

Java设计模式—单例设计模式(Singleton Pattern)完全解析

原创

奋斗之路

于 2015-12-16 16:15:05 发布

32325

收藏 341

版权

分类专栏:

JAVA设计模式

文章标签:


Java

设计模式

单例设计模式

Singleton

实现思想

 JAVA设计模式 专栏收录该

3 订阅 1 篇文章

订阅专栏

转载请注明出处：<http://blog.csdn.net/dmk877/article/details/50311791>

相信大家都知道设计模式，听的最多的也应该是单例设计模式，这种模式也是在开发中用的最多的设计模式，可能有很多人会写几种设计模式，那么你是否知道什么是设计模式？为什么会有单例设计模式即它的作用是什么？单例模式有哪些写法？对于这样的问题，可能有部分童鞋并不能很好的回答，没关系今天就和大家一起来详细的学习下单例设计模式，相信通过学习本篇你将对单例设计模式有个详细的理解。**如有谬误欢迎批评指正，如有疑问欢迎留言。**

通过本篇博客你将学到以下内容

- ①什么是设计模式
- ②为什么会有单例设计模式即它的用处, 以及它解决了什么问题
- ③怎样实现单例, 即它的设计思想是什么
- ④单例模式有哪些写法
- ⑤单例模式在面试中要注意哪些事项

1、什么是设计模式？

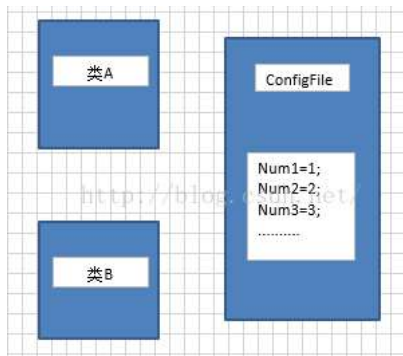
首先我们来看第一个问题什么是设计模式？在百度百科中它的定义是这样的：设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。（百度百科）

其实设计模式是人们实践的产物，在初期的开发过程中好多人发现再进行重复的代码书写，那些开发大神们就不断总结、抽取最终得到了大家的认可于是就产生了设计模式，其实设计模式的种类可以分为23种左右，今天主要和大家一起学习下单例设计模式，因为这种设计模式是使用的最多的设计模式。在以后的文章中会带来其他模式的讨论。

2、为什么会有单例设计模式？

我们都知道单例模式是在开发中用的最多的一种设计模式，那么究竟为什么会有单例设计模式呢？对于这个问题相信有很多会写单例的人都会有个这个疑问。在这里先说一下单例的用途，然后举一个例子大家就会明白为什么会有单例了。单例模式主要是为了避免因为创建了多个实例造成资源的浪费，且多个实例由于多次调用容易导致结果出现错误，**而使用单例模式能够保证整个应用中有且只有一个实例**。从其名字中我们就可以看出所谓单例，就是单个实例也就是说它可以解决的问题是：可以保证一个类在内存中的对象的唯一性，在一些常用的工具类、线程池、缓存，数据库，账户登录系统、配置文件等程序中可能只允许我们创建一个对象，一方面如果创建多个对象可能引起程序的错误，另一方面创建多个对象也造成资源的浪费。在这种基础之上单例设计模式就产生了因为使用单例能够保证整个应用中有且只有一个实例，看到这大家可能有些疑惑，没关系，我们来举一个例子，相信看完后你就会非常明白，为什么会有单例。

假如有一个有这么一个需求，有一个类A和一个类B它们共享配置文件的信息，在这个配置文件中有很多数据如下图所示



如上图所示现在类ConfigFile中存在共享的数据Num1，Num2，Num3等。假如在类A中修改ConfigFile中数据，在类A中应该有如下代码

```
1 ConfigFile configFile=new ConfigFile();
2 configFile.Num1=2;
```

