

# 单例模式 - - 反射 - - 防止序列化破坏单例模式

2017-03-03 23:21 by ttylinux, 2175 阅读, 2 评论, 收藏, 编辑

本文牵涉到的概念:

- 1.单例模式 - - - - - 唯一最佳实现方式，使用枚举类实现
- 2.单例模式的几种实现，各自的缺点
- 3.反射;反射是如何破坏单例模式
- 4.序列化；序列化如何破坏单例模式

## 单例模式

单例模式，是指在任何时候，该类只能被实例化一次，在任何时候，访问该类的对象，对象都是同一的，只有一个。

## 单例模式的实现方式：

### a.使用类公有的静态成员来保存该唯一对象



```
public class EagerSingleton {  
    // jvm保证在任何线程访问uniqueInstance静态变量之前一定先创建了此实例  
    public static EagerSingleton uniqueInstance = new EagerSingleton();  
  
    // 私有的默认构造子，保证外界无法直接实例化  
    private EagerSingleton() {  
    }  
}
```



### b.使用公有的静态成员工厂方法



```
public class EagerSingleton {  
    // jvm保证在任何线程访问uniqueInstance静态变量之前一定先创建了此实例  
    private static EagerSingleton uniqueInstance = new EagerSingleton();  
  
    // 私有的默认构造子，保证外界无法直接实例化
```

## About

你可以通过以下方式了解和联系我：

邮箱：albertxiaoyu@163.com

[译言网](#)

[我的Github](#)

[豆瓣](#)

昵称：[ttylinux](#)

园龄：[7年4个月](#)

粉丝：[5](#)

关注：[13](#)

[±加关注](#)

SEARCH

## 日历

2018年5月						
<	日	一	二	三	四	五
	29	30	1	2	3	4
	6	7	8	9	10	11
	13	14	15	16	17	18
	20	21	22	23	24	25
	27	28	29	30	31	1
	3	4	5	6	7	8

## 随笔分类

[Android\(29\)](#)

[cpp\(1\)](#)

[effective\\_java\(26\)](#)

[编码\(14\)](#)

[产品-交互\(4\)](#)

```
private EagerSingleton() {
}

// 提供全局访问点获取唯一的实例
public static EagerSingleton getInstance() {
    return uniqueInstance;
}
}
```

```
//懒汉式
同步一个方法可能造成程序执行效率下降100倍，完全没有必要每次调用getInstance都加锁，事实上我们只想
保证一次初始化成功，其余的快速返回而已,如果在getInstance频繁使用的地方就要考虑重新优化了.
public class LazySingleton {
    private static LazySingleton uniqueInstance;

    private LazySingleton() {
    }

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

3)"双检锁"(Double-Checked Lock)尽量将"加锁"推迟,只在需要时"加锁"(仅适用于Java 5.0 以上版本,volatile保证原子操作) happens-before:"什么什么一定在什么什么之前运行",也就是保证顺序性. 现在的CPU有乱序执行的能力(也就是指令会乱序或并行运行,可以不按我们写代码的顺序执行内存的存取过程),并且多个CPU之间的缓存也不保证实时同步,只有上面的 happens-before所规定的情况下才保证顺序性.

JVM能够根据CPU的特性(CPU的多级缓存系统、多核处理器等)适当的重新排序机器指令,使机器指令更符合CPU的执行特点，最大限度的发挥机器的性能.

如果没有volatile修饰符则可能出现一个线程t1的B操作和另一线程t2的C操作之间对instance的读写没有happens-before，可能会造成的现象是t1的B操作还没有完全构造成功，但t2的C已经看到instance为非空，这样t2就直接返回了未完全构造的instance的引用，t2想对instance进行操作就会出问题.

- volatile 的功能:
1. 避免编译器将变量缓存在寄存器里
  2. 避免编译器调整代码执行的顺序

优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。

架构(5)
设计模式(5)
随想(10)
疑难杂症(1)
音视频多媒体(3)
重构【改善既有代码的设计】(5)
转载(4)

### 推荐排行榜

1. Activity与Service的回收(1)
2. JVM-内存回收算法--复制算法(1)
3. 重载equals方法时要遵守的通用约定--自反性,对称性,传递性，一致性，非空性(1)
4. 单例模式 - - 反射 - - 防止序列化破坏单例模式(1)

### 阅读排行榜

1. onCreateView的一个细节--Fragment(6822)
2. Android项目分包---总结-----直接使用(5084)
3. 单例模式 - - 反射 - - 防止序列化破坏单例模式(2175)
4. 如何让一个类不能被实例化(1503)
5. android:process结合activity启动模式的一次实践(1409)



