

## 重要声明

1: 本文并非本人原创, 原创作者为 JavaEye 上的一位大牛人, 网名为 chjavach, 博客地址为: <http://chjavach.javaeye.com/>

2: 这位牛人在 JavaEye 发表一系列关于设计模式的文章, 广受欢迎, 是 JavaEye 网站上设计模式类的第一牛人, 几乎每篇博客, 都在 JavaEye 的博客首页长时间排第一, 超受欢迎。

3: 个人觉得是看过的设计模式类最好的了, 特为它制作电子书, 跟大家分享。其实就是从网页上把内容 copy 下来, 再稍稍排了一下版, 原有的文字, 一个字都没敢动, 保证原汁原味。

4: 如果大家喜欢, 我会陆续把这位大牛的其他关于设计模式的文章都制作成电子书, 跟大家分享, 先跟大家分享单例模式。

## 部分 JavaEye 网友对单例模式的评论

直接从该牛人博客单例模式部分的评论里面摘录几个, 看看 JavaEye 网友的评价:

[ktex263](#) 2010-07-26 [引用](#)

简洁明了, 楼主写得不错, 期待后续大作 🍌

[gstarwd](#) 2010-07-30 [引用](#)

很精彩啊~~一步一步 引人入胜啊

[rather\\_lonely](#) 2010-07-29 [引用](#)

每天第一件事就是看楼主文章有没有更新

[dayday\\_up1](#) 2010-08-04 [引用](#)

比国内某些所谓的大师写的设计模式书好多了, 比国外的大牛写的书似乎更容易让人懂, 顶博主。

[ilovehome](#) 2010-08-02 [引用](#)

看了你所有的文章, 写得超级棒, 要是出书了, 一定买一本, 很多内容值得细细咀嚼. 🍌

[EnterLee](#) 2010-08-02 [引用](#)

🍌 每天在开始工作前都要来看一次您的博客, 看看有没有更新, 期待楼主的后续模式剖析, 期待楼主出书, 如果出书了请告诉我一声。谢谢

[kairos](#) 2010-08-03 [引用](#)

写的非常的棒, 每一篇都非常的精彩!

更欣赏博主这种 share 的精神。

牛人很多, 可是能与大家 share 的人并不多。

非常非常的赞, 希望楼主继续。

[cxqdji](#) 2010-08-14 [引用](#)

🍌🍌 好得没话说

以下是原始博文

## 研磨设计模式之单例模式-1

原文地址: <http://chjavach.javaeye.com/blog/721076>

看到很多朋友在写单例，也来凑个热闹，虽然很简单，但是也有很多知识点在单例里面，看看是否能写出点不一样来。

### 单例模式 (Singleton)

#### 1 场景问题

##### 1.1 读取配置文件的内容

考虑这样一个应用，读取配置文件的内容。

很多应用项目，都有与应用相关的配置文件，这些配置文件多是由项目开发人员自定义的，在里面定义一些应用需要的参数数据。当然在实际的项目中，这种配置文件多采用 xml 格式的。也有采用 properties 格式的，毕竟使用 Java 来读取 properties 格式的配置文件比较简单。


现在要读取配置文件的内容，该如何实现呢？

##### 1.2 不用模式的解决方案

有些朋友会想，要读取配置文件的内容，这也不是个什么困难的事情，直接读取文件的内容，然后把文件内容存放在相应的数据对象里面就可以了。真的这么简单吗？先实现看看吧。

为了示例简单，假设系统是采用的 properties 格式的配置文件。

(1) 那么直接使用 Java 来读取配置文件，示例代码如下：

Java 代码 

```
1. /**
2.  * 读取应用配置文件
3.  */
4. public class AppConfig {
5.     /**
```

```

6.         * 用来存放配置文件中参数 A 的值
7.         */
8.     private String parameterA;
9.     /**
10.        * 用来存放配置文件中参数 B 的值
11.        */
12.     private String parameterB;
13.
14.     public String getParameterA() {
15.         return parameterA;
16.     }
17.     public String getParameterB() {
18.         return parameterB;
19.     }
20.     /**
21.        * 构造方法
22.        */
23.     public AppConfig() {
24.         //调用读取配置文件的方法
25.         readConfig();
26.     }
27.     /**
28.        * 读取配置文件，把配置文件中的内容读出来设置到属性上
29.        */
30.     private void readConfig() {
31.         Properties p = new Properties();
32.         InputStream in = null;
33.         try {
34.             in = AppConfig.class.getResourceAsStream(
35. "AppConfig.properties");
36.             p.load(in);
37.             //把配置文件中的内容读出来设置到属性上
38.             this.parameterA = p.getProperty("paramA");
39.             this.parameterB = p.getProperty("paramB");
40.         } catch (IOException e) {
41.             System.out.println("装载配置文件出错了，具体堆栈
信息如下：");
42.             e.printStackTrace();
43.         } finally {
44.             try {
45.                 in.close();
46.             } catch (IOException e) {
47.                 e.printStackTrace();
48.             }

```

```
49.         }
50.     }
51. }
```


注意：只有访问参数的方法，没有设置参数的方法。

(2) 应用的配置文件，名字是 AppConfig.properties，放在 AppConfig 相同的包里面，简单示例如下：

Java 代码 

```
1. paramA=a
2. paramB=b
```

(3) 写个客户端来测试一下，示例代码如下：

Java 代码 

```
1. public class Client {
2.     public static void main(String[] args) {
3.         //创建读取应用配置的对象
4.         AppConfig config = new AppConfig();
5.
6.         String paramA = config.getParameterA();
7.         String paramB = config.getParameterB();
8.
9.         System.out.println("paramA="+paramA+", paramB="+paramB);
10.    }
11. }
```

运行结果如下：

```
1. paramA=a, paramB=b
```

### 1.3 有何问题

上面的实现很简单嘛，很容易的就实现了要求的功能。仔细想想，有没有什么问题呢？

看看客户端使用这个类的地方，是通过 new 一个 AppConfig 的实例来得到一个操作配置文件内容的对象。如果在系统运行中，有很多地方都需要使用配置文件的内容，也就是很多地方都需要创建 AppConfig 这个对象的实例。

换句话说，在系统运行期间，系统中会存在很多个 AppConfig 的实例对象，这有什么问题吗？

当然有问题了，试想一下，每一个 AppConfig 实例对象，里面都封装着配置文件的内容，系统中有多个 AppConfig 实例对象，也就是说系统中会同时存在多份配置文件的内容，这会

严重浪费内存资源。如果配置文件内容较少，问题还小一点，如果配置文件内容本来就多的话，对于系统资源的浪费问题就大了。事实上，对于 AppConfig 这种类，在运行期间，只需要一个实例对象就够了。

