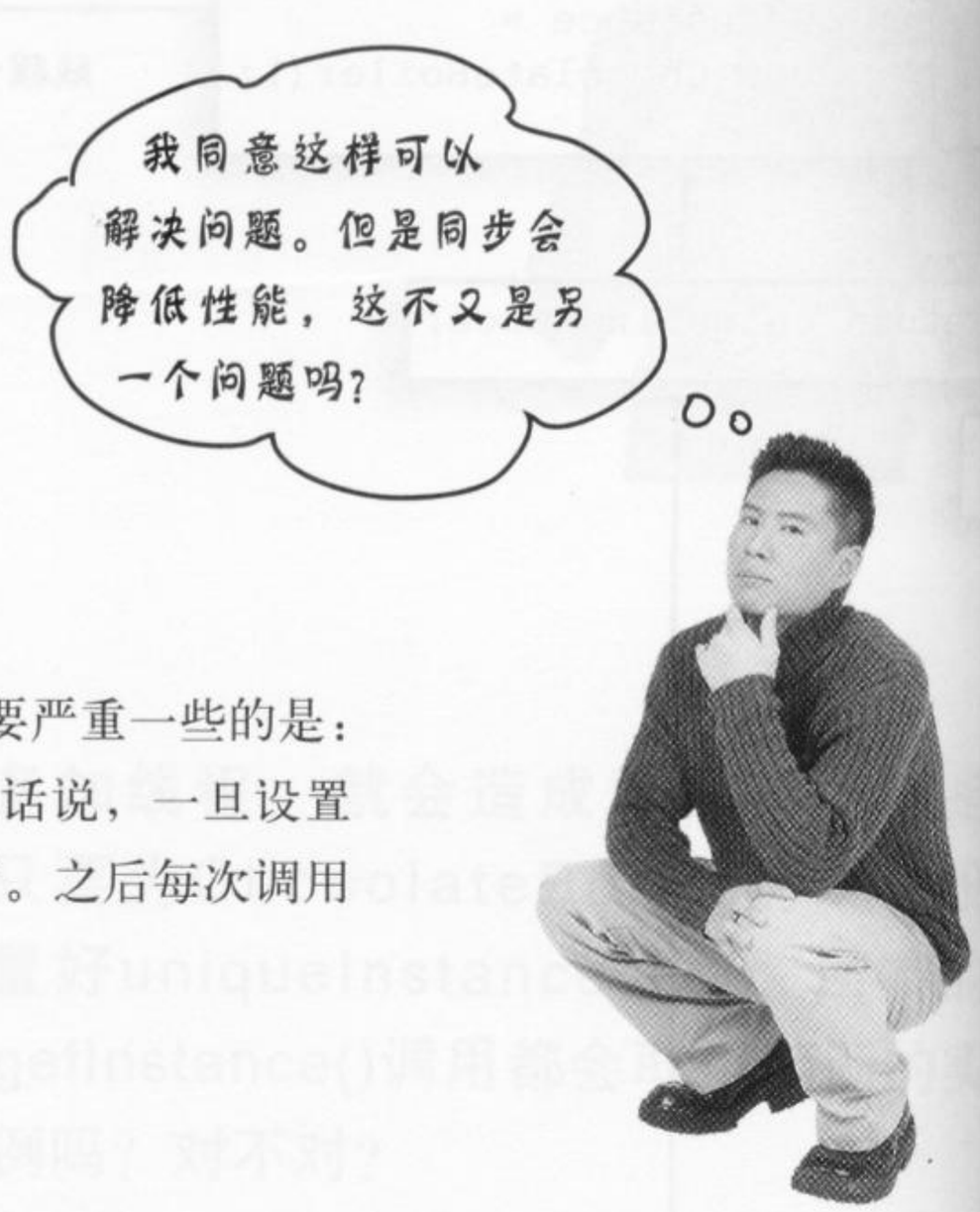
```
public class Singleton {
    private static Singleton uniqueInstance;

    // 其他有用的实例化的变量
    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // 其他有用的方法
}
```

通过增加synchronized关键字到getInstance()方法中，我们迫使每个线程在进入这个方法之前，要先等候别的线程离开该方法。也就是说，不会有两个线程可以同时进入这个方法。



说得很对，的确是有一点不好。而比你所想象的还要严重一些的是：只有第一次执行此方法时，才真正需要同步。换句话说，一旦设置好uniqueInstance变量，就不再需要同步这个方法了。之后每次调用这个方法，同步都是一种累赘。

