JAVA认证:深入浅出单实例Singleton设计模式Java认证考试
PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/559/2021_2022_JAVA_E8_AE_A4_E8_AF_81_c104_559590.htm 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。我们姑且把这具版本叫做1.0版 // version 1.0 public class Singleton { private static final Singleton singleton = null. private Singleton() { } public static Singleton getInstance() { if (singleton== null) { singleton= new Singleton(). } return singleton. } } 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） 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。 即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。 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。 所形成的实例保存在自己类中的私有成员中。 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，那我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： // version 1.1 public class Singleton { private static final Singleton singleton = null. private Singleton() { } public static Singleton getInstance() { if (singleton==null) { synchronized (Singleton.class) { singleton= new Singleton(). } } return singleton. } } 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： // version 1.2 public class Singleton { private static final Singleton singleton = null. private Singleton() { } public static Singleton getInstance() { synchronized (Singleton.class) { if (singleton== null) { singleton= new Singleton(). } } return singleton. } } 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. // version 1.3 public class Singleton { private static final Singleton singleton = null. private Singleton() { } public static Singleton getInstance() { if (singleton== null) { synchronized (Singleton.class) { if (singleton== null) { singleton= new Singleton(). } } } return singleton. } } 感觉代码开始

变得有点罗嗦和复杂了，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：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。 不然，我们就开始同步线程。 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！ 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、序例化。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： public class Singleton implements Serializable { .... protected Object readResolve() { return getInstance(). } } 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？ templategt. class Singleton { public: static Tlt.ONLYONElt.ONLYONEamp.) {} }. int main( ) { cout lt. OnlyOne::Instance().GetExampleData()lt. endl. return 0. } 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com