目录

这本是试读版本，为了给关注公众号的读者一点小福利，所以把完整版本放在了公众号。获取完整版本姿势：关注公众号『LieBrother』，后台回复【设计模式】即可获得。

欢迎大家关注我的公众号『LieBrother』，生活不止代码，还有诗和远方！

1. 六大原则

1.1. 单一职责原则（方法：修改名字还是密码？接口：洗碗、买菜还是倒垃圾？类：注册、登录和注销）

1.1.1. 简介

姓名： 单一职责原则

英文名： Single Responsibility Principle

座右铭： There should never be more than one reason for a class to change. 应当有且仅有一个原因引起类的变更。。。意思就是不管干啥，我都只干一件事，你叫我去买菜，我就只买菜，叫我顺便去倒垃圾就不干了，就这么拽

脾气： 一个字“拽”，两个字“特拽“

伴侣： 老子职责单一，哪来的伴侣？

个人介绍： 在这个人兼多责的社会里，我显得那么的特立独行，殊不知，现在社会上发生的很多事情都是因为没有处理好职责导致的，比如，经常有些父母带着小孩，一边玩手机，导致小孩弄丢、发生事故等等

1.1.2. 单一职责应用范围

单一职责原则适用的范围有接口、方法、类。按大家的说法，接口和方法必须保证单一职责，类就不必保证，只要符合业务就行。

1.1.2.1. 方法

设想一下这个场景：假设我们要做一个用户修改名字以及修改密码的功能，可以有多种实现方案，比如下面列举 2 种实现方式

1.1.2.1.1. 第一种实现方式

```
/**
 * 错误的示范
 */
enum OprType {
    /**
     * 更新密码
     */
    UPDATE_PASSWORD,
    /**
     * 更新名字
     */
    UPDATE_NAME;
}

interface UserOpr {
    boolean updateUserInfo(User user, OprType oprType);
}

class UserOprImpl implements UserOpr {

    @Override
    public boolean updateUserInfo(User user, OprType oprType) {
        if (oprType == OprType.UPDATE_NAME) {
            // update name
        } else if (oprType == OprType.UPDATE_PASSWORD) {
            // update password
        }
        return true;
```

```
    }
}
```

1.1.2.1.2. 第二种实现方式

```
/**
 * 正确的示范
 */
interface UserOpr2 {
    boolean updatePassword(User user, String password);
    boolean updateUserInfo(User user);
}

class UserOprImpl2 implements UserOpr2 {

    @Override
    public boolean updatePassword(User user, String password) {
        user.setPassword(password);
        // update password
        return true;
    }

    @Override
    public boolean updateUserInfo(User user) {
        // update user info
        return true;
    }
}
```

2 种实现有什么区别呢？ 第一种实现通过 OprType 类型的不同来做不同的事情，把修改密码和修改名字耦合在一起，容易引起问题，只要稍不注意，传错枚举值就悲剧了，在代码中也没法很直接看到是做什么操作，也就是这个方法的职责不明确。而第二种实现，把修改密码和修改名字分离开来，也就是把修改密码和修改名字都当做独自的职责处理，这样子就很清晰明了，你调用哪个方法，就很明确的知道这个方法是实现什么逻辑。结论是啥呢？用第二种方式实习才符合单一职责原则。现实中看到很多像第一种实现的代码，而且是枚举有十来个的情况，看代码真费劲。

1.1.2.2. 接口

设想一下这个场景，假设我们让小明去倒垃圾，小红去买菜，小红回来后再叫小红去洗碗。下面也举 2 个实现的例子。

1.1.2.2.1. 第一种实现方式

```
/**
 * 错误的示范
 */
interface Housework {
    void shopping();
    void pourGarbage();
}

class XiaoMing implements Housework {

    @Override
    public void shopping() {
        // 不购物
    }

    @Override
    public void pourGarbage() {
        System.out.println("pourGarbage ...");
    }
}

class XiaoHong implements Housework {

    @Override
    public void shopping() {
        System.out.println("shopping ...");
    }

    @Override
    public void pourGarbage() {
        // 从不倒垃圾
    }
}
```

中途回来小红去洗碗，要怎么实现？按这个写法，就在 Housework 接口添加 washingUp() 方法，然后小明和小红依次都实现洗碗这个方法，只是小明不做具体实现代码，这样子是不是觉得很别扭，不符合单一职责原则的，修改一个

地方，不影响其他不需要改变的地方，只对需要用到的地方做修改。小明本来就不用洗碗，却要去实现洗碗这个方法。

1.1.2.2.2. 第二种实现方式

```
/**
 * 正确的示范
 */
interface Shopping {
    void doShopping();
}

interface PourGarbage {
    void doPourGarbage();
}

interface WashingUp {
    void doWashingUp();
}

class XiaoMing2 implements PourGarbage {

    @Override
    public void doPourGarbage() {
        System.out.println("pourGarbage ...");
    }
}

class XiaoHong2 implements Shopping, WashingUp {

    @Override
    public void doShopping() {
        System.out.println("shopping ...");
    }

    @Override
    public void doWashingUp() {
        System.out.println("washing up ...");
    }
}
```

可以看到，这种实现把不同的家务都当做不同的职责，分离开来，这种实现可以按需实现做家务的类型，小明只需要去倒垃圾，就实现 PourGarbage 接口，小红去购物和洗碗，就实现 Shopping 和 WashingUp 接口，完全不会影响到对方，这才是完美的根据单一职责原则编写出来的代码。

1.1.2.3. 类

类这个看了一些资料都说没法硬性要求一定按单一职责原则分，或者说类的职责可大可小，没有很明确的像上面接口那样按照单一职责原则分就很清晰也很有道理。 设想一下这个场景：我们要实现一个用户注册、登录、注销操作，可以像如下 2 种实现方式

1.1.2.3.1. 第一种实现方式

从用户的角度考虑，这些操作都是用户的行为，可以放在一个统一的类 UserBiz

```
class UserBiz {

    public boolean register(User user){
        // 注册操作
        return true;
    }

    public boolean login(User user) {
        // 登录操作
        return true;
    }

    public boolean logout(User user) {
        // 注销操作
        return true;
    }

}
```


1.1.2.3.2. 第二种实现方式

有人又说，不是说单一职责么？从业务操作考虑，需要把注册、登录、注销分开

```
class UserRegisterBiz {

    public boolean register(User user){
```

```
        // 注册操作
        return true;
    }

}

class UserLoginBiz {

    public boolean login(User user) {
        // 登录操作
        return true;
    }

}

class UserLogoutBiz {

    public boolean logout(User user) {
        // 注销操作
        return true;
    }

}
```


感觉像是在抬杠，其实这个没有好坏之分，根据具体业务具体分析，你说你的登录、注册、注销操作代码很多，需要分开，那就分开，无可厚非。

1.1.3. 好处

1. 类的复杂性降低，实现什么职责都有清晰明确的定义
2. 可读性提高，复杂性降低，那当然可读性提高了
3. 可维护性提高，可读性提高，那当然更容易维护了
4. 变更引起的风险降低，变更是必不可少的，如果接口的单一职责做得好，一个接口修改只对相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助 (来自《设计模式之禅》)

1.1.4. 总结

这个单一职责原则，目的就是提高代码的可维护性、可读性、扩展性，如果为了单一职责而破坏了这 3 个特性，可能会得不偿失。

1.2. 里氏替换原则（我儿来自新东方烹饪）

1.2.1. 简介

姓名： 里氏替换原则

英文名： Liskov Substitution Principle

座右铭： 1. If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T,the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T. 如果对每一个类型为 S 的对象 o1，都有类型为 T 的对象 o2，使得以 T 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 S 是类型 T 的子类型。

2. Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. 所有引用基类的地方必须能透明地使用其子类的对象。

这 2 个定义来自《设计模式之禅》，比较干巴巴，不认真思考起来可能不太容易懂。简单来说就是定义了什么是父子。在现实生活中，什么是父子？就是生你的那个男人和你的关系就是父子（父女）。而这里定义的就是假如 A 能胜任 B 干的所有事情，那 B 就是 A 的父亲，也就是儿子要会父亲的所有能活，儿子活得再烂也要有父亲的水平。

价值观：很显然，比较传统，严父出孝子。儿子必须要有父亲的能耐，最好青出于蓝胜于蓝。

伴侣：估计有个贤惠的老婆，才能有这么优秀的儿子。

个人介绍：我比较严厉，也是为了生存没办法，只有一辈一辈地变优秀，一直坚持下去，家族就会越来越好。这样就可以富过三代，你看你们人类不是经常说富不过三代。。。扎心了老铁，老子还是富零代。

1.2.2. 老爹开车，前方注意

里氏替换原则定义了什么是父子，还有一点要注意的，就是儿子不能在父亲会的技能上搞“创新”。比如父亲会做红烧排骨，儿子在新东方烹饪学校中学到了一招，在红烧排骨里面加糖和醋，变成红烧糖醋排骨，更加美味，看代码，儿子在父亲的基础红烧排骨上加了糖醋，好像没啥问题。

```
class Father1 {

    public void braisedRibs(){
        System.out.println("红烧排骨");
    }

}


class Son1 extends Father1 {

    public void braisedRibs(){
        System.out.println("红烧糖醋排骨");
    }

}
```

运行下面代码，会打印：红烧排骨

```
Father1 father1 = new Father1();
father1.braisedRibs();
```

我们上面说过，所有在使用父亲的地方，都能够替换成儿子，并且效果是一样的，那接下来我们改一下代码。

```
Son1 son1 = new Son1();
son1.braisedRibs();
```

结果是啥？打印出：红烧糖醋排骨，出乎意料吧。。。这结果完全不一样。想一下上面说的：老爸会的老子也要会，很明显，上面的例子老子不会红烧排骨，只会红烧糖醋排骨，所以这根本不是父子关系。

那应该怎么实现呢？其实红烧排骨和红烧糖醋排骨这压根就是 2 道菜，你去餐馆吃饭的时候，你点红烧排骨服务员给你送来红烧糖醋排骨，或者你点红烧糖醋排骨服务员给你送来红烧排骨，你这时候不生气，算我输。

来看看 Son2，Son2 将红烧糖醋改为 braisedSweetAndSourPorkRibs （翻译不好找 Google 算账去哈，反正不是我翻译的）。

```
class Son2 extends Father1 {

    public void braisedSweetAndSourPorkRibs(){
        System.out.println("红烧糖醋排骨");
    }

}
```

测试一下是不是好儿子

```
Son2 son2 = new Son2();
son2.braisedRibs();
son2.braisedSweetAndSourPorkRibs();
```

打印出：
红烧排骨
红烧糖醋排骨

这才是 Father1 的好儿子嘛，不仅会红烧排骨，还会红烧糖醋排骨。所以说里氏替换原则就是在定义父子关系，大家都遵守这个定义，就会一代比一代好，不遵守大家也看到了，把前辈传下来的都毁于一旦了。

