4，创建者模式

创建型模式的主要关注点是“怎样创建对象？”，它的主要特点是“将对象的创建与使用分离”。

这样可以降低系统的耦合度，使用者不需要关注对象的创建细节。

创建型模式分为：

- 单例模式
- 工厂方法模式
- 抽象工程模式
- 原型模式
- 建造者模式

4.1 单例设计模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

4.1.1 单例模式的结构

单例模式的主要有以下角色：

- 单例类。只能创建一个实例的类
- 访问类。使用单例类

4.1.2 单例模式的实现

> 单例设计模式分类两种：
>
> 饿汉式：类加载就会导致该单实例对象被创建
>
> 懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建

1. 饿汉式-方式1（静态变量方式）

```
/**
 * 饿汉式
 *       静态变量创建类的对象
 */
public class Singleton {
    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance = new Singleton();

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

说明：

该方式在成员位置声明Singleton类型的静态变量，并创建Singleton类的对象instance。instance对象是随着类的加载而创建的。如果该对象足够大的话，而一直没有使用就会造成内存的浪费。

2. 饿汉式-方式2（静态代码块方式）

```
/**
 * 恶汉式
 *       在静态代码块中创建该类对象
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance;

    static {
        instance = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return instance;
    }
}
```

说明：

该方式在成员位置声明Singleton类型的静态变量，而对象的创建是在静态代码块中，也是对着类的加载而创建。所以和饿汉式的方式1基本上一样，当然该方式也存在内存浪费问题。

3. 懒汉式-方式1（线程不安全）

```
/**
 * 懒汉式
 *  线程不安全
 */
public class Singleton {
    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance;

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {

        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

说明：

从上面代码我们可以看出该方式在成员位置声明Singleton类型的静态变量，并没有进行对象的赋值操作，那么什么时候赋值的呢？当调用getInstance()方法获取Singleton类的对象的时候才创建Singleton类的对象，这样就实现了懒加载的效果。但是，如果是多线程环境，会出现线程安全问题。

4. 懒汉式-方式2（线程安全）

```
/**
 * 懒汉式
 *  线程安全
 */
public class Singleton {
    //私有构造方法
    private Singleton() {}

    //在成员位置创建该类的对象
    private static Singleton instance;

    //对外提供静态方法获取该对象
    public static synchronized Singleton getInstance() {

        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

说明：

该方式也实现了懒加载效果，同时又解决了线程安全问题。但是在getInstance()方法上添加了synchronized关键字，导致该方法的执行效果特别低。从上面代码我们可以看出，其实就是在初始化instance的时候才会出现线程安全问题，一旦初始化完成就不存在了。

5. 懒汉式-方式3（双重检查锁）

再来讨论一下懒汉模式中加锁的问题，对于 `getInstance()` 方法来说，绝大部分的操作都是读操作，读操作是线程安全的，所以我们没必让每个线程必须持有锁才能调用该方法，我们需要调整加锁的时机。由此也产生了一种新的实现模式：双重检查锁模式

```
/**
 * 双重检查方式
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static Singleton instance;

   //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例
```

```
        if(instance == null) {
            synchronized (Singleton.class) {
                //抢到锁之后再次判断是否为null
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

双重检查锁模式是一种非常好的单例实现模式，解决了单例、性能、线程安全问题，上面的双重检测锁模式看上去完美无缺，其实是存在问题，在多线程的情况下，可能会出现空指针问题，出现问题的原因是JVM在实例化对象的时候会进行优化和指令重排序操作。

要解决双重检查锁模式带来空指针异常的问题，只需要使用 `volatile` 关键字， `volatile` 关键字可以保证可见性和有序性。

```
/**
 * 双重检查方式
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static volatile Singleton instance;

   //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实际
        if(instance == null) {
            synchronized (Singleton.class) {
                //抢到锁之后再次判断是否为空
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

小结:

添加 `volatile` 关键字之后的双重检查锁模式是一种比较好的单例实现模式，能够保证在多线程的情况下线程安全也不会有性能问题。

6. 懒汉式-方式4（静态内部类方式）

静态内部类单例模式中实例由内部类创建，由于 JVM 在加载外部类的过程中， 是不会加载静态内部类的， 只有内部类的属性/方法被调用时才会被加载， 并初始化其静态属性。静态属性由于被 `static` 修饰，保证只被实例化一次，并且严格保证实例化顺序。

```
/**
```

```
 * 静态内部类方式
 */
public class Singleton {

    //私有构造方法
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

说明：

第一次加载Singleton类时不会去初始化INSTANCE，只有第一次调用getInstance，虚拟机加载SingletonHolder

并初始化INSTANCE，这样不仅能确保线程安全，也能保证 Singleton 类的唯一性。

小结：

静态内部类单例模式是一种优秀的单例模式，是开源项目中比较常用的一种单例模式。在没有加任何锁的情况下，保证了多线程下的安全，并且没有任何性能影响和空间的浪费。

7. 枚举方式

枚举类实现单例模式是极力推荐的单例实现模式，因为枚举类型是线程安全的，并且只会装载一次，设计者充分的利用了枚举的这个特性来实现单例模式，枚举的写法非常简单，而且枚举类型是所用单例实现中唯一一种不会被破坏的单例实现模式。

```
/**
 * 枚举方式
 */
public enum Singleton {
    INSTANCE;
}
```

说明：

枚举方式属于恶汉式方式。

4.1.3 存在的问题

4.1.3.1 问题演示

破坏单例模式：

使上面定义的单例类（Singleton）可以创建多个对象，枚举方式除外。有两种方式，分别是序列化和反射。

- 序列化反序列化

  Singleton类：

```
public class Singleton implements Serializable {

    //私有构造方法
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

Test类：