把上面的描述进一步抽象一下，问题就出来了：在一个系统运行期间，某个类只需要一个类实例就可以了，那么应该怎么实现呢？

## 2 解决方案

### 2.1 单例模式来解决

- 用来解决上述问题的一个合理的解决方案就是单例模式。那么什么是单例模式呢？
- (1) 单例模式定义
  - 保证一个类仅有一个实例，并提供一个访问它的全局访问点。
  - (2) 应用单例模式来解决的思路
  - 仔细分析上面的问题，现在一个类能够被创建多个实例，问题的根源在于类的构造方法是公开的，也就是可以让类的外部来通过构造方法创建多个实例。换句话说，只要类的构造方法能让类的外部访问，就没有办法去控制外部来创建这个类的实例个数。
  - 要想控制一个类只被创建一个实例，那么首要的问题就是要把创建实例的权限收回来，让类自身来负责自己类实例的创建工作，然后由这个类来提供外部可以访问这个类实例的方法，这就是单例模式的实现方式。

### 2.2 模式结构和说明

单例模式结构见图 1 所：

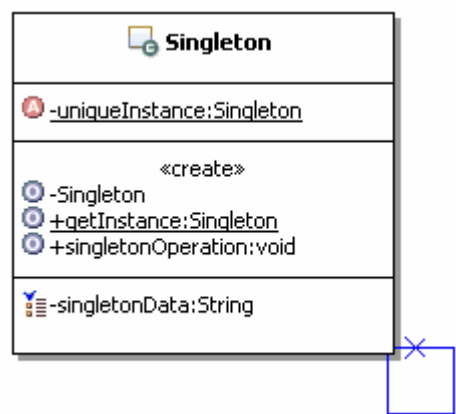


图 1 单例模式结构图


Singleton:

负责创建 Singleton 类自己的唯一实例，并提供一个 getInstance 的方法，让外部来访问这个类的唯一实例。

## 2.3 单例模式示例代码

在 Java 中，单例模式的实现又分为两种，一种称为懒汉式，一种称为饿汉式，其实就是在具体创建对象实例的处理上，有不同的实现方式。下面分别来看这两种实现方式的代码示例。为何这么写，具体的在后面再讲述。

(1) 懒汉式实现，示例代码如下：

Java 代码 


```
1. /**
2.  * 懒汉式单例实现的示例
3.  */
4. public class Singleton {
5.     /**
6.      * 定义一个变量来存储创建好的类实例
7.      */
8.     private static Singleton uniqueInstance = null;
9.     /**
10.    * 私有化构造方法，好在内部控制创建实例的数目
11.    */
12.    private Singleton() {
13.        //
14.    }
15.    /**
16.    * 定义一个方法来为客户端提供类实例
17.    * @return 一个 Singleton 的实例
18.    */
19.    public static synchronized Singleton getInstance() {
20.        //判断存储实例的变量是否有值
21.        if(uniqueInstance == null){
22.            //如果没有，就创建一个类实例，并把值赋值给存储类
实例的变量
23.                uniqueInstance = new Singleton();
24.            }
25.            //如果有值，那就直接使用
26.            return uniqueInstance;
27.        }
28.    /**
29.    * 示意方法，单例可以有自己的操作
30.    */
31.    public void singletonOperation() {
32.        //功能处理
33.    }
34.    /**
```

```

35.         * 示意属性，单例可以有自己的属性
36.         */
37.     private String singletonData;
38.     /**
39.         * 示意方法，让外部通过这些方法来访问属性的值
40.         * @return 属性的值
41.         */
42.     public String getSingletonData() {
43.         return singletonData;
44.     }
45. }

```

(2) 饿汉式实现，示例代码如下：

Java 代码 

```

1.  /**
2.     * 饿汉式单例实现的示例
3.     */
4.  public class Singleton {
5.      /**
6.         * 定义一个变量来存储创建好的类实例，直接在这里创建类实例，只会
        创建一次
7.         */
8.         private static Singleton uniqueInstance = new Singleton();

9.         /**
10.            * 私有化构造方法，好在内部控制创建实例的数目
11.            */
12.        private Singleton() {
13.            //
14.        }
15.        /**
16.            * 定义一个方法来为客户端提供类实例
17.            * @return 一个 Singleton 的实例
18.            */
19.        public static Singleton getInstance() {
20.            //直接使用已经创建好的实例
21.            return uniqueInstance;
22.        }
23.
24.        /**
25.            * 示意方法，单例可以有自己的操作
26.            */
27.        public void singletonOperation() {

```

```

28.             //功能处理
29.         }
30.         /**
31.          * 示意属性，单例可以有自己的属性
32.          */
33.         private String singletonData;
34.         /**
35.          * 示意方法，让外部通过这些方法来访问属性的值
36.          * @return 属性的值
37.          */
38.         public String getSingletonData() {
39.             return singletonData;
40.         }
41.     }

```

## 2.4 使用单例模式重写示例

要使用单例模式来重写示例，由于单例模式有两种实现方式，这里选一种来实现就好了，就选择饿汉式的实现方式来重写示例吧。

采用饿汉式的实现方式来重写实例的示例代码如下：

Java 代码 

```

1.  /**
2.   * 读取应用配置文件，单例实现
3.   */
4.  public class AppConfig {
5.      /**
6.       * 定义一个变量来存储创建好的类实例，直接在这里创建类实例，只会
        创建一次
7.       */
8.       private static AppConfig instance = new AppConfig();
9.       /**
10.        * 定义一个方法来为客户端提供 AppConfig 类的实例
11.        * @return 一个 AppConfig 的实例
12.        */
13.        public static AppConfig getInstance() {
14.            return instance;
15.        }
16.
17.        /**
18.         * 用来存放配置文件中参数 A 的值
19.         */
20.        private String parameterA;

```



```

21.      /**
22.         * 用来存放配置文件中参数 B 的值
23.         */
24.     private String parameterB;
25.     public String getParameterA() {
26.         return parameterA;
27.     }
28.     public String getParameterB() {
29.         return parameterB;
30.     }
31.     /**
32.         * 私有化构造方法
33.         */
34.     private AppConfig() {
35.         //调用读取配置文件的方法
36.         readConfig();
37.     }
38.     /**
39.         * 读取配置文件，把配置文件中的内容读出来设置到属性上
40.         */
41.     private void readConfig() {
42.         Properties p = new Properties();
43.         InputStream in = null;
44.         try {
45.             in = AppConfig.class.getResourceAsStream(
46. "AppConfig.properties");
47.             p.load(in);
48.             //把配置文件中的内容读出来设置到属性上
49.             this.parameterA = p.getProperty("paramA");
50.             this.parameterB = p.getProperty("paramB");
51.         } catch (IOException e) {
52.             System.out.println("装载配置文件出错了，具体堆栈
信息如下：");
53.             e.printStackTrace();
54.         }finally{
55.             try {
56.                 in.close();
57.             } catch (IOException e) {
58.                 e.printStackTrace();
59.             }
60.         }
61.     }
62. }

