单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

首先我们写一个简单的单例类：

[java] view plaincopy

```java
public class Singleton {

    /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 */
    private static Singleton instance = null;

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 静态工程方法，创建实例 */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
    public Object readResolve() {
        return instance;
    }
}
```

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

[java] view plaincopy

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

但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

[java] view plaincopy

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

似乎解决了之前提到的问题，将synchronized关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在Java指令中创建对象和赋值操作是分开进行的，也就是说instance = new Singleton();语句是分两步执行的。但是JVM并不保证这两个操作的先后顺序，也就是说有可能JVM会为新的Singleton实例分配空间，然后直接赋值给instance成员，然后再去初始化这个Singleton实例。这样就可能出错了，我们以A、B两个线程为例：

a>A、B线程同时进入了第一个if判断

b>A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton();

c>由于JVM内部的优化机制，JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance成员（注意此时JVM没有开始初始化这个实例），然后A离开了synchronized块。

d>B进入synchronized块，由于instance此时不是null，因此它马上离开了synchronized块并将结果返回给调用该方法的程序。

e>此时B线程打算使用Singleton实例，却发现它没有被初始化，于是错误发生了。

所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：

[java] view plaincopy

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

实际情况是，单例模式使用内部类来维护单例的实现，JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕，这样我们就不用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式：

[java] view plaincopy

```
    public class Singleton {

        /* 私有构造方法，防止被实例化 */
        private Singleton() {
        }
```

```
    /* 此处使用一个内部类来维护单例 */
    private static class SingletonFactory {
        private static Singleton instance = new Singleton();
    }

    /* 获取实例 */
    public static Singleton getInstance() {
        return SingletonFactory.instance;
    }

    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
    public Object readResolve() {
        return getInstance();
    }
}
```

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加synchronized关键字，也是可以的：

[java] view plaincopy

```
public class SingletonTest {

    private static SingletonTest instance = null;

    private SingletonTest() {
    }

    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new SingletonTest();
        }
    }
```

```
    public static SingletonTest getInstance() {
        if (instance == null) {
            syncInit();
        }
        return instance;
    }
}
```

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。

补充：采用"影子实例"的办法为单例对象的属性同步更新

[java] view plaincopy

```
public class SingletonTest {

    private static SingletonTest instance = null;
    private Vector properties = null;

    public Vector getProperties() {
        return properties;
    }

    private SingletonTest() {
    }

    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new SingletonTest();
        }
    }

    public static SingletonTest getInstance() {
        if (instance == null) {
```

```
                syncInit();
            }
            return instance;
        }

        public void updateProperties() {
            SingletonTest shadow = new SingletonTest();
            properties = shadow.getProperties();
        }
    }
```

通过单例模式的学习告诉我们：

1、单例模式理解起来简单，但是具体实现起来还是有一定的难度。

2、synchronized关键字锁定的是对象，在用的时候，一定要在恰当的地方使用（注意需要使用锁的对象和过程，可能有的时候并不是整个对象及整个过程都需要锁）。

到这儿，单例模式基本已经讲完了，结尾处，笔者突然想到另一个问题，就是采用类的静态方法，实现单例模式的效果，也是可行的，此处二者有什么不同？

首先，静态类不能实现接口。（从类的角度说是可以的，但是那样就破坏了静态了。因为接口中不允许有static修饰的方法，所以即使实现了也是非静态的）

其次，单例可以被延迟初始化，静态类一般在第一次加载是初始化。之所以延迟加载，是因为有些类比较庞大，所以延迟加载有助于提升性能。

再次，单例类可以被继承，他的方法可以被覆写。但是静态类内部方法都是static，无法被覆写。

最后一点，单例类比较灵活，毕竟从实现上只是一个普通的Java类，只要满足单例的基本需求，你可以在里面随心所欲的实现一些其它功能，但是静态类不行。从上面这些概括中，基本可以看出二者的区别，但是，从另一方面讲，我们上面最后实现的那个单例模式，内部就是用一个静态类来实现的，所以，二者有很大的关联，只是我们考虑问题的层面不同罢了。两种思想的结合，才能造就出完美的解决方案，就像HashMap采用数组+链表来实现一样，其实生活中很多事情都是这样，单用不同的方法来处理问题，总是有优点也有缺点，最完美的方法是，结合各个方法的优点，才能最好的解决问题！