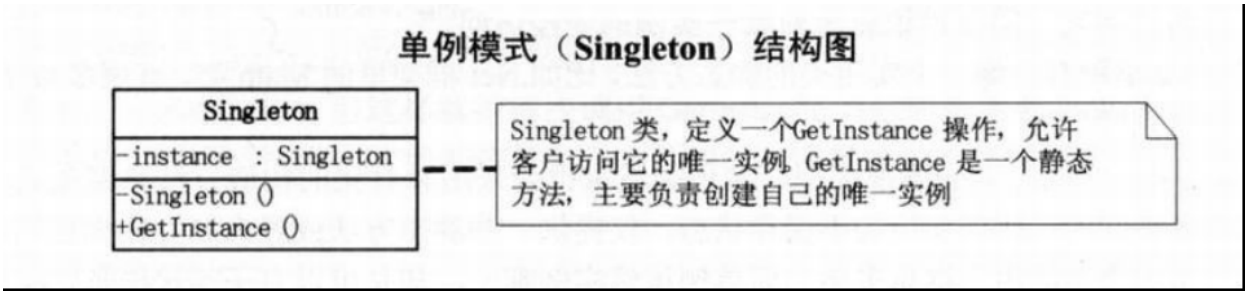
一、什么是单例模式

单例模式是一种常用的软件设计模式，其定义是单例对象的类只能允许一个实例存在。

许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。



单例的实现主要是通过以下两个步骤：

- 将该类的构造方法定义为私有方法，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；
- 在该类内提供一个静态方法，当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

二、单例模式的应用场景

举一个小例子，在我们的windows桌面上，我们打开了一个回收站，当我们试图再次打开一个新的回收站时，Windows系统并不会为你弹出一个新的回收站窗口。，也就是说在整个系统运行的过程中，系统只维护一个回收站的实例。这就是一个典型的单例模式运用。

继续说回收站，我们在实际使用中并不存在需要同时打开两个回收站窗口的必要性。假如我每次创建回收站时都需要消耗大量的资源，而每个回收站之间资源是共享的，那么在没有必要多次重复创建该实例的情况下，创建了多个实例，这样做就会给系统造成不必要的负担，造成资源浪费。

再举一个例子，网站的计数器，一般也是采用单例模式实现，如果你存在多个计数器，每一个用户的访问都刷新计数器的值，这样的话你的实计数的值是难以同步的。但是如果采用单例模式实现就不会存在这样的问题，而且还可以避免线程安全问题。同样多线程的线程池的设计一般也是采用单例模式，这是由于线程池需要方便对池中的线程进行控制

同样，对于一些应用程序的日志应用，或者web开发中读取配置文件都适合使用单例模式，如HttpApplication 就是单例的典型应用。

从上述的例子中我们可以总结出适合使用单例模式的场景和优缺点：

适用场景：

- 需要生成唯一序列的环境
- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
- 方便资源相互通信的环境

三、单例模式的优缺点

优点：

- 在内存中只有一个对象，节省内存空间；
- 避免频繁的创建销毁对象，可以提高性能；
- 避免对共享资源的多重占用，简化访问；
- 为整个系统提供一个全局访问点。

缺点：

- 不适用于变化频繁的对象；
- 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；
- 如果实例化的对象长时间不被利用，系统会认为该对象是垃圾而被回收，这可能会导致对象状态的丢失；

四、单例模式的实现

1.饿汉式

```
// 饿汉式单例
public class Singleton1 {

    // 指向自己实例的私有静态引用，主动创建
    private static Singleton1 singleton1 = new Singleton1();

    // 私有的构造方法
    private Singleton1(){}
```

```
    // 以自己实例为返回值的静态的公有方法，静态工厂方法
    public static Singleton1 getSingleton1(){
        return singleton1;
    }
}
```

我们知道，类加载的方式是按需加载，且加载一次。。因此，在上述单例类被加载时，就会实例化一个对象并交给自己的引用，供系统使用；而且，由于这个类在整个生命周期中只会被加载一次，因此只会创建一个实例，即能够充分保证单例。

优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。
缺点：在类装载的时候就完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费。

2.懒汉式

```
// 懒汉式单例
public class Singleton2 {

    // 指向自己实例的私有静态引用
    private static Singleton2 singleton2;

    // 私有的构造方法
    private Singleton2(){}

    // 以自己实例为返回值的静态的公有方法，静态工厂方法
    public static Singleton2 getSingleton2(){
        // 被动创建，在真正需要使用时才去创建
        if (singleton2 == null) {
            singleton2 = new Singleton2();
        }
        return singleton2;
    }
}
```

我们从懒汉式单例可以看到，单例实例被延迟加载，即只有在真正使用的时候才会实例化一个对象并交给自己的引用。

这种写法起到了Lazy Loading的效果，但是只能在单线程下使用。如果在多线程下，一个线程进入了if (singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。所以在多线程环境下不可使用这种方式。

3.双重加锁机制

```
public class DoubleCheck {

    private static volatile DoubleCheck doubleCheck;

    private DoubleCheck() {}

    public static DoubleCheck getInstance() {
        if (doubleCheck == null) {
            synchronized (DoubleCheck.class) {
                if (doubleCheck == null) {
                    doubleCheck = new DoubleCheck();
                }
            }
        }
        return doubleCheck;
    }
}
```

4.静态内部类初始化

```
public class StaticInnerClass {

    private StaticInnerClass() {}

    // 外部类装载的时候，外部类不会装载，在调用getInstance装载内部类的时候，才会装载内部类，装载类的时候是线程安全的
    private static class InnerClass {
        private static final StaticInnerClass INSTANCE = new StaticInnerClass();
    }

    public static StaticInnerClass getInstance() {
        return InnerClass.INSTANCE;
    }

}
```

当然，单例模式的实现方法还有很多。但是，这四种是比较经典的实现，也是我们应该掌握的几种实现方式。

从这四种实现中，我们可以总结出，要想实现效率高的线程安全的单例，我们必须注意以下两点：

- 尽量减少同步块的作用域；

- 尽量使用细粒度的锁。