```

当然，测试的客户端也需要相应的变化，示例代码如下：

Java 代码 

```
1. public class Client {
2.     public static void main(String[] args) {
3.         //创建读取应用配置的对象
4.         AppConfig config = AppConfig.getInstance();
5.
6.         String paramA = config.getParameterA();
7.         String paramB = config.getParameterB();
8.
9.         System.out.println("paramA="+paramA+", paramB="+paramB);
10.    }
11. }
```

去测试看看，是否能满足要求。

未完待续，精彩稍后继续

## 研磨设计模式之单例模式-2

原文地址：<http://chjavach.javaeye.com/blog/723901>

### 3 模式讲解

#### 3.1 认识单例模式

##### (1) 单例模式的功能

单例模式的功能是用来保证这个类在运行期间只会被创建一个类实例，另外单例模式还提供了全局唯一访问这个类实例的访问点，就是那个 `getInstance` 的方法。不管采用懒汉式还是饿汉式的实现方式，这个全局访问点是一样的。

对于单例模式而言，不管采用何种实现方式，它都是只关心类实例的创建问题，并不关心具体的业务功能。

## (2) 单例模式的范围

也就是在多大范围内是单例呢？

观察上面的实现可以知道，目前 Java 里面实现的单例是一个虚拟机的范围。因为装载类的功能是虚拟机的，所以一个虚拟机在通过自己的 ClassLoader 装载饿汉式实现的单例类的时候就会创建一个类的实例。

这就意味着如果一个机器上有多个虚拟机，那么每个虚拟机里面都应该有一个这个类的实例，但是整个机器上就有很多个实例了。

另外请注意一点，这里讨论的单例模式并不适用于集群环境，对于集群环境下的单例这里不去讨论，那不属于这里的内容范围。

## (3) 单例模式的命名

一般建议单例模式的方法命名为：getInstance()，这个方法的返回类型肯定是单例类的类型了。getInstance 方法可以有参数，这些参数可能是创建类实例所需要的参数，当然，大多数情况下是不需要的。

单例模式的名称：单例、单件、单体等等，翻译的不同，都是指的同一个模式。

## 3.2 懒汉式和饿汉式实现

前面提到了单例模式有两种典型的解决方案，一种叫懒汉式，一种叫饿汉式，这两种方式究竟是如何实现的，下面分别来看看。为了看得更清晰一点，只是实现基本的单例控制部分，不再提供示例的属性和方法了；而且暂时也不去考虑线程安全的问题，这个问题在后面会重点分析。

### 1: 第一种方案 懒汉式

#### (1) 私有化构造方法

要想在运行期间控制某一个类的实例只有一个，那首先的任务就是要控制创建实例的地方，也就是不能随随便便就可以创建类实例，否则就无法控制创建的实例个数了。现在是让使用类的地方来创建类实例，也就是在类外部来创建类实例。


那么怎样才能让类的外部不能创建一个类的实例呢？很简单，私有化构造方法就可以了！示例代码如下：

Java 代码 

```
1. private Singleton() {  
2. }
```

#### (2) 提供获取实例的方法

构造方法被私有化了，外部使用这个类的地方不干了，外部创建不了类实例就没有办法调用这个对象的方法，就实现不了功能处理，这可不行。经过思考，单例模式决定让这个类提供一个方法来返回类的实例，好让外面使用。示例代码如下：


Java 代码 

```
1. public Singleton getInstance() {  
2. }
```

### (3) 把获取实例的方法变成静态的

又有新的问题了，获取对象实例的这个方法是个实例方法，也就是说客户端要想调用这个方法，需要先得到类实例，然后才可以调用，可是这个方法就是为了得到类实例，这样一来不就形成一个死循环了吗？这不就是典型的“先有鸡还是先有蛋的问题”嘛。

解决方法也很简单，在方法上加上 `static`，这样就可以直接通过类来调用这个方法，而不需要先得到类实例了，示例代码如下：

Java 代码 

```
1. public static Singleton getInstance() {  
2. }
```

### (4) 定义存储实例的属性


方法定义好了，那么方法内部如何实现呢？如果直接创建实例并返回，这样行不行呢？示例代码如下

Java 代码 

```
1. public static Singleton getInstance() {  
2.         return new Singleton();  
3. }
```

当然不行了，如果每次客户端访问都这样直接 `new` 一个实例，那肯定会有多个实例，根本实现不了单例的功能。

怎么办呢？单例模式想到了一个办法，那就是用一个属性来记录自己创建好的类实例，当第一次创建过后，就把这个实例保存下来，以后就可以复用这个实例，而不是重复创建对象实例了。示例代码如下：


Java 代码 

```
1. private Singleton instance = null;
```

### (5) 把这个属性也定义成静态的

这个属性变量应该在什么地方用呢？肯定是第一次创建类实例的地方，也就是在前面那个返回对象实例的静态方法里面使用。

由于要在一个静态方法里面使用，所以这个属性被迫成为一个类变量，要强制加上 `static`，也就是说，这里并没有使用 `static` 的特性。示例代码如下：

Java 代码 

```
1. private static Singleton instance = null;
```

## (6) 实现控制实例的创建

现在应该到 `getInstance` 方法里面实现控制实例创建了，控制的方式很简单，只要先判断一下，是否已经创建过实例了。如何判断？那就看存放实例的属性是否有值，如果有值，说明已经创建过了，如果没有值，那就是应该创建一个，示例代码如下：

Java 代码 

```
1. public static Singleton getInstance() {
2.     //先判断 instance 是否有值
3.     if(instance == null){
4.         //如果没有值，说明还没有创建过实例，那就创建一个
5.         //并把这个实例设置给 instance
6.         instance = new Singleton ();
7.     }
8.     //如果有值，或者是创建了值，那就直接使用
9.     return instance;
10. }
```