1.2.3. 优缺点

下面再贴一下书本上的一些优缺点

1.2.3.1. 优点

1. 代码共享，减少创建类的工作量，每个子类都拥有父类的方法和属性；
2. 提高代码的重用性；

3. 子类可以形似父类，但又异于父类，“龙生龙，凤生凤，老鼠生来会打洞”是说子拥有父的“种”，“世界上没有两片完全相同的叶子”是指明子与父的不同；
4. 提高代码的可扩展性，实现父类的方法就可以“为所欲为”了，君不见很多开源框架的扩展接口都是通过继承父类来完成的；
5. 提高产品或项目的开放性。

1.2.3.2. 缺点

1. 继承是侵入性的。只要继承，就必须拥有父类的所有属性和方法；
2. 降低代码的灵活性。子类必须拥有父类的属性和方法，让子类自由的世界中多了些约束；
3. 增强了耦合性。当父类的常量、变量和方法被修改时，需要考虑子类的修改，而且在缺乏规范的环境下，这种修改可能带来非常糟糕的结果——大段的代码需要重构。(来自《设计模式之禅》)

1.2.4. 总结

好了，里氏替换原则的大概原理讲得差不多，大家只要记住是在定义“父子关系”，就像游戏规则一样，定义后让大家遵守，会让大家的程序在后面越来越复杂的时候也能清晰，而不会越来越乱。

1.3. 依赖倒置原则（抠门的饭店老板）

1.3.1. 简介

姓名： 依赖倒置原则

英文名： Dependence Inversion Principle

价值观： 大男子主义的典型代表，什么都得通过老大或者老爸同意。

伴侣： 一定是个温柔体贴的女子。

个人介绍：

1. High level modules should not depend upon low level modules.Both should depend upon abstractions. 高层模块不应该依赖低层模块，两者都应该依赖其抽象（模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的）
2. Abstractions should not depend upon details. 抽象不应该依赖细节（接口或抽象类不依赖于实现类）
3. Details should depend upon abstractions. 细节应该依赖抽象（实现类依赖接口或抽象类）

给大家讲个故事，我胡乱想的，如有雷同，肯定是英雄所见略同。那必须交个朋友。

一个小村里，有两家饭馆，虽然挂着不同的牌子，挨在一起，但是老板确是表兄弟。这两兄弟抠得很，为了节省成本，密谋了一个想法：在两家饭馆谁家忙的时候，可以让不忙的那家的员工过去支援一下。这样子，本来每家饭馆都需要 2 个洗碗工，总共需要 4 个，他们就只招了 3 个，省了 1 个洗碗工的成本，当然不止洗碗工，还有服务员等等。两兄弟约定了规则：

1. A 饭馆需要支援的时候，B 饭馆老板，让 B 饭馆老板选哪个员工去支援，不能直接让 A 饭馆的员工直接找 B 饭馆的员工去帮忙，但可以让 A 饭馆员工找 B 饭馆老板告知需要支援。
2. 虽然老板权利大，但是也不能说 A 饭馆老板直接叫 B 饭馆的员工去帮忙。
3. 员工没有真实的老板，今天为 A 饭馆工作就是 A 饭馆的员工，没有跟定哪个老板。

大概通过这个小故事，描述了依赖倒置原则的基本内容。

1.3.2. 代码体现

下面通过代码来模拟这个故事。

1.3.2.1. 错误的示范

这个错误的示范将就看哈，可能有些问题没描述清楚。

1.3.2.1.1. 老板和员工抽象

```
abstract class Boss {

    abstract void support();

    abstract void askHelp(Boss boss);
}

abstract class Staff {

    private String name;

    abstract void service();

    abstract void askHelp(Boss boss);

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

1.3.2.1.2. 老板具体类

```
class BossA extends Boss {

    private StaffA staffA;

    public BossA(StaffA staffA) {
        this.staffA = staffA;
    }

    @Override
    void support() {
```

```
        staffA.service();
    }

    @Override
    void askHelp(Boss boss) {
        boss.support();
    }

}

class BossB extends Boss {

    private StaffB staffB;

    public BossB(StaffB staffB) {
        this.staffB = staffB;
    }

    @Override
    void support() {
        staffB.service();
    }

    @Override
    void askHelp(Boss boss) {
        boss.support();
    }
}
```

1.3.2.1.3. 员工具体类

```
class StaffA extends Staff {

    public StaffA(String name) {
        this.setName(name);
    }

    @Override
    void service() {
        System.out.println(this.getName() + "提供服务");
```

```
    }

    @Override
    void askHelp(Boss boss) {
        boss.support();
    }
}

class StaffB extends Staff {

    public StaffB(String name) {
        this.setName(name);
    }

    @Override
    void service() {
        System.out.println(this.getName() + "提供服务");
    }

    @Override
    void askHelp(Boss boss) {
        boss.support();
    }
}
```


1.3.2.1.4. 测试代码

```
/** 初始化老板和员工 */
StaffA staffA = new StaffA("A 员工");
StaffB staffB = new StaffB(" B 员工");
Boss bossA = new BossA(staffA);
Boss bossB = new BossB(staffB);

/** A 老板向 B 老板求支援 */
bossA.askHelp(bossB); // 打印出：B 员工提供服务

/** B 员工向 A 老板求支援 */
staffB.askHelp(bossA); // 打印出：A 员工提供服务
```

好像看起来实现了要求了，但是其实这段代码没有按照上面的 3 点规则编写，破坏了第 3 点规则，老板们的员工没有用员工的抽象类，破坏了细节依赖抽象

这一点。设想一下，假如现在 A 老板把 A 员工辞退了，重新招了个 C 员工，那么怎么实现呢？是不是需要再新增一个 StaffC 类，然后再修改 BossA 类代码，把 StaffA 换成 StaffC。这样超级麻烦，在平时写项目中要时刻考虑这一点：在具体实现类使用其他类，是不是可以用其抽象类？

1.3.3. 正确的示范

看了上面那个憋屈的代码，再来看下面简洁的代码，才会发现依赖倒置原则是多么强大。

1.3.3.1. 老板和员工抽象类

```
abstract class Boss2 {

    private Staff2 staff;

    public Boss2(Staff2 staff) {
        this.staff = staff;
    }

    abstract void support();

    abstract void askHelp(Boss2 boss);

    public void setStaff(Staff2 staff) {
        this.staff = staff;
    }

    public Staff2 getStaff() {
        return staff;
    }
}

abstract class Staff2 {

    private String name;

    abstract void service();

    abstract void askHelp(Boss2 boss);
```

```
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

1.3.3.2. 老板类

```
class BossImpl extends Boss2 {

    public BossImpl(Staff2 staff) {
        super(staff);
    }

    @Override
    void support() {
        this.getStaff().service();
    }

    @Override
    void askHelp(Boss2 boss) {
        boss.support();
    }
}
```

1.3.3.3. 员工类

```
class StaffImpl extends Staff2{

    public StaffImpl(String name) {
        this.setName(name);
    }

    @Override
    void service() {
        System.out.println(this.getName() + "提供服务");
    }
```

```
    @Override
    void askHelp(Boss2 boss) {
        boss.support();
    }
}
```

1.3.3.4. 测试类

```
/** 正确示范 */
Staff2 staffA2 = new StaffImpl("A 员工");
Staff2 staffB2 = new StaffImpl("B 员工");
Boss2 bossA2 = new BossImpl(staffA2);
Boss2 bossB2 = new BossImpl(staffB2);

/** A 老板向 B 老板求支援 */
bossA2.askHelp(bossB2); // 打印出：B 员工提供服务

/** B 员工向 A 老板求支援 */
staffB2.askHelp(bossA2); // 打印出：A 员工提供服务

/** A 老板辞退了 A 员工，换成了 C 员工 */
Staff2 staffC2 = new StaffImpl("C 员工");
bossA2.setStaff(staffC2);

/** B 员工向 A 老板求支援 */
staffB2.askHelp(bossA2); // 打印出：C 员工提供服务
```

这代码相比上面错误的示范，简洁了很多，实现的功能却更灵活，这就是依赖倒置原则强大的地方，它可以将类的耦合性降低，提供灵活的处理。

1.3.4. 最佳实践

1. 变量的表面类型尽量是接口或者是抽象类
2. 任何类都不应该从具体类派生
3. 尽量不要覆写基类的方法
4. 结合里氏替换原则使用（来自《设计模式之禅》）

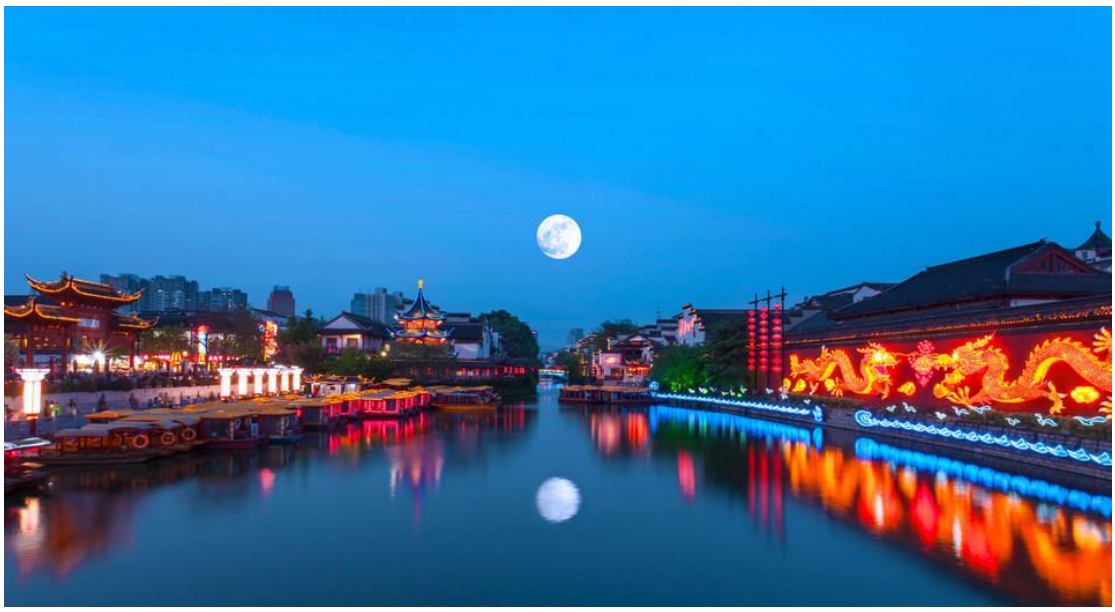
1.3.5. 总结

总的来说，要实现依赖倒置原则，要有『面向接口编程』这个思维，掌握好这个思维后，就可以很好的运用依赖倒置原则。

---

2. 五大创建型模式

2.1. 单例模式（小明就只有 1 辆车）



2.1.1. 简介

姓名：单例模式

英文名：Singleton Pattern

价值观：我的生活我主宰（只允许自己实例化，不愿意被其他对象实例化）

个人介绍：

Ensure a class has only one instance, and provide a global point of access to it.
（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。）
(来自《设计模式之禅》)

