

## 第 15 章 单例（Singleton）模式

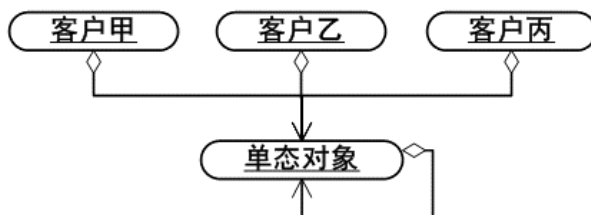
作为对象的创建模式[GOF95]，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

### 15.1 引言

#### 单例模式的要点

单例单例

显然单例模式的要点有三个：一是某各类只能有一个实例；二是它必须自行创建这个事例；三是它必须自行向整个系统提供这个实例。在下面的对象图中，有一个“单例对象”，而“客户甲”、“客户乙”和“客户丙”是单例对象的三个客户对象。可以看到，所有的客户对象共享一个单例对象。而且从单例对象到自身的连接线可以看出，单例对象持有对自己的引用。



#### 资源管理

一些资源管理器常常设计成单例模式。

在计算机系统中，需要管理的资源包括软件外部资源，譬如每台计算机可以有若干个打印机，但只能有一个 **Printer Spooler**，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干传真卡，但是只应该有一个软件负责管理传真卡，以避免出现两份传真作业同时传到传真卡中的情况。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

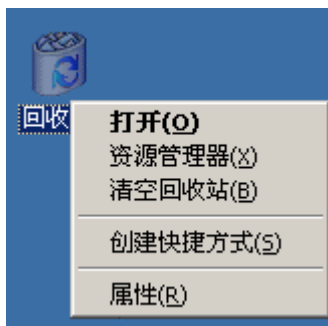
需要管理的资源包括软件内部资源，譬如，大多数的软件都有一个（甚至多个）属性（**properties**）文件存放系统配置。这样的系统应当由一个对象来管理一个属性文件。

需要管理的软件内部资源也包括譬如负责记录网站来访人数的部件，记录软件系统内部事件、出错信息的部件，或是对系统的表现进行检查的部件等。这些部件都必须集中管理，不可政出多头。

这些资源管理器构件必须只有一个实例，这是其一；它们必须自行初始化，这是其二；允许整个系统访问自己这是其三。因此，它们都满足单例模式的条件，是单例模式的应用。

## 一个例子：Windows 回收站

Windows 9x 以后的视窗系统中都有一个回收站，下图就显示了 Windows 2000 的回收站。



在整个视窗系统中，回收站只能有一个实例，整个系统都使用这个惟一的实例，而且回收站自行提供自己的实例。因此，回收站是单例模式的应用。

## 双重检查成例

在本章最后的附录里研究了双重检查成例。双重检查成例与单例模式并无直接的关系，但是由于很多 C 语言设计师在单例模式里面使用双重检查成例，所以这一做法也被很多 Java 设计师所模仿。因此，本书在附录里提醒读者，双重检查成例在 Java 语言里并不能成立，详情请见本章的附录。

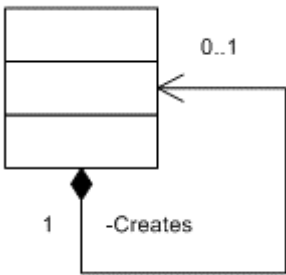
## 15.2 单例模式的结构

单例模式有以下的特点：

- 单例类只可有一个实例。
- 单例类必须自己创建自己这惟一的实例。
- 单例类必须给所有其他对象提供这一实例。

虽然单例模式中的单例类被限定只能有一个实例，但是单例模式和单例类可以很容易被推广到任意且有限多个实例的情况，这时候称它为多例模式（Multiton Pattern）和多例类（Multiton Class），请见“专题：多例（Multiton）模式与多语言支持”一章。单例类的简

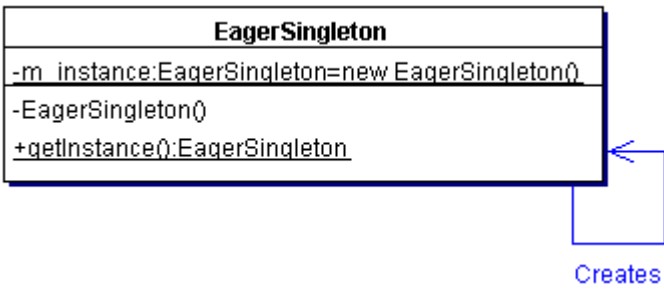
略类图如下所示。



由于 Java 语言的特点，使得单例模式在 Java 语言的实现上有自己的特点。这些特点主要表现在单例类如何将自己实例化上。

饿汉式单例类

饿汉式单例类是在 Java 语言里实现得最为简便的单例类，下面所示的类图描述了一个饿汉式单例类的典型实现。



从图中可以看出，此类已经自己将自己实例化。

代码清单 1：饿汉式单例类

```
public class EagerSingleton
{
    private static final EagerSingleton m_instance =
        new EagerSingleton();
    /**
     * 私有的默认构造子
     */
    private EagerSingleton() { }
    /**
     * 静态工厂方法
     */
    public static EagerSingleton getInstance()
    {
```

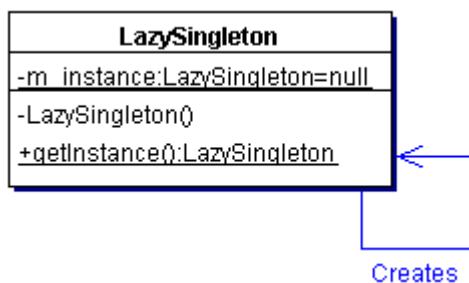
```
        return m_instance;
    }
}
```

读者可以看出，在这个类被加载时，静态变量 `m_instance` 会被初始化，此时类的私有构造子会被调用。这时候，单例类的唯一实例就被创建出来了。

Java 语言中单例类的一个最重要的特点是类的构造子是私有的，从而避免外界利用构造子直接创建出任意多的实例。值得指出的是，由于构造子是私有的，因此，此类不能被继承。

## 懒汉式单例类

与饿汉式单例类相同之处是，类的构造子是私有的。与饿汉式单例类不同的是，懒汉式单例类在第一次被引用时将自己实例化。如果加载器是静态的，那么在懒汉式单例类被加载时不会将自己实例化。如下图所示，类图中给出了一个典型的饿汉式单例类实现。



代码清单 2：懒汉式单例类

```
package com.javapatterns.singleton.demos;
public class LazySingleton
{
    private static LazySingleton
        m_instance = null;
    /**
     * 私有的默认构造子，保证外界无法直接实例化
     */
    private LazySingleton() { }
    /**
     * 静态工厂方法，返还此类的唯一实例
     */
    synchronized public static LazySingleton
        getInstance()
    {
        if (m_instance == null)
        {
```

```
        m_instance = new LazySingleton();
    }
    return m_instance;
}
}
```