## (7) 完整的实现

至此，成功解决了：在运行期间，控制某个类只被创建一个实例的要求。完整的代码如下，为了大家好理解，用注释标示了代码的先后顺序，示例代码如下：

Java 代码 

```
1. public class Singleton {
2.     //4: 定义一个变量来存储创建好的类实例
3.     //5: 因为这个变量要在静态方法中使用，所以需要加上 static 修饰
4.     private static Singleton instance = null;
5.     //1: 私有化构造方法，好在内部控制创建实例的数目
6.     private Singleton() {
7.     }
8.     //2: 定义一个方法来为客户端提供类实例
9.     //3: 这个方法需要定义成类方法，也就是要加 static
10.    public static Singleton getInstance() {
11.        //6: 判断存储实例的变量是否有值
12.        if(instance == null){
13.            //6.1: 如果没有，就创建一个类实例，并把值赋值给存
            储类实例的变量
14.            instance = new Singleton();
15.        }
16.        //6.2: 如果有值，那就直接使用
17.        return instance;
18.    }
19. }
```

## 2: 第二种方案 饿汉式

这种方案跟第一种方案相比，前面的私有化构造方法，提供静态的 `getInstance` 方法来返回实例等步骤都一样。差别在如何实现 `getInstance` 方法，在这个地方，单例模式还想到了另外一种方法来实现 `getInstance` 方法。

不就是要控制只创建一个实例吗？那么有没有什么现成的解决办法呢？很快，单例模式回忆起了 Java 中 `static` 的特性：

- `static` 变量在类装载的时候进行初始化
- 多个实例的 `static` 变量会共享同一块内存区域。

这就意味着，在 Java 中，`static` 变量只会被初始化一次，就是在类装载的时候，而且多个实例都会共享这个内存空间，这不就是单例模式要实现的功能吗？真是得来全不费功夫啊。根据这些知识，写出了第二种解决方案的代码，示例代码如下：

Java 代码 

```
1. public class Singleton {
2.     //4: 定义一个静态变量来存储创建好的类实例
3.     //直接在这里创建类实例，只会创建一次
4.     private static Singleton instance = new Singleton();
5.     //1: 私有化构造方法，好在内部控制创建实例的数目
6.     private Singleton() {
7.     }
8.     //2: 定义一个方法来为客户端提供类实例
9.     //3: 这个方法需要定义成类方法，也就是要加 static
10.    //这个方法里面就不需要控制代码了
11.    public static Singleton getInstance() {
12.        //5: 直接使用已经创建好的实例
13.        return instance;
14.    }
15. }
```

**注意一下**，这个方案是用到了 `static` 的特性的，而第一个方案是没有用到的，因此两个方案的步骤会有一些不同，在第一个方案里面，强制加上 `static` 也是算作一步的，而在这个方案里面，是主动加上 `static`，就不单独算作一步了。

所以在查看上面两种方案的代码的时候，仔细看看编号，顺着编号的顺序看，可以体会出两种方案的不一样来。

不管是采用哪一种方式，在运行期间，都只会生成一个实例，而访问这些类的一个全局访问点，就是那个静态的 `getInstance` 方法。

## 3: 单例模式的调用顺序示意图

由于单例模式有两种实现方式，那么它的调用顺序也分成两种。先看懒汉式的调用顺序，如图 2 所示：

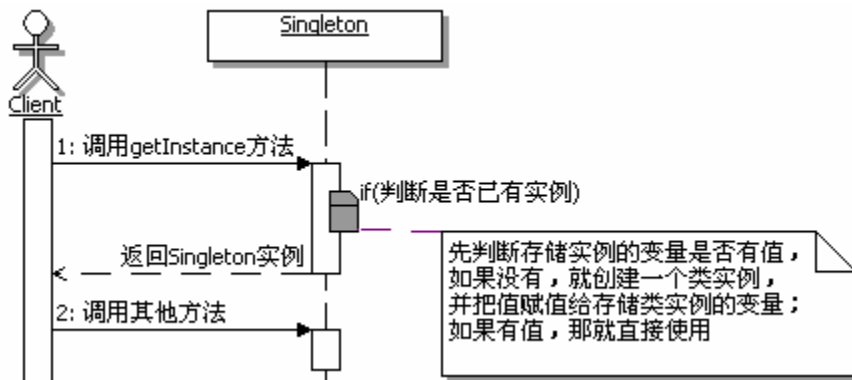


图 2 懒汉式调用顺序示意图

饿汉式的调用顺序，如图 3 所示：

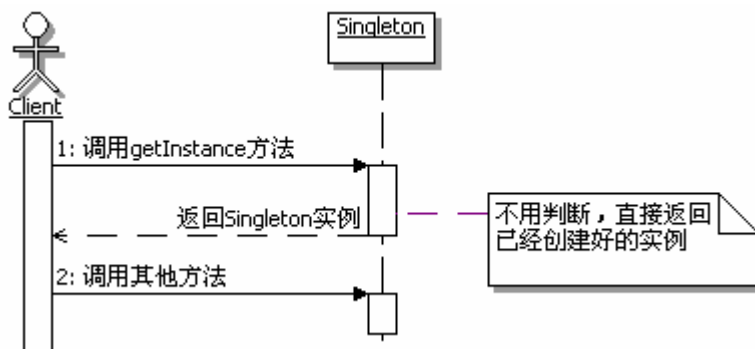


图 3 饿汉式调用顺序示意图

未完待续

## 研磨设计模式之单例模式-3

原文地址：<http://chjavach.javaeye.com/blog/726788>

### 3.3 延迟加载的思想

单例模式的懒汉式实现方式体现了延迟加载的思想，什么是延迟加载呢？

通俗点说，就是一开始不要加载资源或者数据，一直等，等到马上就要使用这个资源或者数据了，躲不过去了才加载，所以也称 Lazy Load，不是懒惰啊，是“延迟加载”，这在实际开发中是一种很常见的思想，尽可能的节约资源。

体现在什么地方呢？看如下代码：

```
public static Singleton getInstance(){  
      
    if(instance == null){  
        instance = new Singleton();  
    }  
    return instance;  
}
```

这里就体现了延迟加载，马上就要使用这个实例了，还不知道有没有呢，所以判断一下，如果没有，没办法了，赶紧创建一个吧

### 3.4 缓存的思想

单例模式的懒汉式实现还体现了缓存的思想，缓存也是实际开发中非常常见的功能。

简单讲就是，如果某些资源或者数据会被频繁的使用，而这些资源或数据存储的系统外部，比如数据库、硬盘文件等，那么每次操作这些数据的时候都从数据库或者硬盘上去获取，速度会很慢，会造成性能问题。

一个简单的解决方法就是：把这些数据缓存到内存里面，每次操作的时候，先到内存里面找，看有没有这些数据，如果有，那么就直接使用，如果没有那么就获取它，并设置到缓存中，下一次访问的时候就可以直接从内存中获取了。从而节省大量的时间，当然，缓存是一种典型的空间换时间的方案。