这里的关注点有 3 个，分别是： 1. 只有一个实例 2. 自行实例化（也就是主动实例化） 3. 向整个系统提供这个实例

2.1.2. 你要的故事

我们脑洞大开来用一个故事讲解一番。

小明家里有一辆小汽车，具体什么牌子就不知道了，咱也不关注，反正他家里就这么一辆车，小明比较懒，只要一出门都会开车，例如去旅游、去学校、去聚会都会开车去。下面模拟小明出去的场景。

```
class Car {
    public void run() {
        System.out.println("走。。。。");
    }
}


class XiaoMing {
    public Car travel() {
        System.out.println("小明去旅游");
        Car car = new Car();
        car.run();
        return car;
    }

    public Car goToSchool() {
        System.out.println("小明去学校");
        Car car = new Car();
        car.run();
        return car;
    }

    public Car getTogether() {
        System.out.println("小明参加聚会");
        Car car = new Car();
        car.run();
        return car;
    }
}
```

```
public class SingletonErrorTest {

    public static void main(String[] args) {
        XiaoMing xiaoMing = new XiaoMing();
        Car car1 = xiaoMing.travel();
        Car car2 = xiaoMing.goToSchool();
        Car car3 = xiaoMing.getTogether();
    }

}
```

上面小汽车只有一个方法，就是走。小明去旅游、去学校、参加聚会都开着他唯一的一辆汽车车去。是不是有人有疑问？为什么每个方法都返回 Car 对象？其实只是想在下面做一次检查，检查小明去旅游、去学校和参加聚会的车是不是同一辆。下面是检查代码：

```
System.out.println("car1 == car2 ? " + (car1 == car2));
System.out.println("car2 == car3 ? " + (car2 == car3));
```

最终结果是啥？很明显是 2 个 false。小明去旅游、去学校和参加聚会的车都不相同，小明不是只有 1 辆车？关键在于 `Car car = new Car();` 这一句代码，其实这一句是创建一辆车，每次都重新创建一辆。那应该怎么实现小明只有一辆车呢？这时候就引入了**单例模式**。

上面我们说到了单例模式需要具备的 3 个点：**只有 1 个实例**，很显然，上面的代码不止 1 个实例，而是有 3 个 Car 实例；**自行实例化**，Car 本身没有主动实例化，而是在小明需要用到的时候才实例化；**向整个系统提供这个实例**，因为 Car 没有主动实例化，所以它没法向外部暴露提供自己出来。

我们的代码完全不符合单例模式的要求。我们要通过修改，使之符合单例模式的 3 个要点。首先需要实现的是第 2 点，把 Car 实例化从小明转为 Car 本身，如下代码

```
class Car1{

    private static Car1 car1 = new Car1();

    private Car1() {

    }
```

```
    public void run(){
        System.out.println("走。。。。");
    }
}
```

上面代码使用 private 修饰构造方法，使得 Car1 不能被其他使用方实例化，通过 `Car1 car1 = new Car1();` 主动实例化自己。

接下来再实现第 3 点，向整个系统暴露这个实例，也就是暴露它自己。每个使用方都调用 `Car1.getInstance()` 方法来获取实例。

```
class Car1{

    private static Car1 car1 = new Car1();

    public static Car1 getInstance() {
        return car1;
    }

    private Car1() {

    }

    public void run(){
        System.out.println("走。。。。");
    }
}
```


上面代码就实现了单例模式的 2 和 3 要点，第 1 要点要怎么实现呢？告诉你，不用实现，只要满足了 2 和 3 要点就可以，第 1 要点是用来检验是否是单例模式的好思路。我们检验一下

```
class Car1{

    private static Car1 car1 = new Car1();

    public static Car1 getInstance() {
        return car1;
    }

    private Car1() {
```

```
    }

    public void run(){
        System.out.println("走。。。。");
    }
}

class XiaoMing1 {
    public Car1 travel() {
        System.out.println("小明去旅游");
        Car1 car = Car1.getInstance();
        car.run();
        return car;
    }

    public Car1 goToSchool() {
        System.out.println("小明去学校");
        Car1 car = Car1.getInstance();
        car.run();
        return car;
    }

    public Car1 getTogether() {
        System.out.println("小明参加聚会");
        Car1 car = Car1.getInstance();
        car.run();
        return car;
    }
}

public class SingletonRightHungryTest {

    public static void main(String[] args) {
        XiaoMing1 xiaoMing1 = new XiaoMing1();
        Car1 car1 = xiaoMing1.travel();
        Car1 car2 = xiaoMing1.goToSchool();
        Car1 car3 = xiaoMing1.getTogether();

        System.out.println("car1 == car2 ? " + (car1 == car2));
        System.out.println("car2 == car3 ? " + (car2 == car3));
```

```
    }

}
```

上面代码最后两行打印出来的结果是啥？是我们想要的：2 个 true。说明小明这几次外出开的车都是同一辆。这是最简单的单例模式的实现方式，我们经常称作**饿汉式单例模式**。为什么起这么古怪的名字呢？其实和对应的**懒汉式单例模式**有关，这是 2 个实现方式的差别，饿汉式单例模式实现方式在类加载到内存的时候，就创建好对象了，而懒汉式则是在第一次使用的时候才创建对象，也就是把创建对象的时机从加载延迟到第一次使用，所以才有懒饿之分。

下面我们来看怎么实现懒汉式单例模式。先描述一下场景：小明还没有汽车，他也不知道什么时候要买汽车，突然某一天，他想去旅游，觉得是时候买辆车了，然后他就买车去旅游了，旅游回来又开车去学校和参加聚会。

```
class Car2{

    private static Car2 car2;

    public static synchronized Car2 getInstance() {
        if (null == car2) {
            System.out.println("买车啦。。。");
            car2 = new Car2();
        }
        return car2;
    }

    private Car2() {

    }

    public void run(){
        System.out.println("走。。。。");
    }
}

class XiaoMing2
{
    public Car2 travel() {
        System.out.println("小明去旅游");
        Car2 car = Car2.getInstance();
```

```
        car.run();
        return car;
    }

    public Car2 goToSchool() {
        System.out.println("小明去学校");
        Car2 car = Car2.getInstance();
        car.run();
        return car;
    }

    public Car2 getTogether() {
        System.out.println("小明参加聚会");
        Car2 car = Car2.getInstance();
        car.run();
        return car;
    }
}

public class SingletonRightLazyTest {

    public static void main(String[] args) {
        XiaoMing2 xiaoMing2 = new XiaoMing2();
        Car2 car1 = xiaoMing2.travel();
        Car2 car2 = xiaoMing2.goToSchool();
        Car2 car3 = xiaoMing2.getTogether();

        System.out.println("car1 == car2 ? " + (car1 == car2));
        System.out.println("car2 == car3 ? " + (car2 == car3));
    }

}
```

```
小明去旅游
买车啦。。。
走。。。。
小明去学校
走。。。。
小明参加聚会
```

走。。。。

```
car1 == car2 ? true
car2 == car3 ? true
```

上面附带了打印出来的结果，小明要去旅游的时候，才去买车。这就是**懒汉式单例模式**的实现方式。

要注意懒汉式单例模式有个很关键的一点就是 getInstance() 方法带上了 synchronized，这个是为什么呢？

首先得了解关键字 synchronized 的作用是什么：用于修饰执行方法同步，也就是说多线程并发的情况下，在一个时间点，只允许一个线程执行这个方法。

不加上这个会有什么结果？在多线程并发情况下，如果有 2 个线程同时执行到 if(null == car2)，那么都判断为 true，这时 2 个线程都会执行 car2 = new Car2()，这样子就不是单例了。

2.1.3. 总结

单例模式可以说是设计模式中最简单的一个，也是在工作中很多场景下经常用到的，比如：项目的配置文件加载、各种工具类等等。我们对于单例模式最重要的一点就是要考虑**多线程并发**，没有考虑这点就容易引发单例对象不单例的情况。而单例给我们带来最大的好处就是**节约内存**。

上面实现的两种方法是单例模式中最最最简单的 2 种实现，相信也是用得最多的实现方式。网上有不少网友分享了单例模式的很多种实现方法，大家也可以去了解，在了解之前务必已经搞懂文中这 2 种最简单的实现方式，不然会头晕的。

2.2. 工厂方法（小明家的车库）

2.2.1. 简介

姓名：工厂方法

英文名：Factory method Pattern

价值观：扩展是我的专属

个人介绍：

Define an interface for creating an object,but let subclasses decide which class to instantiate.Factory Method lets a class defer instantiation to subclasses. (定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。) (来自《设计模式之禅》)

2.2.2. 你要的故事

还记得上一篇 单例模式 中的故事么？小明开着汽车去旅游、去学校、去聚会。这一次还是延续小明的故事，一个故事能讲 2 个设计模式，不容易呀。。。（每次想故事都想破脑袋，每一篇文章至少有 3 个故事从脑子里闪过，但最终留下的只有一个适合，为了就是能比较清晰简单的说明设计模式中的关键要点。）

2.2.2.1. 简单工厂

小明家里以前不算很富裕，但是还是有一个不错的车库，什么汽车、摩托车、自行车啥的都放在这个车库里。小明每次要出去，都会到车库里面挑合适的车出发。比如，小明最近期末考试了，骑摩托车去学校考试，考完试之后，小明就准备去旅游，这次决定自驾游，开着自己家的小汽车去。这个场景我们用代码描述下。

```
public class SimpleFactoryTest {

    public static void main(String[] args) {
        XiaoMing xiaoMing = new XiaoMing();
        // 小明骑摩托车去学校
        IVehicle motorcycle = GarageFactory.getVehicle("motorcycle");
        xiaoMing.goToSchool(motorcycle);

        // 小明开汽车去旅游
        IVehicle car = GarageFactory.getVehicle("car");
        xiaoMing.travel(car);
    }

}

/**
 * 车库
 */
class GarageFactory {

    public static IVehicle getVehicle(String type) {
        if ("car".equals(type)) {
            return new Car();
        } else if ("motorcycle".equals(type)) {
            return new Motorcycle();
        }
        throw new IllegalArgumentException("请输入车类型");
    }

}

/**
```

```
 * 交通工具
 */
interface IVehicle {
    void run();
}

/**
 * 汽车
 */
class Car implements IVehicle {

    @Override
    public void run() {
        System.out.println("开汽车去。。。。");
    }
}

/**
 * 摩托车
 */
class Motorcycle implements IVehicle {

    @Override
    public void run() {
        System.out.println("骑摩托车去。。。。");
    }
}


class XiaoMing {

    public void goToSchool(IVehicle vehicle) {
        System.out.println("小明去学校");
        vehicle.run();
    }

    public void travel(IVehicle vehicle) {
        System.out.println("小明去旅游");
        vehicle.run();
    }
```

```
}
```