读者可能会注意到，在上面给出懒汉式单例类实现里对静态工厂方法使用了同步化，以处理多线程环境。有些设计师在这里建议使用所谓的“双重检查成例”。必须指出的是，“双重检查成例”不可以在 Java 语言中使用。不十分熟悉的读者，可以看看后面给出的小节。

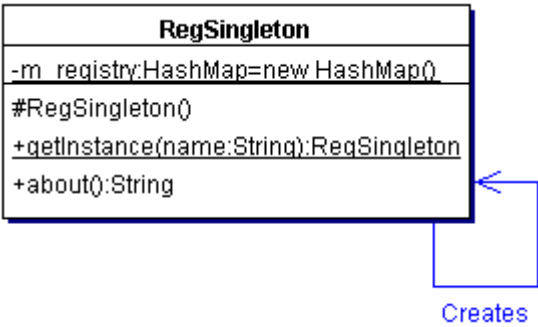
同样，由于构造子是私有的，因此，此类不能被继承。

饿汉式单例类在自己被加载时就将自己实例化。即便加载器是静态的，在饿汉式单例类被加载时仍会将自己实例化。单从资源利用效率角度来讲，这个比懒汉式单例类稍差些。从速度和反应时间角度来讲，则比懒汉式单例类稍好些。然而，懒汉式单例类在实例化时，必须处理好在多个线程同时首次引用此类时的访问限制问题，特别是当单例类作为资源控制器，在实例化时必然涉及资源初始化，而资源初始化很有可能耗费时间。这意味着出现多线程同时首次引用此类的机率变得较大。

饿汉式单例类可以在 Java 语言内实现，但不易在 C++ 内实现，因为静态初始化在 C++ 里没有固定的顺序，因而静态的 m\_instance 变量的初始化与类的加载顺序没有保证，可能会出问题。这就是为什么 GoF 在提出单例类的概念时，举的例子是懒汉式的。他们的书影响之大，以致 Java 语言中单例类的例子也大多是懒汉式的。实际上，本书认为饿汉式单例类更符合 Java 语言本身的特点。

## 登记式单例类

登记式单例类是 GoF 为了克服饿汉式单例类及懒汉式单例类均不可继承的缺点而设计的。本书把他们的例子翻译为 Java 语言，并将它自己实例化的方式从懒汉式改为饿汉式。只是它的子类实例化的方式只能是懒汉式的，这是无法改变的。如下图所示是登记式单例类的一个例子，图中的关系线表明，此类已将自己实例化。



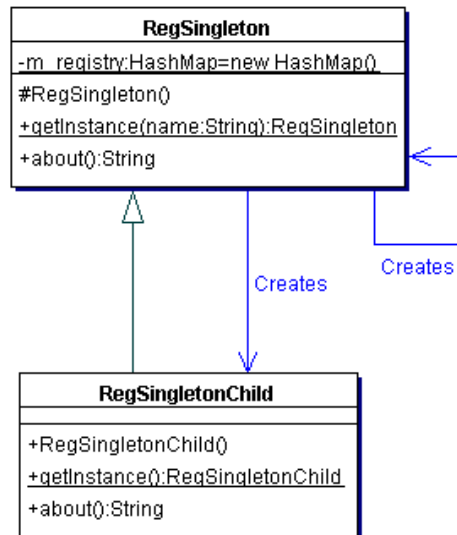
代码清单 3：登记式单例类

```

import java.util.HashMap;
public class RegSingleton
{
    static private HashMap m_registry = new HashMap();
    static
    {
        RegSingleton x = new RegSingleton();
        m_registry.put( x.getClass().getName() ,  x);
    }
    /**
     * 保护的默认构造子
     */
    protected RegSingleton() {}
    /**
     * 静态工厂方法， 返还此类惟一的实例
     */
    static public RegSingleton getInstance(String name)
    {
        if (name == null)
        {
            name = "com.javapatterns.singleton.demos.RegSingleton";
        }
        if (m_registry.get(name) == null)
        {
            try
            {
                m_registry.put( name,
                               Class.forName(name).newInstance() );
            }
            catch(Exception e)
            {
                System.out.println("Error happened.");
            }
        }
        return  (RegSingleton) (m_registry.get(name) );
    }
    /**
     * 一个示意性的商业方法
     */
    public String about()
    {
        return "Hello,  I am RegSingleton.";
    }
}

```

它的子类 `RegSingletonChild` 需要父类的帮助才能实例化。下图所示是登记式单例类子类的一个例子。图中的关系表明，此类是由父类将子类实例化的。



下面是子类的源代码。

代码清单 4：登记式单例类的子类

```

import java.util.HashMap;
public class RegSingletonChild extends RegSingleton
{
    public RegSingletonChild() {}
    /**
     * 静态工厂方法
     */
    static public RegSingletonChild getInstance()
    {
        return (RegSingletonChild)
            RegSingleton.getInstance(
                "com.javapatterns.singleton.demos.RegSingletonChild" );
    }
    /**
     * 一个示意性的商业方法
     */
    public String about()
    {
        return "Hello, I am RegSingletonChild.";
    }
}
  
```

在 GoF 原始的例子中，并没有 `getInstance()` 方法，这样得到子类必须调用的 `getInstance(String name)` 方法并传入子类的名字，因此很不方便。本章在登记式单例类子类的例子里，加入了 `getInstance()` 方法，这样做的好处是 `RegSingletonChild` 可以通过这个方法，返还自己的实例。而这样做的缺点是，由于数据类型不同，无法在 `RegSingleton` 提供

这样一个方法。

由于子类必须允许父类以构造子调用产生实例，因此，它的构造子必须是公开的。这样一来，就等于允许了以这样方式产生实例而不在父类的登记中。这是登记式单例类的一个缺点。

GoF 曾指出，由于父类的实例必须存在才可能有子类的实例，这在有些情况下是一个浪费。这是登记式单例类的另一个缺点。

## 15.3 在什么情况下使用单例模式

### 使用单例模式的条件

使用单例模式有一个很重要的必要条件：

在一个系统要求一个类只有一个实例时才应当使用单例模式。反过来说，如果一个类可以有几个实例共存，那么就没有必要使用单例类。

但是有经验的读者可能会看到很多不当地使用单例模式的例子，可见做到上面这一点并不容易，下面就是一些这样的情况。

#### 例子一

问：我的一个系统需要一些“全程”变量。学习了单例模式后，我发现可以使用一个单例类盛放所有的“全程”变量。请问这样做对吗？

答：这样做是违背单例模式的用意的。单例模式只应当在有真正的“单一实例”的需求时才可使用。

一个设计得当的系统不应当有所谓的“全程”变量，这些变量应当放到它们所描述的实体所对应的类中去。将这些变量从它们所描述的实体类中抽出来，放到一个不相干的单例类中去，会使得这些变量产生错误的依赖关系和耦合关系。