缓存在单例模式的实现中怎么体现的呢？

```
public class Singleton {  
    private static Singleton instance = null;  
    private Singleton() {  
    }  
    public static Singleton getInstance(){  
        //判断存储实例的变量是否有值  
        if(instance == null){  
            //如果没有，就创建一个类实例，并把值赋值给存储类实例的变量  
            instance = new Singleton();  
        }  
        //如果有值，那就直接使用  
        return instance;  
    }  
}
```

这个属性就是用来缓存实例的

缓存的实现




### 3.5 Java 中缓存的基本实现

引申一下，看看在 Java 开发中的缓存的基本实现，在 Java 中最常见的一种实现缓存的方式就是使用 Map，基本的步骤是：

- 先到缓存里面查找，看看是否存在需要使用的数据
- 如果没有找到，那么就创建一个满足要求的数据，然后把这个数据设置回到缓存中，以备下次使用
- 如果找到了相应的数据，或者是创建了相应的数据，那就直接使用这个数据。

还是看看示例吧，示例代码如下：

Java 代码 

```
1. /**
2.  * Java 中缓存的基本实现示例
3.  */
4. public class JavaCache {
5.     /**
6.      * 缓存数据的容器，定义成 Map 是方便访问，直接根据 Key 就可以获取
7.      * key 选用 String 是为了简单，方便演示
8.      */
9.     private Map<String, Object> map=new HashMap<String, Object>();
10.    /**
11.     * 从缓存中获取值
12.     * @param key 设置时候的 key 值
13.     * @return key 对应的 Value 值
14.     */
15.    public Object getValue(String key){
16.        //先从缓存里面取值
17.        Object obj = map.get(key);
18.        //判断缓存里面是否有值
19.        if(obj == null){
20.            //如果没有，那么就去获取相应的数据，比如读取数据
21.            //这里只是演示，所以直接写个假的值
22.            obj = key+",value";
23.            //把获取的值设置回到缓存里面
24.            map.put(key, obj);
25.        }
26.        //如果有值了，就直接返回使用
27.        return obj;
28.    }
```

29. }

这里只是缓存的基本实现，还有很多功能都没有考虑，比如缓存的清除，缓存的同步等等。当然，Java 的缓存还有很多实现方式，也是非常复杂的，现在有很多专业的缓存框架，更多缓存的知识，这里就不再去讨论了。

### 3.6 利用缓存来实现单例模式

其实应用 Java 缓存的知识，也可以变相实现 Singleton 模式，算是一个模拟实现吧。每次都先从缓存中取值，只要创建一次对象实例过后，就设置了缓存的值，那么下次就不用再创建了。

虽然不是很标准的做法，但是同样可以实现单例模式的功能，为了简单，先不去考虑多线程的问题，示例代码如下：

Java 代码 

```
1.  /**
2.   * 使用缓存来模拟实现单例
3.   */
4.  public class Singleton {
5.      /**
6.       * 定义一个缺省的 key 值，用来标识在缓存中的存放
7.       */
8.      private final static String DEFAULT_KEY = "One";
9.      /**
10.     * 缓存实例的容器
11.     */
12.     private static Map<String, Singleton> map =
13. new HashMap<String, Singleton>();
14.     /**
15.     * 私有化构造方法
16.     */
17.     private Singleton() {
18.         //
19.     }
20.     public static Singleton getInstance() {
21.         //先从缓存中获取
22.         Singleton instance = (Singleton)map.get(DEFAULT_KEY);
23.         //如果没有，就新建一个，然后设置回缓存中
24.         if(instance==null){
25.             instance = new Singleton();
26.             map.put(DEFAULT_KEY, instance);
27.         }
28.         //如果有就直接使用
```

```
29.         return instance;
30.     }
31. }
```

是不是也能实现单例所要求的功能呢？其实实现模式的方式有很多种，并不是只有模式的参考实现所实现的方式，上面这种也能实现单例所要求的功能，只不过实现比较麻烦，不是太好而已，但在后面扩展单例模式的时候会有用。

另外，模式是经验的积累，模式的参考实现并不一定是最优的，对于单例模式，后面会给大家一些更好的实现方式。

### 3.7 单例模式的优缺点

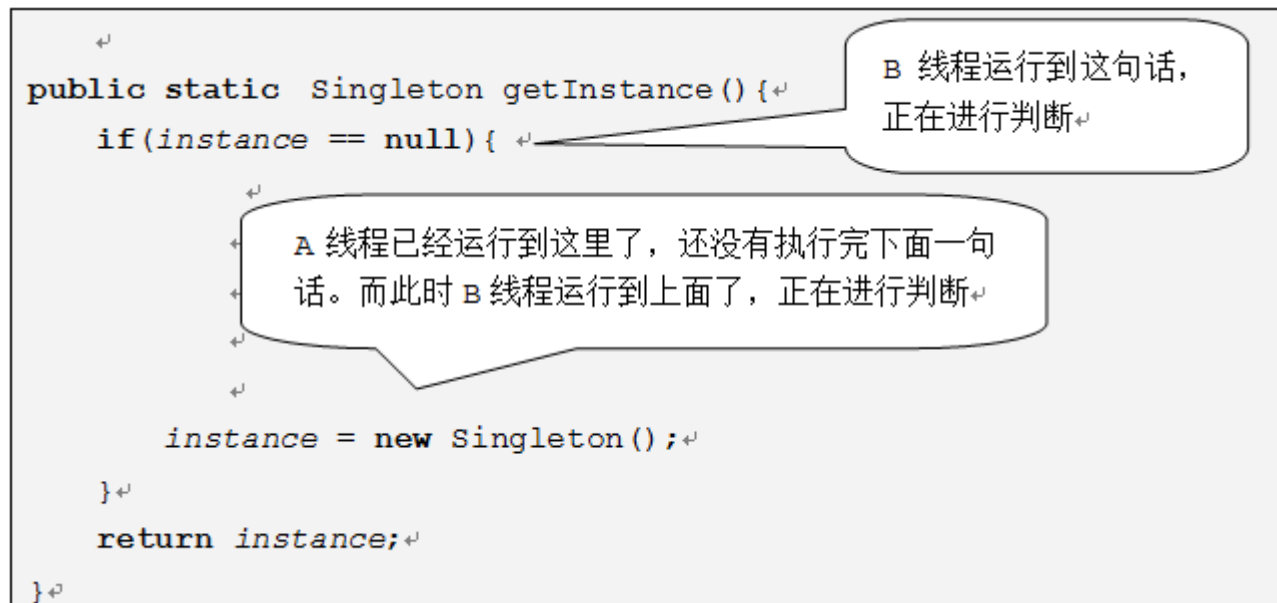
#### 1: 时间和空间

比较上面两种写法：**懒汉式是典型的时间换空间**，也就是每次获取实例都会进行判断，看是否需要创建实例，费判断的时间，当然，如果一直没有人使用的话，那就不会创建实例，节约内存空间。

