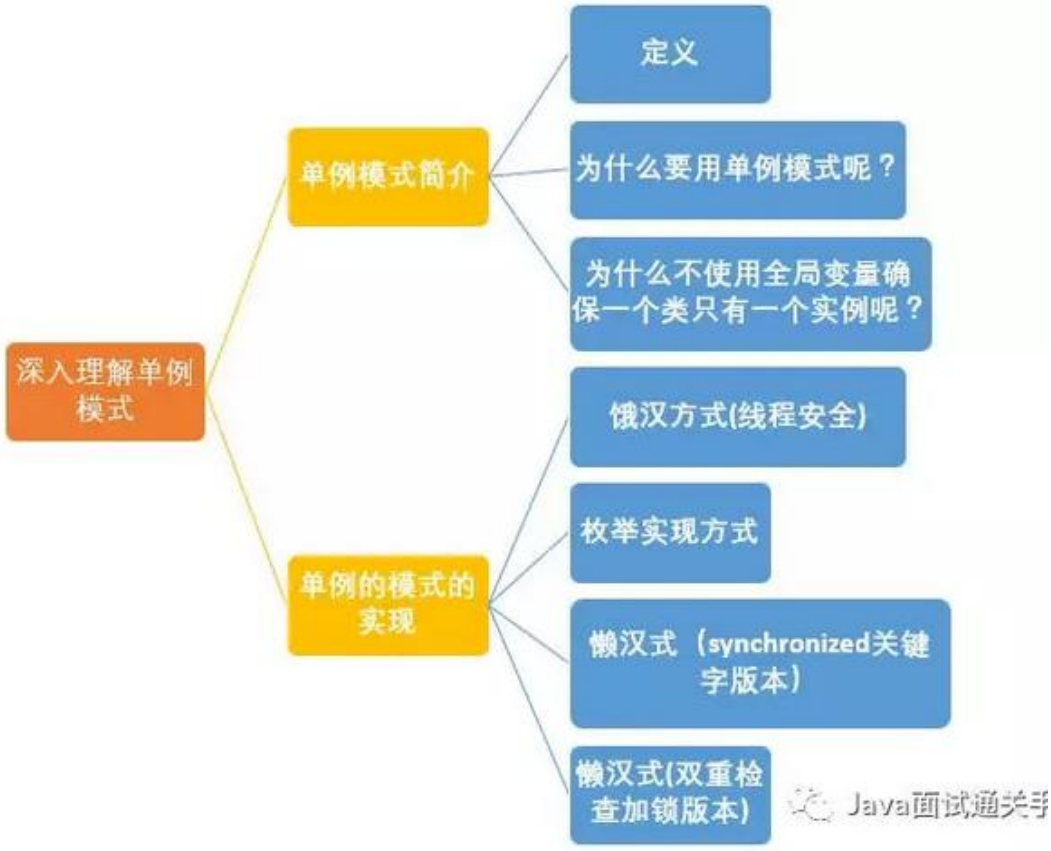
设计模式：单例模式



1 单例模式简介

1.1 定义

保证一个类仅有一个实例，并提供一个访问它的全局访问点

1.2 为什么要用单例模式呢？

在我们的系统中，有一些对象其实我们只需要一个，比如说：线程池、缓存、对话框、注册表、日志对象、充当打印机、显卡等设备驱动程序的对象。事实上，这一类对象只能有一个实例，如果制造出多个实例就可能会导致一些问题的产生，比如：程序的行为异常、资源使用过量、或者不一致性的结果。

简单来说使用单例模式可以带来下面几个好处:

- **对于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级对象而言，是非常可观的一笔系统开销；**
- **由于 new 操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻 GC 压力，缩短 GC 停顿时间。**

1.3 为什么不使用全局变量确保一个类只有一个实例呢？

我们知道全局变量分为静态变量和实例变量，静态变量也可以保证该类的实例只存在一个。 只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。