能够改善多线程吗？

为了要符合大多数Java应用程序，很明显地，我们需要确保单件模式能在多线程的状况下正常工作。但是似乎同步getInstance()的做法将拖垮性能，该怎么办呢？

可以有一些选择……

1. 如果getInstance()的性能对应用程序不是很关键，就什么都别做

没错，如果你的应用程序可以接受getInstance()造成的额外负担，就忘了这件事吧。同步getInstance()的方法既简单又有效。但是你必须知道，同步一个方法可能造成程序执行效率下降100倍。因此，如果将getInstance()的程序使用在频繁运行的地方，你可能就得重新考虑了。

2. 使用“急切”创建实例，而不用延迟实例化的做法

如果应用程序总是创建并使用单件实例，或者在创建和运行时方面的负担不太繁重，你可能想要急切（eagerly）创建此单件，如下所示：

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

在静态初始化器（static initializer）中创建单件。这段代码保证了线程安全（thread safe）。

已经有实例了，直接使用它。

利用这个做法，我们依赖JVM在加载这个类时马上创建此唯一的单件实例。JVM保证在任何线程访问uniqueInstance静态变量之前，一定先创建此实例。

3.用“双重检查加锁”，在getInstance()中减少使用同步

利用双重检查加锁（double-checked locking），首先检查是否实例已经创建了，如果尚未创建，“才”进行同步。这样一来，只有第一次会同步，这正是我们想要的。