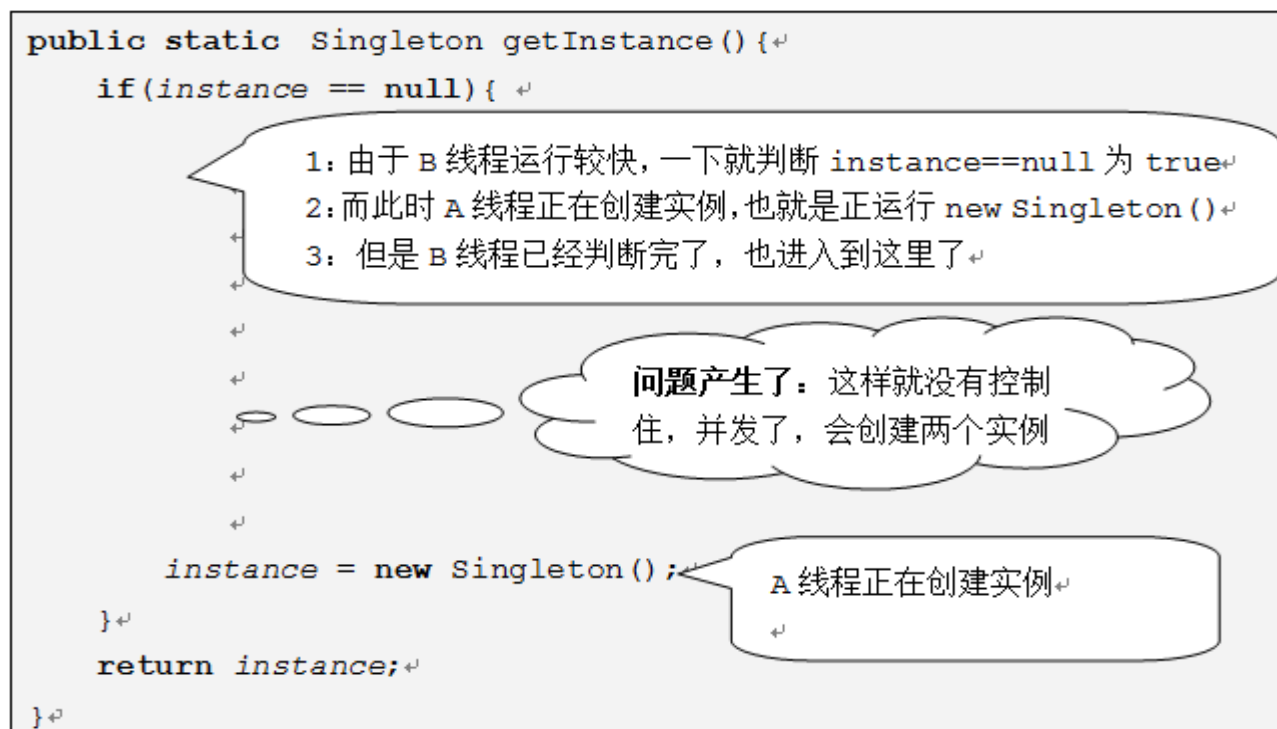
**饿汉式是典型的空间换时间**，当类装载的时候就会创建类实例，不管你用不用，先创建出来，然后每次调用的时候，就不需要再判断了，节省了运行时间。

#### 2: 线程安全

(1) 从线程安全性上讲，**不加同步的懒汉式是线程不安全的**，比如说：有两个线程，一个是线程 A，一个是线程 B，它们同时调用 getInstance 方法，那就可能导致并发问题。如下示例：