这个时候configFile中的Num1=2，但是请注意这里是new ConfigFile是一个对象，想象一下在进行了上述操作后类B中进行如下操作

```
1 ConfigFile configFile=new ConfigFile();
2 System.out.println("configFile.Num1=" +configFile.Num1);
```

即直接new ConfigFile();然后打印Num1，大家思考一下这时候打印出的数据为几？我想你应该知道它打印的结果是这样的：configFile.Num1=1；也就是说因为每次调用都创建了一个ConfigFile对象，所以导致了在类A中的修改并不会真正改变ConfigFile中的值，它所更改的只是在类A中说创建的那个对象的值。假如现在要求在类A中修改数据后，要通知类B，即在类A和类B中操作的数据是同一个数据，类A改变一个数据，类B也会得到这个数据，并在类A修改后的基础上进行操作，那么我们应该怎么做呢？看到这大家可能会说so easy，把ConfigFile中的数据设置为静态不就Ok了吗？对，有这种想法很好，这样做也没有错。但是我们都知道静态数据的生命周期是很长的，假如ConfigFile中有很多数据时，如果将其全部设成静态的，那将是对内存的极大损耗。所以全部设置成静态虽然可行但并不是一个很好的解决方法。那么我们应该怎么做呢？要想解决上面的问题，其实不难，只要能保证对象是唯一的就可以解决上面的问题，那么问题来了如何保证对象的唯一性呢？这样就需要用单例设计模式了。

3、单例模式的设计思想

在上面我们说到现在解决问题的关键就是保证在应用中只有一个对象就行了，那么怎么保证只有一个对象呢？

其实只需要三步就可以保证对象的唯一性

(1) 不允许其他程序用new对象。

因为new就是开辟新的空间，在这里更改数据只是更改的所创建的对象的数据，如果可以new的话，每一次new都产生一个对象，这样肯定保证不了对象的唯一性。

(2) 在该类中创建对象

因为不允许其他程序new对象，所以这里的对象需要在本类中new出来

(3) 对外提供一个可以让其他程序获取该对象的方法

因为对象是在本类中创建的，所以需要提供一个方法让其它的类获取这个对象。

那么这三步怎么用代码实现呢？将上述三步转换成代码描述是这样的

(1) 私有化该类的构造函数

(2) 通过new在本类中创建一个本类对象

(3) 定义一个公有的方法，将在该类中所创建的对象返回

4、单例模式的写法

经过3中的分析我们理解了单例所解决的问题以及它的实现思想，接着来看看它的实现代码，单例模式的写法大的方面可以分为5种①懒汉式②饿汉式③双重校验锁④静态内部类⑤枚举。接下来我们就一起来看看这几种单例设计模式的代码实现，以及它们的优缺点

4.1 单例模式的饿汉式[可用]

```
1 public class Singleton {
2
3     private static Singleton instance=new Singleton();
4     private Singleton(){};
5     public static Singleton getInstance(){
6         return instance;
7     }
8 }
```

访问方式

```
Singleton instance = Singleton.getInstance();
```

得到这个实例后就可以访问这个类中的方法了。

优点：从它的实现中我们可以看到，这种方式的实现比较简单，在类加载的时候就完成了实例化，避免了线程的同步问题。

缺点：由于在类加载的时候就实例化了，所以没有达到Lazy Loading(懒加载)的效果，也就是说可能我没有用到这个实例，但是它

也会加载，会造成内存的浪费(但是这个浪费可以忽略，所以这种方式也是推荐使用的)。

4.2 单例模式的饿汉式变换写法[可用]

```
1 public class Singleton{
2
3     private static Singleton instance = null;
4
5     static {
6         instance = new Singleton();
7     }
8
9     private Singleton() {};
10
11     public static Singleton getInstance() {
12         return instance;
13     }
14 }
```

访问方式:

```
Singleton instance = Singleton.getInstance();
```

得到这个实例后就可以访问这个类中的方法了。

可以看到上面的代码是按照在2中分析的那三步来实现的，这中写法被称为饿汉式，因为它在类创建的时候就已经实例化了对象。其实4.2和4.1只是写法有点不同，都是在类初始化时创建对象的，它的优缺点和4.1一样，可以归为一种写法。