#### 例子二

问：我的一个系统需要管理与数据库的连接。学习了单例模式后，我发现可以使用一个单例类包装一个 `Connection` 对象，并在 `finalize()` 方法中关闭这个 `Connection` 对象。这样的话，在这个单例类的实例没有被人引用时，这个 `finalize()` 对象就会被调用，因此，`Connection` 对象就会被释放。这多妙啊。

答：这样做是不恰当的。除非有单一实例的需求，不然不要使用单例模式。在这里 `Connection` 对象可以同时有几个实例共存，不需要是单一实例。

单例模式有很多的错误使用案例都与此例子相似，它们都是试图使用单例模式管理共



---

享资源的生命周期，这是不恰当的。

## 15.4 单例类的状态

### 有状态的单例类

一个单例类可以有状态的 (stateful)，一个有状态的单例对象一般也是可变 (mutable) 单例对象。

有状态的可变的单例对象常常当做状态库 (repository) 使用。比如一个单例对象可以持有一个 int 类型的属性，用来给一个系统提供一个数值惟一的序列号码，作为某个贩卖系统的账单号码。

当然，一个单例类可以持有一个聚集，从而允许存储多个状态。

### 没有状态的单例类

另一方面，单例类也可以是没有状态的 (stateless)，仅用做提供工具性函数的对象。既然是为了提供工具性函数，也就没有必要创建多个实例，因此使用单例模式很合适。一个没有状态的单例类也就是不变 (Immutable) 单例类；关于不变模式，读者可以参见本书的“不变 (Immutable) 模式”一章。

### 多个 JVM 系统的分散式系统

EJB 容器有能力将一个 EJB 的实例跨过几个 JVM 调用。由于单例对象不是 EJB，因此，单例类局限于某一个 JVM 中。换言之，如果 EJB 在跨过 JVM 后仍然需要引用同一个单例类的话，这个单例类就会在数个 JVM 中被实例化，造成多个单例对象的实例出现。一个 J2EE 应用系统可能分布在数个 JVM 中，这时候不一定需要 EJB 就能造成多个单例类的实例出现在不同 JVM 中的情况。

如果这个单例类是没有状态的，那么就没有问题。因为没有状态的对象是没有区别的。但是如果这个单例类是有状态的，那么问题就来了。举例来说，如果一个单例对象可以持有一个 int 类型的属性，用来给一个系统提供一个数值惟一的序列号码，作为某个贩卖系统的账单号码的话，用户会看到同一个号码出现好几次。

在任何使用了 EJB、RMI 和 JINI 技术的分散式系统中，应当避免使用有状态的单例模式。

### 多个类加载器

同一个 JVM 中会有多个类加载器，当两个类加载器同时加载同一个类时，会出现两个

实例。在很多 J2EE 服务器允许同一个服务器内有几个 Servlet 引擎时，每一个引擎都有独立的类加载器，经有不同的类加载器加载的对象之间是绝缘的。

比如一个 J2EE 系统所在的 J2EE 服务器中有两个 Servlet 引擎：一个作为内网给公司的网站管理人员使用；另一个给公司的外部客户使用。两者共享同一个数据库，两个系统都需要调用同一个单例类。如果这个单例类是有状态的单例类的话，那么内网和外网用户看到的单例对象的状态就会不同。

除非系统有协调机制，不然在这种情况下应当尽量避免使用有状态的单例类。

## 15.5 一个实用的例子：属性管理器

### 什么是属性文件

这里给出一个读取属性 (properties) 文件的单例类，作为单例模式的一个实用的例子。属性文件如同老式的视窗编程时的 .ini 文件，用于存放系统的配置信息。配置信息在属性文件中以属性的方式存放，一个属性就是两个字符串组成的对子，其中一个字符串是键 (key)，另一个字符串是这个键的值 (value)。

大多数的系统都有一些配置常量，这些常量如果是存储在程序内部的，那么每一次修改这些常量都需要重新编译程序。将这些常量放在配置文件中，系统通过访问这个配置文件取得配置常量，就可以通过修改配置文件而无需修改程序而达到更改系统配置的目的。

系统也可以在配置文件中存储一些工作环境信息，这样在系统重启时，这些工作信息可以延续到下一个运行周期中。

假定需要读取的属性文件就在当前目录中，且文件名为 singleton.properties。这个文件中有如下的一些属性项。

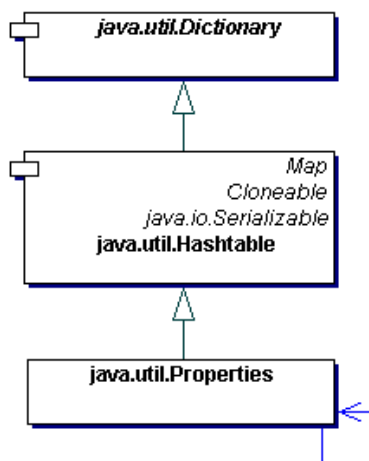
代码清单 5：属性文件内容

```
node1.item1=How  
node1.item2=are  
node2.item1=you  
node2.item2=doing  
node3.item1=?
```

例如，node1.item1 就是一个键，而 How 就是这个键所对应的值。

### Java 属性类

Java 提供了一个工具类，称做属性类，可以用来完成 Java 属性和属性文件的操作。这个属性类的继承关系可以从下面的类图中看清楚。



属性类提供了读取属性和设置属性的各种方法。其中读取属性的方法有：

- `contains(Object value)`、`containsKey(Object key)`：如果给定的参数或属性关键字在属性表中有定义，该方法返回 `True`，否则返回 `False`。
- `getProperty(String key)`、`getProperty(String key, String default)`：根据给定的属性关键字获取关键字值。
- `list(PrintStream s)`、`list(PrintWriter w)`：在输出流中输出属性表内容。
- `size()`：返回当前属性表中定义的属性关键字个数。

设置属性的方法有：

- `put(Object key, Object value)`：向属性表中追加属性关键字和关键字的值。
- `remove(Object key)`：从属性表中删除关键字。

从属性文件加载属性的方法为 `load(InputStream inStream)`，可以从一个输入流中读入一个属性列，如果这个流是来自一个文件的话，这个方法就从文件中读入属性。

