

[High一下!](#)

[酷壳 - CoolShell.cn](#)

享受编程和技术所带来的快乐 - <http://coolshell.cn>

- [首页](#)
- [推荐文章](#)
- [本站插件](#)
- [留言小本](#)
- [关于酷壳](#)
- [关于陈皓](#)
-



[首页](#) > [Java语言](#), [程序设计](#), [编程语言](#) > 深入浅出单实例Singleton设计模式

深入浅出单实例Singleton设计模式

2009年3月26日 [陈皓](#) [发表评论](#) [阅读评论](#) 14,947 人阅读

单实例Singleton设计模式可能是被讨论和使用的最广泛的一个设计模式了，这可能也是面试中问得最多的一个设计模式了。这个设计模式主要目的是想在整个系统中只能出现一个类的实例。这样做当然是有必然的，比如你的软件的全局配置信息，或者是一个Factory，或是一个主控类，等等。你希望这个类在整个系统中只能出现一个实例。当然，作为一个技术负责人的你，你当然有权利通过使用非技术的手段来达到你的目的。比如：你在团队内部明文规定，“XX类只能有一个全局实例，如果某人使用两次以上，那么该人将被处于2000元的罚款！”（呵呵），你当然有权这么做。但是如果你的设计的是东西是一个类库，或是一个需要提供给用户使用的API，恐怕你的这项规定将会失效。因为，你无权要求别人会那么做。所以，这就是为什么，我们希望通过使用技术的手段来达成这样一个目的的原因。

本文会带着你深入整个Singleton的世界，当然，我会放弃使用C++语言而改用Java语言，因为使用Java这个语言可能更容易让我说明一些事情。

Singleton的教学版本

这里，我将直接给出一个Singleton的简单实现，因为我相信你已经有这方面的一些基础了。我们姑且把这个版本叫做1.0版

```
1  // version 1.0
2  public class Singleton {
3      private static Singleton singleton = null;
4      private Singleton() { }
5      public static Singleton getInstance() {
6          if (singleton == null) {
7              singleton = new Singleton();
8          }
9          return singleton;
10     }
11 }
```

在上面的实例中，我想说明下面几个Singleton的特点：（下面这些东西可能是尽人皆知的，没有什么新鲜的）

1. 私有（private）的构造函数，表明这个类是不可能形成实例了。这主要是怕这个类会有多个实例。
2. 既然这个类是不可能形成实例，那么，我们需要一个静态的方式让其形成实例：getInstance()。注意这个方法是在new自己，因为其可以访问私有的构造函数，所以他是可以保证实例被创建出来的。
3. 在getInstance()中，先做判断是否已形成实例，如果已形成则直接返回，否则创建实例。
4. 所形成的实例保存在自己类中的私有成员中。
5. 我们取实例时，只需要使用Singleton.getInstance()就行了。

当然，如果你觉得知道了上面这些事情后就学成了，那得给你当头棒喝一下了，事情远远没有那么简单。

Singleton的实际版本

上面的这个程序存在比较严重的问题，因为是全局性的实例，所以，在多线程情况下，所有的全局共享的东西都会变得非常的危险，这个也一样，在多线程情况下，如果多个线程同时调用getInstance()的话，那么，可能会有多个进程同时通过 (singleton == null) 的条件检查，于是，多个实例就创建出来，并且很可能造成内存泄露问题。嗯，熟悉多线程的你一定会说——“我们需要线程互斥或同步”，没错，我们需要这个事情，于是我们的Singleton升级成1.1版，如下所示：

```
1  // version 1.1
2  public class Singleton
3  {
4      private static Singleton singleton = null;
5      private Singleton() { }
6      public static Singleton getInstance() {
7          if (singleton == null) {
8              synchronized (Singleton.class) {
9                  singleton = new Singleton();
10             }
11     }
```

```
12 |         return singleton;
13 |     }
14 | }
```