4.3 单例模式的懒汉式[线程不安全，不可用]

```
1 public class Singleton {
2
3     private static Singleton instance=null;
4
5     private Singleton() {};
6
7     public static Singleton getInstance(){
8
9         if(instance==null){
```

```
10 |         instance=new Singleton();
    |                                     11 |         }
12 |     return instance;
13 | }
14 | }
```

这种方式是在调用getInstance方法的时候才创建对象的，所以它比较懒因此被称为懒汉式。

在上述两种写法中懒汉式其实是存在线程安全问题的，喜欢刨根问题的同学可能会问，存在怎样的线程安全问题？怎样导致这种问题的？好，我们来说一下什么情况下这种写法会有问题。在运行过程中可能存在这么一种情况：有多个线程去调用getInstance方法来获取Singleton的实例，那么就有可能发生这样一种情况当第一个线程在执行if(instance==null)这个语句时，此时instance是为null的进入语句。在还没有执行instance=new Singleton()时(此时instance是为null的)第二个线程也进入if(instance==null)这个语句，因为之前进入这个语句的线程中还没有执行instance=new Singleton()，所以它会执行instance=new Singleton()来实例化Singleton对象，因为第二个线程也进入了if语句所以它也会实例化Singleton对象。这样就导致了实例化了两个Singleton对象。所以单例模式的懒汉式是存在线程安全问题的，既然它存在问题，那么可能有解决这个问题的方法，那么究竟怎么解决呢？对这种问题可能很多人会想到加锁于是出现了下面这种写法。

4. 4懒汉式线程安全的[线程安全，效率低不推荐使用]

```
1 | public class Singleton {
2 |
3 |     private static Singleton instance=null;
4 |
5 |     private Singleton() {};
6 |
7 |     public static synchronized Singleton getInstance(){
8 |
9 |         if(instance==null){
10 |             instance=new Singleton();
11 |         }
12 |         return instance;
13 |     }
14 | }
```

缺点：效率太低了，每个线程在想获得类的实例时候，执行getInstance()方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接return就行了。方法进行同步效率太低要改进。

4. 5单例模式懒汉式[线程不安全，不可用]

对于上述缺陷的改进可能有的人会想到如下的代码

```
1 | public class Singleton7 {
2 |
3 |     private static Singleton instance=null;
4 |
5 |     public static Singleton getInstance() {
6 |         if (instance == null) {
7 |             synchronized (Singleton.class) {
8 |                 instance = new Singleton();
9 |             }
10 |         }
11 |         return instance;
12 |     }
13 | }
```

其实这种写法跟4.3一样是线程不安全的，当一个线程还没有实例化Singleton时另一个线程执行到if(instance==null)这个判断语句时就会进入if语句，虽然加了锁，但是等到第一个线程执行完instance=new Singleton()跳出这个锁时，另一个进入if语句的线程同样会实例化另外一个Singleton对象，线程不安全的原理跟4.3类似。因此这种改进方式并不可行，经过大神们一步一步的探索，写出了懒汉式的双重校验锁。

4. 6单例模式懒汉式双重校验锁[推荐用]

```
1 public class Singleton {
2     /**
3      * 懒汉式变种，属于懒汉式中最好的写法，保证了：延迟加载和线程安全
4      */
5     private static Singleton instance=null;
6
7     private Singleton() {};
8
9     public static Singleton getInstance(){
10         if (instance == null) {
11             synchronized (Singleton.class) {
12                 if (instance == null) {
13                     instance = new Singleton();
14                 }
15             }
16         }
17         return instance;
18     }
19 }
```

访问方式

```
Singleton instance = Singleton.getInstance();
```

得到这个实例后就可以访问这个类中的方法了。

Double-Check概念对于多线程开发者来说不会陌生，如代码中所示，我们进行了两次if (instance== null)检查，这样就可以保证线程安全了。这样，实例化代码只用执行一次，后面再次访问时，判断if (instance== null)，直接return实例化对象。