将属性存入属性文件的方法有几个，重要的一个是 `store(OutputStream out, String header)`，将当前的属性列写入一个输出流，如果这个输出流是导向一个文件的，那么这个方法就将属性流存入文件。

## 为什么需要使用单例模式

属性是系统的一种“资源”，应当避免有多余一个的对象读取特别是存储属性。此外，属性的读取可能会在很多地方发生，创建属性对象的地方应当在哪里不是很清楚。换言之，属性管理器应当自己创建自己的实例，并且自己向系统全程提供这一事例。

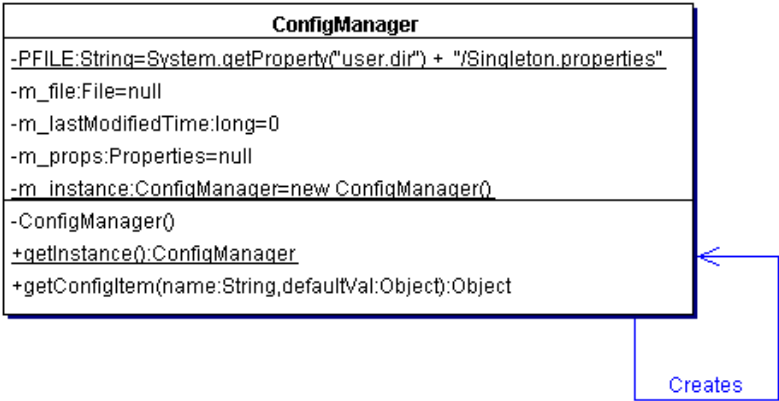
因此，属性文件管理器应当是一个单例模式负责。

## 系统设计

系统的核心是一个属性管理器，也就是一个叫做 `ConfigManager` 的类，这个类应当是

一个单例类。因此，这个类应当有一个静态工厂方法，不妨叫做 `getInstance()`，用于提供自己的实例。

为简单起见，本书在这里采取“饿汉”方式实现 `ConfigManager`。例子的类图如下所示。



本例子的源代码如下所示。

代码清单 6: `ConfigManager` 的源代码

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.File;
public class ConfigManager
{
    /**
     * 属性文件全名
     */
    private static final String PFILE =
        System.getProperty("user.dir")
        + File.Separator + "Singleton.properties";
    /**
     * 对应于属性文件的文件对象变量
     */
    private File m_file = null;
    /**
     * 属性文件的最后修改日期
     */
    private long m_lastModifiedTime = 0;
    /**
     * 属性文件所对应的属性对象变量
     */
    private Properties m_props = null;
    /**
     * 本类可能存在的惟一的一个实例
     */
    private static ConfigManager m_instance =
```

```
        new ConfigManager();
    /**
     * 私有的构造子， 用以保证外界无法直接实例化
     */
    private ConfigManager()
    {
        m_file = new File(PFILE);
        m_lastModifiedTime = m_file.lastModified();
        if(m_lastModifiedTime == 0)
        {
            System.err.println(PFILE +
                " file does not exist!");
        }
        m_props = new Properties();
        try
        {
            m_props.load(new FileInputStream(PFILE));
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    /**
     * 静态工厂方法
     * @return 返回 ConfigManager 类的单一实例
     */
    synchronized public static ConfigManager
        getInstance()
    {
        return m_instance;
    }
    /**
     * 读取一特定的属性项
     *
     * @param name 属性项的项名
     * @param defaultVal 属性项的默认值
     * @return 属性项的值（如此项存在）， 默认值（如此项不存在）
     */
    final public Object getConfigItem(
        String name,  Object defaultVal)
    {
        long newTime = m_file.lastModified();
        // 检查属性文件是否被其他程序
        // （多数情况是程序员手动）修改过
        // 如果是，重新读取此文件
```

```
        if(newTime == 0)
        {
            // 属性文件不存在
            if(m_lastModifiedTime == 0)
            {
                System.err.println(PFILE
                    + " file does not exist!");
            }
            else
            {
                System.err.println(PFILE
                    + " file was deleted!!");
            }
            return defaultVal;
        }
        else if(newTime > m_lastModifiedTime)
        {
            // Get rid of the old properties
            m_props.clear();
            try
            {
                m_props.load(new FileInputStream(PFILE));
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
        m_lastModifiedTime = newTime;
        Object val = m_props.getProperty(name);
        if( val == null )
        {
            return defaultVal;
        }
        else
        {
            return val;
        }
    }
}
```

在上面直接使用了一个局域的常量储存储属性文件的路径。在实际的系统中，读者可以采取更灵活的方式将属性文件的路径传入。

读者可以看到，这个管理器类有一个很有意思的功能，即在每一次调用时，检查属性文件是否已经被更新过。如果确实已经被更新过的话，管理器会自动重新加载属性文件，从而保证管理器的内容与属性文件的内容总是一致的。

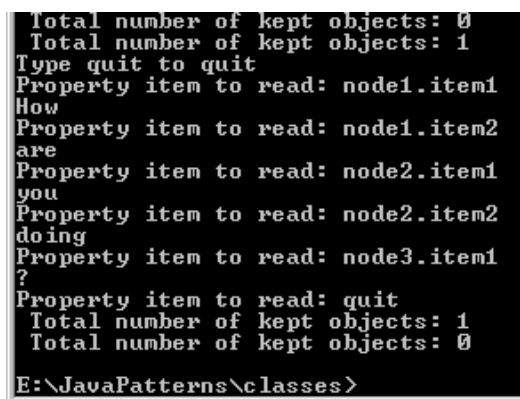
## 怎样调用属性管理器

下面的源代码演示了怎样调用 `ConfigManager` 来读取属性文件。

代码清单 7：怎样调用 `ConfigManager` 类以读取属性文件

```
BufferedReader reader = new BufferedReader(  
    new InputStreamReader(System.in));  
System.out.println("Type quit to quit");  
do  
{  
    System.out.print("Property item to read: ");  
    String line = reader.readLine();  
    if(line.equals("quit"))  
    {  
        break;  
    }  
    System.out.println(ConfigManager.getInstance()  
        .getConfigItem(line, "Not found."));  
} while(true);
```

上面代码运行时的情况如下图所示。