上面代码看懂了么？小明家里有一个车库 GarageFactory，里面放着汽车 Car 和摩托车 Motorcycle，小明要出去的时候，就到车库选择车，通过传递参数给 GarageFactory.getVehicle()，指明要什么车，然后小明就骑着车出发了。

这个代码真正的术语叫：**简单工厂模式**（Simple Factory Pattern），也叫做**静态工厂模式**。它是工厂方法中的一个实现方式，从字面理解就可以知道，它是最简单的工厂方法实现方式。它有一点点小缺陷，就是**扩展性不够好**，在上面代码中，小明只能骑摩托车或者开汽车，如果小明要骑单车出去呢？势必得在 GarageFactory 中添加 if 是自行车的逻辑。这违反了哪条规则了？是不是那个**允许扩展，拒绝修改的**开闭原则？

不是说简单工厂这种实现方式不好，而是扩展性不够，在平时的开发中，简单工厂模式也用得不少。在这个小明家里车不多的情况下，用一个车库也是合适的。

2.2.2.2. 工厂方法

小明老爸近几年赚了不少，车迷的两父子一直买车，家里的车越来越多，这时候，他们决定多建几个车库，按车类型放置。比如，有一个汽车库，一个摩托车库。这时候小明要开汽车就去汽车库，要骑摩托车就去摩托车库。代码实现如下。

```
public class FactoryMethodTest {

    public static void main(String[] args) {
        XiaoMing xiaoMing = new XiaoMing();
        // 小明骑摩托车去学校
        VehicleGarage motorcycleGarage = new MotorcycleGarage();
        IVehicle motorcycle = motorcycleGarage.getVehicle();
        xiaoMing.goToSchool(motorcycle);

        // 小明开汽车去旅游
        VehicleGarage carGarage = new CarGarage();
        IVehicle car = carGarage.getVehicle();
        xiaoMing.travel(car);
    }

}
```

```
interface VehicleGarage {
    IVehicle getVehicle();
}

/**
 * 汽车车库
 */
class CarGarage implements VehicleGarage {

    @Override
    public IVehicle getVehicle() {
        return new Car();
    }
}

/**
 * 摩托车车库
 */
class MotorcycleGarage implements VehicleGarage {

    @Override
    public IVehicle getVehicle() {
        return new Motorcycle();
    }
}
```


上面代码重用了简单工厂实现方式的交通接口以及摩托车和汽车的实现类。代码中有 2 个车库，一个是汽车车库 CarGarage，一个是摩托车库 MotorcycleGarage。如果小明要骑自行车，只需要建一个自行车车库，完全不用去修改汽车车库或者摩托车车库，就非常符合开闭原则，扩展性大大的提高。

2.2.3. 总结

工厂方法模式可以说在你能想到的开源框架源码中必定会使用的一个设计模式，因为开源框架很重要一点就是要有扩展性，而工厂方法模式恰恰具有**可扩展性**。弄懂了工厂方法模式，以后看开源代码就很得心应手啦。

3. 十一大行为型模式

3.1. 模板方法（运动鞋制造过程）

3.1.1. 简介

姓名： 模板方法

英文名： Template Method Pattern

价值观： 在我的掌控下，任由你发挥

个人介绍：

Define the skeleton of an algorithm in an operation,deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. 定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。（来自《设计模式之禅》）

解释一下上面的介绍，意思是由父类来定义框架，让子类来具体实现。

3.1.2. 你要的故事

刚过完春节，大家都买新鞋了么？今天要讲的故事和鞋子有关。一双鞋子从表面来看，由鞋底、鞋垫、鞋面、鞋带组成，同一系列的鞋子这几个部分都是一样的，用同样的材料做出来，不同系列的鞋子就大相径庭了。根据模板方法模式，组装一双鞋子的制造过程可以归并为固定的框架，至于用什么材料，那由每个系列的鞋子去具体实现。我们先看定义组装鞋子的框架代码。

```
/**
 * 定义鞋子制造的工序框架
 */
abstract class ShoeInstallTemplate {

    public abstract void installSole();
    public abstract void installInsole();
    public abstract void installVamp();
    public abstract void installShoelace();

    public void installShot(){
        System.out.println("组装一双鞋，步骤如下：");
        // 组装鞋底
        installSole();
        // 组装鞋垫
        installInsole();
        // 组装鞋面
        installVamp();
        // 组装鞋带
        installShoelace();
    }

}
```

定义了一个组装鞋子框架的抽象类 ShoeInstallTemplate，里面有 4 个工序未具体实现，由鞋子制造商去实现，因为只有鞋子制造商才知道鞋子要用什么材料来做。 下面举 2 个比较出名的鞋子：Adidas 的 Boost 系列和 Nike 的 Jordan 系列。下面分别实现这 2 个系列鞋子的制造代码。

```
/**
 * Adidas Boost 鞋制造
 */
```

```
class AdidasBoostShoeInstall extends ShoeInstallTemplate {
    @Override
    public void installSole() {
        System.out.println("组装白色 Boost 鞋底");
    }

    @Override
    public void installInsole() {
        System.out.println("组装黑色 Boost 鞋垫");
    }

    @Override
    public void installVamp() {
        System.out.println("组装黑色 Boost 鞋面");
    }

    @Override
    public void installShoelace() {
        System.out.println("组装黑色 Boost 鞋带");
    }
}

/**
 * Nike Jordan 鞋制造
 */
class NikeJordanShoeInstall extends ShoeInstallTemplate {

    @Override
    public void installSole() {
        System.out.println("组装黑色 Jordan 鞋底");
    }

    @Override
    public void installInsole() {
        System.out.println("组装黑色 Jordan 鞋垫");
    }

    @Override
    public void installVamp() {
```

```
        System.out.println("组装红色 Jordan 鞋面");
    }

    @Override
    public void installShoelace() {
        System.out.println("组装红色 Jordan 鞋带");
    }
}
```

实现了制造商制造鞋子的代码之后，我们通过代码测试怎么制造 Boost 和 Jordan 鞋子。

```
public class TemplateMethodTest {

    public static void main(String[] args) {
        ShoeInstallTemplate adidasBoost = new AdidasBoostShoeInstall
();
        adidasBoost.installShot();

        ShoeInstallTemplate nikeJordan = new NikeJordanShoeInstall();
        nikeJordan.installShot();
    }

}
```


打印结果：

```
组装一双鞋，步骤如下：
组装白色 Boost 鞋底
组装黑色 Boost 鞋垫
组装黑色 Boost 鞋面
组装黑色 Boost 鞋带

组装一双鞋，步骤如下：
组装黑色 Jordan 鞋底
组装黑色 Jordan 鞋垫
组装红色 Jordan 鞋面
组装红色 Jordan 鞋带
```

模板方法模式就这么简单。是不是掌握了？

3.1.3. 总结

模板方法是一个比较实用的模式，为什么说实用呢？举个现实的例子，Java 能有如今的发展，离不开各大开源框架，比如 Dubbo，有看过源码的朋友就知道，里面大量代码运用了模板方法设计模式，为什么 Dubbo 可以支持很多种注册中心？其实本质就是用了模板方法设计模式，使得可以扩展多种注册中心。掌握好模板方法，对读源码有非常大的帮助，很多人包括我在内，在刚开始阅读源码的时候，有相当长的一段时间怀疑人生，怎么这些代码那么绕？调来调去的。当你了解了常用的设计模式之后，看源代码就可以直截了当的知道是用什么设计模式，为什么用这个设计模式？原来是为了什么什么。。。有了这层思考，就像有一条线将以前散落在各地的知识点连接起来，成了可以推敲的知识。

3.2. 中介者模式（租房找中介）

3.2.1. 简介

姓名： 中介者模式

英文名： Mediator Pattern

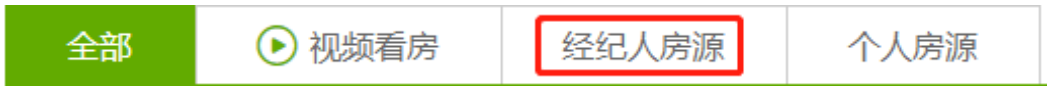
价值观： 让你体验中介是无所不能的存在

个人介绍：

Define an object that encapsulates how a set of objects interact.Mediator promotes loose coupling by keeping objects from referring to each other explicitly,and it lets you vary their interaction independently. 用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互。（来自《设计模式之禅》）

3.2.2. 你要的故事

看了这小伙子的名字，大家会很直观的想到那些拿了我们半个月租的租房中介同学。在这不讲讲房租中介同学，以后可没机会了。大家现在找房子，不管是买还是租，一登录什么安居客、58 同城，是不是有 80% 是经纪人房源，说 80% 还是比较保守的，经历了 4 次找房，发现个人房源越来越少。每个网站都有个选项：经纪人房源。如下图：



房源

(图片截自：安居客网站)

经纪人就扮演着中介的角色，和本文要讲的中介者模式完全吻合。我们在找房子的时候，经纪人扮演什么角色呢？我们通过个人房源和经纪人房源的租房案例来简单描述经纪人的角色。

3.2.2.1. 个人房源

我们通过个人房源找房子的方式是这样的：在网上找个人房源的房东，然后挨个联系，和房东约定好时间去看房，我们跟房东的关系是一对多的关系。小明就在网上看了个人房源，联系了房东，分别去看了农民房和小区房，用代码表示如下。

```
public class PersonalTest {

    public static void main(String[] args) {
        Tenant xiaoMing = new Tenant("小明");
        xiaoMing.lookAtHouse();
    }

}

class Tenant {
```

```
    private String name;
    private XiaoQuFangLandlord xiaoQuFangLandlord2 = new XiaoQuFangL
andlord();
    private NongMinFangLandlord nongMinFangLandlord2 = new NongMinFa
ngLandlord();

    public Tenant(String name) {
        this.name = name;
    }

    public void lookAtHouse() {
        System.out.println(this.name +"想看农民房");
        nongMinFangLandlord2.supply();
        System.out.println(this.name + "想看小区房");
        xiaoQuFangLandlord2.supply();
    }

}

/**
 * 房东
 */
abstract class Landlord {
    // 提供房子
    public abstract void supply();
}

class XiaoQuFangLandlord extends Landlord {

    @Override
    public void supply() {
        System.out.println("小区房的房东提供一间小区房");
    }
}

class NongMinFangLandlord extends Landlord {

    @Override
    public void supply() {
```

```
        System.out.println("农民房的房东提供一间小区房");
    }
}
```

打印结果如下：
小明想看农民房
农民房的房东提供一间小区房
小明想看小区房
小区房的房东提供一间小区房

小明分别联系小区房的房东和农民房的房东，然后依次去看了农民房和小区房。这样子有个弊端就是小明和房东是强关联的关系，其实小明只是去看一下房，看完不想租就和房东没啥关系了。这个时候经纪人就派上用场了，经纪人的主要任务就是把房子租出去，所以他和房东应该是强关系，直到把房子成功租出去了，才和房东脱离关系，而小明也不用去挨个找房东看房子了，这个职责转给经纪人，小明只需要联系一个人，那就是经纪人，跟他说我要看小区房和农民房，经纪人就带他去看。下面就介绍经纪人房源的方式，也就是本文要讲的中介者模式。

