# 特点

单例类只能有一个实例。

# 测试代码

public class TTestThread extends Thread {  
 private static Object CreateObj() {  
 return LasySingleObj.*getInstance*();  
 }  
  
 public static void MyRun() {  
 for(int i = 0; i < 2; ++i){  
 *CreateObj*();  
 try {  
 Thread.*sleep*(1);  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 }  
  
 public void run(){  
 *MyRun*();  
 }  
}

public class main {  
 public static void main(String argv[]){  
 System.*out*.println("enter main()");  
 for(int i = 0; i < 3; ++i){  
 new TTestThread().start();  
 }  
 TTestThread.*MyRun*();  
 System.*out*.println("exit main()");  
 return;  
 }  
}

# 单例的几种模式

## 饿汉式

public class HungerSingleObj {  
 private HungerSingleObj(){  
 System.*out*.println("enter HungerSingleObj()");  
   
 //休眠100毫秒，尽量引发多线程并发导致的构造过个对象  
 try {  
 Thread.*sleep*(100);  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 private static HungerSingleObj *instance* = new HungerSingleObj();  
 public static HungerSingleObj getInstance(){  
 return *instance*;  
 }  
}

enter main()

enter HungerSingleObj()

exit main()

备注：

不是在程序初始化时就构造该对象，而是在该类被调用（就是该类的任何函数，任何变量被引用时，就构造该对象）

## 懒汉式

### 线程不安全

public class LasySingleObj {  
 private LasySingleObj(){  
 System.*out*.println("enter LasySingleObj()");  
 try {  
 Thread.*sleep*(100);  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 private static LasySingleObj *instance* = null;  
 public static LasySingleObj getInstance(){  
 if(null == *instance*)  
 {  
 {  
 *instance* = new LasySingleObj();  
 }  
 }  
 return *instance*;  
 }  
}

enter main()

enter LasySingleObj()

enter LasySingleObj()

enter LasySingleObj()

enter LasySingleObj()

exit main()

备注：只有getInstance被调用时，才构造对象

### 线程安全

import com.sun.source.tree.SynchronizedTree;  
  
public class LasyThreadSafeSingleObj {  
 private LasyThreadSafeSingleObj(){  
 System.*out*.println("enter LasyThreadSafeSingleObj()");  
 try {  
 Thread.*sleep*(100);  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 private static volatile LasyThreadSafeSingleObj *instance* = null;  
 public static LasyThreadSafeSingleObj getInstance(){  
 if(null == *instance*)  
 {  
 synchronized(LasyThreadSafeSingleObj.class) {  
 if (null == *instance*) {  
 *instance* = new LasyThreadSafeSingleObj();  
 }  
 }  
 }  
 return *instance*;  
 }  
}

enter main()

enter LasyThreadSafeSingleObj()

exit main()

注意：

双检测+锁

Volatile避免指令重排序

二级缓存的内容修改后立即更新到内存中，导致其他线程感觉到这个变化

### 内部私有类

public class InnerPrivateClass\_SingleObj {  
  
 private InnerPrivateClass\_SingleObj(){  
 System.*out*.println("enter InnerPrivateClass\_SingleObj()");  
  
 //休眠100毫秒，尽量引发多线程并发导致的构造过个对象  
 try {  
 Thread.*sleep*(100);  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 public static InnerPrivateClass\_SingleObj getInstance(){  
 return InnerPrivateClass.*instance*;  
 }  
  
 private final static class InnerPrivateClass{  
 public static final InnerPrivateClass\_SingleObj *instance* = new InnerPrivateClass\_SingleObj();  
 }  
}

enter main()

enter InnerPrivateClass\_SingleObj()

exit main()

可以看到是线程安全的。

## 注册式

### 枚举式单例

public enum EnumInstance {

INSTANCE,;//该句在jdk被展开成public static final EnumInstance INSTANCE;再用一个静态块，new出该对象static {INSTANCE = new EnumInstance("INSTANCE", 0);}

private Date birthDay;

public Date getBirthDay() {

return birthDay;

}

public void setBirthDay(Date birthDay) {

this.birthDay = birthDay;

}

public static EnumInstance getInstance()

{

return INSTANCE;

}}

枚举采用了静态成员的办法（饿汉式）来保证线程安全。也就是枚举使用jdk来实现了静态成员的饿汉式，天然就是单例模式。

枚举式单例，为啥不能反射和序列化破坏？愿意你是反射的JDK代码，判断了需要反射的对象是否是枚举，如果是枚举，直接抛出异常，所以反射就无法破坏单例。

而反序列会检查getResolve，JDK保证枚举类有getResolve方法，所以反序列虽然会分配内存，但是不会构造新的对象。而是释放分配的内存。返回getResolve返回的对象。

### 容器式单例

## ThreadLocal(不是单例)

# 反射破坏

## 测试代码

public class TTestReflection {  
 public static void test(){  
 try {  
 Class<?> classX = InnerPrivateClass\_SingleObj.class;  
 Constructor c = classX.getDeclaredConstructor();  
 c.setAccessible(true);  
 c.newInstance();  
  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
  