程序继续运行，两个线程都向前走了一步，如下：



可能有些朋友会觉得文字描述还是不够直观，再来画个图说明一下，如图 4 所示：

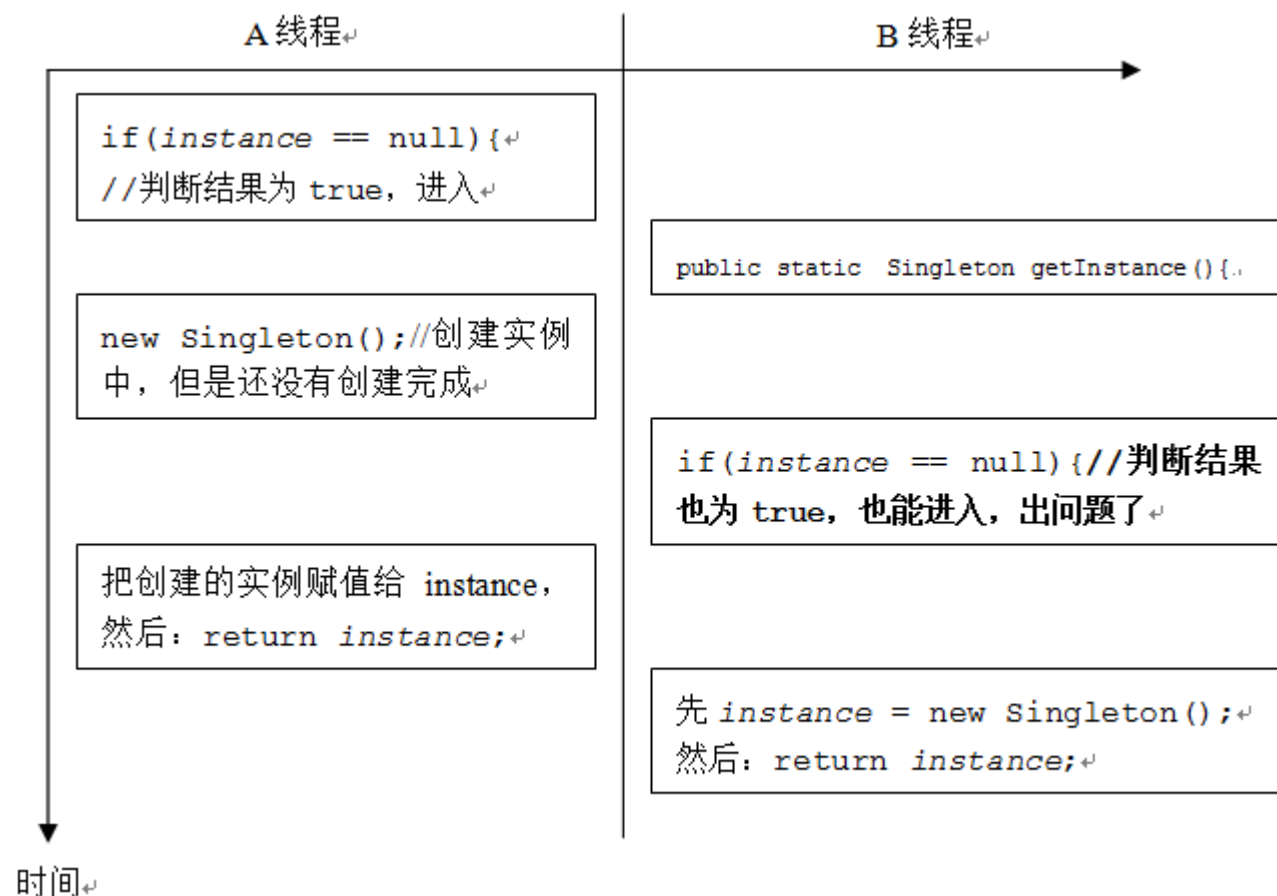


图 4 懒汉式单例的线程问题示意图

通过图 4 的分解描述，明显可以看出，当 A、B 线程并发的情况下，会创建出两个实例来，也就是单例的控制在此并发情况下失效了。

**(2) 饿汉式是线程安全的**，因为虚拟机保证了只会装载一次，在装载类的时候是不会发生并发的。

### **(3) 如何实现懒汉式的线程安全呢？**

当然懒汉式也是可以实现线程安全的，只要加上 `synchronized` 即可，如下：

Java 代码 

```
1. public static synchronized Singleton getInstance() {}
```

但是这样一来，会降低整个访问的速度，而且每次都要判断，也确实是稍微慢点。那么有没有更好的方式来实现呢？

### **(4) 双重检查加锁**

可以使用“双重检查加锁”的方式来实现，就可以既实现线程安全，又能够使性能不受太大的影响。那么什么是“双重检查加锁”机制呢？

所谓双重检查加锁机制，指的是：并不是每次进入 `getInstance` 方法都需要同步，而是先不同步，进入方法过后，先检查实例是否存在，如果不存在才进入下面的同步块，这是第一重检查。进入同步块过后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。

双重检查加锁机制的实现会使用一个关键字 `volatile`，它的意思是：被 `volatile` 修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存，从而确保多个线程能正确的处理该变量。

**注意：**在 Java1.4 及以前版本中，很多 JVM 对于 `volatile` 关键字的实现有问题，会导致双重检查加锁的失败，因此双重检查加锁的机制只能用在 Java5 及以上的版本。

看看代码可能会更清楚些，示例代码如下：

Java 代码 

```
1. public class Singleton {
2.     /**
3.      * 对保存实例的变量添加 volatile 的修饰
4.      */
5.     private volatile static Singleton instance = null;
6.     private Singleton() {
7.     }
8.     public static Singleton getInstance() {
9.         //先检查实例是否存在，如果不存在才进入下面的同步块
10.        if(instance == null){
11.            //同步块，线程安全的创建实例
12.            synchronized(Singleton.class) {
```

```

13.                                     //再次检查实例是否存在，如果不存在才真的
    创建实例
14.                                     if(instance == null){
15.                                     instance = new Singleton();

16.                                     }
17.                                     }
18.                                     }
19.                                     return instance;
20.                                     }
21. }

```

这种实现方式既可使实现线程安全的创建实例，又不会对性能造成太大的影响，它只是在第一次创建实例的时候同步，以后就不需要同步了，从而加快运行速度。

**提示：**由于 `volatile` 关键字可能会屏蔽掉虚拟机中一些必要的代码优化，所以运行效率并不是很高，因此一般建议，没有特别的需要，不要使用。也就是说，虽然可以使用双重加锁机制来实现线程安全的单例，但并不建议大量采用，根据情况来选用吧。

未完待续

## 研磨设计模式之单例模式-4

原文地址：<http://chjavach.javaeye.com/blog/732368>

### 3.8 在 Java 中一种更好的单例实现方式

根据上面的分析，常见的两种单例实现方式都存在小小的缺陷，那么有没有一种方案，既能够实现延迟加载，又能够实现线程安全呢？

还真有高人想到这样的解决方案了，这个解决方案被称为 `Lazy initialization holder class` 模式，这个模式综合使用了 Java 的类级内部类和多线程缺省同步锁的知识，很巧妙的同时实现了延迟加载和线程安全。

#### 1: 先来看点相应的基础知识

先简单的看看类级内部类相关的知识。

- 什么是类级内部类？

简单点说,类级内部类指的是:有 static 修饰的成员式内部类。如果没有 static 修饰的成员式内部类被称为对象级内部类。

- 类级内部类相当于其外部类的 static 成分,它的对象与外部类对象间不存在依赖关系,因此可直接创建。而对象级内部类的实例,是绑定在外部对象实例中的。
- 类级内部类中,可以定义静态的方法,在静态方法中只能够引用外部类中的静态成员方法或者成员变量。
- 类级内部类相当于其外部类的成员,只有在第一次被使用的时候才会被装载

### 再来看看多线程缺省同步锁的知识。

大家都知道,在多线程开发中,为了解决并发问题,主要是通过使用 synchronized 来加互斥锁进行同步控制。但是在某些情况中,JVM 已经隐含地为您执行了同步,这些情况下就不用自己再来进行同步控制了。这些情况包括:


- 由静态初始化器(在静态字段上或 static {} 块中的初始化器)初始化数据时
- 访问 final 字段时
- 在创建线程之前创建对象时
- 线程可以看见它将要处理的对象时

## 2: 接下来看看这种解决方案的思路

要想很简单的实现线程安全,可以采用静态初始化器的方式,它可以由 JVM 来保证线程安全性。比如前面的“饿汉式”实现方式,但是这样一来,不是会浪费一定的空间吗?因为这种实现方式,会在类装载的时候就初始化对象,不管你需不需要。

如果现在有一种方法能够让类装载的时候不去初始化对象,那不就解决问题了?一种可行的方式就是采用类级内部类,在这个类级内部类里面去创建对象实例,这样一来,只要不使用到这个类级内部类,那就不会创建对象实例。从而同时实现延迟加载和线程安全。

看看代码示例可能会更清晰,示例代码如下:

