创建者模式

创建型模式的主要关注点时“怎么创建对象“，它的主要特点是”将对象的创建与使用分离“。这样可以减低系统的耦合度，使用者不需要关注对象的创建细节。

创建者模式分为

- 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

单例设计模式

单例模式是Java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对象的方式。可以直接访问，不需要实例化该类的对象。

单利模式的实现

饿汉式：类加载就会导致改单实例对象被创建

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被创建

饿汉式（如果对象太大，而一直没有使用，会造成内存浪费）：

- `饿汉式（静态成员变量）`
- `饿汉式（静态代码块）`

> 对于JDK1.2后的JVM HotSpot来说，判断对象可以回收需要经过可达性分析，由于单例对象被其类中的静态变量引用，所以JVM认为对象是可达的，不会被回收。另外，**对于JVM方法区回收，由堆中存在单例对象，所以单例类也不会被卸载，其静态变量引用也不会失效**。

懒汉式（存在线程不安全的情况）

- 方法加`synchronized` (对于绝大多数的操作都是读操作，读操作是线程安全的，所以没有必要让每个线程都持有所才去操作)
- `双重检查锁`

```
public class Singleton2 {

    private static Singleton2 singleton2;

    public Singleton2 getInstance() {
        if (Objects.isNull(singleton2)) {
            synchronized (Singleton2.class) {
                if (Objects.isNull(singleton2)) {
                    return new Singleton2();
                }
            }
        }
        return singleton2;
    }

    private Singleton2() {
    }
}
```

> *双重检查锁* 是一种非常好的单例实现模式，解决了单利、性能、线程安全问题，但是也存在问题，在多线程的情况下，可能会出现空指针的问题，出现问题的原因是JVM在实例化对象的时候可能会进行优化和指令重排序。所以要解决这个问题，需要用到`volatile`，保证可见性和有序性。

- *静态内部类*

```
public class Singleton3 {

    private static class SingletonHolder {
        private static final Singleton3 SINGLETON = new Singleton3();
    }

    public Singleton3 getInstance() {
        return SingletonHolder.SINGLETON;
    }
}
```

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/方法被调用的时候才会加载，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，并且保证实例化的顺序。

- *枚举* *(恶汉模式，存在内存浪费)*

  枚举类类型是线程安全的，并且只会加载一次，设计者充分利用枚举这个特性来实现单例。而且枚举是***所有单例实现中唯一不会被破坏的单例模式***

破坏单例模式

- 序列化

  解决：添加readResolve()方法

```
        }

        handles.finish(passHandle);

        if (obj != null &&
            handles.lookupException(passHandle) == null &&
            desc.hasReadResolveMethod())
        {
            Object rep = desc.invokeReadResolve(obj);
            if (unshared && rep.getClass().isArray()) {
                rep = cloneArray(rep);
            }
            if (rep != obj) {
                // Filter the replacement object
```

```
                    new Class<?>[] { ObjectInputStream.class },
                    Void.TYPE);
                readObjectNoDataMethod = getPrivateMethod(
                    cl, name: "readObjectNoData", argTypes: null, Void.TYPE);
                hasWriteObjectData = (writeObjectMethod != null);
            }
            domains = getProtectionDomains(cons, cl);
            writeReplaceMethod = getInheritableMethod(
                cl, name: "writeReplace", argTypes: null, Object.class);
            readResolveMethod = getInheritableMethod(
                cl, name: "readResolve", argTypes: null, Object.class);
            return null;
        }
    });
} else {
    suid = Long.valueOf( l: 0);
```

```
if (obj != null &&
    handles.lookupException(passHandle) == null &&
    desc.hasReadResolveMethod())
{
    Object rep = desc.invokeReadResolve(obj);
    if (unshared && rep.getClass().isArray()) {
        rep = cloneArray(rep);
    }
    if (rep != obj) {
        // Filter the replacement object
```

- 反射

```
public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException, InstantiationException, IllegalAcc
        Constructor<Singleton2> declaredConstructor = Singleton2.class.getDeclaredConstructor();
        // 取消访问检查
        declaredConstructor.setAccessible(true);
        Singleton2 s1 = declaredConstructor.newInstance();
        Singleton2 s2 = declaredConstructor.newInstance();
        System.out.println(s1 == s2);
    }
```

