单例模式

单例模式常被用于一个类在系统中最多只允许存在一个实例的场合

多线程时的单例模式：

一.双重校验锁式

（分别在代码前后进行判空检验，避免多个有机会进入临界区的线程都创建对象，同时也避免了后来线程在先来线程创建对象后仍未退出临界区的情况下等待）

```
public class Singleton{
    private volatile static Singleton instance=null;
    private Singleton(){

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

1.sychronized中的Singleton.class换否换成this?

不能，因为getInstance()为静态方法，静态方法中不可以使用this，this代表当前对象的引用，而静态方法是依赖于类的，它优于对象而存在

2.singleton为什么要使用 volatile关键字?

- **保证共享变量在多线程下的可见性：**防止被实例化多次（这里由于synchronized锁的是Singleton.class对象，而不是singleton对象，所以synchronized只能保证Singleton.class对象的内存可见性，但并不能保证Singleton对象的内存可见性；这里用volatile声明Singleton，可以保证Singleton对象的内存可见性。(避免因为线程1 创建实例后还只存在自己线程的工作内存，未更新到主存。线程 2 判断对象为空，创建实例，从而存在多实例错误)
- **禁止指令重排序：**（实例化一个对象其实可以分为三个步骤：（1）分配内存空间。（2）初始化对象。（3）将内存空间的地址赋值给对应的引用。但是由于操作系统可以对指令进行重排序，所以上面的过程也可能会变成如下过程：（1）分配内存空间。（2）将内存空间的地址赋值给对应的引用。（3）初始化对象。如果是这个流程，多线程环境下就可能将一个未初始化的对象引用暴露出来，从而导致不可预料的结果（如题目的描述，这里就是因为 instance = new Singleton()；不是原子操作，编译器存在指令重排，从而存在线程1 创建实例后（初始化未完成），线程2判断对象不为空后对其操作，但实际对象仍为空，造成错误）。因此，为了防止这个过程的重排序，我们需要将变量设置为volatile类型的变量，volatile的禁止重排序保证了操作的有序性。

3.两次判空的原因：第二次是为了防止创建多个实例，第一次是为了降低 synchronized块被多个线程执行到的几率，减小锁的开销。

二.饿汉式单例

```
public class Singleton{
    private static Singleton instance=new Singleton();
    private Singleton(){

    }
```

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

单例模式的饿汉式，在定义自身类型的成员变量时就将其实例化，使得在Singleton单例类被系统加载时就已经被实例化出一个单例对象，从而一劳永逸地避免了线程安全的问题。

ClassLoader加载Singleton类时，饿汉式单例就被创建，虽然饿汉式单例是线程安全的，但也有其不足之处。饿汉式单例在类被加载时就创建单例对象并且长驻内存，不管你需不需要它；如果单例类占用的资源比较多，就会降低资源利用率以及程序的运行效率。

三.懒汉式单例

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

在需要的时候才创建单例对象，而不是随着软件系统的运行或者当类被加载器加载的时候就创建。

这样做相当于在方法上添加synchronized关键字，会影响程序效率，因为当有多个线程访问getInstance()方法时，多个线程必须有序地进入方法内，这样导致了多个线程需要耗费等待进入临界区（被所锁住的代码块）的时间，因此，提出了双重校验锁式

四.静态内部类

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

使用静态内部类很好地将懒汉式和饿汉式结合起来，即能实现延迟加载，保证系统性能，又能实现线程安全，但是当遇到序列化对象时结果还是多例

五.枚举单例

```
public class Singleton{
    public enum enumSingleton{
        INSTANCE;
        private Singleton singleton;
        private enumSingleton(){
```

```
            singleton=new Singleton();
        }
        public Singleton getInstnce(){
            return singleton;
        }
    }
    public static Singleton getInseance(){
        return enumSingleton.INSTANCE.getInstnce();
    }
}
```

使用枚举单例模式的好处：

- 实例的创建线程安全，确保单例
- 防止被反射创建多个实例
- 没有序列化的问题。

工厂模式

许多类型对象的创造需要一系列的步骤: 你可能需要计算或取得对象的初始设置，选择生成哪个子对象实例，或在生成你需要的对象之前必须先生成一些辅助功能的对象。 在这些情况，新对象的建立就是一个 “过程”，不仅是一个操作，像一部大机器中的一个齿轮传动。

一.简单工厂模式

一个抽象产品类，可以派生出多个具体产品类

一个简单工厂类可以创建多个具体产品类的实例

缺点：没有遵循开闭原则，当增加产品时，工厂类的业务逻辑也要进行修改

二.工厂方法模式

一个抽象产品类，可以派生出多个具体产品类

一个抽象工厂类，可以派生出多个具体工厂类
一个具体工厂类只能创建一个具体产品类的实例

三.抽象工厂模式

多个抽象产品类，每个抽象产品类可以派生出多个具体产品类
一个抽象工厂类，可以派生出多个具体工厂类
每个具体工厂类可以创建多个具体产品类的实例

抽象工厂模式是工厂方法模式的升级版本，他用来创建一组相关或者相互依赖的对象。他与工厂方法模式的区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则是针对的多个产品等级结构。

工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。

代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。

适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。

模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。

除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。