单例模式

1.单例模式

单例模式是应用最广的模式之一，也可能是很多初级工程师唯一会使用的设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个全局对象，这样有利于我们协调系统整体的行为。

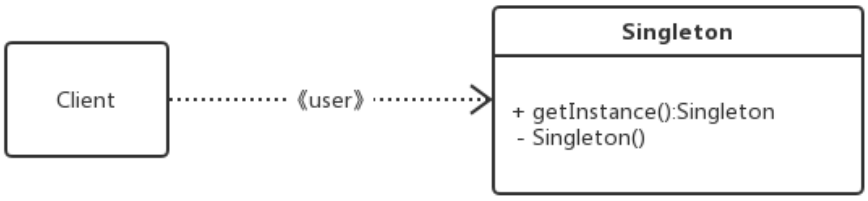
2.单例模式的定义

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

3.单例模式的使用场景

确保某个类有且只有一个对象的场景，避免产生多个对象消耗过多资源，或者某种类型的对象只应该有且只有一个。例如，创建一个对象需要消耗的资源过多，如要访问IO和数据库等资源，这时就要考虑使用单例模式。

4.单例模式UML类图



角色介绍：

- Client：高层客户端；
- Singleton：单例类。

实现单例模式主要有如下几个关键点：

1. 构造函数不对外开放，一般为Private;
2. 通过一个静态类方法或者枚举返回单例类对象；
3. 确保单例类的对象有且只有一个，尤其是在多线程环境下；

4. 确保单例类对象在反序列化时不会重新构造对象。

通过将单例类的构造函数私有化，使得客户端代码不能通过new的形式手动构造单例类的对象。单例类会暴露一个公有静态方法，客户端需要调用这个静态方法获取到单例类的唯一对象，在获取这个单例对象的过程中需要确保线程安全，即在多线程环境下构造单例类的对象也是有且只有一个，这也是单例模式实现中比较困难的地方。

5.单例模式的简单实现

单例模式是设计模式中比较简单的，只有一个单例类，没有其它的层次结构与抽象。该模式需要确保该类只能生成一个对象，通常是该类需要消耗较多的资源或者没有多个实例的情况下。例如，一个公司只有一个CEO、一个应用只有一个Application对象等。下面以公司里的CEO为例来简单演示一下，一个公司可以有几个VP、无数个员工，但是CEO只有一个，代码如下：

```
//普通员工
public class Staff {
        public void work(){
                //干活
        }
}

//副总
public class VP extends Staff {
        public void work() {
                //管理下面的经理
        }
}

//CEO,饿汉单例模式
public class CEO extends Staff {

        private static final CEO mCeo=new CEO();
        //构造函数私有
        private CEO(){

        }

        //公有的静态函数，对外暴露获取单例对象的接口
        public static CEO getCeo(){
                return mCeo;
        }

        public void work() {
                //管理VP
        }
}
//公司类
public class Company {

        private List<Staff> allStaff=new ArrayList();
        public void addStaff(Staff per){
                allStaff.add(per);
        }
        public void showAllStaffs(){
```

```
                for (Staff per:allStaff) {
                        System.out.println("Obj: "+per.toString());

                }
        }
}

```

测试类

```
public class Test {
        public static void main(String[] args) {
                Company cp = new Company();
                // CEO对象只能通过getCeo函数获取
                Staff ceo1 = CEO.getCeo();
                Staff ceo2 = CEO.getCeo();

                cp.addStaff(ceo1);
                cp.addStaff(ceo2);

                // 通过new 创建VP对象
                Staff vp1 = new VP();
                Staff vp2 = new VP();
                // 通过new创建Staff对象
                Staff staff1 = new Staff();
                Staff staff2 = new Staff();

                cp.addStaff(vp1);
                cp.addStaff(vp2);
                cp.addStaff(staff1);
                cp.addStaff(staff2);
                cp.showAllStaffs();
        }
}


```

输出结果如下：

```
Obj: Singleton.CEO@15db9742
Obj: Singleton.CEO@15db9742
Obj: Singleton.VP@6d06d69c
Obj: Singleton.VP@7852e922
Obj: Singleton.Staff@4e25154f
Obj: Singleton.Staff@70dea4e

```

从上述的代码中可以看出，CEO类不能通过new的形式构造对象，只能通过CEO.getCEO()函数来获取，而这个CEO对象是静态函数，并且在声明的时候已经初始化了，这就保证了CEO对象的唯一性。从输出结果中发现，CEO两次输出的CEO对象都是一样的，而VP、Staff等类型的对象都是不同的。这个实现的核心在于 **将CEO类的构造方法私有化，使得外部程序不能通过构造函数来构造CEO对象，而CEO类通过一个静态方法返回一个静态对象。**

> 1 懒汉模式

懒汉模式是声明一个静态对象，并且在用户第一次调用getInstance时进行初始化，而上述的恶汉模式是在声明静态对象时就已经初始化。懒汉单例模式实现如下：

```
public class Singleton {
        private static Singleton instance;

        private Singleton() {
        }

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