解决：添加一个标识符

```
public class Singleton2 {

    private static boolean flag = false;

    private static volatile Singleton2 singleton2;

    public Singleton2 getInstance() {
        if (Objects.isNull(singleton2)) {
            synchronized (Singleton2.class) {
                if (Objects.isNull(singleton2)) {
```

```
                    return new Singleton2();
                }
            }
        }
        return singleton2;
    }

    private Singleton2() {
        synchronized (this) {
            if (flag) {
                throw new RuntimeException("多实例");
            }
            flag = true;
        }
    }

    public static void main(String[] args) throws NoSuchMethodException, InvocationTargetException, InstantiationException, Illega
        Constructor<Singleton2> declaredConstructor = Singleton2.class.getDeclaredConstructor();
        // 取消访问检查
        declaredConstructor.setAccessible(true);
        Singleton2 s1 = declaredConstructor.newInstance();
        Singleton2 s2 = declaredConstructor.newInstance();
        System.out.println(s1 == s2);
    }
}
```

工厂模式

如果创建对象的时候用new，需要更换对象，则需要更改，这违背了软件设计的开闭原则。工厂模式最大的优点：***解耦***

- 简单工厂（不属于GOF，开发常用）
- 工厂方法
- 抽象工厂

简单工厂

不是一种设计模式，比较像编程习惯

优点：封装了创建对象的过程，可以通过参数直接获取对象。把对象的创建和业务逻辑层分开，更加容易扩展

缺点：还是需要修改工厂类的代码，违背了开闭原则

工厂方法

定义一个用户创建对象的接口，让子类决定实例化那个对象。工厂方法让实例化延迟其工厂子类

优点：

- 只需要知道工厂的名称就可以（通过构造方法或者set方法），无须知道具体的创建过程
- 如果后续需要添加新的类型，直接添加对应类和工厂类，不需要对原工厂修改。满足开闭原则

缺点：

- 每添加一个类型，就需要一个具体类和工厂类，增加系统的复杂度

抽象工厂类

是一种为访问类提供一个创建一组相关或互相依赖的接口，切访问类无需制定产品的具体类就能得到同族的不同等级的产品模式。抽象工厂是工厂方法的升级版本，工厂方法只能生产一个等级的产品，而抽象工厂可以生产多个等级的产品

优点：当一个产品族中多个对象被设计成一起工作时，它能保证客户端始终使用一个产品族中的对象

缺点：添加一个新实现，需要修改所有的工厂类

原型模式

用一个已经创建的实例作为原型，通过复制该原型对象来创建一个相同的新对象（BeanCopy）

原型模式克隆分为浅克隆和深克隆

- 浅克隆：创建一个新对象，新的对象属性和原来对象完成相同，对于非基本类型***属性***，内存地址是相同的
- 深克隆：创建一个新对象，属性也克隆，内存地址不相同（new）

建造者模式

将一个对象的构建与表示分离，使得同样的构建过程可以创建不同的表示

- 分离了部件的构造（Builder）和装配（Director）。从而可以构造出复杂的对象。这个模式适用于：构造出复杂的对象
- 由于实现了构建和装配的解耦。不同的构造器，相同的装配，可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用
- 建造者模式可以将部件和其组装过程分开，一步步创建一个复杂的对象。用户只需要制定复杂的对象类型就可以得到该对象，从而无需知道内部的构造细节。

```
public class Director {

    private Builder builder;

    public Director setBuilder(Builder builder) {
        this.builder = builder;
        return this;
    }

    public Report construct() {
        builder.buildScreening();
        builder.buildCommon();
        return builder.createReport();
    }

    public static abstract class Builder {

        public Report report = new Report();

        public abstract void buildScreening();

        public abstract void buildCommon();

        public abstract Report createReport();
    }

    public static class kindergartenBuilder extends Builder {
        @Override
        public void buildScreening() {
            report.setScreening("幼儿园-筛查");
        }

        @Override
        public void buildCommon() {
            report.setCommon("幼儿园-常见病");
        }

        @Override
        public Report createReport() {
            return report;
        }
    }

    public static class PrimaryBuilder extends Builder {
        @Override
        public void buildScreening() {
            report.setScreening("小学以上-筛查");
        }

        @Override
        public void buildCommon() {
            report.setCommon("小学以上-常见病");
        }

        @Override
        public Report createReport() {
            return report;
        }
    }
```

```
    @Getter
    @Setter
    @ToString
    public static class Report {

        private String screening;

        private String common;
    }

    public static void main(String[] args) {
        System.out.println(new Director().setBuilder(new PrimaryBuilder()).construct());
        System.out.println(new Director().setBuilder(new kindergartenBuilder()).construct());
    }
}
```

优点：

- 建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在指挥者类中对整体而言可以取得比较好的稳定性。
- 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。
- 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
- 建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。符合开闭原则。

缺点：

- 造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。

扩展：对象set值