3.2.2.2. 经纪人房源

用经纪人房源找房子，小明就省心很多了，小明就只联系了一个经纪人，跟他描述了自己要的房源：小区房和农民房都可以，经纪人里面和他约定了一个下午的时间，把小明所有想看的房让他看完，最终小明决定租了一间房。看代码。

```
public class MediatorTest {

    public static void main(String[] args) {
        System.out.println("小明想要看小区房和农民房");
        Tenant2 xiaoMing = new Tenant2("小明", Arrays.asList("XiaoQuF
ang", "NongMinFang"));
        xiaoMing.lookAtHouse();
    }


}

/**
 * 租客
 */
class Tenant2 {
```

```
    private String name;
    private List<String> wantTypes;

    private RentingMediator rentingMediator = new RentingMediator();

    public Tenant2(String name, List<String> wantTypes) {
        this.name = name;
        this.wantTypes = wantTypes;
    }

    public void lookAtHouse() {
        rentingMediator.supplyHouse(wantTypes);
    }

}

/**
 * 中介抽象类
 */
abstract class Mediator {
    // 看房
    public abstract void supplyHouse(List<String> types);
}

/**
 * 租房中介
 */
class RentingMediator extends Mediator {

    private XiaoQuFangLandlord xiaoQuFangLandlord;
    private NongMinFangLandlord nongMinFangLandlord;

    public RentingMediator() {
        xiaoQuFangLandlord = new XiaoQuFangLandlord();
        nongMinFangLandlord = new NongMinFangLandlord();
    }

    @Override
    public void supplyHouse(List<String> types) {
```

```
        System.out.println("经纪人提供了如下房源");
        if (types.contains("XiaoQuFang")) {
            xiaoQuFangLandlord.supply();
        }
        if (types.contains("NongMinFang")) {
            nongMinFangLandlord.supply();
        }
    }
}
```

打印结果：
小明想要看小区房和农民房
经纪人提供了如下房源
小区房的房东提供一间小区房
农民房的房东提供一间小区房

在代码中，我们可以看到小明和经纪人是一对一关系，经纪人和房东是一对多关系。小明找房经历也轻松多了，只花了一下午就把房子都看了并看中了。这也是中介者模式的优点，**减少了不必要的依赖，降低了类间的耦合**。

3.2.3. 总结

中介者模式通过在互相依赖的对象中间加了一层，让原本强依赖的对象变成弱依赖。在软件编程中，有一个中介者模式的典型的例子，就是 MVC 框架，也称三层架构，通过 Controller (控制层) 将 Model (业务逻辑层) 和 View (视图层) 的依赖给分离开，协调 Model 和 View 中的数据和界面交互工作。看看你工作中的代码，想想看有没有哪些对象之间的关系特紧密特混乱，考虑是不是可以通过中介者模式来把依赖关系剥离，让代码更清晰。

3.3. 命令模式（技术经理分配任务）

3.3.1. 简介

姓名： 命令模式

英文名： Command Pattern

价值观： 军令如山

个人介绍：

Encapsulate a request as an object,thereby letting you parameterize clients with different requests,queue or log requests,and support undoable operations. 将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。（来自《设计模式之禅》）

3.3.2. 你要的故事

作为一个程序猿，我们每天都在经历着命令模式，技术经理把需求任务分配给工程师开发，有时因为第三方或者其他不可抗拒的因素导致需求停止开发。这种工作模式就是命令模式。好了，开始故事了。小明在 XX 科技公司做一个安静的程序猿，有一天技术经理给他分配了一个任务：新增黑名单，也就是在他们系统的某个模块里面可以手工对电话打黑名单标签的功能。小明接到任务后就

立马开发，在开发了 2 天之后，因为战略原因，技术经理大明暂停了这个开发任务，接下来我们通过非命令模式和命令模式 2 种代码实现来体现这个过程。在这个场景中，为了简单，我们假定技术经理大明手下只有小明一个开发人员。

3.3.2.1. 非命令模式

非命令模式也就是不使用命令模式的代码实现。代码中，我们出现了 Developer 开发人，开发同学是接受技术经理传达的任务，技术经理让他开发哪个需求就开发哪个需求，如果项目有问题需要中断，也需要技术经理评估后传达给开发同学，所以 Developer 有 2 个方法，分别是 develop() 开发需求和 suspend() 暂停需求。 Requirement 则为需求类，TechnicalManager1 则为技术经理类，他有一个方法 action()，通过这个方法来指定开发同学开发任务或者暂停任务。

```
public class NoCommandTest {

    public static void main(String[] args) {
        Developer xiaoMing = new Developer("小明");
        Requirement requirement = new Requirement("新增黑名单");
        TechnicalManager1 technicalManager2 = new TechnicalManager1
("大明");
        technicalManager2.setDeveloper(xiaoMing);
        technicalManager2.action(requirement, "develop");
        System.out.println("开发了 2 天，需求变故，需要暂停。。。");
        technicalManager2.action(requirement, "suspend");
    }

}

/**
 * 开发人员
 */
class Developer {

    private String name;

    public Developer(String name) {
        this.name = name;
    }

    public void develop(Requirement requirement) {
```

```
        System.out.println(this.name + " 开始开发需求：" + requiremen
t.getName());
    }

    public void suspend(Requirement requirement) {
        System.out.println(this.name + " 停止开发需求：" + requiremen
t.getName());
    }

    public String getName() {
        return name;
    }

}

/**
 * 需求
 */
class Requirement {
    private String name;

    public Requirement(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

}

/**
 * 技术经理
 */
class TechnicalManager1 {

    private String name;

    private Developer developer;
```

```
    public TechnicalManager1(String name) {
        this.name = name;
    }

    public void setDeveloper(Developer developer) {
        this.developer = developer;
    }

    public void action(Requirement requirement, String type) {
        if ("develop".equals(type)) {
            this.developer.develop(requirement);
        } else if ("suspend".equals(type)) {
            this.developer.suspend(requirement);
        }
    }

}
```

打印结果：
小明 开始开发需求：新增黑名单
开发了 2 天，需求变故，需要暂停。。。
小明 停止开发需求：新增黑名单

通过代码，我们可以发现技术经理和开发同学是强依赖关系。如果技术经理下达了一个任务，要求小明写一下周报，这时候得怎么写？是不是小明需要一个写周报的方法，大明也需要新增一个处理事务类型？有没有更好的方法让技术经理不需要做任何改变？命令模式就来解决这个问题。

3.3.2.2. 命令模式

在这个例子中，不管大明叫小明做什么事情，其实都是一样的，就是下达任务命令，让小明去执行命令。我们可以利用命令模式把下达任务这个抽象起来，当做父类，下达开发命令、下达暂停命令、下达写周报等等都是不同的子命令。代码如下。

```
public class CommandTest {

    public static void main(String[] args) {
        Developer xiaoMing = new Developer("小明");
        Command developCommand = new DevelopCommand(xiaoMing);
```

```
        Command suspendCommand = new SuspendCommand(xiaoMing);
        Requirement requirement = new Requirement("新增黑名单");
        TechnicalManager2 technicalManager = new TechnicalManager2("
大明");
        technicalManager.setCommand(developCommand);
        technicalManager.action(requirement);
        System.out.println("开发了 2 天，需求变故，需要暂停。。。");
        technicalManager.setCommand(suspendCommand);
        technicalManager.action(requirement);

    }

}

/**
 * 命令
 */
abstract class Command {

    protected Developer developer;

    public Command(Developer developer) {
        this.developer = developer;
    }

    public abstract void execute(Requirement requirement);
}

/**
 * 开始开发
 */
class DevelopCommand extends Command {

    public DevelopCommand(Developer developer) {
        super(developer);
    }

    @Override
    public void execute(Requirement requirement) {
        this.developer.develop(requirement);
```

```
    }
}

/**
 * 开发中断
 */
class SuspendCommand extends Command {

    public SuspendCommand(Developer developer) {
        super(developer);
    }

    @Override
    public void execute(Requirement requirement) {
        this.developer.suspend(requirement);
    }
}

/**
 * 技术经理
 */
class TechnicalManager2 {

    private String name;
    private Command command;

    public TechnicalManager2(String name) {
        this.name = name;
    }

    public void action(Requirement requirement) {
        this.command.execute(requirement);
    }

    public void setCommand(Command command) {
        this.command = command;
    }
}
```

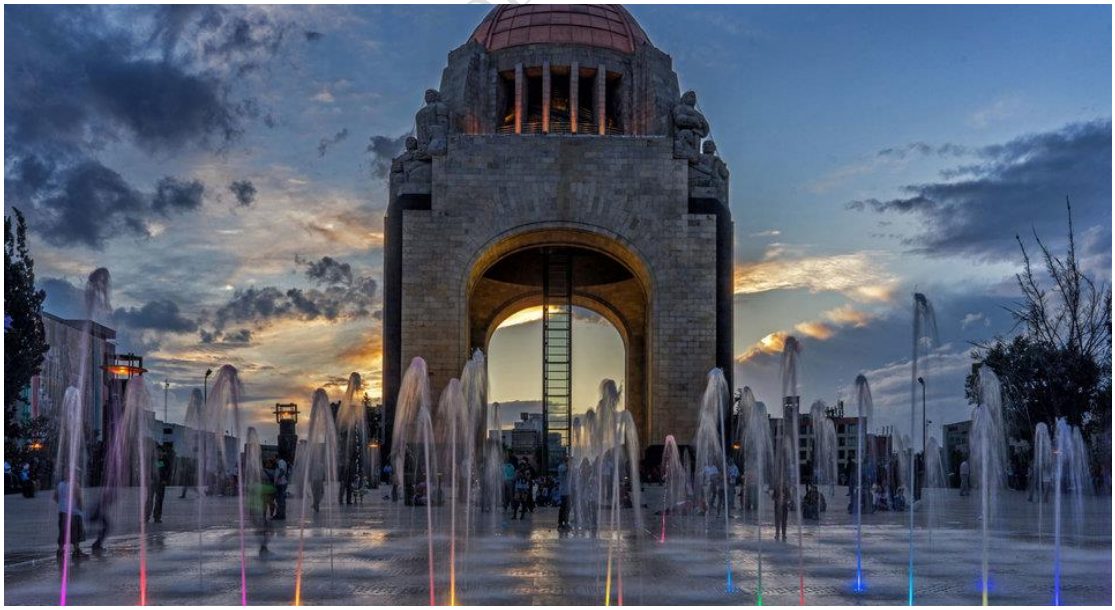
打印结果：
小明 开始开发需求：新增黑名单
开发了 2 天，需求变故，需要暂停。。。
小明 停止开发需求：新增黑名单