但是，如果说这个对象非常消耗资源，而且程序某次的执行中一直没用，这样就造成了资源的浪费。**利用单例模式的话，我们就可以实现在需要使用时才创建对象，这样就避免了不必要的资源浪费。** 不仅仅是因为这个原因，在程序中我们要尽量避免全局变量的使用，大量使用全局变量给程序的调试、维护等带来困难。

2 单例的模式的实现

通常单例模式在 Java 语言中，有两种构建方式：

- **饿汉方式**。指全局的单例实例在类装载时构建
- **懒汉方式**。指全局的单例实例在第一次被使用时构建。

不管是那种创建方式，它们通常都存在下面几点相似处：

- 单例类必须要有一个 private 访问级别的构造函数，只有这样，才能确保单例不会在系统中的其他代码内被实例化;
- instance 成员变量和 uniqueInstance 方法必须是 static 的。

2.1 饿汉方式(线程安全)

```
public class Singleton {
    //在静态初始化器中创建单例实例，这段代码保证了线程安全
     private static Singleton uniqueInstance = new Singleton();
     private Singleton(){}
     public static Singleton getInstance(){
         return uniqueInstance;
     }
}
```

所谓 **“饿汉方式”** 就是说 JVM 在加载这个类时就马上创建此唯一的单例实例，不管你用不用，先创建了再说，如果一直没有被使用，便浪费了空间，典型的空间换时间，每次调用的时候，就不需要再判断，节省了运行时间。

2.2 懒汉式（非线程安全和 synchronized 关键字线程安全版本 ）

```
public class Singleton {
      private static Singleton uniqueInstance;
      private Singleton (){
      }
      //没有加入synchronized关键字的版本是线程不安全的
      public static Singleton getInstance() {
          //判断当前单例是否已经存在，若存在则返回，不存在则再建立单例
          if (uniqueInstance == null) {
              uniqueInstance = new Singleton();
          }
          return uniqueInstance;
      }
 }
```

```
public static synchronized Singleton getInstance() {
    if (instance == null) {
        uniqueInstance = new Singleton();
    }
    return uniqueInstance;
}
```

所谓 **“饿汉方式”** 就是说单例实例在第一次被使用时构建，而不是在 JVM 在加载这个类时就马上创建此唯一的单例实例。

但是上面这种方式很明显是线程不安全的，如果多个线程同时访问 getInstance()方法时就会出现问题。如果想要保证线程安全，一种比较常见的方式就是在 getInstance() 方法前加上 synchronized 关键字，如下：

我们知道 synchronized 关键字偏重量级锁。虽然在 JavaSE1.6 之后 synchronized 关键字进行了主要包括：为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升。

但是在程序中每次使用 getInstance() 都要经过 synchronized 加锁这一层，这难免会增加 getInstance()的方法的时间消费，而且还可能会发生阻塞。我们下面介绍到的 **双重检查加锁版本** 就是为了解决这个问题而存在的。

2.3 懒汉式(双重检查加锁版本)

利用双重检查加锁（double-checked locking），首先检查是否实例已经创建，如果尚未创建，“才”进行同步。这样以来，只有一次同步，这正是我们想要的效果。

```
public class Singleton {

    //volatile保证，当uniqueInstance变量被初始化成Singleton实例时，多个线程可以正确处理
    uniqueInstance变量
     private volatile static Singleton uniqueInstance;
     private Singleton() {
     }
     public static Singleton getInstance() {
        //检查实例，如果不存在，就进入同步代码块
         if (uniqueInstance == null) {
             //只有第一次才彻底执行这里的代码
             synchronized(Singleton.class) {
                //进入同步代码块后，再检查一次，如果仍是null，才创建实例
                 if (uniqueInstance == null) {
                     uniqueInstance = new Singleton();
                 }
             }
         }
         return uniqueInstance;
     }
}
```

很明显，这种方式相比于使用 synchronized 关键字的方法，可以大大减少 getInstance() 的时间消费。

一般用懒汉模式的双重加锁版本还挺多的

2.4 其他方式（枚举）

除了上面说的几种创建方式之外，还有挺多种其他的创建方式这里稍微多提一点使用枚举的方式，其他创建方式我们就不管了，没有什么实质性的作用。

不叙述