嗯，使用了Java的synchronized方法，看起来不错哦。应该没有问题了吧？！错！这还是有问题！为什么呢？前面已经说过，如果有多个线程同时通过(singleton == null)的条件检查（因为他们并行运行），虽然我们的synchronized方法会帮助我们同步所有的线程，让我们并行线程变成串行的一个一个去new，那不还是一样的吗？同样会出现很多实例。嗯，确实如此！看来，还得把那个判断(singleton == null)条件也同步起来。于是，我们的Singleton再次升级成1.2版本，如下所示：

```
1 | // version 1.2
2 | public class Singleton
3 | {
4 |     private static Singleton singleton = null;
5 |     private Singleton() { }
6 |     public static Singleton getInstance() {
7 |         synchronized (Singleton.class) {
8 |             if (singleton == null) {
9 |                 singleton = new Singleton();
10 |             }
11 |         }
12 |         return singleton;
13 |     }
14 | }
```

不错不错，看似很不错了。在多线程下应该没有什么问题了，不是吗？的确是这样的，1.2版的Singleton在多线程下的确没有问题了，因为我们同步了所有的线程。只不过嘛.....，什么？！还不行？！是的，还是有点小问题，我们本来只是想让new这个操作并行就可以了，现在，只要是进入getInstance()的线程都得同步啊，注意，创建对象的动作只有一次，后面的动作全是读取那个成员变量，这些读取的动作不需要线程同步啊。这样的作法感觉非常极端啊，为了一个初始化的创建动作，居然让我们达上了所有的读操作，严重影响后续的性能啊！

还得改！嗯，看来，在线程同步前还得加一个(singleton == null)的条件判断，如果对象已经创建了，那么就不需要线程的同步了。OK，下面是1.3版的Singleton。

```
1 | // version 1.3
2 | public class Singleton
3 | {
4 |     private static Singleton singleton = null;
5 |     private Singleton() { }
6 |     public static Singleton getInstance() {
7 |         if (singleton == null) {
8 |             synchronized (Singleton.class) {
9 |                 if (singleton == null) {
10 |                     singleton = new Singleton();
11 |                 }
12 |             }
13 |         }
14 |     }
15 | }
```

```
13     }  
14     return singleton;  
15 }  
16 }
```

感觉代码开始变得有点啰嗦和复杂了，不过，这可能是最不错的一个版本了，这个版本又叫“双重检查” Double-Check。下面是说明：

1. 第一个条件是说，如果实例创建了，那就不需要同步了，直接返回就好了。
2. 不然，我们就开始同步线程。
3. 第二个条件是说，如果被同步的线程中，有一个线程创建了对象，那么别的线程就不用再创建了。

相当不错啊，干得非常漂亮！请大家为我们的1.3版起立鼓掌！

但是，如果你认为这个版本大功告成，你就错了。

主要在于 **singleton = new Singleton()** 这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情。

1. 给 singleton 分配内存
2. 调用 Singleton 的构造函数来初始化成员变量，形成实例
3. 将 singleton 对象指向分配的内存空间（执行完这步 singleton 才是非 null 了）

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

对此，我们只需要把 singleton 声明成 volatile 就可以了。下面是 1.4 版：

```
1 // version 1.4  
2 public class Singleton  
3 {  
4     private volatile static Singleton singleton = null;  
5     private Singleton() { }  
6     public static Singleton getInstance() {  
7         if (singleton == null) {  
8             synchronized (Singleton.class) {  
9                 if (singleton == null) {  
10                    singleton = new Singleton();  
11                }  
12            }  
13        }  
14        return singleton;  
15    }  
16 }
```

使用 volatile 有两个功用：

- 1) 这个变量不会在多个线程中存在复本，直接从内存读取。
- 2) 这个关键字会禁止指令重排序优化。也就是说，在 `volatile` 变量的赋值操作后面会有一个内存屏障（生成的汇编代码上），读操作不会被重排序到内存屏障之前。

但是，这个事情仅在Java 1.5版后有用，1.5版之前用这个变量也有问题，因为老版本的Java的内存模型是有缺陷的。

Singleton 的简化版本

上面的玩法实在是太复杂了，一点也不优雅，下面是一种更为优雅的方式：

这种方法非常简单，因为单例的实例被声明成 `static` 和 `final` 变量了，在第一次加载类到内存中时就会初始化，所以创建实例本身是线程安全的。

```
1 // version 1.5
2 public class Singleton
3 {
4     private volatile static Singleton singleton = new Singl
5     private Singleton() { }
6     public static Singleton getInstance() {
7         return singleton;
8     }
9 }
```

但是，这种玩法的最大问题是——当这个类被加载的时候，`new Singleton()` 这句话就会被执行，就算是`getInstance()`没有被调用，类也被初始化了。

