Java：单例模式的七种写法

第一种（懒汉，线程不安全）：

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

这种写法lazy loading很明显，但是致命的是在多线程不能正常工作。

第二种（懒汉，线程安全）：

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

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

第三种（饿汉）：

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

这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance显然没有达到lazy loading的效果。

第四种（饿汉，变种）：

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

表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。

第五种（静态内部类）：

```
1 public class Singleton {
2     private static class SingletonHolder {
3     private static final Singleton INSTANCE = new Singleton();
4     }
5     private Singleton (){}
6     public static final Singleton getInstance() {
7         return SingletonHolder.INSTANCE;
8     }
9 }
```

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder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SingletonHolder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Singleton类加载时就实例化，因为我不能确保Singleton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，这种方式相比第三和第四种方式就显得很合理。

第六种（枚举）：

```
1 public enum Singleton {
2     INSTANCE;
3     public void whateverMethod() {
4     }
5 }
```

这种方式是Effective Java作者Josh Bloch 提倡的方式，它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。

第七种（双重校验锁）：

```
1 public class Singleton {
2     private volatile static Singleton singleton;
3     private Singleton (){}
4     public static Singleton getSingleton() {
5     if (singleton == null) {
6         synchronized (Singleton.class) {
7         if (singleton == null) {
```

```
 8              singleton = new Singleton();
 9          }
10          }
11      }
12      return singleton;
13      }
14  }
```