代码中用 Command 来抽象下达任务，而技术经理 TechnicalManager2 并没有和 Developer 有直接的关系，而是 TechnicalManager2 和 Command 建立的联系，Command 和 Developer 建立了联系。这样子把大明和小明的强依赖关系给剥离开，而新增一个下达写周报的任务也很简单，在 Developer 中新增一个处理写周报的方法，新增一个写周报的 Command 子类，就可以了，TechnicalManager2 如上面所愿不用修改。这就是完整的一个命令模式代码。

3.3.3. 总结

从文章中我们就可以看到，利用命令模式能够**进行类的解耦**，让调用者和接受者没有任何关系，也通过对行为的抽象，让新增其他行为变得清晰容易，也就是**可扩展性大大增加**。

3.4. 责任链模式（面试过五关斩六将）



3.4.1. 简介

姓名： 责任链模式

英文名： Chain of Responsibility Pattern

价值观： 责任归我

个人介绍： Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.Chain the receiving objects and pass the request along the chain until an object handles it. 使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。（来自《设计模式之禅》）

3.4.2. 你要的故事

快要金三银四了，很多同学蠢蠢欲动想要去外面看看世界，而大家有没有提前了解各大企业的面试流程呢？这里我就给大家科普一下大多数互联网企业的面试流程，正好责任链模式用得上。

在互联网企业中，程序员这个岗位的招聘流程大同小异，而一般至少都会有 3 轮面试，分别是 2 轮技术面和 1 轮 HR 面。而这几轮面试都是层层递进的，最开始面试一般是组长面试，通过之后就是部门领导面试，再通过之后就是 HR 面试，HR 面试通过之后就可以成功拿到 Offer 了。

故事从小明参加某公司的面试开始，某公司的招聘流程就是上面说的 3 轮面试。招聘流程的面试官分别是：第一面是组长老刚，第二面是部门经理老孙，第三面也就是终面是 HR 老刘。为什么说这个场景符合责任链模式呢？首先不管是组长还是部门经理还是 HR，他们都作为面试官，面试官赋予他们的权利是去面试来公司面试的同学，而面试的结果是可传递性的，也就是如果面试通过，就会到下一轮面试，最终成为一条传递链。我们用代码模拟这个过程。

```
public class ChainOfResponsibilityTest {

    public static void main(String[] args) {
        Interviewee interviewee = new Interviewee("小明");
        TeamLeader teamLeader = new TeamLeader("老刚");
        DepartMentManager departMentManager = new DepartMentManager
("老孙");
        HR hr = new HR("老刘");
        // 设置面试流程
        teamLeader.setNextInterviewer(departMentManager);
        departMentManager.setNextInterviewer(hr);
        // 开始面试
        teamLeader.handleInterview(interviewee);
```

```
    }

}

/**
 * 面试者
 */
class Interviewee {

    private String name;

    private boolean teamLeaderOpinion;
    private boolean departMentManagerOpinion;
    private boolean hrOpinion;

    public Interviewee(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isTeamLeaderOpinion() {
        return teamLeaderOpinion;
    }

    public void setTeamLeaderOpinion(boolean teamLeaderOpinion) {
        this.teamLeaderOpinion = teamLeaderOpinion;
    }

    public boolean isDepartMentManagerOpinion() {
        return departMentManagerOpinion;
    }

    public void setDepartMentManagerOpinion(boolean departMentManage
```

```
rOpinion) {
        this.departMentManagerOpinion = departMentManagerOpinion;
    }

    public boolean isHrOpinion() {
        return hrOpinion;
    }

    public void setHrOpinion(boolean hrOpinion) {
        this.hrOpinion = hrOpinion;
    }
}

/**
 * 面试官
 */
abstract class Interviewer {

    protected String name;
    protected Interviewer nextInterviewer;

    public Interviewer(String name) {
        this.name = name;
    }

    public Interviewer setNextInterviewer(Interviewer nextInterviewe
r) {
        this.nextInterviewer = nextInterviewer;
        return this.nextInterviewer;
    }

    public abstract void handleInterview(Interviewee interviewee);

}

/**
 * 组长
 */
class TeamLeader extends Interviewer {
```

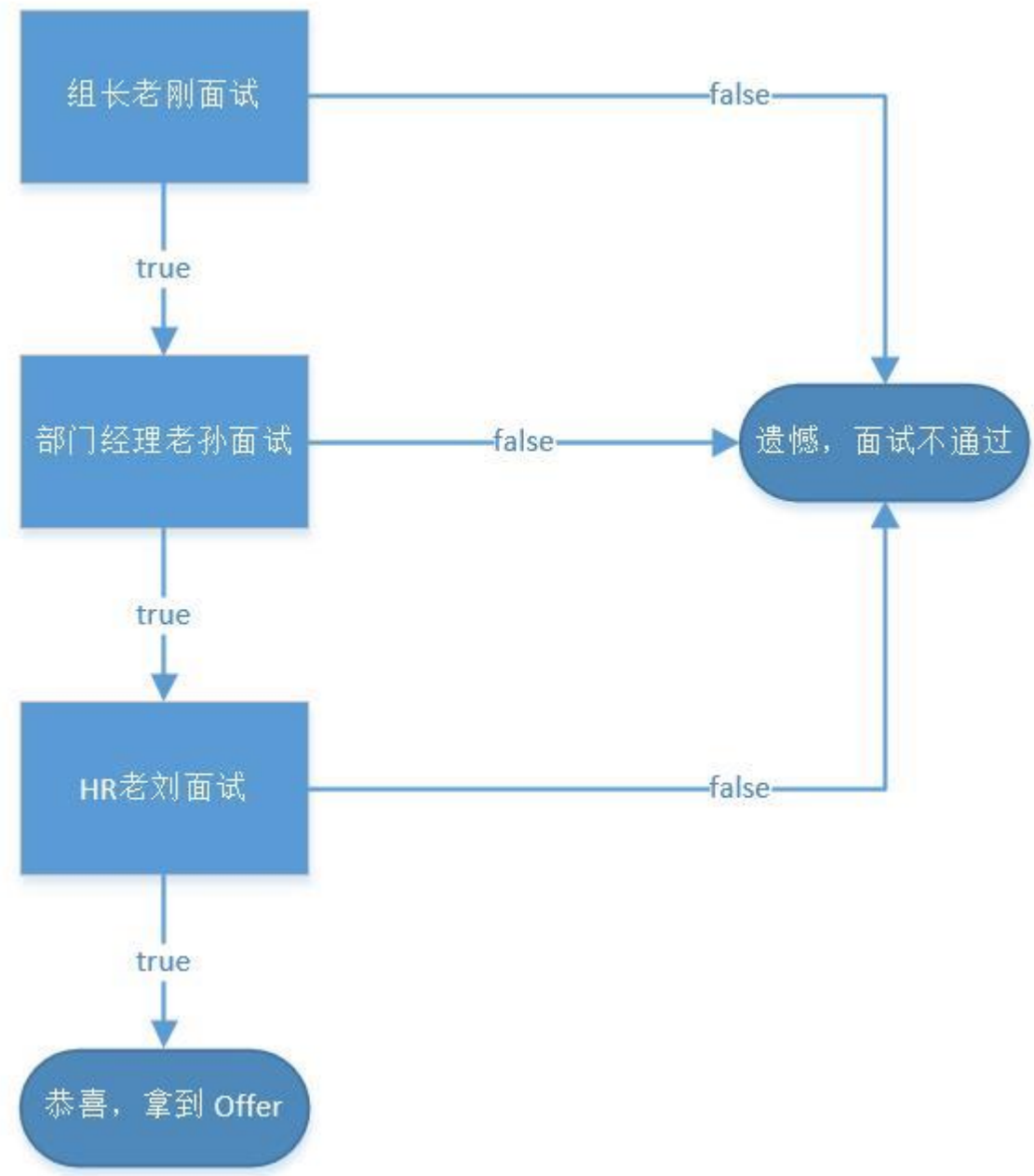
```
    public TeamLeader(String name) {
        super(name);
    }

    @Override
    public void handleInterview(Interviewee interviewee) {
        System.out.println("组长[" + this.name + "]面试[" + interview
ee.getName() + "]同学");
        interviewee.setTeamLeaderOpinion(new Random().nextBoolean());
        if (interviewee.isTeamLeaderOpinion()) {
            System.out.println("[" + interviewee.getName() + "]同学组
长轮面试通过");
            this.nextInterviewer.handleInterview(interviewee);
        } else {
            System.out.println("[" + interviewee.getName() + "]同学组
长轮面试不通过");
        }
    }
}

/**
 * 部门经理
 */
class DepartMentManager extends Interviewer {

    public DepartMentManager(String name) {
        super(name);
    }

    @Override
    public void handleInterview(Interviewee interviewee) {
        System.out.println("部门经理[" + this.name + "]面试[" + inter
viewee.getName() + "]同学");
        interviewee.setDepartMentManagerOpinion(new Random().nextBoo
lean());
        if (interviewee.isDepartMentManagerOpinion()) {
            System.out.println("[" + interviewee.getName() + "]同学部
门经理轮面试通过");
            this.nextInterviewer.handleInterview(interviewee);
```

```
        } else {
            System.out.println("[" + interviewee.getName() + "]同学部
门经理轮面试不通过");
        }
    }
}

/**
 * HR
 */
class HR extends Interviewer {

    public HR(String name) {
        super(name);
    }

    @Override
    public void handleInterview(Interviewee interviewee) {
        System.out.println("HR[" + this.name + "]面试[" + interviewee.
getName() + "]同学");
        interviewee.setHrOpinion(new Random().nextBoolean());
        if (interviewee.isHrOpinion()) {
            System.out.println("[" + interviewee.getName() + "]同学 H
R 轮面试通过，恭喜拿到 Offer");
        } else {
            System.out.println("[" + interviewee.getName() + "]同学 H
R 轮面试不通过");
        }
    }
}
```

打印结果：

```
组长[老刚]面试[小明]同学
[小明]同学组长轮面试通过
部门经理[老孙]面试[小明]同学
[小明]同学部门经理轮面试通过
HR[老刘]面试[小明]同学
[小明]同学 HR 轮面试通过，恭喜拿到 Offer
```

上面代码打印结果是小明通过层层筛选，通过了面试，拿到了 Offer。下面的图来展现这次面试的流程。



流程图

讲解一下代码，Interviewee 是面试者，对于企业来说这个面试者来面试的过程会有 3 个关键标识，就是 3 位面试官对这位面试者的评价，只有都评价好才能通过面试拿到 Offer，所以 Interviewee 类有 3 位面试官的面试结果。
Interviewer 是面时官，企业中面试官不是一个职位，而是一个工作，帮企业招到合适的人才，所以它是一个抽象类，有一个抽象方法就是去面试应聘者，具

