**单例模式**

【转】http://www.cnblogs.com/rwxwsblog/p/6662951.html

**Java内存模型的抽象示意图:**

所有单例模式都有一个共性，那就是这个类没有自己的状态。也就是说无论这个类有多少个实例，都是一样的；然后除此以外更重要的是，这个类如果有两个或两个以上的实例的话程序会产生错误。

**非线程安全的模式**

public class Singleton {

private static Singleton instance;

private Singleton(){

}

public static Singleton getInstance() {

if (instance == null) //1：A线程执行

instance = new Singleton(); //2：B线程执行

return instance;

}

}

**普通加锁**

public class SafeLazyInitialization {

private static Singleton instance;

public synchronized static Singleton getInstance() {

if (instance == null)

instance = new Singleton();

return instance;

}

}

出于性能考虑，采用双重检查加锁的模式

**双重检查加锁模式**

public class Singleton{

private static Singleton singleton;

private Singleton(){

}

public static Singleton getInstance(){

if(null == singleton){ //第一次检查

synchronized(Singleton.class){ //加锁

if(null == singleton){ //第二次检查

singleton = new Singleton();//问题的根源出在这里

}

}

}

return singleton;

}

}

双重检查加锁模式相对于普通的单例和加锁模式而言，从性能和线程安全上来说都有很大的提升和保障。然而双重检查加锁模式也存在一些隐蔽不易被发现的问题。首先我们要明白在JVM创建新的对象时，主要要经过三个步骤。

* 分配内存
* 初始化构造器
* 将对象指向分配的内存地址

这样的顺序在双重加锁模式下是么有问题的，对象在初始化完成之后再把内存地址指向对象。

**问题的根源**

但是现代的JVM为了追求执行效率会针对字节码（编译器级别）以及指令和内存系统重排序（处理器重排序）进行调优，这样的话就有可能(注意是有可能)导致2和3的顺序是相反的，一旦出现这样的情况问题就来了。

java源代码到最终实际执行的指令序列:

前面的双重检查锁定示例代码的（instance = new Singleton();）创建一个对象。这一行代码可以分解为如下的三行伪代码：

memory = allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance = memory; //3：设置instance指向刚分配的内存地址

上面三行伪代码中的2和3之间，可能会被重排序（在一些JIT编译器上，这种重排序是真实发生的，详情见参考文献1的“Out-of-order writes”部分）。2和3之间重排序之后的执行时序如下：

memory = allocate(); //1：分配对象的内存空间

instance = memory; //3：设置instance指向刚分配的内存地址

//注意，此时对象还没有被初始化！

ctorInstance(memory); //2：初始化对象

多线程并发执行的时候的情况:

**解决方案**

**基于Volatile的解决方案**

先来说说Volatile这个关键字的含义：

* 可以很好地解决可见性问题
* 但不能确保原子性问题（通过 synchronized 进行解决）
* 禁止指令的重排序（单例主要用到此JVM规范）

Volatile 双重检查加锁模式

public class Singleton{

private volatile static Singleton singleton;

private Singleton(){

}

public static Singleton getInstance(){

if(null == singleton){

synchronized(Singleton.class){

if(null == singleton){

singleton = new Singleton();

}

}

}

return singleton;

}

}

**基于类初始化的解决方案**

利用静态内部类的方式来创建，因为静态属性由JVM确保第一次初始化时创建，因此也不用担心并发的问题出现。当初始化进行到一半的时候，别的线程是无法使用的，因为JVM会帮我们强行同步这个过程。另外由于静态变量只初始化一次，所以singleton仍然是单例的。

这个方案的实质是：允许“问题的根源”的三行伪代码中的2和3重排序，但不允许非构造线程（这里指线程B）“看到”这个重排序。

静态内部类的方式

public class Singleton{

private Singleton(){}

public static Singleton getInstance(){

return InnerClassSingleton.singleton;

}

private class InnerClassSingleton{

protected static Singleton singleton = new Singleton();