```
Total number of kept objects: 0  
Total number of kept objects: 1  
Type quit to quit  
Property item to read: node1.item1  
How  
Property item to read: node1.item2  
are  
Property item to read: node2.item1  
you  
Property item to read: node2.item2  
doing  
Property item to read: node3.item1  
?  
Property item to read: quit  
Total number of kept objects: 1  
Total number of kept objects: 0  
E:\JavaPatterns\classes>
```

感兴趣的读者可以参考阅读本书的“专题：XMLProperties 与适配器模式”一章，那里对使用 Java 属性类和 XML 文件格式做了有用的讨论。

## 15.6 Java 语言中的单例模式

Java 语言中就有很多的单例模式的应用实例，这里讨论比较有名的几个。



## Java 的 Runtime 对象

在 Java 语言内部, `java.lang.Runtime` 对象就是一个使用单例模式的例子。在每一个 Java 应用程序里面, 都有惟一的一个 `Runtime` 对象。通过这个 `Runtime` 对象, 应用程序可以与其运行环境发生相互作用。

`Runtime` 类提供一个静态工厂方法 `getRuntime():`

```
public static Runtime getRuntime();
```

通过调用此方法, 可以获得 `Runtime` 类惟一的一个实例:

```
Runtime rt = Runtime.getRuntime();
```

`Runtime` 对象通常的用途包括: 执行外部命令; 返回现有内存即全部内存; 运行垃圾收集器; 加载动态库等。下面的例子演示了怎样使用 `Runtime` 对象运行一个外部程序。

代码清单 8: 怎样使用 `Runtime` 对象运行一个外部命令

```
import java.io.*;

public class CmdTest
{
    public static void main(String[] args) throws IOException
    {
        Process proc = Runtime.getRuntime().exec("notepad.exe");
    }
}
```

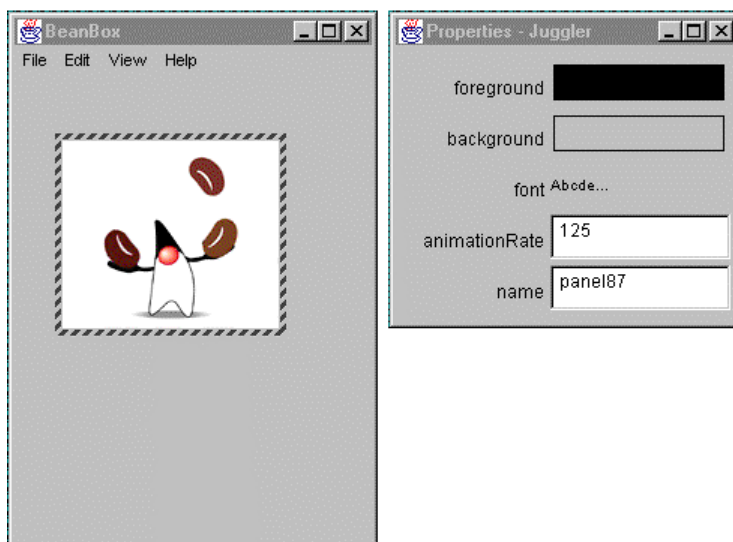
上面的程序在运行时会打开 `notepad` 程序。应当指出的是, 在 Windows 2000 的环境中, 如果需要打开一个 Word 文件, 而又不想指明 Word 软件安装的位置时, 可以使用下面的做法:

```
Process proc = Runtime.getRuntime().exec(
    "cmd /E:ON /c start MyDocument.doc");
```

在上面, 被执行的命令是 `start MyDocument.doc`, 开关 `E:ON` 指定 DOS 命令处理器允许命令扩展, 而开关 `/C` 指明后面跟随的字符串是命令, 并在执行命令后关闭 DOS 窗口, `start` 命令会开启一个单独的窗口执行所提供的命令。

## Introspector 类

一般的应用程序可能永远也不会直接用到 `Introspector` 类, 但读者应该知道 `Introspector` 是做什么的。Sun 提供了一个叫做 `BeanBox` 的系统, 允许动态地加载 `JavaBean`, 并动态地修改其性质。`BeanBox` 在运行时的情况如下图所示。



在上面的图中显示了 BeanBox 最重要的两个视窗，一个叫做 BeanBox 视窗，另一个叫做性质视窗。在上面的 BeanBox 视窗中显示了一个 Juggler Bean 被放置到视窗中的情况。相应的，在性质视窗中显示了 Juggler Bean 的所有性质。所有的 Java 集成环境都提供这种功能，这样的系统就叫做 BeanBox 系统。

BeanBox 系统使用一种自省（Introspection）过程来确定一个 Bean 所输出的性质、事件和方法。这个自省机制是通过自省者类，也即 `java.util.Introspector` 类实现的；这个机制是建立在 Java 反射（Reflection）机制和命名规范的基础之上的。比如，`Introspector` 类可以确定 Juggler Bean 所支持的所有的性质，这是因为 `Introspector` 类可以得到所有的方法，然后将其中的取值和赋值方法以及它们的特征加以比较，从而得出结果。

显然，在整个 BeanBox 系统中只需要一个 `Introspector` 对象，下面所示就是这个类的结构图。

java.beans.Introspector
<pre> +getBeanInfo:BeanInfo +getBeanInfo:BeanInfo +getBeanInfo:BeanInfo +decapitalize:String +getBeanInfoSearchPath:String[] +setBeanInfoSearchPath:void +flushCaches:void +flushFromCaches:void -Introspector -getBeanInfo:GenericBeanInfo -findInformant:BeanInfo -getTargetPropertyInfo:PropertyDescriptor[] addProperty:void -getTargetEventInfo:EventSetDescriptor[] addEvent:void -getTargetMethodInfo:MethodDescriptor[] -addMethod:void -makeQualifiedMethodName:String -getTargetDefaultEventIndex:int -getTargetDefaultPropertyIndex:int -getTargetBeanDescriptor:BeanDescriptor -isEventHandler:boolean -getPublicDeclaredMethods:Method[] -internalFindMethod:Method -internalFindMethod:Method findMethod:Method findMethod:Method isSubclass:boolean -throwException:boolean instantiate:Object </pre>

可以看出，Introspector 类的构造子是私有的，一个静态工厂方法 instantiate() 提供了 Introspector 类的唯一实例。换言之，这个类是单例模式的应用。

## java.awt.Toolkit 类

Toolkit 类是一个非常有趣的单例模式的例子。Toolkit 使用单例模式创建所谓的 Toolkit 的默认对象，并且确保这个默认实例在整个系统中是唯一的。Toolkit 类提供了一个静态的方法 getDefaultToolkit() 来提供这个唯一的实例，这个方法相当于懒汉式的单例方法，因此整个方法都是同步化的。

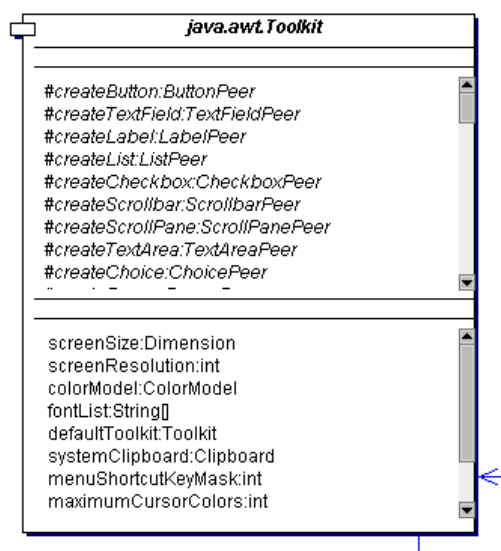
代码清单 9: getDefaultToolkit() 方法