来看看代码：

```
public class Singleton {
    private volatile★ static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

检查实例，如果不存在，就进入同步区块。

注意，只有第一次才彻底执行这里的代码。

进入区块后，再检查一次。如果仍是null，才创建实例。

★ volatile关键词确保：当uniqueInstance变量被初始化成Singleton实例时，多个线程正确地处理uniqueInstance变量。

如果性能是你关心的重点，那么这个做法可以帮你大大地减少getInstance()的时间耗费。

注意！ **双重检查加锁不适用于1.4及更早版本的Java！**

很不幸地，在1.4及更早版本的Java中，许多JVM对于volatile关键字的实现会导致双重检查加锁的失效。如果你不能使用Java 5，而必须使用旧版的Java，就请不要利用此技巧实现单件模式。

再度回到巧克力工厂……

在研究如何摆脱多线程的梦魇同时，巧克力锅炉也被清理干净可以再度开工了。首先，得处理多线程的问题。我们有一些选择方案，每个方案都有优缺点，到底该采用哪一个？

Sharpen your pencil

描述每一种方案对于修改巧克力锅炉代码所遇到的问题的适用性。

同步getInstance()方法：

急切实例化

双重检查加锁

恭喜！

此刻，巧克力工厂的问题已经解决了，而且Choc-O-Holic很高兴在锅炉的代码中能够采用这些专业知识。不管你使用哪一种多线程解决方案，锅炉都能顺畅工作，不会有闪失。恭喜你，不但避免了500磅热巧克力的危机，也认清了单件所带来的所有潜在问题。

there are no Dumb Questions

问： 单件模式只有一个类，应该是很简单的模式，但是问题似乎不少。

答： 唉呀！我们只是提前警告，读者不要因为这点儿问题而泄气。固然正确地实现单件模式需要一点技巧，但是在阅读完本章之后，你已经具备了用正确的方法实现单件模式的能力。当你需要控制实例个数时，还是应当使用单件模式。

问： 难道我不能创建一个类，把所有的方法和变量都定义为静态的，把类直接当做一个单件？

答： 如果你的类自给自足，而且不依赖于复杂的初始化，那么你可以这么做。但是，因为静态初始化的控制权是在Java手上，这么做有可能导致混乱，特别是当有许多类牵涉其中的时候。这么做常常会造成一些微妙的、不容易发现的和初始化的次序有关的bug。除非你有绝对的必要使用类的单件，否则还是建议使用对象的单件，比较保险。

问： 那么类加载器（class loader）呢？听说两个类加载器可能有机会各自创建自己的单件实例。

答： 是的。每个类加载器都定义了一个命名空间，如果有两个以上的类加载器，不同的类加载器可能会加载同一个类，从整个程序来看，同一个类会被加载多次。如果这样的事情发生在单件上，就会产生多个单件并存的怪异现象。所以，如果你的程序有多个类加载器又同时使用了单件模式，请小心。有一个解决办法：自行指定类加载器，并指定同一个类加载器。

谣传垃圾收集器会吃掉单件，这过分夸大了！