Java 代码 

```
1. public class Singleton {
2.     /**
3.         * 类级的内部类,也就是静态的成员式内部类,该内部类的实例与外部
      类的实例
4.         * 没有绑定关系,而且只有被调用到才会装载,从而实现了延迟加载
5.         */
6.     private static class SingletonHolder{
7.         /**
8.         * 静态初始化器,由 JVM 来保证线程安全
9.         */
10.        private static Singleton instance=new Singleton();
11.    }
12.    /**
13.    * 私有化构造方法
14.    */
```

```

15.         private Singleton() {
16.         }
17.         public static Singleton getInstance() {
18.             return SingletonHolder.instance;
19.         }
20.     }

```

仔细想想，是不是很巧妙呢！

当 `getInstance` 方法第一次被调用的时候，它第一次读取 `SingletonHolder.instance`，导致 `SingletonHolder` 类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建 `Singleton` 的实例，由于是静态的域，因此只会被虚拟机在装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

这个模式的优势在于，`getInstance` 方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。

### 3.9 单例和枚举

按照《高效 Java 第二版》中的说法：单元素的枚举类型已经成为实现 Singleton 的最佳方法。

为了理解这个观点，先来了解一点相关的枚举知识，这里只是强化和总结一下枚举的一些重要观点，更多基本的枚举的使用，请参看 Java 编程入门资料：

- Java 的枚举类型实质上是功能齐全的类，因此可以有自己的属性和方法
- Java 枚举类型的基本思想：通过公有的静态 `final` 域为每个枚举常量导出实例的类
- 从某个角度讲，枚举是单例的泛型化，本质上是单元素的枚举

用枚举来实现单例非常简单，只需要编写一个包含单个元素的枚举类型即可，示例代码如下：

Java 代码 

```

1.  /**
2.   *  使用枚举来实现单例模式的示例
3.   */
4.  public enum Singleton {
5.      /**
6.       *  定义一个枚举的元素,它就代表了 Singleton 的一个实例
7.       */
8.      uniqueInstance;
9.
10.     /**
11.      *  示意方法，单例可以有自己的操作
12.      */

```



```

13.         public void singletonOperation() {
14.             //功能处理
15.         }
16. }

```

使用枚举来实现单实例控制，会更加简洁，而且无偿的提供了序列化的机制，并由 JVM 从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。


### 3.10 思考单例模式

#### 1: 单例模式的本质

单例模式的本质：**控制实例数目**。

单例模式是为了控制在运行期间，某些类的实例数目只能有一个。可能有人就会想了，那么我能不能控制实例数目为 2 个，3 个，或者是任意多个呢？目的都是一样的，节省资源啊，有些时候单个实例不能满足实际的需要，会忙不过来，根据测算，3 个实例刚刚好，也就是说，现在要控制实例数目为 3 个，怎么办呢？

其实思路很简单，就是利用上面通过 Map 来缓存实现单例的示例，进行变形，一个 Map 可以缓存任意多个实例，新的问题就是，Map 中有多个实例，但是客户端调用的时候，到底返回那一个实例呢，也就是实例的调度问题，我们只是想要来展示设计模式，对于这个调度算法就不去深究了，做个最简单的，循环返回就好了，示例代码如下：

Java 代码 

```

1.  /**
2.   *  简单演示如何扩展单例模式，控制实例数目为 3 个
3.   */
4.  public class OneExtend {
5.      /**
6.       *  定义一个缺省的 key 值的前缀
7.       */
8.      private final static String DEFAULT_PREKEY = "Cache";
9.      /**
10.     *  缓存实例的容器
11.     */
12.     private static Map<String, OneExtend> map =
13. new HashMap<String, OneExtend>();
14.     /**
15.     *  用来记录当前正在使用第几个实例，到了控制的最大数目，就返回从 1
    开始
16.     */
17.     private static int num = 1;

```

```

18.      /**
19.      * 定义控制实例的最大数目
20.      */
21.      private final static int NUM_MAX = 3;
22.      private OneExtend() {}
23.      public static OneExtend getInstance() {
24.          String key = DEFAULT_PREKEY+num;
25.          //缓存的体现，通过控制缓存的数据多少来控制实例数目
26.          OneExtend oneExtend = map.get(key);
27.          if(oneExtend==null) {
28.              oneExtend = new OneExtend();
29.              map.put(key, oneExtend);
30.          }
31.          //把当前实例的序号加 1
32.          num++;
33.          if(num > NUM_MAX) {
34.              //如果实例的序号已经达到最大数目了，那就重复从 1
开始获取
35.              num = 1;
36.          }
37.          return oneExtend;
38.      }
39.
40.      public static void main(String[] args) {
41.          //测试是否能满足功能要求
42.          OneExtend t1 = getInstance ();
43.          OneExtend t2 = getInstance ();
44.          OneExtend t3 = getInstance ();
45.          OneExtend t4 = getInstance ();
46.          OneExtend t5 = getInstance ();
47.          OneExtend t6 = getInstance ();
48.
49.          System.out.println("t1=="+t1);
50.          System.out.println("t2=="+t2);
51.          System.out.println("t3=="+t3);
52.          System.out.println("t4=="+t4);
53.          System.out.println("t5=="+t5);
54.          System.out.println("t6=="+t6);
55.      }
56. }

```

测试一下，看看结果，如下：

Java 代码 

1. t1==cn.javass.dp.singleton.example9.OneExtend@6b97fd
2. t2==cn.javass.dp.singleton.example9.OneExtend@1c78e57
3. t3==cn.javass.dp.singleton.example9.OneExtend@5224ee
4. t4==cn.javass.dp.singleton.example9.OneExtend@6b97fd
5. t5==cn.javass.dp.singleton.example9.OneExtend@1c78e57
6. t6==cn.javass.dp.singleton.example9.OneExtend@5224ee

第一个实例和第四个相同，第二个与第五个相同，第三个与第六个相同，也就是说一共只有三个实例，而且调度算法是从第一个依次取到第三个，然后回来继续从第一个开始取到第三个。

当然这里我们不去考虑复杂的调度情况，也不去考虑何时应该创建新实例的问题。

**注意：**这种实现方式同样是线程不安全的，需要处理，这里就不再展开去讲了。

## 2: 何时选用单例模式

建议在如下情况中，选用单例模式：

- 当需要控制一个类的实例只能有一个，而且客户只能从一个全局访问点访问它时，可以选用单例模式，这些功能恰好是单例模式要解决的问题

## 3.11 相关模式

很多模式都可以使用单例模式，只要这些模式中的某个类，需要控制实例为一个的时候，就可以很自然的使用上单例模式。比如抽象工厂方法中的具体工厂类就通常是一个单例。

单例模式结束, 谢谢各位的捧场, 鞠躬 ing