http://wuchong.me/blog/2014/08/28/how-to-correctly-write-singleton-pattern/

单例模式算是设计模式中最容易理解，也是最容易手写代码的模式了吧。但是其中的坑却不少，所以也常作为面试题来考。本文主要对几种单例写法的整理，并分析其优缺点。很多都是一些老生常谈的问题，但如果你不知道如何创建一个线程安全的单例，不知道什么是双检锁，那这篇文章可能会帮助到你。

懒汉式，线程不安全

当被问到要实现一个单例模式时，很多人的第一反应是写出如下的代码，包括教科书上也是这样教我们的。

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

这段代码简单明了，而且使用了懒加载模式，但是却存在致命的问题。当有多个线程并行调用 getInstance() 的时候，就会创建多个实例。也就是说在多线程下不能正常工作。

懒汉式，线程安全

为了解决上面的问题，最简单的方法是将整个 getInstance() 方法设为同步（synchronized）。

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

虽然做到了线程安全，并且解决了多实例的问题，但是它并不高效。因为在任何时候只能有一个线程调用 getInstance() 方法。但是同步操作只需要在第一次调用时才被需要，即第一次创建单例实例对象时。这就引出了双重检验锁。

双重检验锁

双重检验锁模式（double checked locking pattern），是一种使用同步块加锁的方法。程序员称其为双重检查锁，因为会有两次检查 `instance == null`，一次是在同步块外，一次是在同步块内。为什么在同步块内还要再检验一次？因为可能会有多个线程一起进入同步块外的 if，如果在同步块内不进行二次检验的话就会生成多个实例了。

```
public static Singleton getSingleton() {
   if (instance == null) {                    //Single Checked
      synchronized (Singleton.class) {
         if (instance == null) {              //Double Checked
            instance = new Singleton();
         }
      }
   }
   return instance;
}
```

这段代码看起来很完美，很可惜，它是有问题。主要在于`instance = new Singleton()`这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情。

1. 给 instance 分配内存
2. 调用 Singleton 的构造函数来初始化成员变量
3. 将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

我们只需要将 instance 变量声明成 volatile 就可以了。

```
public class Singleton {
   private volatile static Singleton instance; //声明成 volatile
```

```
    private Singleton (){}
    public static Singleton getSingleton() {
        //如果有thread1,thread2同时都进入了第一个if块,只有一个线程能进入synchronzied块,
                //若thread1进入synchronized块,
                //thread1第二个if发现instance==null,则new Singleton(),thread1时间戳到了结束.
                //thread2进入synchronized块,第二个if时发现instance!=null,则直接退出后return
instance
                if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

有些人认为使用 volatile 的原因是可见性，也就是可以保证线程在本地不会存有 instance 的副本，每次都是去主内存中读取。但其实是不对的。使用 volatile 的主要原因是其另一个特性：禁止指令重排序优化。也就是说，在 volatile 变量的赋值操作后面会有一个内存屏障（生成的汇编代码上），读操作不会被重排序到内存屏障之前。比如上面的例子，取操作必须在执行完 1-2-3 之后或者 1-3-2 之后，不存在执行到 1-3 然后取到值的情况。从「先行发生原则」的角度理解的话，就是对于一个 volatile 变量的写操作都先行发生于后面对这个变量的读操作（这里的“后面”是时间上的先后顺序）。

但是特别注意在 Java 5 以前的版本使用了 volatile 的双检锁还是有问题的。其原因是 Java 5 以前的 JMM （Java 内存模型）是存在缺陷的，即时将变量声明成 volatile 也不能完全避免重排序，主要是 volatile 变量前后的代码仍然存在重排序问题。这个 volatile 屏蔽重排序的问题在 Java 5 中才得以修复，所以在这之后才可以放心使用 volatile。

相信你不会喜欢这种复杂又隐含问题的方式，当然我们有更好的实现线程安全的单例模式的办法。

饿汉式 static final field

这种方法非常简单，因为单例的实例被声明成 static 和 final 变量了，在第一次加载类到内存中时就会初始化，所以创建实例本身是线程安全的。