于是，这个可能会与我们想要的行为不一样，比如，我的类的构造函数中，有一些事可能需要依赖于别的类干的一些事（比如某个配置文件，或是某个被其它类创建的资源），我们希望他能在我第一次`getInstance()`时才被真正的创建。这样，我们可以控制真正的类创建的时刻，而不是把类的创建委托给了类装载器。

好吧，我们还得绕一下：

下面的这个1.6版是老版《Effective Java》中推荐的方式。

```
1 // version 1.6
2 public class Singleton {
3     private static class SingletonHolder {
4         private static final Singleton INSTANCE = new Singl
5     }
6     private Singleton (){}
7     public static final Singleton getInstance() {
8         return SingletonHolder.INSTANCE;
9     }
10 }
```

上面这种方式，仍然使用JVM本身机制保证了线程安全问题；由于 `SingletonHolder` 是私

有的，除了 `getInstance()` 之外没有办法访问它，因此它只有在`getInstance()`被调用时才会真正创建；同时读取实例的时候不会进行同步，没有性能缺陷；也不依赖 JDK 版本。

Singleton 优雅版本

```
1 | public enum Singleton{
2 |     INSTANCE;
3 | }
```

居然用枚举！！看上去好牛逼，通过`EasySingleton.INSTANCE`来访问，这比调用`getInstance()`方法简单多了。

默认枚举实例的创建是线程安全的，所以不需要担心线程安全的问题。但是在枚举中的其他任何方法的线程安全由程序员自己负责。还有防止上面的通过反射机制调用私有构造器。

这个版本基本上消除了绝大多数的问题。代码也非常简单，实在无法不用。这也是新版的《Effective Java》中推荐的模式。

Singleton的其它问题

怎么？还有问题？！当然还有，请记住下面这条规则——“**无论你的代码写得有多好，其只能在特定的范围内工作，超出这个范围就要出Bug了**”，这是“陈式第一定理”，呵呵。你能想一想还有什么情况会让这个我们上面的代码出问题吗？

在C++下，我不是很好举例，但是在Java的环境下，嘿嘿，还是让我们来看看下面的一些反例和一些别的事情的讨论（当然，有些反例可能属于钻牛角尖，可能有点学院派，不过也不排除其实际可能性，就算是提个醒吧）：

其一、Class Loader。不知道你对Java的Class Loader熟悉吗？“类装载器”？！C++可没有这个东西啊。这是Java动态性的核心。顾名思义，类装载器是用来把类(class)装载进JVM的。JVM规范定义了两种类型的类装载器：启动内装载器(bootstrap)和用户自定义装载器(user-defined class loader)。在一个JVM中可能存在多个ClassLoader，每个ClassLoader拥有自己的NameSpace。一个ClassLoader只能拥有一个class对象类型的实例，但是不同的ClassLoader可能拥有相同的class对象实例，这时可能产生致命的问题。如ClassLoaderA，装载了类A的类型实例A1，而ClassLoaderB，也装载了类A的对象实例A2。逻辑上讲A1=A2，但是由于A1和A2来自于不同的ClassLoader，它们实际上是完全不同的，如果A中定义了一个静态变量c，则c在不同的ClassLoader中的值是不同的。

于是，如果咱们的Singleton 1.3版本如果面对着多个Class Loader会怎么样？呵呵，多个实例同样会被多个Class Loader创建出来，当然，这个有点牵强，不过他确实存在。难道我们还要整出个1.4版吗？可是，我们怎么可能在我的Singleton类中操作Class Loader啊？是的，你根本不可能。在这种情况下，你能做的只有是——“保证多个Class Loader不会装载同一个Singleton”。

其二、序列化。如果我们的这个Singleton类是一个关于我们程序配置信息的类。我们需要它有序列化的功能，那么，当反序列化的时候，我们将无法控制别人不多次反序列化。不过，我们可以利用一下Serializable接口的readResolve()方法，比如：

```
1 public class Singleton implements Serializable
2 {
3     .....
4     .....
5     protected Object readResolve()
6     {
7         return getInstance();
8     }
9 }
```

其三、多个Java虚拟机。如果我们的程序运行在多个Java的虚拟机中。什么？多个虚拟机？这是一种什么样的情况啊。嗯，这种情况是有点极端，不过还是可能出现，比如EJB或RMI之流的东西。要在这种环境下避免多实例，看来只能通过良好的设计或非技术来解决了。