```
public class DoubleCheckedLockingSingleton {  
    // java中使用双重检查锁定机制,由于Java编译器和JIT的优化的原因系统无法保证我们期望的执行  
    次序。  
  
    // 在java5.0修改了内存模型,使用volatile声明的变量可以强制屏蔽编译器和JIT的优化工作  
    private volatile static DoubleCheckedLockingSingleton uniqueInstance;  
  
    private DoubleCheckedLockingSingleton() {  
    }  
  
    public static DoubleCheckedLockingSingleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (DoubleCheckedLockingSingleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new  
DoubleCheckedLockingSingleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```



4)Lazy initialization holder class 满足所有 Double-Checked Locking 满足的条件，并且没有显示的同步操作



```
public class LazyInitHolderSingleton {  
    private LazyInitHolderSingleton() {  
    }  
  
    private static class SingletonHolder {  
        private static final LazyInitHolderSingleton INSTANCE = new  
LazyInitHolderSingleton();  
    }  
  
    public static LazyInitHolderSingleton getInstance() {  
        return SingletonHolder.INSTANCE;  
    }  
}
```



根据jvm规范，当某对象第一次调用LazyInitHolderSingleton.getInstance()时，LazyInitHolderSingleton类被首次主动使用，jvm对其进行初始化（此时并不会调用LazyInitHolderSingleton()构造方法;进行LazyInitHolderSingleton的类加载，初始化静态变量），然后LazyInitHolderSingleton调用getInstance()方法，该方法中，又首次主动使用了SingletonHolder类，所以要对SingletonHolder类进行初始化（类的静态变量首先加载，进行初始化），初始化中，INSTANCE常量被赋值时才调用了LazyInitHolderSingleton的构造方法LazyInitHolderSingleton()，完成了实例化并返回该实例。

当再有对象（也许是在别的线程中）再次调用LazyInitHolderSingleton.getInstance()时，因为已经初始化过了，不会再进行初始化步骤，所以直接返回INSTANCE常量即同一个LazyInitHolderSingleton实例。

C 使用枚举类的方式来实现单例

推荐做法



```
public enum SingletonClass {
    INSTANCE;

    private String name;
    public void test() {
        System.out.println("The Test!");
    }

    public void setName(String name){

        this.name= name;
    }

    public String getName(){

        return name;
    }
}

public class TestMain {

    public static void main(String[] args) {

        SingletonClass one = SingletonClass.INSTANCE;
        SingletonClass two = SingletonClass.INSTANCE;
```

```
one.test();

one.setName("I am a SingletonClass Instance");

System.out.println(one.getName());

if (one == two) {

System.out.println("There are same");
}
}
}
```



反射

反射是如何破坏单例模式的，单例模式的目标是，任何时候该类都只有唯一的一个对象

比如，实现一个单例:



```
package com.effective.singleton;

public class Elvis
{
    private static boolean flag = false;

    private Elvis(){

    }

    private static class SingletonHolder{
        private static final Elvis INSTANCE = new Elvis();
    }

    public static Elvis getInstance()
    {
        return SingletonHolder.INSTANCE;
    }

    public void doSomethingElse()
    {

    }

}
```



使用反射的方式来实例化该类



```
package com.effective.singleton;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class ElvisReflectAttack
{

    public static void main(String[] args) throws InstantiationException,
    IllegalAccessException, IllegalArgumentException, InvocationTargetException,
    NoSuchMethodException, SecurityException
    {

        Class<?> classType = Elvis.class;

        Constructor<?> c = classType.getDeclaredConstructor(null);
        c.setAccessible(true);
        Elvis e1 = (Elvis)c.newInstance();
        Elvis e2 = Elvis.getInstance();
        System.out.println(e1==e2);

    }


}
```