体面试过程又各面试官实现，而因为这个面试会有结果反馈，结果好的会进入下一轮面试，所以会有下一个面试官的对象引用，责任链模式也就在这里体现。TeamLeader、DepartMentManager、HR 则为公司的不同职位，而这些职位当公司需要招聘时，都需要去充当面试官，所以都继承了 Interviewer。这整个过程就构成了责任链模式代码示例，希望在金三银四各位蠢蠢欲动的朋友们都能闯到最后一关拿下 HR 姐姐。

3.4.3. 总结

责任链模式很好的**把处理的逻辑封装起来**，在代码中，我们看到的只是组长面试，但是其实背后隐藏着部门经理和 HR 的面试。责任链是不是看起来很熟悉？在开发 Java Web 项目的时候是不是有用到过？Filter 过滤器里面就是用责任链模式来实现的。上面代码还用到了另一个模式，不明确指出来了，考验大家学习这些设计模式的效果，知道的同学可以留言说一下。

4. 七大结构型模式

4.1. 适配器模式（你用过港式插座转换器么？）

4.1.1. 简介

姓名： 适配器模式

英文名： Adapter Pattern

价值观： 老媒人，牵线搭桥

个人介绍： Convert the interface of a class into another interface clients expect.Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. 将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工作的两个类能够在一起工作。（来自《设计模式之禅》）

4.1.2. 你要的故事

大家有买过港式的 Apple 产品么？在深圳的同学估计买过，毕竟港式的 Apple 产品基本比国内便宜 500 以上。我手机和平板都是在香港买的，买来后这充电器是没法直接充电的，因为港版的电子产品都是英式的插头，而咱们国内是中式的，所以用上港版电子产品的同学免不了要用上这么一个转换器：将英式的插孔转为中式的插孔，方可插入咱家里的插座充电。这个转换器就是今天想讲的适配器。

没见过的同学可以看看图片熟悉一下，下图右边为港版苹果手机充电器，插头比较大，左边为某品牌转换器，插头为中国家用标准形状。

下图为使用时的图片

在这描述一下这个场景。用港式插头要在国内充电，因为插头和插座大小对不上，所以需要加一个适配器，这个适配器充当插头和插座，它的插头可以插入国内标准的插座，它的插座可以插入港式标准的插头，这样子就可以用港式充电器在国内为手机充电。

下面用适配器模式代码实现这个场景。

首先需要找到被适配的对象是什么？在这里我们的被适配对象是英式充电器。

```
/**
 * 英式充电器
 */
class BritishCharger {
```

```
    public void chargeByBritishStandard(){
        System.out.println("用英式充电器充电");
    }

}
```

在这个场景的目的是什么？在中国为港式手机充电，因此目的是让英式充电器能够在中国标准的插座充电。

```
/**
 * 使用中式插座充电
 */
interface Target {

    void chargeByChineseStandard();

}
```

接下来是这个设计模式的主角：适配器。它需要连接中式插座以及英式充电器，在中间做适配功能。

```
/**
 * 充电器适配器
 */
class ChargerAdapter implements Target {

    private BritishCharger britishCharger;

    public ChargerAdapter(BritishCharger britishCharger) {
        this.britishCharger = britishCharger;
    }

    @Override
    public void chargeByChineseStandard() {
        System.out.println("使用中英式插头转换器");
        britishCharger.chargeByBritishStandard();
    }
}
```

上面是适配器模式的一个简单的例子，要学习适配器模式也可以看看 Java 的 IO 实现源码，里面是应用适配器模式的官方很好的代码。

4.1.3. 总结

适配器很好的将 2 个无法关联的类结合起来，在中间起桥梁作用。另外新增适配器代码不会影响原来被适配者的正常使用，他们可以一起被使用。在工作中和外部系统对接的时候，大可能外部系统的数据格式和自己系统的数据格式并不相同，这时候就可以利用适配器模式来实现。

4.2. 桥接模式（IOS、Android 二分天下）

4.2.1. 简介

姓名： 桥接模式

英文名： Bridge Pattern

价值观： 解耦靠我

个人介绍： Decouple an abstraction from its implementation so that the two can vary independently. 将抽象和实现解耦，使得两者可以独立地变化。（来自《设计模式之禅》）

4.2.2. 你要的故事

现在手机二分天下，安卓手机和苹果手机目前占有率高居 98.45%，其中安卓手机占有率为 70.21%，苹果手机占有率为 28.24%，如下图所示。

Platform	Share
Android	70.21%
iOS	28.24%
Unknown	1.29%
Series 40	0.09%
Windows Phone OS	0.09%
Linux	0.03%
RIM OS	0.03%
Symbian	0.01%
Bada	0.01%
Windows Mobile	0.00%

（数据从 netmarketshare 来）

因为有这 2 个系统，所以很多软件商都不得不开发 2 个系统的 APP。我们就拿这个案例来讲，目前手机有安卓手机和苹果手机，软件有谷歌浏览器和火狐浏览器，通过手机打开软件这一过程来讲讲桥接模式。

从个人介绍可见，需要抽象化和实现化，然后使用桥接模式将抽象和实现解耦。

抽象化：把一类对象共有的东西抽象到一个类里面，该类作为这类对象的基类。在这里我们可以抽象化的便是**手机**。

实现化：将接口或抽象类的未实现的方法进行实现。在这里我们可以实现化的就是**软件**。

将抽象和实现解耦：有了上面的抽象化和实现化，通过桥接模式来实现解耦。在这里，我们把打开软件 `open()` 放到软件实现中，而抽象的手机利用模板方法模式定义 `openSoftware()` 供手机子类去实现，手机子类也是调用软件的 `open()` 方法，并没有自己实现打开逻辑，也就是解耦了这个打开软件过程。

下面给出案例的代码。

Phone 手机抽象类代码。属性 system 代表系统名称，software 代表要打开的软件，openSoftware() 对外提供打开软件的方法。

```
abstract class Phone {

    private String system;
    private Software software;

    public abstract void openSoftware();

    public String getSystem() {
        return system;
    }

    public void setSystem(String system) {
        this.system = system;
    }

    public Software getSoftware() {
        return software;
    }

    public void setSoftware(Software software) {
        this.software = software;
    }

}
```

AndroidPhone 安卓系统手机代码。

```
class AndroidPhone extends Phone {

    public AndroidPhone(Software software){
        this.setSystem("Android");
        this.setSoftware(software);
    }

    @Override
    public void openSoftware() {
        this.getSoftware().open(this);
```

```
    }
}
```

IOSPhone IOS 系统手机代码（也就是苹果手机）。

```
class IOSPhone extends Phone {

    public IOSPhone(Software software) {
        this.setSystem("IOS");
        this.setSoftware(software);
    }

    @Override
    public void openSoftware() {
        this.getSoftware().open(this);
    }
}
```

Software 软件接口代码。它有一个方法 open()，用于打开该软件。

```
interface Software {
    void open(Phone phone);
}
```

Chrome 谷歌浏览器软件代码。

```
class Chrome implements Software {

    @Override
    public void open(Phone phone) {
        System.out.println("打开 " + phone.getSystem() + " 手机的 Chr
ome 浏览器");
    }

}
```

FireFox 火狐浏览器软件代码。

```
class FireFox implements Software {

    @Override
    public void open(Phone phone) {
        System.out.println("打开 " + phone.getSystem() + " 手机的 Fir
```

```
efox 浏览器");
    }

}
```

测试代码如下。

```
public class BridgeTest {

    public static void main(String[] args) {
        Software chrome = new Chrome();
        Software firefox = new FireFox();

        Phone androidPhone = new AndroidPhone(chrome);
        androidPhone.openSoftware();

        androidPhone.setSoftware(firefox);
        androidPhone.openSoftware();

        Phone iosPhone = new IOSPhone(chrome);
        iosPhone.openSoftware();

        iosPhone.setSoftware(firefox);
        iosPhone.openSoftware();
    }

}
```

打印结果：

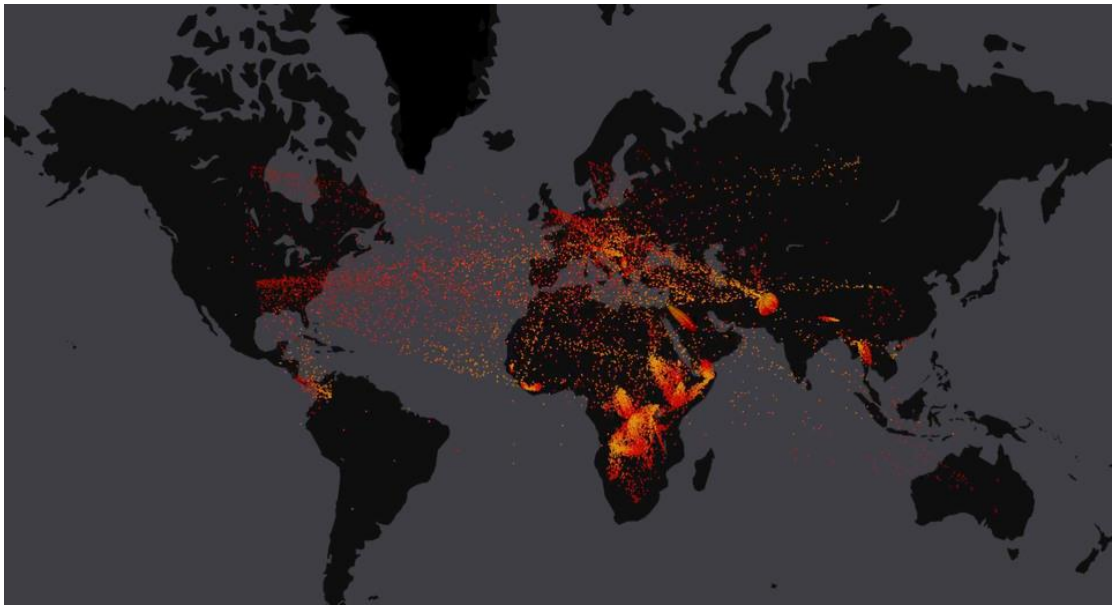
```
打开 Android 手机的 Chrome 浏览器
打开 Android 手机的 Firefox 浏览器
打开 IOS 手机的 Chrome 浏览器
打开 IOS 手机的 Firefox 浏览器
```

桥接模式代码已经写完。为什么叫桥接模式呢？因为它将打开软件的具体实现放到了软件实现里面，而不是放在了手机，通过聚合方式去调用软件打开的方法，这就像一条桥一样连接手机和软件。