其四，volatile变量。关于volatile这个关键字所声明的变量可以被看作是一种“程度较轻的同步synchronized”；与synchronized块相比，volatile变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是synchronized的一部分。当然，如前面所述，我们需要的Singleton只是在创建的时候线程同步，而后面的读取则不需要同步。所以，volatile变量并不能帮助我们即能解决问题，又有好的性能。而且，这种变量只能在JDK 1.5+版后才能使用。

其五、关于继承。是的，继承于Singleton后的子类也有可能造成多实例的问题。不过，因为我们早把Singleton的构造函数声明成了私有的，所以也就杜绝了继承这种事情。

其六，关于代码重用。也话我们的系统中有很多个类需要用到这个模式，如果我们在每一个类都中有这样的代码，那么就显得有点傻了。那么，我们是否可以使用一种方法，把这具模式抽象出去？在C++下这是很容易的，因为有模板和友元，还支持栈上分配内存，所以比较容易一些（程序如下所示），Java下可能比较复杂一些，聪明的你知道怎么做吗？

```
1 template class Singleton
2 {
3     public:
4         static T& Instance()
5         {
6             static T theSingleInstance; //假设T有一个protected
7             return theSingleInstance;
8         }
9 };
10
11 class OnlyOne : public Singleton
12 {
13     friend class Singleton;
14     int example_data;
```

```
15
16     public:
17         int GetExampleData() const {return example_data;}
18     protected:
19         OnlyOne(): example_data(42) {}    // 默认构造函数
20         OnlyOne(OnlyOne&) {}
21 };
22
23 int main( )
24 {
25     cout << OnlyOne::Instance().GetExampleData() << endl;
26     return 0;
27 }
```

(转载时请注明作者和出处。未经许可，请勿用于商业用途)

(全文完)



欢迎关注CoolShell微信公众账号

(转载本站文章请注明作者和出处 [酷壳 - CoolShell.cn](http://coolshell.cn) , 请勿用于任何商业用途)

——=== 访问 [酷壳404页面](#) 寻找遗失儿童。 ===——

11

分类: [Java语言](#), [程序设计](#), [编程语言](#) 标签: [Java](#), [Singleton](#), [设计模式](#)

★★★★★ (4 人打了分, 平均分: 4.50)

相关文章

- 2009年09月03日 [编程真难啊](#)
- 2009年06月18日 [如何在Java中避免equals方法的隐藏陷阱](#)
- 2009年04月22日 [Java如何取源文件中文件名和行号](#)
- 2011年12月28日 [由一个问题到 Resin ClassLoader 的学习](#)
- 2009年03月02日 [101个设计模式](#)
- 2009年05月23日 [20非常有用的Java程序片段](#)
- 2009年07月03日 [Java构造时成员初始化的陷阱](#)
- 2012年12月13日 [如此理解面向对象编程](#)

[评论 \(24\)](#) [Trackbacks \(5\)](#) [发表评论](#) [Trackback](#)



wahaha

2009年3月26日16:30 | [#1](#)

[回复](#) | [引用](#)

version 1.3中错把if (singleton != null) 写成 if (singleton == null) 了
平常一般都是1.0版，学习了~谢



耗子

2009年3月26日17:25 | [#2](#)

[回复](#) | [引用](#)

[wahaha](#) :

version 1.3中错把if (singleton != null) 写成 if (singleton == null) 了
平常一般都是1.0版，学习了~谢

嗯？！可能并没有错哦？你再仔细理解一下哦。：)



adam

2009年3月27日09:20 | [#3](#)

[回复](#) | [引用](#)

对于Singleton还有一种偷懒的方式你可以提一下，就是放弃这种可能引发一大堆同步问题的new操作，提前加载。

比如我可以这么写：

```
1 | public class Singleton
2 | {
3 |     private static final Singleton singleton = new Sin
```

```
4
5     private Singleton()
6     {
7     }
8     public static Singleton getInstance() {
9         return singleton;
10    }
11 }
```

或者可以这么写：

```
1 public class Singleton
2 {
3     private static final Singleton singleton = null;
4
5     static{
6         singleton = new Singleton();
7     }
8     private Singleton() { }
9     public static Singleton getInstance() {
10         return singleton;
11     }
12 }
```

第二种写法在好像某些极端的情况下也会出现同步问题，不过已经是很极端了。



4.