输出结果为false，说明e1和e2不是同一个对象，到这里，单例模式不起作用。如果e1和e2都是指向同一个对象的，那么它们的引用值相等。

序列化

使用序列化的方式，单例模式是如何失效的



```
package com.serialize;

import java.io.Serializable;

public class SerSingleton implements Serializable
{

    private static final long serialVersionUID = 1L;

    String name;
```

```

private SerSingleton()
{
    System.out.println("Singleton is create");
    name="SerSingleton";
}

private static SerSingleton instance = new SerSingleton();

public static SerSingleton getInstance()
{
    return instance;
}

public static void createString()
{
    System.out.println("createString in Singleton");
}
}

@Test
public void test() throws IOException, ClassNotFoundException
{
    SerSingleton s1= null;
    SerSingleton s = SerSingleton.getInstance();

    FileOutputStream fos = new FileOutputStream("SerSingleton.obj");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(s);
    oos.flush();
    oos.close();

    FileInputStream fis = new FileInputStream("SerSingleton.obj");
    ObjectInputStream ois = new ObjectInputStream(fis);
    s1 = (SerSingleton)ois.readObject();
    System.out.println(s==s1);
}

```



输出结果为false。s和s1指向的对象不是同一个。

如何避免单例模式被破坏

1.反射

第二次实例化的时候，抛出异常



```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class Elvis {
    private static boolean flag = false;

    private Elvis() {
        synchronized (Elvis.class) {
            System.out.println(" try to instance");
            if (flag == false) {
                System.out.println("first time instance");
                flag = !flag;
            } else {
                throw new RuntimeException("单例模式被侵犯!");
            }
        }
    }

    private static class SingletonHolder {
        // jvm保证在任何线程访问INSTANCE静态变量之前一定先创建了此实例
        private static final Elvis INSTANCE = new Elvis();
    }

    public static Elvis getInstance() {
        System.out.println("in getInstance");
        return SingletonHolder.INSTANCE;
    }

    public void doSomethingElse() {

    }


    public static void main(String[] args) throws InstantiationException,
        IllegalAccessException,
        IllegalArgumentException, InvocationTargetException, NoSuchMethodException,
        SecurityException {

        Class<?> classType = Elvis.class;
        Constructor<?> c = classType.getDeclaredConstructor(null);
        c.setAccessible(true);
        Elvis e1 = (Elvis) c.newInstance();
```



```
Elvis e2 = Elvis.getInstance();

System.out.println(e1 == e2);
}
}
```



输出结果

**try to instance**

**first time instance**

in getInstance

try to instance

Exception in thread "main" java.lang.ExceptionInInitializerError

at chapterOne.Elvis.getInstance(Elvis.java:28)

at chapterOne.Elvis.main(Elvis.java:43)

Caused by: java.lang.RuntimeException: 单例模式被侵犯！

at chapterOne.Elvis.<init>(Elvis.java:16)

at chapterOne.Elvis.<init>(Elvis.java:9)

at chapterOne.Elvis\$SingletonHolder.<clinit>(Elvis.java:23)

... 2 more

分析：

a.因为，反射执行了，会创建一个对象。这时候，是不走静态方法getInstance的

b.然后，我们尝试去获得一个单例，会失败。因为，我们调用静态方法getInstance，会尝试创建一个实例。而此时，实例已经创建过了。

这样，就可以保证只有一个实例。**是达到效果了，但是，在这种情况下，反射先于静态方法getInstance执行。这导致，我们无法获得已经该实例。**

所以，其实这种方法，是不好的。**除非，你可以保证，你的getInstance方法，一定先于反射代码执行。否则虽然有效果，但是你得不到指向该实例的引用。**

2.序列化

在被序列化的类中添加readResolve方法

Deserializing an object via readUnshared invalidates the stream handle associated with the returned object. Note that this in itself does not always guarantee that the reference returned by readUnshared is unique; the deserialized object may define a readResolve method which returns an object visible to other parties, or readUnshared may return a Class object or enum constant obtainable elsewhere in the stream or through external means. If the deserialized object defines a readResolve method and the invocation of that method returns an array, then readUnshared returns a shallow clone of that array; this guarantees that the returned array object is unique and cannot be obtained a second time from an invocation of readObject or readUnshared on the ObjectInputStream, even if the underlying data stream has been manipulated.



```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class SerSingleton implements Serializable {

    private static final long serialVersionUID = 1L;
    String name;

    private SerSingleton() {
        System.out.println("Singleton is create");
        name = "SerSingleton";
    }

    private static SerSingleton instance = new SerSingleton();

    public static SerSingleton getInstance() {
        return instance;
    }

    public static void createString() {
        System.out.println("createString in Singleton");
    }

    private Object readResolve(){
        return instance;
    }

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        SerSingleton s1 = null;
        SerSingleton s = SerSingleton.getInstance();

        FileOutputStream fos = null;
        ObjectOutputStream oos = null;

        FileInputStream fis = null;
        ObjectInputStream ois = null;
        try {
            fos = new FileOutputStream("SerSingleton.obj");
            oos = new ObjectOutputStream(fos);
```

```
oos.writeObject(s);
} finally {
oos.flush();
oos.close();
fos.close();
}

try{
fis = new FileInputStream("SerSingleton.obj");
ois = new ObjectInputStream(fis);
s1 = (SerSingleton) ois.readObject();
}finally{
ois.close();
fis.close();
}
System.out.println(s == s1);
}

}
```