```

public static synchronized Toolkit
    getDefaultToolkit()
{
    .....
}

```

Toolkit 类的类图如下所示。



其中性质 `defaultToolkit` 实际上就是静态的 `getDefaultToolkit` 类。

有趣的是，由于 `Toolkit` 是一个抽象类，因此其子类如果提供一个私有的构造子，那么其子类便是一个正常的单例类；而如果其子类作为具体实现提供一个公开的构造子，这时候这个具体子类便是“不完全”的单例类。关于“不完全”的单例类的讨论请见本章后面的“专题：不完全的单例类”一节。

## 模版方法模式

同时，熟悉模版方法模式的读者可以看出，`getDefaultToolkit()`方法实际上是一个模版方法。私有构造子是推迟到子类实现的剩余逻辑，根据子类对这个剩余逻辑的不同实现，子类就可以提供完全不同的行为。对 `Toolkit` 的子类而言，私有构造子依赖于操作系统，不同的子类可以根据不同的操作系统而给出不同的逻辑，从而使 `Toolkit` 的子类对不同的操作系统给出不同的行为。

## `javax.swing.TimerQueue` 类

这是一个不完全的单例类，由于这个类是在 `Swing` 的定时器类中使用的，因此将在“观察者模式与 `Swing` 定时器”一章中介绍。关于“不完全”的单例类的讨论请见本章下面的“专题：不完全的单例类”一节。

## 15.7 专题：不完全的单例类

### 什么是不完全的单例类

估计有些读者见过下面这样的“不完全”的单例类。

代码清单 10：“不完全”单例类

```
package com.javapatterns.singleton.demos;
public class LazySingleton
{
    private static LazySingleton
        m_instance = null;
    /**
     * 公开的构造子，外界可以直接实例化
     */
    public LazySingleton() { }
    /**
     * 静态工厂方法
     * @return 返回 LazySingleton 类的惟一实例
     */
    synchronized public static
        LazySingleton getInstance()
    {
        if (m_instance == null)
        {
            m_instance = new LazySingleton();
        }
        return m_instance;
    }
}
```

上面的代码乍看起来是一个“懒汉”式单例类，仔细一看，发现有一个公开的构造子。由于外界可以使用构造子创建出任意多个此类的实例，这违背了单例类只能有一个（或有限个）实例的特性，因此这个类不是完全的单例类。这种情况有时会出现，比如 `javax.swing.TimerQueue` 便是一例，关于这个类，请参见“观察者模式与 Swing 定时器”一章。

造成这种情况出现的原因有以下几种可能：

（1）初学者的错误。许多初学者没有认识到单例类的构造子不能是公开的，因此犯下这个错误。有些初学 Java 语言的学员甚至不知道一个 Java 类的构造子可以不是公开的。在这种情况下，设计师可能会通过自我约束，也就是说不去调用构造子的办法，将这个不完全的单例类在使用中作为一个单例类使用。

在这种情况下，一个简单的矫正办法，就是将公开的构造子改为私有的构造子。

(2) 当初出于考虑不周，将一个类设计成为单例类，后来发现此类应当有多于一个的实例。为了弥补错误，干脆将构造子改为公开的，以便在需要多于一个的实例时，可以随时调用构造子创建新的实例。

要纠正这种情况较为困难，必须根据具体情况做出改进的决定。如果一个类在最初被设计成为单例类，但后来发现实际上此类应当有有限多个实例，这时候应当考虑是否将单例类改为多例类 (Multiton)。

(3) 设计师的 Java 知识很好，而且也知道单例模式的正确使用方法，但是还是有意使用这种不完全的单例模式，因为他意在使用一种“改良”的单例模式。这时候，除去共有的构造子不符合单例模式的要求之外，这个类必须是很好的单例模式。

## 默认实例模式

有些设计师将这种不完全的单例模式叫做“默认实例模式” (Default Instance Pattern)。在所谓的“默认实例模式”里面，一个类提供静态的方法，如同单例模式一样，同时又提供一个公开的构造子，如同普通的类一样。

这样做的惟一好处是，这种模式允许客户端选择如何将类实例化：创建新的自己独有的实例，或者使用共享的实例。

这样一来，由于没有任何的强制性措施，客户端的选择不一定是合理的选择。其结果是设计师往往不会花费时间在如何提供最好的选择上，而是不恰当地将这种选择交给客户端的程序员，这样必然会导致不理想的设计和欠考虑的实现。

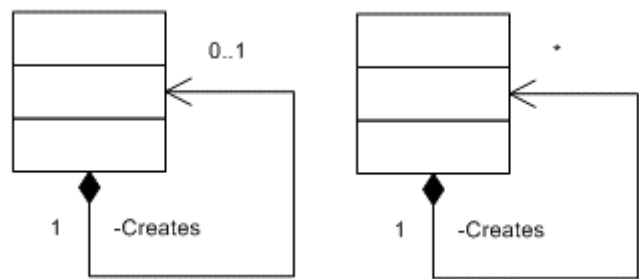
本书建议读者不要这样做。

## 15.8 相 关 模 式

有一些模式可以使用单例模式，如抽象工厂模式可以使用单例模式，将具体工厂类设计成单例类；建造模式可以使用单例模式，将具体建造类设计成单例类。

## 多例（Multiton）模式

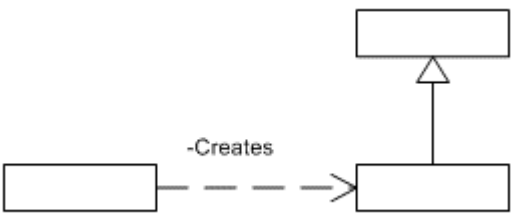
正如同本章所说的，单例模式的精神可以推广到多于一个实例的情况。这时候这种类叫做多例类，这种模式叫做多例模式。单例类（左）和多例类（右）的类图如下所示。



关于多例模式，请见本书“专题：多例（Multiton）模式与多语言支持”一章。

## 简单工厂（Simple Factory）模式

单例模式使用了简单工厂模式（又称为静态工厂方法模式）来提供自己的实例。在上面 ConfigManager 例子的代码中，静态工厂方法 getInstance()就是静态工厂方法。在 java.awt.Toolkit 类中，getDefaultToolkit()方法就是静态工厂方法。简单工厂模式的简略类图如下所示。



本章讨论了单例模式的结构和实现方法。

单例模式是一个看上去很简单的模式，很多设计师最先学会的往往是单例模式。然而，随着 Java 系统日益变得复杂化和分散化，单例模式的使用变得比过去困难。

本书提醒读者在分散式的 Java 系统中使用单例模式时，尽量不要使用有状态的。

## 问答题

1. 为什么不使用一个静态的“全程”原始变量，而要建一个类？一个静态的原始变量当然只能有一个值，自然而然不就是“单例”的吗？

2. 举例说明如何调用 EagerSingleton 类。
3. 举例说明如何调用 RegSingleton 类和 RegSingletonChild 类。
4. 请问 java.lang.Math 类和 java.lang.StrictMath 类是否是单例模式？
5. 我们公司只购买了一个 JDBC 驱动软件的单用户使用许可，可否使用单例模式管理通过 JDBC 驱动软件连接的数据库？

## 问答题答案

1. 单例模式可以提供很复杂的逻辑，而一个原始变量不能自己初始化，不可能有继承的关系，没有内部结构。因此单例模式有很多优越之处。

在 Java 语言里并没有真正的“全程”变量，一个变量必须属于某一个类或者某一个实例。而在复杂的程序当中，一个静态变量的初始化发生在哪里常常是一个不易确定的问题。

当然，使用“全程”原始变量并没有什么错误，就好像选择使用 Fortran 语言而非 Java 语言编程并不是一种对错的问题一样。

2. 几种单例类的使用方法如下。

代码清单 11：几种单例类的使用方法

