

#### 咕泡学院（每天20：30腾讯课堂公开课）

今日课程：Java设计模式之单例模式的七种写法 

咕泡学院-Tom老师 QQ : 441221062

咕泡学院-VIP咨询 QQ : 2082233439

咕泡学院-乐乐老师 QQ ：2279002544 (视频下载联系)

咕泡学院-技术交流 QQ ：608583947

打开腾讯课堂 搜索 咕泡学院，立即报名即可天天听老师讲课啦。

格言：不只做一个技术者，更要做一个思考者。

课程安排

# 什么是单例模式

Java中单例模式是一种常见的设计模式，单例模式的写法有好几种，这里主要介绍三种：懒汉式单例、饿汉式单例、登记式单例。

　　单例模式有以下特点：

　　1、单例类只能有一个实例。

　　2、单例类必须自己创建自己的唯一实例。

　　3、单例类必须给所有其他对象提供这一实例。

　　单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。

在讲单例之前，要做一次基础知识的科普行动。大家都知道Java类加载器加载内容的顺序：

1、从上往下（Java的变量需要先声明才能使用）

2、静态后动态(对象实例化)（静态块和static关键字修饰在实例化以前分配内存空间）

3、先属性后方法（成员变量不能定义在方法中，只能定义在class下）

# 二、懒汉式单例（4种写法）

|  |
| --- |
| //懒汉式单例类.在第一次调用的时候实例化自己  **public** **class** Singleton1 {  //1、第一步先将构造方法私有化  **private** Singleton1() {}  //2、然后声明一个静态变量保存单例的引用  **private** **static** Singleton1 *single* = **null**;  //3、通过提供一个静态方法来获得单例的引用  **public** **static** Singleton1 getInstance() {  **if** (*single* == **null**) {  *single* = **new** Singleton1();  }  **return** *single*;  }  } |

Singleton1 通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton1 的唯一实例只能通过getInstance()方法访问。

（事实上，通过Java反射机制是能够实例化构造方法为private的类的，那基本上会使所有的Java单例实现失效。此问题在此处不做讨论，姑且掩耳盗铃地认为反射机制不存在。）

但是以上懒汉式单例的实现没有考虑线程安全问题，它是线程不安全的，并发环境下很可能出现多个Singleton1 实例，要实现线程安全，有以下三种方式，都是对getInstance这个方法改造，保证了懒汉式单例的线程安全，如果你第一次接触单例模式，对线程安全不是很了解，可以先跳过下面这三小条，去看饿汉式单例，等看完后面再回头考虑线程安全的问题：

1. 在getInstance方法上加同步

|  |
| --- |
| //懒汉式单例.保证线程安全  **public** **class** Singleton2 {  //1、第一步先将构造方法私有化  **private** Singleton2() {}  //2、然后声明一个静态变量保存单例的引用  **private** **static** Singleton2 *single*=**null**;  //3、通过提供一个静态方法来获得单例的引用  //为了保证多线程环境下正确访问，给方法加上同步锁synchronized  **public** **static** **synchronized** Singleton2 getInstance() {  **if** (*single* == **null**) {  *single* = **new** Singleton2();  }  **return** *single*;  }  } |

# 双重检查锁定

|  |
| --- |
| //懒汉式单例.双重锁检查  **public** **class** Singleton3 {  //1、第一步先将构造方法私有化  **private** Singleton3() {}  //2、然后声明一个静态变量保存单例的引用  **private** **static** Singleton3 *single*=**null**;  //3、通过提供一个静态方法来获得单例的引用  //为了保证多线程环境下的另一种实现方式，双重锁检查  **public** **static** Singleton3 getInstance() {  **if** (*single* == **null**) {  **synchronized** (Singleton3.**class**) {  **if** (*single* == **null**) {  *single* = **new** Singleton3();  }  }  }  **return** *single*;  }  } |

1. 静态内部类

|  |
| --- |
| //懒汉式（静态内部类）  **public** **class** Singleton4 {  //1、先声明一个静态内部类  **private** **static** **class** LazyHolder {  **private** **static** **final** Singleton4 ***INSTANCE*** = **new** Singleton4();  }  //2、将默认构造方法私有化  **private** Singleton4 (){}    //3、同样提供静态方法获取实例  **public** **static** **final** Singleton4 getInstance() {  **return** LazyHolder.***INSTANCE***;  }  } |

这种方式是Singleton4 类被装载了，instance不一定被初始化。因为LazyHolder 类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载LazyHolder 类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton4 类加载时就实例化，因为我不能确保Singleton4 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种比上面1、2都好一些，既实现了线程安全，又避免了同步带来的性能影响。

# 三、饿汉式单例（1种写法）

|  |
| --- |
| //饿汉式单例类.在类初始化时，已经自行实例化  **public** **class** Singleton5 {  //1、 同样也是将默认构造方法私有化  **private** Singleton5() {}  //2、声明静态变量，在类实例化之前就初始化变量,将对象引用保存  **private** **static** **final** Singleton5 ***single*** = **new** Singleton5();  //3、开放静态方法，获取实例  **public** **static** Singleton5 getInstance() {  **return** ***single***;  }  } |

饿汉式在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

# 枚举式单例（1种写法，不常用）

|  |
| --- |
| **public** **enum** Singleton6 {  ***INSTANCE***;  **public** **void** getInstance() {  }  } |