```
public class Singleton{
```

```
    //类加载时就初始化
    private static final Singleton instance = new Singleton();

    private Singleton(){}
    public static Singleton getInstance(){
        return instance;
    }
}
```

这种写法如果完美的话，就没必要在啰嗦那么多双检锁的问题了。缺点是它不是一种懒加载模式（lazy initialization），单例会在加载类后一开始就被初始化，即使客户端没有调用 getInstance()方法。饿汉式的创建方式在一些场景中将无法使用：譬如 Singleton 实例的创建是依赖参数或者配置文件的，在 getInstance() 之前必须调用某个方法设置参数给它，那样这种单例写法就无法使用了。

静态内部类 static nested class

我比较倾向于使用静态内部类的方法，这种方法也是《Effective Java》上所推荐的。

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

这种写法仍然使用JVM本身机制保证了线程安全问题；由于 SingletonHolder 是私有的，除了 getInstance() 之外没有办法访问它，因此它是懒汉式的；同时读取实例的时候不会进行同步，没有性能缺陷；也不依赖 JDK 版本。

枚举 Enum实现单例

http://blog.csdn.net/yy254117440/article/details/52305175

引言

单例模式比较常见的实现方法有懒汉模式，DCL模式公有静态成员等，从Java 1.5版本起，单元素枚举实现单例模式成为最佳的方法。

Java枚举

基本用法

枚举的用法比较多，本文主要旨在介绍利用枚举实现单例模式的原理，所以这里也主要介绍一些相关的基础内容。

首先，枚举类似类，一个枚举可以拥有成员变量，成员方法，构造方法。先来看枚举最基本的用法：

```
enum Type{
    A,B,C,D;
}
```

创建enum时，编译器会自动为我们生成一个继承自Java.lang.Enum的类，我们上面的enum可以简单看作：

```
class Type extends Enum{
    public static final Type A;
    public static final Type B;
    ...
}
```

对于上面的例子，我们可以把Type看作一个类，而把A，B，C，D看作类的Type的实例。

当然，这个构建实例的过程不是我们做的，一个enum的构造方法限制是private的，也就是不允许我们调用。

“类”方法和“实例”方法

上面说到，我们可以把Type看作一个类，而把A，B。。。看作Type的一个实例。同样，在enum中，我们可以定义类和实例的变量以及方法。看下面的代码：

```
enum Type{
    A,B,C,D;

    static int value;
    public static int getValue() {
        return value;
    }

    String type;
    public String getType() {
        return type;
    }
```

```
}
```

在原有的基础上，添加了类方法(static方法)和实例方法。我们把Type看做一个类，那么enum中静态的域和方法，都可以视作类方法。和我们调用普通的静态方法一样，这里调用类方法也是通过 Type.getValue()即可调用，访问类属性也是通过Type.value即可访问。

下面的是实例方法，也就是每个实例才能调用的方法。那么实例是什么呢？没错，就是A，B，C，D。所以我们调用实例方法，也就通过 Type.A.getType()来调用就可以了。

最后，对于某个实例而言，还可以实现自己的匿名内部类。再看下下面的代码：

```
enum Type{
A{
   public String getType() {
      return "I will not tell you";
   }
},B,C,D;
static int value;

public static int getValue() {
   return value;
}

String type;
public String getType() {
   return type;
 }
}
```

这里，A实例后面的{...}就是属于A的匿名内部类，可以通过覆盖原本的方法，实现属于自己的定制。

除此之外，我们还可以添加抽象方法在enum中，强制ABCD都实现各自的处理逻辑：

```
enum Type{
   A{
      public String getType() {
         return "A";
      }
   },B {
      @Override
      public String getType() {
```

```
            return "B";
        }
    },C {
        @Override
        public String getType() {
            return "C";
        }
    },D {
        @Override
        public String getType() {
            return "D";
        }
    };
    public abstract String getType();
}
```

枚举单例

有了上面的基础，我们可以来看一下枚举单例的实现方法：

```
class Resource{
}

public enum SomeThing {
    INSTANCE;
    private Resource instance;
    SomeThing() {
        instance = new Resource();
    }
    public Resource getInstance() {
        return instance;
    }
}
```

上面的类Resource是我们要应用单例模式的资源，具体可以表现为网络连接，**数据库**连接，线程池等等。

获取资源的方式很简单，只要 SomeThing.INSTANCE.getInstance() 即可获得所要实例。下面我们来看看单例是如何被保证的：

首先，在枚举中我们明确了构造方法限制为私有，在我们访问枚举实例时会执行构造

方法，同时每个枚举实例都是static final类型的，也就表明只能被实例化一次。在调用构造方法时，我们的单例被实例化。
也就是说，因为enum中的实例被保证只会被实例化一次，所以我们的INSTANCE也被保证实例化一次。
最后借用《Effective Java》一书中的话，

> 单元素的枚举类型已经成为实现Singleton的最佳方法。

总结

一般来说，单例模式有五种写法：懒汉、饿汉、双重检验锁、静态内部类、枚举。上述所说都是线程安全的实现，文章开头给出的第一种方法不算正确的写法。
就我个人而言，一般情况下直接使用饿汉式就好了，如果明确要求要懒加载（lazy initialization）会倾向于使用静态内部类，如果涉及到反序列化创建对象时会试着使用枚举的方式来实现单例。