优点：线程安全；延迟加载；效率较高。

4. 7内部类[推荐用]

```
1 public class Singleton{
2
3
4     private Singleton() {};
5
6     private static class SingletonHolder{
7         private static Singleton instance=new Singleton();
8     }
9
10    public static Singleton getInstance(){
11        return SingletonHolder.instance;
12    }
13 }
```

访问方式

```
Singleton instance = Singleton.getInstance();
```

得到这个实例后就可以访问这个类中的方法了。

这种方式跟饿汉式方式采用的机制类似，但又有不同。两者都是采用了类装载的机制来保证初始化实例时只有一个线程。不同

的地方在饿汉式方式是只要Singleton类被装载就会实例化，没有Lazy-Loading的作用，而静态内部类方式在Singleton类被装载时

并不会立即实例化，而是在需要实例化时，调用getInstance方法，才会装载SingletonHolder类，从而完成Singleton的实例化。

类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们保证了线程的安全性，在类进行初始化时，别的线程是

无法进入的。

优点：避免了线程不安全，延迟加载，效率高。

4.8枚举[极推荐使用]

```
1 public enum SingletonEnum {  
2  
3     instance;  
4  
5     private SingletonEnum() {}  
6  
7     public void method(){  
8     }  
9 }
```

访问方式

```
SingletonEnum.instance.method();
```

可以看到枚举的书写非常简单，访问也很简单在这里SingletonEnum.instance这里的instance即为SingletonEnum类型的引用所以得到它就可以调用枚举中的方法了。

借助JDK1.5中添加的枚举来实现单例模式。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。可能是因为枚举在JDK1.5中才添加，所以在实际项目开发中，很少见人这么写过，这种方式也是最好的一种方式，如果在开发中JDK满足要求的情况下建议使用这种方式。

5、总结

在真正的项目开发中一般采用4.1、4.6、4.7、4.8看你最喜欢哪种写法了，一般情况下这几种模式是没有问题的，为了装逼我一般采用4.6这种写法，我们经常用的Android-

Universal-Image-Loader这个开源项目也是采用的4.6这种写法，其实最安全的写法是4.8即枚举，它的实现非常简单而且最安全可谓很完美，但是可能是因为只支持JDK1.5吧又或者是因为枚举大家不熟悉所以目前使用的人并不多，但是大家可以尝试下。另外当我们使用反射机制时可能不能保证实例的唯一性，但是枚举始终可以保证唯一性，具体请参考次博客：

http://blog.csdn.net/java2000_net/article/details/3983958但是一般情况下很少遇到这种情况。

6、单例模式的在面试中的问题

单例模式在面试中会常常的被遇到，因为它是考察一个程序员的基础的扎实程度的，如果说你跟面试官说你做过项目，面试官让你写几个单例设计模式，你写不出来，你觉着面试官会相信吗？在面试时一定要认真准备每一次面试，靠忽悠即使你被录取了，你也很有可能会对这家公司不满意，好了我们言归正传，其实单例设计模式在面试中很少有人会问饿汉式写法，一般都会问单例设计模式的懒汉式的线程安全问题，所以大家一定要充分理解单例模式的线程安全的问题，就这几种模式花点时间，认真学透，面试中遇到任何关于单例模式的问题你都不会害怕是吧。

如果发现博客中有任何问题或者您还有什么疑问欢迎留言，您的支持是我前进的动力

如果本篇博客对你有帮助请赞一个或者留个言呗

转载请注明出处：<http://blog.csdn.net/dmk877/article/details/50311791>

参考博客：

<https://segmentfault.com/q/1010000003732558>

<http://tianweili.github.io/blog/2015/03/02/singleton-pattern/>

文章知识点与官方知识档案匹配，可进一步学习相关知识

Java技能树 首页 概览 116158 人正在系统学习中

显示推荐内容



奋斗之路

关注



153



341



41



专栏目录