```
public class RegSingletonTest
{
    public static void main(String[] args)
    {
        //(1) Test eager
        System.out.println( EagerSingleton.getInstance());
        //(2) Test reg
        System.out.println(
            RegSingleton.getInstance(
                "com.javapatterns.singleton.demos.RegSingleton").about());
        System.out.println( RegSingleton.getInstance(null).about() );
        System.out.println(
            RegSingleton.getInstance(
                "com.javapatterns.singleton.demos.RegSingletonChild").about());
        System.out.println( RegSingletonChild.getInstance().about());
    }
}
```

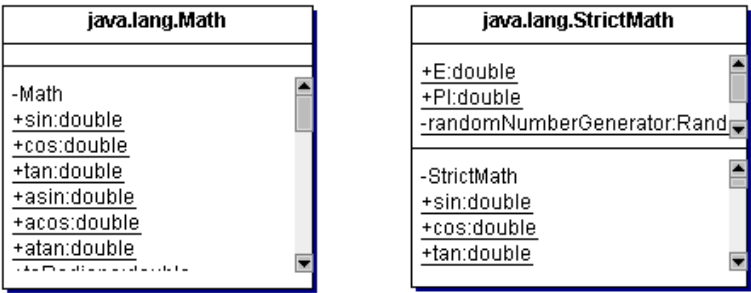
3. 见上题答案。

4. 它们都不是单例类。原因如下：

这两个类均有一个私有的构造子。但是这仅仅是单例模式的必要条件，而不是充分条件。回顾在本章开始提出的单例模式的三个特性可以看出，无论是 Math 还是 StrictMath 都没有为外界提供任何自身的实例。实际上，这两个类都是被设计来提供静态工厂方法和常量的，因此从来就不需要它们的实例，这才是它们的构造子是私有的原因。Math 和 StrictMath



类的类图如下所示。



5. 这样做是可行的，只是必须注意当使用在分散式系统中的时候，不一定能保证单例类实例的惟一性。

## 15.9 附录：双重检查成例的研究

成例是一种代码层次上的模式，是在比设计模式的层次更具体的层次上的代码技巧。成例往往与编程语言密切相关。双重检查成例（Double Check Idiom）是从 C 语言移植过来的一种代码模式。在 C 语言里，双重检查成例常常用在多线程环境中类的晚实例化（Late Instantiation）里。

本节之所以要介绍这个成例（严格来讲，是介绍为什么这个成例不成立），是因为有很多人认为双重检查成例可以使用在“懒汉”单例模式里面。

### 什么是双重检查成例

为了解释什么是双重检查成例，请首先看看下面没有使用任何线程安全考虑的错误例子。

#### 从单线程的程序谈起

首先考虑一个单线程的版本。

代码清单 13：没有使用任何线程安全措施的一个例子

```
// Single threaded version
class Foo
{
    private Helper helper = null;
    public Helper getHelper()
    {
        if (helper == null)
        {
            helper = new Helper();
        }
    }
}
```

```

    }
    return helper;
}
// other functions and members...
}

```

这是一个错误的例子，详情请见下面的说明。

写出这样的代码，本意显然是要保持在整个 JVM 中只有一个 `Helper` 的实例；因此，才会有 `if(helper == null)` 的检查。非常明显的是，如果在多线程的环境中运行，上面的代码会有两个甚至两个以上的 `Helper` 对象被创建出来，从而造成错误。

但是，想像一下在多线程环境中的情形就会发现，如果有两个线程 A 和 B 几乎同时到达 `if(helper == null)` 语句的外面的话，假设线程 A 比线程 B 早一点点，那么：

(1) A 会首先进入 `if(helper == null)` 块的内部，并开始执行 `new Helper()` 语句。此时，`helper` 变量仍然是 `null`，直到线程 A 的 `new Helper()` 语句返回并给 `helper` 变量赋值为止。

(2) 但是，线程 B 并不会在 `if(helper == null)` 语句的外面等待，因为此时 `helper == null` 是成立的，它会马上进入 `if(helper == null)` 语句块的内部。这样，线程 B 会不可避免地执行 `helper = new Helper()` 语句，从而创建出第二个实例来。

(3) 线程 A 的 `helper = new Helper()` 语句执行完毕后，`helper` 变量得到了真实的对象引用，`(helper == null)` 不再为真。第三个线程不会再进入 `if(helper == null)` 语句块的内部了。

(4) 线程 B 的 `helper = new Helper()` 语句也执行完毕后，`helper` 变量的值被覆盖。但是第一个 `Helper` 对象被线程 A 引用的事实不会改变。

这时，线程 A 和 B 各自拥有一个独立的 `Helper` 对象，而这是错误的。

### 线程安全的版本

为了克服没有线程安全的缺点，下面给出一个线程安全的例子。

代码清单 14：这是一个正确的答案