耗子

2009年3月27日17:32 | [#4](#)

[回复](#) | [引用](#)

to adam: 非常不错！看来，我还是在用C++的方式思考Java，受教了。



5.

sure-one

2009年12月18日00:20 | [#5](#)

[回复](#) | [引用](#)

在《effective JAVA》第一版中作者给出了一个更好的方法：

```
1 public class Singleton{
2     private Singleton(){}
3     private static class SingletonHolder{
4         static final Singleton instance=new Singleton(
5     }
6     public static Singleton getInstance() {
7         return SingletonHolder.instance;
8     }
9 }
```

然后在第二版中给出了一个更好的方案：

```
1 | public enum Singleton{  
2 |     instance;  
3 |     //other methods  
4 | }
```

当然这些也只是解决多线程的问题，其他的问题还是要靠别的手段来解决的。



6.

Link028

2011年11月22日09:19 | [#6](#)

[回复](#) | [引用](#)

[@耗子](#)

private static final Singleton singleton = null;

这个是有问题的，已经初始化了，以后不能再new 了



7.

amwtke

2011年12月6日22:15 | [#7](#)

[回复](#) | [引用](#)

[@adam](#)

这种提前加载的方式会在多线程（高并发）环境下造成麻烦。如果private static final Singleton singleton = new Singleton();中的构造方法涉及到异步的网络数据交换如读取服务器配置或者数据库，则此构造过程可能会被操作系统打断而没有完成加载，其他访问singleton实例的线程会脏，而且错误很难查到。



8.

amwtke

2011年12月6日22:24 | [#8](#)

[回复](#) | [引用](#)

相对而言Double-Check是正确的做法。



9.

[核桃博客](#)

2011年12月23日12:42 | [#9](#)

[回复](#) | [引用](#)

[amwtke](#) :

[@adam](#)

这种提前加载的方式会在多线程（高并发）环境下造成麻烦。如果 `private static final Singleton singleton = new Singleton();` 中的构造方法涉及到异步的网络数据交换如读取服务器配置或者数据库，则此构造过程可能会被操作系统打断而没有完成加载，其他访问singleton实例的线程脏，而且错误很难查到。

这个有更详细一点的说明么？为什么构造函数会被操作系统打断？

我开始学的时候用double check，后来感觉简单问题复杂化了，直接用提前加载的方式，而且我实际接触的项目里面需要这样用singleton的都很少，而且基本都是必须一开始就起来的；



10.

[lmyanglei](#)2012年5月17日23:52 | [#10](#)[回复](#) | [引用](#)

的确，修改成“`private static final Singleton singleton;`”就好了
[@Link028](#)



11.

[姚嘉俊](#)2012年6月9日23:41 | [#11](#)[回复](#) | [引用](#)

这篇文章对C++的double check有很好的说明,scott meyers写的喔
http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf



12.

[54yuri](#)2012年12月14日23:03 | [#12](#)[回复](#) | [引用](#)

<http://www.cnblogs.com/kamome/archive/2010/02/02/1661605.html>

“双重检测锁”模式。如182页所示，这种看似“聪明”的方式，其实有着巨大的漏洞。简单的说，在1.5之前的JVM中，代码会进行“重整”，单例引用uniqueInstance有时尽管不为null，但是此时所引用的那个“单例对象”，并没有被完全初始化。也就是new Singleton()函数未正式完成其工作之前，JVM可以根据Java规范，重整代码，使得uniqueInstance先获得这个“单例对象”的引用，这样一来，第二个线程直接判定单例已完成实例化，故接下来的客户代码会直接使用单例对象的数据，但是有些数据并没有被正确的初始化，因为new Singleton()尚未正式

完成。

— 也就是说，double check对判断singleton是不是为null没有任何问题，但是在使用上就会有问题！第二个线程因为通过double check认为现在的singleton变量已经不是null，可以直接使用了；但是实际上可是第一个线程还未完全初始化好的实例，它仅仅是不为null而已，也就是singleton指向的实例还不可用！但是此时第二个线程如果立刻使用该singleton 可能会出现问题，因为一些资源还没有真正初始化！

不过这种状况不知道实际当中多不多，反正我是没遇到过。。。



13.

sigh

2013年1月28日13:34 | [#13](#)

[回复](#) | [引用](#)

private static final Singleton singleton = null;

final应该去掉



14.