在Java 1.2之前，垃圾收集器有个bug，会造成当单件在没有全局的引用时被当作垃圾清除。也就是说，如果一个单件只有本单件类引用它本身，那么该单件就会被当做垃圾清除。这造成让人困惑的bug：因为在单件被清除之后，下次调用getInstance()会产生一个“全新的”单件。对很多程序来说，这会造成让人困惑的行为，因为对象的实例变量值都不见了，一切回到最原始的设置（例如：网络连接被重新设置）。

Java 1.2以后，这个bug已经被修正了，也不再需要一个全局引用来保护单件。如果出于某些原因你还在用旧版的Java，要特别注意这个问题。如果你使用1.2以后的Java，就可以高枕无忧了。

问： 我所受到的教育一直是：类应该做一件事，而且只做一件事。类如果能做两件事，就会被认为是不好的OO设计。单件有没有违反这样的观念呢？

答： 你说的是“一个类，一个责任”原则。没错，你是对的。单件类不只负责管理自己的实例（并提供全局访问），还在应用程序中担当角色，所以也可以被视为是两个责任。尽管如此，由类管理自己的实例的做法并不少见。这可以让整体设计更简单。更何况，许多开发人员都已经熟悉了单件模式的这种做法。

问： 我想把单件类当成超类，设计出子类，但是我遇到了问题：究竟可以不可以继承单件类？

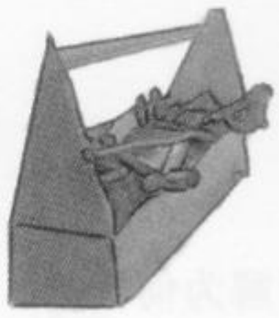
答： 继承单件类会遇到的一个问题，就是构造器是私有的。你不能用私有构造器来扩展类。所以你必须把单件的构造器改成公开的或受保护的。但是这么一来就不算是“真正的”单件了，因为别的类也可以实例化它。

如果你果真把构造器的访问权限改了，还有另一个问题会出现。单件的实现是利用静态变量，直接继承会导致所有的派生类共享同一个实例变量，这可能不是你想要的。所以，想要让子类能工作顺利，基类必须实现注册表（Registry）功能。

在这么做之前，你得想想，继承单件能带来什么好处。就和大多数的模式一样，单件不一定适合设计进入一个库中。而且，任何现有的类，都可以轻易地加上一些代码支持单件模式。最后，如果你的应用程序大量地使用了单件模式，那么你可能需要再好好地检查你的设计。因为通常适合使用单件模式的机会不多。

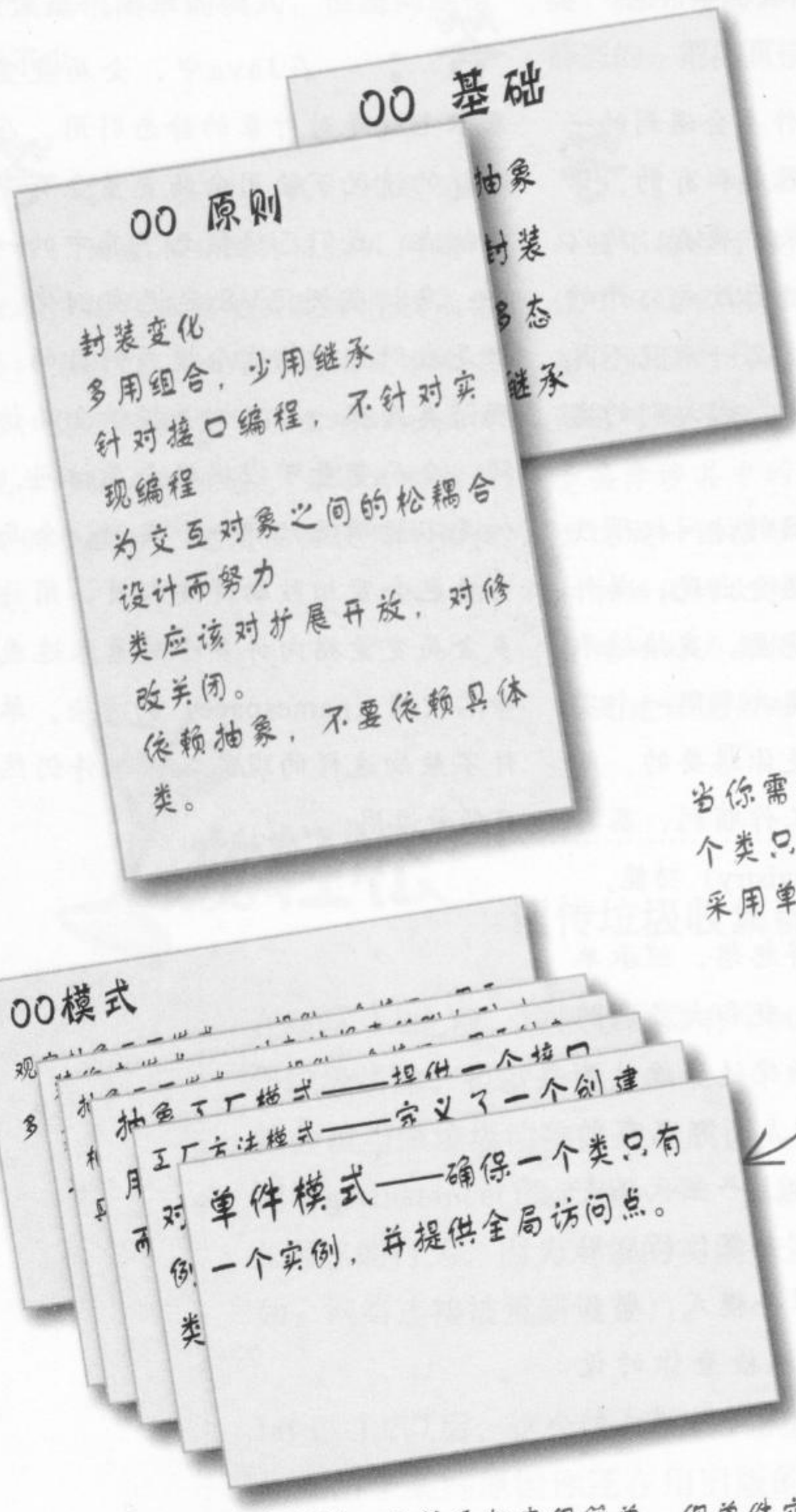
问： 我还是不了解为何全局变量比单件模式差。

答： 在Java中，全局变量基本上就是对对象的静态引用。在这样的情况下使用全局变量会有一些缺点，我们已经提到了其中的一个：急切实例化VS.延迟实例化。但是我们要记住这个模式的目的：确保类只有一个实例并提供全局访问。全局变量可以提供全局访问，但是不能确保只有一个实例。全局变量也会变相鼓励开发人员，用许多全局变量指向许多小对象来造成命名空间（namespace）的污染。单件不鼓励这样的现象，但单件仍然可能被滥用。



设计箱内的工具

你又加了一个新的模式到工具箱里。单件提供另一种创建对象的方法，创建独一无二的对象。

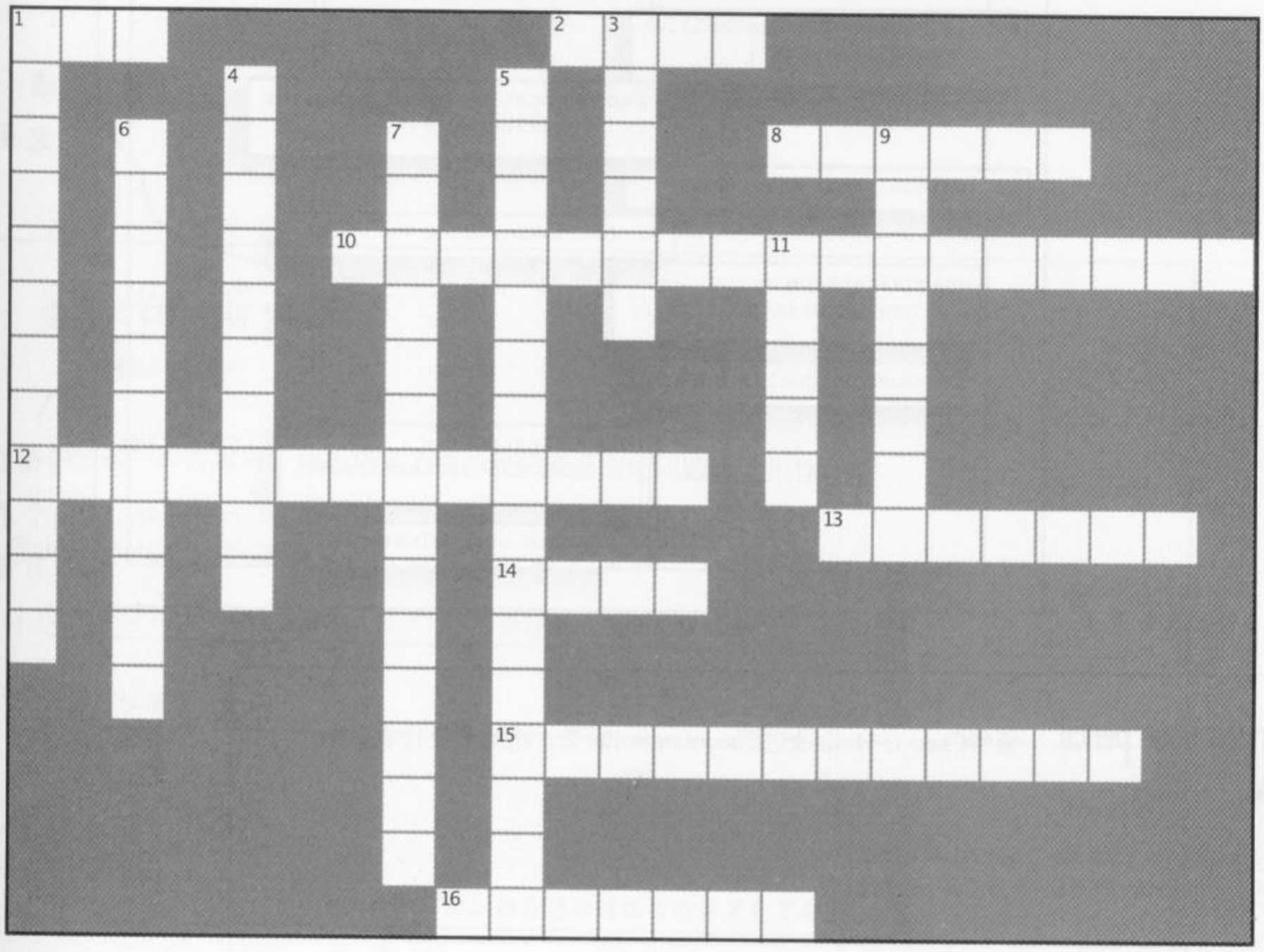


正如你所看到的，尽管看起来很简单，但单件实现中涉及到了很多细节。读完本章后，你就可以使用单件了。

要点

- 单件模式确保程序中一个类最多只有一个实例。
- 单件模式也提供访问这个实例的全局点。
- 在Java中实现单件模式需要私有的构造器、一个静态方法和一个静态变量。
- 确定在性能和资源上的限制，然后小心地选择适当的方案来实现单件，以解决多线 程的问题（我们必须认定所有的程序都是多线程的）。
- 如果不是采用第五版的Java 2，双重检查加锁实现会失效。