```

// Correct multithreaded version
class Foo
{
    private Helper helper = null;
    public synchronized Helper getHelper()
    {
        if(helper == null)
        {
            helper = new Helper();
            return helper;
        }
    }
    // other functions and members...
}

```

显然，由于整个静态工厂方法都是同步化的，因此，不会有两个线程同时进入这个方法。因此，当线程 A 和 B 作为第一批调用者同时或几乎同时调用此方法时：

- (1) 早到一点的线程 A 会率先进入此方法，同时线程 B 会在方法外部等待。
  - (2) 对线程 A 来说，`helper` 变量的值是 `null`，因此 `helper = new Helper();` 语句会被执行。
  - (3) 线程 A 结束对方法的执行，`helper` 变量的值不再是 `null`。
  - (4) 线程 B 进入此方法，`helper` 变量的值不再是 `null`，因此 `helper = new Helper();` 语句不会被执行。线程 B 取到的是 `helper` 变量所含有的引用，也就是对线程 A 所创立的 `Helper` 实例的引用。
- 显然，线程 A 和 B 持有同一个 `Helper` 实例，这是正确的。

### 画蛇添足的“双重检查”

但是，仔细审察上面的正确答案会发现，同步化实际上只在 `helper` 变量第一次被赋值之前才有用。在 `helper` 变量有了值以后，同步化实际上变成了一个不必要的瓶颈。如果能有一个方法去掉这个小小的额外开销，不是更加完美了吗？因此，就有了下面这个设计“巧妙”的双重检查成例。在读者向下继续读之前，有必要提醒一句：正如本小节的标题所标明的那样，这是一个反面教材，因为双重检查成例在 Java 编译器里无法实现。

代码清单 15：使用双重检查成例的懒汉式单例模式

```
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo
{
    private Helper helper = null;
    public Helper getHelper()
    {
        if (helper == null)    //第一次检查(位置 1)
        {
            //这里会有多于一个的线程同时到达 (位置 2)
            synchronized(this)
            {
                //这里在每个时刻只能有一个线程 (位置 3)
                if (helper == null) //第二次检查 (位置 4)
                {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
    // other functions and members...
}
```



这是一个错误的例子，详情请见下面的解释。

对于初次接触双重检查成例的读者来说，这个技巧的思路并不明显易懂。因此，本节在这里给出一个详尽的解释。同样，这里假设线程 A 和 B 作为第一批调用者同时或几乎同时调用静态工厂方法。

(1) 因为线程 A 和 B 是第一批调用者，因此，当它们进入此静态工厂方法时，`helper` 变量是 `null`。因此，线程 A 和 B 会同时或几乎同时到达位置 1。

(2) 假设线程 A 会首先到达位置 2，并进入 `synchronized(this)` 到达位置 3。这时，由于 `synchronized(this)` 的同步化限制，线程 B 无法到达位置 3，而只能在位置 2 等候。

(3) 线程 A 执行 `helper = new Helper()` 语句，使得 `helper` 变量得到一个值，即对一个 `Helper` 对象的引用。此时，线程 B 只能继续在位置 2 等候。

(4) 线程 A 退出 `synchronized(this)`，返回 `Helper` 对象，退出静态工厂方法。

(5) 线程 B 进入 `synchronized(this)` 块，达到位置 3，进而达到位置 4。由于 `helper` 变量已经不是 `null` 了，因此线程 B 退出 `synchronized(this)`，返回 `helper` 所引用的 `Helper` 对象(也就是线程 A 所创建的 `Helper` 对象)，退出静态工厂方法。

到此为止，线程 A 和线程 B 得到了同一个 `Helper` 对象。可以看到，在上面的方法 `getInstance()` 中，同步化仅用来避免多个线程同时初始化这个类，而不是同时调用这个静态工厂方法。如果这是正确的，那么使用这一个成例之后，“懒汉式”单例类就可以摆脱掉同步化瓶颈，达到一个很妙的境界。

代码清单 16：使用了双重检查成例的懒汉式单例类

```
public class LazySingleton
{
    private static LazySingleton m_instance = null;
    private LazySingleton() { }
    /**
     * 静态工厂方法
     */
    public static LazySingleton getInstance()
    {
        if (m_instance == null)
        {
            //More than one threads might be here!!!
            synchronized(LazySingleton.class)
            {
                if (m_instance == null)
                {
                    m_instance = new LazySingleton();
                }
            }
        }
        return m_instance;
    }
}
```



这是一个错误的例子，请见下面的解释。

第一次接触到这个技巧的读者必定会有很多问题，诸如第一次检查或者第二次检查可不可以省掉等。回答是：按照多线程的原理和双重检查成例的预想方案，它们是不可以省掉的。本节不打算讲解的原因在于双重检查成例在 Java 编译器中根本不能成立。

## 双重检查成例对 Java 语言编译器不成立

令人吃惊的是，在 C 语言里得到普遍应用的双重检查成例在多数的 Java 语言编译器里面并不成立[BLOCH01, GOETZ01, DCL01]。上面使用了双重检查成例的“懒汉式”单例类，不能工作的基本原因在于，在 Java 编译器中，LazySingleton 类的初始化与 m\_instance 变量赋值的顺序不可预料。如果一个线程在没有同步化的条件下读取 m\_instance 引用，并调用这个对象的方法的话，可能会发现对象的初始化过程尚未完成，从而造成崩溃。

文献[BLOCH01]指出：一般而言，双重检查成立对 Java 语言来说是不成立的。

## 15.10 给读者的一点建议

有很多非常聪明的人在这个成例的 Java 版本上花费了非常多的时间，到现在为止人们得出的结论是：一般而言，双重检查成例无法在现有的 Java 语言编译器里工作[BLOCH01, GOETZ01, DCL01]。

读者可能会问，是否有可能通过某种技巧对上面的双重检查的实现代码加以修改，从而使某种形式的双重检查成例能在 Java 编译器下工作呢？这种可能性当然不能排除，但是除非读者对此有特别的兴趣，建议不要在这上面花费太多的时间。

在一般情况下使用饿汉式单例模式或者对整个静态工厂方法同步化的懒汉式单例模式足以解决在实际设计工作中遇到的问题。

## 参考文献

[BLOCH01] Joshua Bloch, Effective Java - Programming Language Guide, published by Addison-Wesley, 2001.

[NOBLE97] James Noble, GOF Patterns for GUI Design, preprint of Macquarie University, Sydney Australia, June 1, 1997.

[DCL01] David Bacon et al, The "Double-Checked Locking is Broken" Declaration, preprint of University of Maryland, <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

[GOETZ01] Brian Goetz, Can ThreadLocal solve the double-checked locking problem? ,  
JavaWorld, November 2001,  
[http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl\\_p.html](http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl_p.html)  
[FOX01] Joshua Fox, When is a Singleton Not a Singleton? , JavaWorld, January 2001