```
public class Test {
    public static void main(String[] args) throws Exception {
        //往文件中写对象
        //writeObject2File();
        //从文件中读取对象
        Singleton s1 = readObjectFromFile();
        Singleton s2 = readObjectFromFile();

        //判断两个反序列化后的对象是否是同一个对象
        System.out.println(s1 == s2);
    }

    private static Singleton readObjectFromFile() throws Exception {
        //创建对象输入流对象
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("C:\\Users\\Think\\Desktop\\a.txt"));
        //第一个读取Singleton对象
        Singleton instance = (Singleton) ois.readObject();

        return instance;
    }

    public static void writeObject2File() throws Exception {
        //获取Singleton类的对象
        Singleton instance = Singleton.getInstance();
        //创建对象输出流
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("C:\\Users\\Think\\Desktop\\a.txt"));
        //将instance对象写出到文件中
        oos.writeObject(instance);
    }
}
```

> 上面代码运行结果是`false`，表明序列化和反序列化已经破坏了单例设计模式。

- 反射

Singleton类：

```
public class Singleton {
```

```
    //私有构造方法
    private Singleton() {}

    private static volatile Singleton instance;

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {

        if(instance != null) {
            return instance;
        }

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

Test类：

```
public class Test {
    public static void main(String[] args) throws Exception {
        //获取Singleton类的字节码对象
        Class clazz = Singleton.class;
        //获取Singleton类的私有无参构造方法对象
        Constructor constructor = clazz.getDeclaredConstructor();
        //取消访问检查
        constructor.setAccessible(true);

        //创建Singleton类的对象s1
        Singleton s1 = (Singleton) constructor.newInstance();
        //创建Singleton类的对象s2
        Singleton s2 = (Singleton) constructor.newInstance();

        //判断通过反射创建的两个Singleton对象是否是同一个对象
        System.out.println(s1 == s2);
    }
}
```

> 上面代码运行结果是`false`，表明序列化和反序列化已经破坏了单例设计模式

> 注意：枚举方式不会出现这两个问题。

4.1.3.2 问题的解决

- 序列化、反序列方式破坏单例模式的解决方法

  在`Singleton`类中添加`readResolve()`方法，在反序列化时被反射调用，如果定义了这个方法，就返回这个方法的值，如果没有定义，则返回新`new`出来的对象。

  Singleton类：

```java
public class Singleton implements Serializable {

    //私有构造方法
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }

    /**
     * 下面是为了解决序列化反序列化破解单例模式
     */
    private Object readResolve() {
        return SingletonHolder.INSTANCE;
    }
}
```

源码解析：

ObjectInputStream类

```java
public final Object readObject() throws IOException, ClassNotFoundException{
    ...
    // if nested read, passHandle contains handle of enclosing object
    int outerHandle = passHandle;
    try {
        Object obj = readObject0(false);//重点查看readObject0方法
    .....
}

private Object readObject0(boolean unshared) throws IOException {
    ...
    try {
        switch (tc) {
            ...
            case TC_OBJECT:
                return checkResolve(readOrdinaryObject(unshared));//重点查看readOrdinaryObject方法
            ...
        }
    } finally {
        depth--;
        bin.setBlockDataMode(oldMode);
    }
}

private Object readOrdinaryObject(boolean unshared) throws IOException
{
    ...
    //isInstantiable 返回true，执行 desc.newInstance()，通过反射创建新的单例类，
```

```
    obj = desc.isInstantiable() ? desc.newInstance() : null;
    ...
    // 在Singleton类中添加 readResolve 方法后 desc.hasReadResolveMethod() 方法执行结果为true
    if (obj != null && handles.lookupException(passHandle) == null && desc.hasReadResolveMethod()) {
        // 通过反射调用 Singleton 类中的 readResolve 方法，将返回值赋值给rep变量
        // 这样多次调用ObjectInputStream类中的readObject方法，继而就会调用我们定义的readResolve方法，所以返回的是同一个对象。
        Object rep = desc.invokeReadResolve(obj);
        ...
    }
    return obj;
}
```

- 反射方式破解单例的解决方法

```
public class Singleton {

    //私有构造方法
    private Singleton() {
        /*
            反射破解单例模式需要添加的代码
        */
        if(instance != null) {
            throw new RuntimeException();
        }
    }

    private static volatile Singleton instance;

    //对外提供静态方法获取该对象
    public static Singleton getInstance() {

        if(instance != null) {
            return instance;
        }

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

说明：

这种方式比较好理解。当通过反射方式调用构造方法进行创建创建时，直接抛异常。不运行此中操作。

4.1.4 JDK源码解析-Runtime类

Runtime类就是使用的单例设计模式。

1. 通过源代码查看使用的是哪儿种单例模式

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the <code>Runtime</code> object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}
    ...
}
```

从上面源代码中可以看出Runtime类使用的是恶汉式（静态属性）方式来实现单例模式的。

2. 使用Runtime类中的方法

```
public class RuntimeDemo {
    public static void main(String[] args) throws IOException {
        //获取Runtime类对象
        Runtime runtime = Runtime.getRuntime();

        //返回 Java 虚拟机中的内存总量。
        System.out.println(runtime.totalMemory());
        //返回 Java 虚拟机试图使用的最大内存量。
        System.out.println(runtime.maxMemory());

        //创建一个新的进程执行指定的字符串命令，返回进程对象
        Process process = runtime.exec("ipconfig");
        //获取命令执行后的结果，通过输入流获取
        InputStream inputStream = process.getInputStream();
        byte[] arr = new byte[1024 * 1024* 100];
        int b = inputStream.read(arr);
        System.out.println(new String(arr,0,b,"gbk"));
    }
}
```