////////////////////////////////////

总结，实现单例模式的唯一推荐方法，使用枚举类来实现。使用枚举类实现单例模式，在对枚举类进行序列化时，还不需要添加readRsolve方法就可以避免单例模式被破坏。

使用枚举类来实现



```
public enum SingletonClass implements Serializable {

INSTANCE;

private static final long serialVersionUID = 1L;

private String name;

public void test() {
System.out.println("The Test!");
}

public void setName(String name) {
```

```
this.name = name;
}

public String getName() {

return name;
}

public static void main(String[] args) throws IOException, ClassNotFoundException {
SingletonClass s1 = null;
SingletonClass s = SingletonClass.INSTANCE;

FileOutputStream fos = null;
ObjectOutputStream oos = null;

FileInputStream fis = null;
ObjectInputStream ois = null;
try {
fos = new FileOutputStream("SingletonClass.obj");
oos = new ObjectOutputStream(fos);
oos.writeObject(s);
} finally {
oos.flush();
oos.close();
fos.close();
}

try {
fis = new FileInputStream("SingletonClass.obj");
ois = new ObjectInputStream(fis);
s1 = (SingletonClass) ois.readObject();
} finally {
ois.close();
fis.close();
}
System.out.println(s == s1);
}
}
```



输出:

true

引用:  
<http://blog.csdn.net/u013256816/article/details/50427061>

<http://blog.csdn.net/u013256816/article/details/50474678>

<http://blog.csdn.net/u013256816/article/details/50525335>

[http://blog.csdn.net/zhang\\_yanye/article/details/50344447](http://blog.csdn.net/zhang_yanye/article/details/50344447)

版权声明：  
作者：[ttylinux](#)

出处：<http://www.cnblogs.com/ttylinux/>

本文版权归作者，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

好文要顶

关注我

收藏该文

ttylinux

关注 - 13

粉丝 - 5

1

0

+加关注

« 上一篇：[遇到构造器中有多个可选参数时要考虑用构建器](#)  
» 下一篇：[如何让一个类不能被实例化](#)

分类: [effective\\_java](#)

#1楼 Henry.Hua

2018-01-18 19:23

今天刚接触到用通过反射破坏单例模式。然后就搜到了你这个帖子，如何防止单例模式被破坏。哈哈哈哈哈... 厉害了

支持(0)

反对(0)

#2楼 孤独小洋

2018-04-24 11:59

3)"双检锁"(Double-Checked Lock)尽量将"加锁"推迟,只在需要时"加锁"(仅适用于Java 5.0 以上版本,**volatile保证原子操作**)  
楼主这句话说得不对吧，应该是volatile可以防止指令重排，但是不能保证原子性操作

支持(0)

反对(0)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。



## 腾讯云让移动开发更简单

零代码集成分析，推送，检测，存储等服务

立即体验



最新IT新闻：

- 宝马获上海市智能网联路测牌照，将推进Level 4自动驾驶路试
  - 让语音助手好好说话这项成就，Google还有些事儿没告诉你
  - 38亿收购981个公众号，这家农药兽药企业打的什么算盘？
  - 亚马逊毛利润猛增70亿美元 超五大零售商增幅总和
  - 智能医疗行业，“爆款单品”的概念已经行不通了
- » 更多新闻...



300+篇运维、数据库等  
实战资料免费下载

点击获取



最新知识库文章：

- 如何高效学习
  - 如何成为优秀的程序员？
  - 菜鸟工程师的超神之路 -- 从校园到职场
  - 如何识别人的技术能力和水平？
  - 写给自学者的入门指南
- » 更多知识库文章...