[Jack47](#)

2013年4月12日16:32 | [#14](#)

[回复](#) | [引用](#)

[@Imyanglei](#)

去掉赋初值null，修改成“private static final Singleton singleton;” 应该也是不行的。final修饰的变量只能在构造函数中或者是声明的时候初始化。而咱们讨论的上下文中，是要用Lazy Initialization的，所以必须去掉final。如果要保留final，可以参照这个方法 http://en.wikipedia.org/wiki/Singleton_pattern#Static_block_initialization



15.

冷兵器

2013年5月18日23:14 | [#15](#)

[回复](#) | [引用](#)

学习了，ssh框架的设计提供了很多模式，不小心就出错



16.

Tina


2014年12月5日21:22 | [#16](#)

[回复](#) | [引用](#)

version 1.0 里面把singleton定义为final如何再new？博主至少应该保证示例代码能够编译通过吧。。。

17. [陈皓](#)2015年1月1日13:16 | [#17](#)[回复](#) | [引用](#)[@Tina](#)

谢谢，已修改！

18. 

lance

2015年1月21日14:48 | [#18](#)[回复](#) | [引用](#)

Singleton 优雅版本

这个确实是非常牛逼。

原来的时候对于enum很不了解。

看了之后理解了很多。

```
public enum SingletonEnum {
```

```
    INSTANCE;
```

```
    private int value;
```

```
    private SingletonEnum() {
```

```
        //初始化工作
```

```
        this.value = 10;
```

```
    }
```

```
    public int getValue() {
```

```
        return this.value;
```

```
    }
```

```
    public void doStuff(){
```

```
        System.out.println( "Singleton using Enum:" + this.value);
```

```
    }
```

```
};
```

想问问这样的实现可以吗？

19. 

libin

2015年3月5日16:40 | [#19](#)

[回复](#) | [引用](#)

Singleton 优雅版本

```
public enum Singleton{  
    INSTANCE;  
}
```

居然用枚举！！看上去好牛逼，通过EasySingleton.INSTANCE来访问，这比调用getInstance()方法简单多了。

这一段，看了一下，应该是通过 Singleton.INSTANCE来访问。不知道 EasySingleton是哪里来的，此处是笔误吧

20. 

phil

2015年7月30日10:33 | [#20](#)

[回复](#) | [引用](#)

<http://wuchong.me/blog/2014/08/28/how-to-correctly-write-singleton-pattern/> 这篇文章几乎和您写的这个差不多，原作者是您吗？

21. 

shady

2015年9月25日16:26 | [#21](#)

[回复](#) | [引用](#)

身为刚刚接触java的小白 理解起来这个问题有点吃力。。。

22. 

蜗牛

2016年3月7日15:51 | [#22](#)

[回复](#) | [引用](#)

新手疑问，version 1.5第4行是不是写错了，上面的描述不是说用final吗？

23. 

feimi

2016年3月22日20:30 | [#23](#)

[回复](#) | [引用](#)

此文中一句话【1.2版的Singleton在多线程下的确没有问题了，因为我们同步了所有的线程。】，因为改进为1.3版本之后还不行，又来一个 1.4 版本，所以 这句话就不

合适，真的是多线程没有问题了吗？即使同步了所有线程？版本1.4的候补，很明显给了这个问题一个否定的答案。

关联阅读：http://www.race604.com/java-double-checked-singleton/?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io



24.

feimi

2016年3月23日08:55 | [#24](#)[回复](#) | [引用](#)[@feimi](#)

1.3版本也引入了一个非同步的代码，就是那句 null 的判断，所以，才需要了1.4版本，【1.2版的Singleton在多线程下的确没有问题了，因为我们同步了所有的线程。】也是没有问题的。。。

1. 2014年12月2日03:42 | [#1](#)[\[转\]深入浅出单实例Singleton设计模式 | blog](#)2. 2015年2月11日23:20 | [#2](#)[浅析单例模式与线程安全\(Linux环境c++版本\) - 剑客|关注科技互联网](#)3. 2015年5月27日14:44 | [#3](#)[\[原\]浅析单例模式与线程安全\(Linux环境c++版本\)乐滋坊 | 乐滋坊](#)4. 2015年7月6日22:39 | [#4](#)[设计模式\(3\): 单例模式 | 程序员之家](#)5. 2016年4月19日13:10 | [#5](#)[Singleton设计模式 - Study](#)