这种方式不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

# 登记式单例（1种写法，不常用）

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Map;  //类似Spring里面的方法，将类名注册，下次从里面直接获取。  **public** **class** Singleton6 {  **private** **static** Map<String,Singleton6> *map* = **new** HashMap<String,Singleton6>();  **static**{  Singleton6 single = **new** Singleton6();  *map*.put(single.getClass().getName(), single);  }  //保护的默认构造子  **protected** Singleton6(){}  //静态工厂方法,返还此类惟一的实例  **public** **static** Singleton6 getInstance(String name) {  **if**(name == **null**) {  name = Singleton6.**class**.getName();  }  **if**(*map*.get(name) == **null**) {  **try** {  *map*.put(name, (Singleton6) Class.*forName*(name).newInstance());  } **catch** (InstantiationException e) {  e.printStackTrace();  } **catch** (IllegalAccessException e) {  e.printStackTrace();  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  }  **return** *map*.get(name);  }  } |

登记式单例实际上维护了一组单例类的实例，将这些实例存放在一个Map（登记薄）中，对于已经登记过的实例，则从Map直接返回，对于没有登记的，则先登记，然后返回。

这里我对登记式单例标记了可忽略，我的理解来说，首先它用的比较少，另外其实内部实现还是用的饿汉式单例，因为其中的static方法块，它的单例在类被装载的时候就被实例化了。

# 懒汉式单例跟饿汉式单例的根本区别

从名字上来说，饿汉和懒汉，

饿汉就是类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了，

而懒汉比较懒，只有当调用getInstance的时候，才回去初始化这个单例。

另外从以下两点再区分以下这两种方式：

1、线程安全：

饿汉式天生就是线程安全的，可以直接用于多线程而不会出现问题，

懒汉式本身是非线程安全的，为了实现线程安全有几种写法，分别是上面的1、2、3，这三种实现在资源加载和性能方面有些区别。

2、资源加载和性能：

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成，

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

至于1、2、3这三种实现又有些区别，

第1种，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，

第2种，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗

第3种，利用了classloader的机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗，所以一般我倾向于使用这一种。

# 代码演示线程安全问题

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

或者说：一个类或者程序所提供的接口对于线程来说是原子操作，或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题，那就是线程安全的。

应用

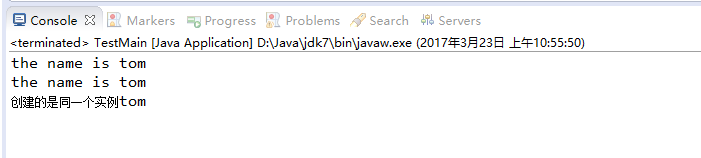
以下是一个单例类使用的例子，以懒汉式为例，这里为了保证线程安全，使用了双重检查锁定的方式：

|  |
| --- |
| **public** **class** TestSingleton {  String name = **null**;  **private** TestSingleton() {}    //注意这里用到了volatile关键字  **private** **static** **volatile** TestSingleton *instance* = **null**;    **public** **static** TestSingleton getInstance() {  **if** (*instance* == **null**) {  **synchronized** (TestSingleton.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** TestSingleton();  }  }  }  **return** *instance*;  }    **public** String getName() {  **return** name;  }    **public** **void** setName(String name) {  **this**.name = name;  }    **public** **void** printInfo() {  System.***out***.println("the name is " + name);  }    } |

可以看到里面加了volatile关键字来声明单例对象，既然synchronized已经起到了多线程下原子性、有序性、可见性的作用，为什么还要加volatile呢，在我们以前的课程中有专门讲到过。

|  |
| --- |
| **public** **class** TestMain {  **public** **static** **void** main(String[] args){  TestSingleton ts1 = TestSingleton.*getInstance*();  ts1.setName("james");  TestSingleton ts2 = TestSingleton.*getInstance*();  ts2.setName("tom");    ts1.print();  ts2.print();    **if**(ts1 == ts2){  System.***out***.println("创建的是同一个实例" + ts1.getName());  }**else**{  System.***out***.println("创建的不是同一个实例" + ts1.getName());  }  }  } |

运行结果



结论：由结果可以得知单例模式为一个面向对象的应用程序提供了对象惟一的访问点，不管它实现何种功能，整个应用程序都会同享一个实例对象。

对于单例模式的几种实现方式，知道饿汉式和懒汉式的区别，线程安全，资源加载的时机，还有懒汉式为了实现线程安全的3种方式的细微差别。

|  |
| --- |
| public class TestThread {    public static void main(String[] args) {  int count = 100;  CountDownLatch latch = new CountDownLatch(count);  final Set<Singleton4> syncSet = Collections.synchronizedSet(new HashSet<Singleton4>());      for (int i = 0; i < count; i++) {  new Thread(){  @Override  public void run() {  syncSet.add(Singleton4.getInstance());  }  }.start();  latch.countDown();  }  try {  latch.await();//等待所有线程全部完成，最终输出结果  System.out.println(syncSet.size());  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

# 八、分布式环境下的单例

有两个问题需要注意：

1.如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些servlet容器对每个servlet使用完全不同的类装载器，这样的话如果有两个servlet访问一个单例类，它们就都会有各自的实例。

2.如果Singleton实现了java.io.Serializable接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。

对第一个问题修复的办法是：

|  |
| --- |
| **private** **static** Class getClass(String classname) **throws** ClassNotFoundException {  ClassLoader classLoader = Thread.*currentThread*().getContextClassLoader();    **if**(classLoader == **null**){  classLoader = Singleton.**class**.getClassLoader();  }  **return** (classLoader.loadClass(classname));  } |

对第二个问题修复的办法是：

|  |
| --- |
| **public** **class** Singleton **implements** java.io.Serializable {  **public** **static** Singleton *INSTANCE* = **new** Singleton();  **protected** Singleton() { }  **private** Object readResolve() {  **return** *INSTANCE*;  }  } |

这个是序列化的知识，可能在以后课程中会讲序列化专题。