```

getInstance()方法中添加synchronized关键字，也就是getInstance()是一个同步方法，这就是上面所说的在多线程情况下保证单例对象唯一性的手段。细想一下，就会发现一个问题，即使instance已经被初始化（第一次调用时就会初始化instance），每次调用getInstance()方法都会进行同步，这样会不必要的资源，这也是懒汉单例模式存在的最大问题。
最后总结一下，**懒汉单例模式的有点是单例只有在使用时才会被实例化，在一定程度上节约了资源；缺点是第一次加载时需要及时进行实例化，反应稍慢，最大问题是每次调用getInstance()都进行同步，造成不必要的同步开销。** 这种模式一般不建议使用。

2. Double Check Lock(DCL)实现单例

DCL方式实现单例模式的优点是既能够在需要时才初始化单例，又能够保证线程安全，且单例对象初始化后调用getInstance()不能进行同步锁。代码如下所示：

```

public class Singleton {
        private static Singleton instance = null;

        private Singleton() {
        }

        public void doSomething() {
                System.out.println("do sth.");
        }

        public static Singleton getInSingleton() {
                if (instance == null) {
                        synchronized (Singleton.class) {
                                if (instance == null) {
                                        instance = new Singleton();

                                }
                        }
                }
                return instance;
```

```
        }
}
```

本程序的亮点自然都在getInstance方法上，可以看到getInstance方法中对instance进行了两次判空：第一层判断主要是为了避免不必要的同步，第二层的判断则是为了在null的情况下创建实例。这是什么意思呢?下面具体分析：
假设线程A执行到sInstance=new Singleton()语句，这里看起来是一句代码，但实际上它并不是一个原子操作，这句代码最终会被编译成多条汇编指令，它大致做了3件事情：
1.给Singleton的实例分配内存；
2.调用Singleton()的构造函数，初始化成员字段；
3.将sInstance对象指向分配的内存空间（此时sInstance就不是 null了）。
但是由于Java编译器允许处理器乱序执行，以及JDK1.5之前JMM中Cache、寄存器到主内存回写顺序的规定，上面的第二和第三的顺序是无法保证的，也就是说，执行顺序可能是1-2-3也可能是1-3-2.如果是后者，并且在3执行完毕后、2未执行之前，被切换到线程B上，这时候sInstance因为已经在线程A执行过第三点，sInstance已经是非空了，所以，线程B直接取走sInstance,在使用是就会报错，这就是DCL失效问题，而且这种难以跟踪难以重现的错误很可能会隐藏很久。
在JDK1.5之后，调整了JVM，具体化了volatile关键字，因此如果在JDK是1.5或之后的版本，只需要将sInstance的定义改为 **private volatile static Singleton sInstance=null** 就可以保证sInstance对象每次都是从主内存中读取，就可以使用DCL的写法来完成单例模式。当然，volatile或多或少也会影响到性能，但考虑到程序的正确性，牺牲这点性能还是值得的。
DCL的优点：资源利用率高，第一次执行getInstace时单例对象才会被实例化，效率高。缺点：第一次加载时反应稍慢，也由于Java内存模型的原因偶尔会失败。在高并发环境下也有一定的缺陷，虽然发生概率很小。DCL模式是使用最多的单例实现方式，它能够在需要时才是实例化单例对象，并且能够在绝大多数场景下保证单例对象的唯一性。

> 3.静态内部类单例模式

DCL虽然在一定程度上解决了资源消耗、多余的同步、线程安全等问题，但是，它还是在某些情况下出现失效的问题。建议使用如下的代码替代：

```
public class Singleton2 {

        public static Singleton2 getInSingleton2() {
                return SingletonHolder.instance;
        }

        /***
         * 静态内部类
         */
        private static class SingletonHolder {
                private static final Singleton2 instance = new Singleton2();
        }
}
```

当第一次加载Singleton类时并不会初始化sInstance，只有在第一次调用Singleton的getInstance方法时才会导致sInstance被初始化。因此，第一次调用getInstance方法会导致虚拟机加载SingletonHolder类，这种方式不仅能够确保线程安全，也能够保证代理对象的唯一性，同时也延迟了单例的实例化，所以这是推荐使用的单例模式实现方式。

6.总结

单例模式是运用频率很高的模式，但是，由于在客户端通常没有高并发的情况，因此，选择哪种实现方式并不会有太大的影响，即便如此，出于效率考虑DCL方式和静态内部类方式。

优点：

1.由于单例模式在内存中只有一个实例，减少了内存的开支，特别是一个对象需要频繁地创建、销毁时，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显。

2.由于单例模式只生成一个实例，所以，减少了系统的性能开销，当一个对象的产生需要比较多的资源时，如读取配置、生产其它依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后用永久驻留内存的方式来解决。

3.单例模式可以避免对资源的多重占用，例如一个写文件操作，由于只有一个实例存在内存中，避免了对同一个资源文件的同时写操作。

4.单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如，可以设计一个单例类，负责所有数据表的映射处理。

缺点：

1.单例模式一般没有接口，扩展很困难，若要扩展，除了修改代码基本上没有第二种途径可以实现。

2.单例对象如果持有Context，那么很容易引起内存泄漏，此时需要注意传递给单例的Context最好是Application Context。

————摘自《Android 源码设计模式解析与实战 第二章》

xyzso1z

原创文章 80　获赞 40　访问量 2万+