昵称 (必填)

电子邮箱 (我们会为您保密) (必填)

网址

[订阅评论](#)

提交评论

[雷人的程序注释](#) [基于JVM的语言正在开始流行](#)
[订阅](#)

[Twitter](#)

本站公告



访问 [酷壳404页面](#) 寻找遗失儿童！



酷壳建议大家多使用RSS访问阅读（本站已经是全文输出，推荐使用cloud.feedly.com 或digg.com）。有相关事宜欢迎电邮：haoel(at)hotmail.com。最后，感谢大家对酷壳的支持和体谅！

最新文章

- [让我们来谈谈分工](#)
- [Cuckoo Filter：设计与实现](#)
- [Docker基础技术：DeviceMapper](#)
- [Docker基础技术：AUFS](#)
- [Docker基础技术：Linux CGroup](#)
- [Docker基础技术：Linux Namespace（上）](#)
- [Docker基础技术：Linux Namespace（下）](#)
- [关于移动端的钓鱼式攻击](#)
- [Linux：为何对象引用计数必须是原子的](#)
- [DHH 谈混合移动应用开发](#)
- [HTML6 展望](#)
- [Google Inbox如何跨平台重用代码？](#)
- [vfork 挂掉的一个问题](#)
- [Leetcode 编程训练](#)
- [State Threads 回调终结者](#)
- [bash代码注入的安全漏洞](#)
- [互联网之子 - Aaron Swartz](#)
- [谜题的答案和活动的心得体会](#)
- [【活动】解谜题送礼物](#)
- [开发团队的效率](#)
- [TCP 的那些事儿（下）](#)
- [TCP 的那些事儿（上）](#)

- [「我只是认真」聊聊工匠情怀](#)
- [面向GC的Java编程](#)
- [C语言的整型溢出问题](#)
- [从LongAdder看更高效的无锁实现](#)
- [从Code Review 谈如何做技术](#)
- [C语言结构体里的成员数组和指针](#)
- [无插件Vim编程技巧](#)
- [Python修饰器的函数式编程](#)

全站热门

- [程序员技术练级攻略](#)
- [简明 Vim 练级攻略](#)
- [做个环保主义的程序员](#)
- [如何学好C语言](#)
- [AWK 简明教程](#)
- [应该知道的Linux技巧](#)
- [“21天教你学会C++”](#)
- [6个变态的C语言Hello World程序](#)
- [TCP 的那些事儿 \(上\)](#)
- [由12306.cn谈谈网站性能技术](#)
- [编程能力与编程年龄](#)
- [“作环保的程序员，从不用百度开始”](#)
- [28个Unix/Linux的命令行神器](#)
- [我是怎么招聘程序员的](#)
- [sed 简明教程](#)
- [Web开发中需要了解的东西](#)
- [性能调优攻略](#)
- [C++ 程序员自信心曲线图](#)
- [Android将允许纯C/C++开发应用](#)
- [如何学好C++语言](#)
- [Lua简明教程](#)
- [如何写出无法维护的代码](#)
- [MySQL性能优化的最佳20+条经验](#)
- [无插件Vim编程技巧](#)
- [20本最好的Linux免费书籍](#)
- [二维码的生成细节和原理](#)
- [Windows编程革命简史](#)
- [编程真难啊](#)
- [深入理解C语言](#)
- [加班与效率](#)

新浪微博

微博



左耳朵耗子

北京 朝阳区

加关注

标签

[agile](#) [AJAX](#) [Algorithm](#) [Android](#) [Bash](#) [C++](#) [Coding](#) [CSS](#) [Database](#) [Design](#) [design](#)
[pattern](#) [ebook](#) [Flash](#) [Game](#) [Go](#) [Google](#) [HTML](#) [IE](#) [Java](#) [Javascript](#) [jQuery](#)
[Linux](#) [MySQL](#) [OOP](#) [password](#) [Performance](#) [PHP](#) [Programmer](#) [Programming](#)
[programming language](#) [Puzzle](#) [Python](#) [Ruby](#) [SQL](#) [TDD](#) [UI](#) [Unix](#) [vim](#) [Web](#) [Windows](#)
[XML](#) [安全](#) [程序员](#) [算法](#) [面试](#)