}

}

然而，虽然静态内部类模式可以很好地避免并发创建出多个实例的问题，但这种方式仍然有其存在的隐患。

**存在的隐患**

* 一旦一个实例被持久化后重新生成的实例仍然有可能是不唯一的。
* 由于java提供了反射机制，通过反射机制仍然有可能生成多个实例。

**序列化和反序列化带来的问题：反序列化后两个实例不一致了。**

private static void singleSerializable() {

try (FileOutputStream fileOutputStream=new FileOutputStream(new File("myObjectFilee.txt"));

ObjectOutputStream objectOutputStream=new ObjectOutputStream(fileOutputStream);) {

// SingletonObject singletonObject = SingletonObject.getInstance();

// InnerClassSingleton singletonObject = InnerClassSingleton.getInstance();

EnumSingleton singletonObject = EnumSingleton.INSTANCE;

objectOutputStream.writeObject(singletonObject);

objectOutputStream.close();

fileOutputStream.close();

System.out.println(singletonObject.hashCode());

} catch (IOException e) {

e.printStackTrace();

}

try (FileInputStream fileInputStream=new FileInputStream(new File("myObjectFilee.txt"));

ObjectInputStream objectInputStream=new ObjectInputStream(fileInputStream);) {

// SingletonObject singleTest=(SingletonObject) objectInputStream.readObject();

// InnerClassSingleton singleTest=(InnerClassSingleton) objectInputStream.readObject();

EnumSingleton singleTest=(EnumSingleton) objectInputStream.readObject();

objectInputStream.close();

fileInputStream.close();

System.out.println(singleTest.hashCode());

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

**问题点及解决办法**  
ObjectInputStream中的readOrdinaryObject。

if (obj != null &&

handles.lookupException(passHandle) == null &&

desc.hasReadResolveMethod())

{

Object rep = desc.invokeReadResolve(obj);

if (unshared && rep.getClass().isArray()) {

rep = cloneArray(rep);

}

if (rep != obj) {

handles.setObject(passHandle, obj = rep);

}

}

调用自定义的readResolve方法

protected Object readResolve(){

System.out.println("调用了readResolve方法！");

return InnerClassSingleton.getInstance();

}

**通过反射机制获取到两个不同的实例**

private static void attack() {

try {

Class<?> classType = InnerClassSingleton.class;

Constructor<?> constructor = classType.getDeclaredConstructor(null);

constructor.setAccessible(true);

InnerClassSingleton singleton = (InnerClassSingleton) constructor.newInstance();

InnerClassSingleton singleton2 = InnerClassSingleton.getInstance();

System.out.println(singleton == singleton2); //false

} catch (NoSuchMethodException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

解决方案 : 私有构造方法中进行添加标志判断。

private InnerClassSingleton() {

synchronized (InnerClassSingleton.class) {

if (false == flag) {

flag = !flag;

} else {

throw new RuntimeException("单例模式正在被攻击");

}

}

}

**单例最优方案，枚举的方式**

枚举实现单例的优势

* 自由序列化；
* 保证只有一个实例（即使使用反射机制也无法多次实例化一个枚举量）；
* 线程安全；

public enum Singleton {

INSTANCE;

private Singleton(){}

}

**Hibernate的解决方案**

通过ThreadLocal的方式

import org.hibernate.HibernateException;

import org.hibernate.Session;

import org.hibernate.cfg.Configuration;

public class HibernateSessionFactory {

private static String CONFIG\_FILE\_LOCATION = "/hibernate.cfg.xml";

private static final ThreadLocal threadLocal = new ThreadLocal();

private static Configuration configuration = new Configuration();

private static org.hibernate.SessionFactory sessionFactory;

private static String configFile = CONFIG\_FILE\_LOCATION;

static {

try {

configuration.configure(configFile);

sessionFactory = configuration.buildSessionFactory();

} catch (Exception e) {

System.err.println("%%%% Error Creating SessionFactory %%%%");

e.printStackTrace();

}

}

private HibernateSessionFactory() {

}

public static Session getSession() throws HibernateException {

Session session = (Session) threadLocal.get();

if (session == null || !session.isOpen()) {

if (sessionFactory == null) {

rebuildSessionFactory();

}

session = (sessionFactory != null) ? essionFactory.openSession() : null;

threadLocal.set(session);

}

return session;

}

// Other methods...

}