- 小心，你如果使用多个类加载器，可能导致单件失效而产生多个实例。
- 如果使用JVM 1.2或之前的版本，你必须建立单件注册表，以免垃圾收集器将单件回收。

坐下来，打开因为解决多线程问题而获赠的巧克力，花一点儿时间解决这个填字游戏，所有的答案都是来自本章的英文词汇。



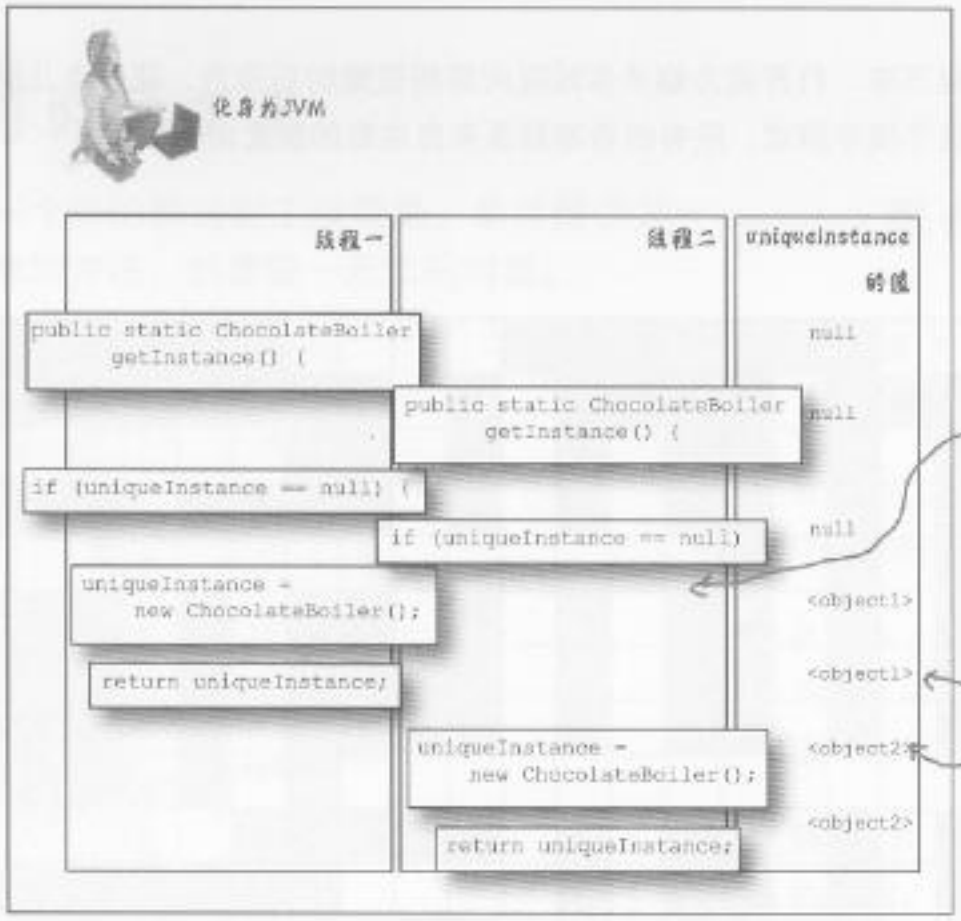
横排提示：

1. It was "one of a kind"
2. Added to chocolate in the boiler
8. An incorrect implementation caused this to overflow
10. Singleton provides a single instance and (three words)
12. Flawed multithreading approach if not using Java 1.5
13. Chocolate capital of the US
14. One advantage over global variables: _____ creation
15. Company that produces boilers
16. To totally defeat the new constructor, we have to declare the constructor _____

竖排提示：

1. Multiple _____ can cause problems
3. A Singleton is a class that manages an instance of _____
4. If you don't need to worry about lazy instantiation, you can create your instance _____
5. Prior to 1.2, this can eat your Singletons (two words)
6. The Singleton was embarassed it had no public _____
7. The classic implementation doesn't handle this
9. Singleton ensures only one of these exist
11. The Singleton Pattern has one

习题解答



Sharpen your pencil 请帮Choc-O-Holic改进ChocolateBoiler类，把此类设计成单件。

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public static ChocolateBoiler getInstance() {

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }
    public static ChocolateBoiler getInstance() {

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            //放牛奶，巧克力混合物填充锅炉
        }
    }
    // 剩余的ChocolateBoiler编码
}
```