分类目录

- [.NET编程](#) (3)
- [Ajax开发](#) (9)
- [C/C++语言](#) (71)
- [Erlang](#) (1)
- [Java语言](#) (32)
- [PHP脚本](#) (11)
- [Python](#) (23)
- [Ruby](#) (5)
- [Unix/Linux](#) (74)
- [Web开发](#) (103)
- [Windows](#) (12)
- [业界新闻](#) (26)
- [企业应用](#) (2)
- [技术新闻](#) (33)
- [技术管理](#) (13)
- [技术读物](#) (117)
- [操作系统](#) (49)
- [数据库](#) (11)
- [杂项资源](#) (267)
- [流程方法](#) (46)

- [程序设计](#) (84)
- [系统架构](#) (8)
- [编程工具](#) (65)
- [编程语言](#) (174)
- [网络安全](#) (27)
- [职场生涯](#) (33)
- [趣味问题](#) (19)
- [轶事趣闻](#) (147)

归档

- [2015年十二月](#) (1)
- [2015年九月](#) (1)
- [2015年八月](#) (2)
- [2015年四月](#) (4)
- [2014年十二月](#) (3)
- [2014年十一月](#) (2)
- [2014年十月](#) (2)
- [2014年九月](#) (2)
- [2014年八月](#) (2)
- [2014年六月](#) (1)
- [2014年五月](#) (4)
- [2014年四月](#) (4)
- [2014年三月](#) (5)
- [2014年二月](#) (3)
- [2014年一月](#) (2)
- [2013年十二月](#) (3)
- [2013年十一月](#) (1)
- [2013年十月](#) (6)
- [2013年八月](#) (1)
- [2013年七月](#) (8)
- [2013年六月](#) (2)
- [2013年五月](#) (3)
- [2013年四月](#) (3)
- [2013年三月](#) (3)
- [2013年二月](#) (5)
- [2013年一月](#) (1)
- [2012年十二月](#) (4)
- [2012年十一月](#) (4)
- [2012年十月](#) (3)
- [2012年九月](#) (4)
- [2012年八月](#) (8)

- [2012年七月](#) (4)
- [2012年六月](#) (7)
- [2012年五月](#) (6)
- [2012年四月](#) (6)
- [2012年三月](#) (6)
- [2012年二月](#) (3)
- [2012年一月](#) (6)
- [2011年十二月](#) (5)
- [2011年十一月](#) (9)
- [2011年十月](#) (6)
- [2011年九月](#) (5)
- [2011年八月](#) (14)
- [2011年七月](#) (6)
- [2011年六月](#) (12)
- [2011年五月](#) (5)
- [2011年四月](#) (18)
- [2011年三月](#) (16)
- [2011年二月](#) (16)
- [2011年一月](#) (18)
- [2010年十二月](#) (11)
- [2010年十一月](#) (11)
- [2010年十月](#) (19)
- [2010年九月](#) (15)
- [2010年八月](#) (10)
- [2010年七月](#) (20)
- [2010年六月](#) (9)
- [2010年五月](#) (13)
- [2010年四月](#) (12)
- [2010年三月](#) (11)
- [2010年二月](#) (7)
- [2010年一月](#) (9)
- [2009年十二月](#) (22)
- [2009年十一月](#) (27)
- [2009年十月](#) (17)
- [2009年九月](#) (14)
- [2009年八月](#) (21)
- [2009年七月](#) (18)
- [2009年六月](#) (19)
- [2009年五月](#) (27)
- [2009年四月](#) (53)
- [2009年三月](#) (43)

- ## 最新评论

- ## 友情链接

- 2016年05月16日 15:09

- [HTML5研究小组](#)
- [朱文昊Albert Zhu](#)
- [C瓜哥的博客](#)
- [开源吧](#)
- [ACMer](#)
- [陈鹏个人博客](#)
- [OneCoder](#)
- [More Than Vimer](#)
- [运维派](#)
- [书巢](#)

功能

- [注册](#)
- [登录](#)
- [文章RSS](#)
- [评论RSS](#)
- [WordPress.org](#)



[回到顶部](#) [WordPress](#)

版权所有 © 2004-2016 酷 壳 – CoolShell.cn

主题由 [NeoEase](#) 提供, 通过 [XHTML 1.1](#) 和 [CSS 3](#) 验证.