 }  
}

## 阻止破坏的代码（红色）

public class InnerPrivateClass\_SingleObj {  
  
 private InnerPrivateClass\_SingleObj(){  
 System.*out*.println("enter InnerPrivateClass\_SingleObj()");  
  
 if(InnerPrivateClass.*instance* != null){  
 throw new RuntimeException("非法创建");  
 }  
  
  
 //休眠100毫秒，尽量引发多线程并发导致的构造过个对象  
 try {  
 Thread.*sleep*(100);  
 }catch (Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 public static InnerPrivateClass\_SingleObj getInstance(){  
 return InnerPrivateClass.*instance*;  
 }  
  
 private final static class InnerPrivateClass{  
 public static final InnerPrivateClass\_SingleObj *instance* = new InnerPrivateClass\_SingleObj();  
 }  
}

## 就是阻止直接调用构造函数

类的加载分为几步：

1. 分配内存，并将内存全部清0，作为该类的静态成员的空间
2. 标记该类正在被加载。其他线程如果想使用该类，就等待加载完成。本线程要使用该类可以直接使用了
3. 初始化类的静态成员函数。

所以直接调用构造函数导致

1. 进入单例的构造函数
2. 需要访问“内部类”，该类还未加载，导致“内部类”被加载
3. 分配内存，并将内存全部清0，作为该类的静态成员的空间
4. 标记该类正在被加载。其他线程如果想使用该类，就等待加载完成。本线程要使用该类可以直接使用了
5. 初始化类的静态成员函数， 此时再次
6. 进入单例的构造函数
7. 单例的构造函数需要访问内部类
8. 此时内部类正在加载，不需要再加载，可以直接使用。
9. 所以此时“内部类”的静态成员==null，单例的构造函数返回成功（第二次的单例构造函数成功）
10. 内部类加载成功，“内部类”的静态成员!=null, 所以单例的构造函数抛出异常（第一次的单例构造函数抛出异常）

通过getInstance成功

1. 进入getInstance
2. 要访问“内部类”，该类还未加载，导致“内部类”被加载
3. 分配内存，并将内存全部清0，作为该类的静态成员的空间
4. 标记该类正在被加载。其他线程如果想使用该类，就等待加载完成。本线程要使用该类可以直接使用了
5. 初始化类的静态成员函数， 此时再次
6. 进入单例的构造函数
7. 单例的构造函数需要访问内部类
8. 此时内部类正在加载，不需要再加载，可以直接使用。
9. 所以此时“内部类”的静态成员==null，单例的构造函数返回成功（第二次的单例构造函数成功）
10. getInstance函数直接成功返回

区别是：直接调用构造函数，导致“内部类”加载成功，其静态成员复制成功。检查失败，抛出异常

# 序列化破坏

## 测试代码

public class TTestSeriable {  
 public static void test(){  
 String FileName = "bin.obj";  
 InnerPrivateClass\_SingleObj obj1 = null;  
 InnerPrivateClass\_SingleObj obj2 = null;  
 try{  
 FileOutputStream fos = new FileOutputStream(FileName);  
 ObjectOutputStream oos = new ObjectOutputStream(fos);  
 obj1 = InnerPrivateClass\_SingleObj.*getInstance*();  
 oos.writeObject(obj1);  
 oos.flush();  
 oos.close();  
 }catch(Exception ex){  
 ex.printStackTrace();  
 }  
  
 try{  
 FileInputStream fis = new FileInputStream(FileName);  
 ObjectInputStream ois = new ObjectInputStream(fis);  
 obj2 = (InnerPrivateClass\_SingleObj)ois.readObject();  
  
 }catch(Exception ex){  
 ex.printStackTrace();  
 }  
 if(obj1 == obj2) {  
 System.*out*.println("obj1 == obj2:" + obj1);  
 }  
 else{  
 System.*out*.println("obj1 != obj2:" + obj1 + ":" + obj2);  
 }  
 }  
}

## 运行结果

enter main()

enter InnerPrivateClass\_SingleObj()

obj1 != obj2:InnerPrivateClass\_SingleObj@5ef04b5:InnerPrivateClass\_SingleObj@6477463f

exit main()

说明序列化构建了新的对象

## 阻止序列化

public class InnerPrivateClass\_SingleObj implements Serializable {  
 public Object readResolve(){  
 return InnerPrivateClass.*instance*;  
 }  
}

## 原理

ObjectInputStream.readObject-->readObject0-->checkResolve-->readOrdinaryObject

该函数首先分配一块内存，该内存作为下一步需要new的对象的地址。判断类是否有ReadResolve方法，没有就调用构造函数，返回这个地址。

如果有ReadResolve方法再调用Object rep = desc.invokeReadResolve(obj)方法，得到ReadResolve返回的对象

然后用返回的对象替换前面分配的内存，并且不再调用构造函数，而是直接返回。

# 总结

## 一般建议使用饿汉方式。

## 只有在要明确实现 lazy loading 效果时，才会使用登记/枚举方式。

## 如果涉及到反序列化创建对象时，可以尝试使用枚举方式。

## 如果有其他特殊的需求，可以考虑使用双检锁方式。