4.2.3. 总结

桥接模式利用了聚合的优点去解决继承的缺点，使得抽象和实现进行分离解耦。正由于解耦，使得有更好的扩展性，加手机类型或者加软件都非常容易，也不会破坏原有的代码。

4.3. 组合模式（程序猿组织架构）



4.3.1. 简介

姓名： 组合模式

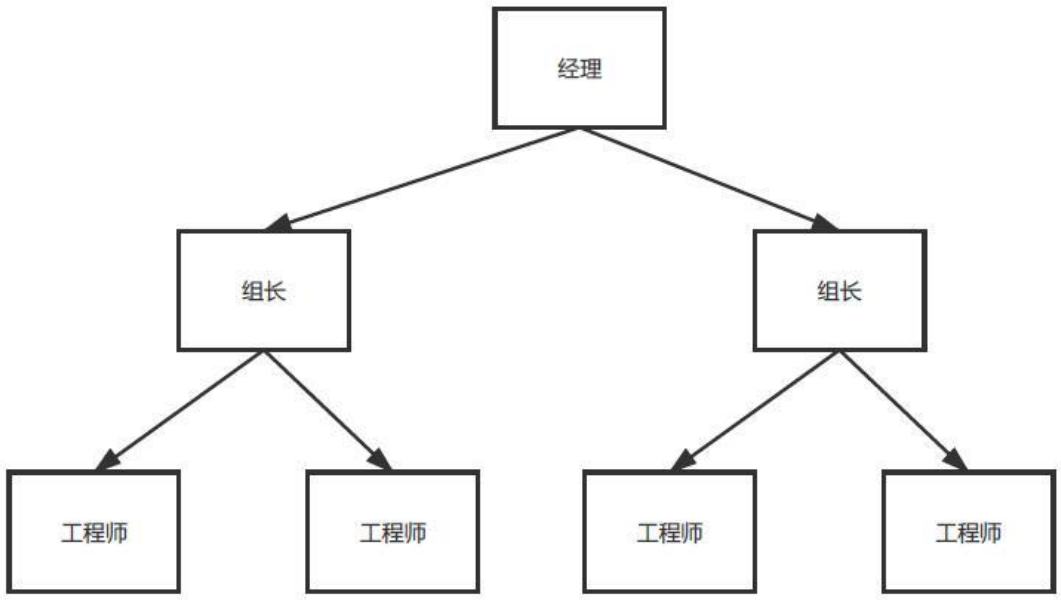
英文名： Composite Pattern

价值观： 专门解决各种树形疑难杂症

个人介绍： Compose objects into tree structures to represent part-whole hierarchies.Composite lets clients treat individual objects and compositions of objects uniformly. 将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。（来自《设计模式之禅》）

4.3.2. 你要的故事

今天咱们再讲讲咱们程序猿的组织架构。技术类的组织架构比较单一，基本上都是这样：经理--->组长--->工程师，如下图所示。



IT 组织架构

各个公司的 title 可能不太一样，但是基本是差不多这种架构，按职业发展，从入职到能独立开发需求便为工程师，从独立开发需求到能带小团队开发便为组长，从带小团队开发到能带几个团队一起协作开发便为经理。

假设目前有一家公司，技术部就 4 个人，大熊担任经理，中熊担任组长，小熊 1 和小熊 2 担任工程师。下面的代码都围绕这个假设编写。

4.3.2.1. 非组合模式

我们先来一个非正常的实现方案：从组织架构里，有 3 个角色，分别是经理、组长、工程师，那么我们就按角色去实现一番。

Manager 为经理类，经理下有多个组长 leaders。

```
/**
 * 经理
 */
class Manager {
```

```
    private String name;
    private List<Leader> leaders;

    public Manager(String name) {
        this.name = name;
        this.leaders = new LinkedList<>();
    }

    public void add(Leader leader) {
        this.leaders.add(leader);
    }

    public void remove(Leader leader) {
        this.leaders.remove(leader);
    }

    public void display(int index) {
        for (int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("经理: " + this.name);
        leaders.forEach(leader -> {
            leader.display(index+1);
        });
    }

}
```

Leader 为组长类，组长下有多个工程师 engineers。

```
/**
 * 组长
 */
class Leader {

    private String name;
    private List<Engineer> engineers;

    public Leader(String name) {
        this.name = name;
```

```
        this.engineers = new LinkedList<>();
    }

    public void add(Engineer engineer) {
        this.engineers.add(engineer);
    }

    public void remove(Engineer engineer) {
        this.engineers.remove(engineer);
    }

    public void display(int index) {
        for (int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("组长：" + this.name);
        engineers.forEach(engineer -> {
            engineer.display(index + 1);
        });
    }
}
```

Engineer 为工程师类，工程师没有下属。

```
/**
 * 工程师
 */
class Engineer {

    private String name;

    public Engineer(String name) {
        this.name = name;
    }

    public void display(int index) {
        for (int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("工程师：" + this.name);
```

```
    }

}
```

测试代码

```
public class NoCompositeTest {

    public static void main(String[] args) {
        Manager manager = new Manager("大熊");
        Leader leader = new Leader("中熊");
        Engineer engineer1= new Engineer("小熊 1");
        Engineer engineer2 = new Engineer("小熊 2");

        manager.add(leader);
        leader.add(engineer1);
        leader.add(engineer2);

        manager.display(0);
    }

}
```


打印结果：

```
经理：大熊
----组长：中熊
--------工程师：小熊 1
--------工程师：小熊 2
```

这份代码看完之后，有什么想法？是不是感觉代码有点冗余？经理和组长的代码几乎一致，而工程师类和经理类、组长类也有共同点，唯一的区别就是工程师没有下属，因此没有对下属的增删操作方法。

4.3.2.2. 安全模式

通过上面一层思考，这 3 个角色有相通性，我们可以抽象出一个 Employee2 类，把 3 个角色共同的特性放到 Employee2 类中，经理和组长合并共用一个类，因为在这个例子里，这 2 个角色完全一样的。下面看代码。

Employee2 抽象类，它有这 3 个角色共有的特性，名称设置获取以及显示数据。

```
abstract class Employee2 {
```

```
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public abstract void display(int index);

}
```

Leader2 领导类，把上面的经理类和组长类都合并到这个领导类，因为他们都是领导层。

```
class Leader2 extends Employee2 {

    private List<Employee2> employees;

    public Leader2(String name) {
        this.setName(name);
        this.employees = new ArrayList<>();
    }

    public void add(Employee2 employee) {
        this.employees.add(employee);
    }

    public void remove(Employee2 employee) {
        this.employees.remove(employee);
    }

    @Override
    public void display(int index) {
        for(int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("领导：" + this.getName());
```

```
        this.employees.forEach(employee -> {
            employee.display(index + 1);
        });
    }
}
```

Engineer2 工程师类，工程师类比较简单，因为名称设置获取在抽象类 Employee2 有了，所以就只需实现显示数据的功能。

```
class Engineer2 extends Employee2 {

    public Engineer2(String name) {
        this.setName(name);
    }

    @Override
    public void display(int index) {
        for(int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("工程师：" + this.getName());
    }
}
```

测试代码

```
public class CompositeTest {

    public static void main(String[] args) {
        // 安全模式
        Leader2 leader1 = new Leader2("大熊");
        Leader2 leader2 = new Leader2("中熊");
        Engineer2 engineer1 = new Engineer2("小熊 1");
        Engineer2 engineer2 = new Engineer2("小熊 2");

        leader1.add(leader2);
        leader2.add(engineer1);
        leader2.add(engineer2);

        leader1.display(0);
    }
```

```
}
```

打印结果：
领导：大熊
----领导：中熊
--------工程师：小熊 1
--------工程师：小熊 2

看下运行结果和上面是一致的，这份代码比第一份代码有更好的封装性，也更符合面向对象的编程方式，经理和组长被合并成 Leader2，也就是咱们今天讲的组合模式，Leader2 为组合对象。上面讲的是安全模式，安全模式指的是抽象类 Employee2 只提供了 3 个角色中共有的特性，安全是相对透明模式所说的，因为这里领导类 Leader2 和工程师类 Engineer2 都只提供了自己能提供的方法，Engineer2 不会有多余的方法，而透明模式则不是。下面讲讲透明模式。

4.3.2.3. 透明模式

透明模式把组合对象（即领导类）使用的方法放到抽象类中，而因为工程师没有下属，则不具体实现对应的方法。代码如下。

Employee3 抽象类，将组合对象的属性 `employees` 和 方法 `add()`、 `remove()` 都放到这个类里面。

```
abstract class Employee3 {

    private String name;
    private List<Employee3> employees;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Employee3> getEmployees() {
        return employees;
    }

    public void setEmployees(List<Employee3> employees) {
```

```
        this.employees = employees;
    }

    public abstract void add(Employee3 employee);

    public abstract void remove(Employee3 employee);

    public abstract void display(int index);

}
```

Leader3 领导类，具体实现 Employee3 提供的所有方法。

```
class Leader3 extends Employee3 {

    public Leader3(String name) {
        this.setName(name);
        this.setEmployees(new ArrayList<>());
    }

    @Override
    public void add(Employee3 employee) {
        this.getEmployees().add(employee);
    }

    @Override
    public void remove(Employee3 employee) {
        this.getEmployees().remove(employee);
    }

    @Override
    public void display(int index) {
        for(int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("领导：" + this.getName());
        this.getEmployees().forEach(employee -> {
            employee.display(index + 1);
        });
```

```
    }
}
```

Engineer3 工程师类，只具体实现 Employee3 中的 display() 方法，add() 和 remove() 方法不是工程师具备的，所以留空，不做具体实现。

```
class Engineer3 extends Employee3 {

    public Engineer3(String name) {
        this.setName(name);
    }

    @Override
    public void add(Employee3 employee) {
        // 没有下属
    }

    @Override
    public void remove(Employee3 employee) {
        // 没有下属
    }

    @Override
    public void display(int index) {
        for(int i = 0; i < index; i++) {
            System.out.print("----");
        }
        System.out.println("工程师：" + this.getName());
    }
}
```

测试代码：

```
public class CompositeTest {

    public static void main(String[] args) {
        // 透明模式
        Leader3 leader3 = new Leader3("大熊");
        Leader3 leader31 = new Leader3("中熊");
        Engineer3 engineer31 = new Engineer3("小熊 1");
        Engineer3 engineer32 = new Engineer3("小熊 2");
```

```
        leader3.add(leader31);
        leader31.add(engineer31);
        leader31.add(engineer32);

        leader3.display(0);

    }
```

打印结果：
领导：大熊
----领导：中熊
--------工程师：小熊 1
--------工程师：小熊 2
}

安全模式把 3 个角色的共同点抽象到 Employee2 中，透明模式则把 3 个角色中的领导者（组合对象）的内容抽象到 Employee3 中。透明模式有些不好的地方在于工程师也有领导者的下属对象和相应的方法，其实工程师并没有这些功能。安全模式把领导者和工程师分开，每个对象都只提供自己具有的功能，这样子在使用的时候也就更安全。

4.3.3. 总结

我们根据 IT 组织架构，从简单的每个角色对应一个类的实现，再到抽象出每个角色共同的功能、组合领导类的安全模式，接着再到抽象起来领导类（组合）所有功能的透明模式，分析了组合模式的完整过程，也讲了安全模式和透明模式的差异。组合模式让对象更加有层次，将对象的划分更加清晰，特别是树形结构的层次，利用组合模式会更加简化。