习题解答

Sharpen your pencil

描述每一种方案对于修改巧克力锅炉代码所遇到的问题的适用性。

同步getInstance()方法：

这是保证可行的最直接做法。对于巧克力锅炉似乎没有性能的考虑，

所以可以用这个方法。

急切实例化

我们一定需要用到一个巧克力锅炉，所以静态地初始化实例并不是不可以。

虽然对于采用标准模式的开发人员来说，此做法可能稍微陌生一点，但它还是可行的。

双重检查加锁

由于没有性能上的考虑，所以这个方法似乎未被用于牛刀。另外，采用这个方法

还得确定使用的是Java5以上的版本。

习题解答

1 C	A	R								2 M	3 I	L	K									
L				4 S					5 G		T											
A		6 C		T			7 M		A		S				8 B	O	9 I	L	E	R		
S		O		A			U		R		E						N					
S		N		T		10 G	L	O	B	A	L	A	C	11 C	E	S	S	P	O	I	N	T
L		S		I			T		A		F			L		T						
O		T		C			I		G					A		A						
A		R		A			T		E					S		N						
12 D	O	U	B	L	E	C	H	E	C	K	E	D		S		C						
E		C		L			R		O						13 H	E	R	S	H	E	Y	
R		T		Y			E		14 L	A	Z	Y										
S		O					A		L													
		R					D		E													
							I		15 C	H	O	C	-	O	-	H	O	L	I	C		
							N		T													
							G		O													
						16 P	R	I	V	A	T	E										