回首页

设计模式也可以这么简单

创建时间: 2017-10-12 00:00:00

一直想写一篇介绍设计模式的文章，让读者可以很快看完，而且一看就懂，看懂就会用，同时不会将各个模式搞混。自认为本文还是写得不错的😂😂😂，花了不少心思来写这文章和做图，力求让读者真的能看着简单同时有所收获。

设计模式是对大家实际工作中写的各种代码进行高层次抽象的总结，其中最出名的当属 *Gang of Four*（*GoF*）的分类了，他们将设计模式分类为 23 种经典的模式，根据用途我们又可以分为三大类，分别为创建型模式、结构型模式和行为型模式。

有一些重要的设计原则在开篇和大家分享下，这些原则将贯通全文：

- 面向接口编程，而不是面向实现。这个很重要，也是优雅的、可扩展的代码的第一步，这就不需要多说了吧。
- 职责单一原则。每个类都应该只有一个单一的功能，并且该功能应该由这个类完全封装起来。
- 对修改关闭，对扩展开放。对修改关闭是说，我们辛辛苦苦加班写出来的代码，该实现的功能和该修复的 bug 都完成了，别人可不能说改就改；对扩展开放就比较好理解了，也就是说在我们写好的代码基础上，很容易实现扩展。

创建型模式比较简单，但是会比较没有意思，结构型和行为型比较有意思。

创建型模式

创建型模式的作用就是创建对象，说到创建一个对象，最熟悉的就是 new 一个对象，然后 set 相关属性。但是，在很多场景下，我们需要给客户端提供更加友好的创建对象的方式，尤其是那种我们定义了类，但是需要提供给其他开发者用的时候。

简单工厂模式

和名字一样简单，非常简单，直接上代码吧：

```
public class FoodFactory {

    public static Food makeFood(String name) {
```

```
        if (name.equals("noodle")) {
            Food noodle = new LanZhouNoodle();
            noodle.addSpicy("more");
            return noodle;
        } else if (name.equals("chicken")) {
            Food chicken = new HuangMenChicken();
            chicken.addCondiment("potato");
            return chicken;
        } else {
            return null;
        }
    }
}
```

其中，LanZhouNoodle 和 HuangMenChicken 都继承自 Food。

简单地说，简单工厂模式通常就是这样，一个工厂类 XxxFactory，里面有一个静态方法，根据我们不同的参数，返回不同的派生自同一个父类（或实现同一接口）的实例对象。

> 我们强调**职责单一**原则，一个类只提供一种功能，FoodFactory 的功能就是只要负责生产各种 Food。

工厂模式

简单工厂模式很简单，如果它能满足我们的需要，我觉得就不要折腾了。之所以需要引入工厂模式，是因为我们往往需要使用两个或两个以上的工厂。

```
public interface FoodFactory {
    Food makeFood(String name);
}
public class ChineseFoodFactory implements FoodFactory {

    @Override
    public Food makeFood(String name) {
        if (name.equals("A")) {
            return new ChineseFoodA();
        } else if (name.equals("B")) {
            return new ChineseFoodB();
        } else {
```

```
            return null;
        }
    }
}
public class AmericanFoodFactory implements FoodFactory {

    @Override
    public Food makeFood(String name) {
        if (name.equals("A")) {
            return new AmericanFoodA();
        } else if (name.equals("B")) {
            return new AmericanFoodB();
        } else {
            return null;
        }
    }
}
```

其中，ChineseFoodA、ChineseFoodB、AmericanFoodA、AmericanFoodB 都派生自 Food。

客户端调用：

```
public class APP {
    public static void main(String[] args) {
        // 先选择一个具体的工厂
        FoodFactory factory = new ChineseFoodFactory();
        // 由第一步的工厂产生具体的对象，不同的工厂造出不一样的对象
        Food food = factory.makeFood("A");
    }
}
```

虽然都是调用 makeFood("A") 制作 A 类食物，但是，不同的工厂生产出来的完全不一样。

第一步，我们需要选取合适的工厂，然后第二步基本上和简单工厂一样。

核心在于，我们需要在第一步选好我们需要的工厂。比如，我们有 LogFactory 接口，实现类有 FileLogFactory 和 KafkaLogFactory，分别对应将日志写入文件和写入 Kafka 中，显然，我们客

户端第一步就需要决定到底要实例化 FileLogFactory 还是 KafkaLogFactory，这将决定之后的所有的操作。

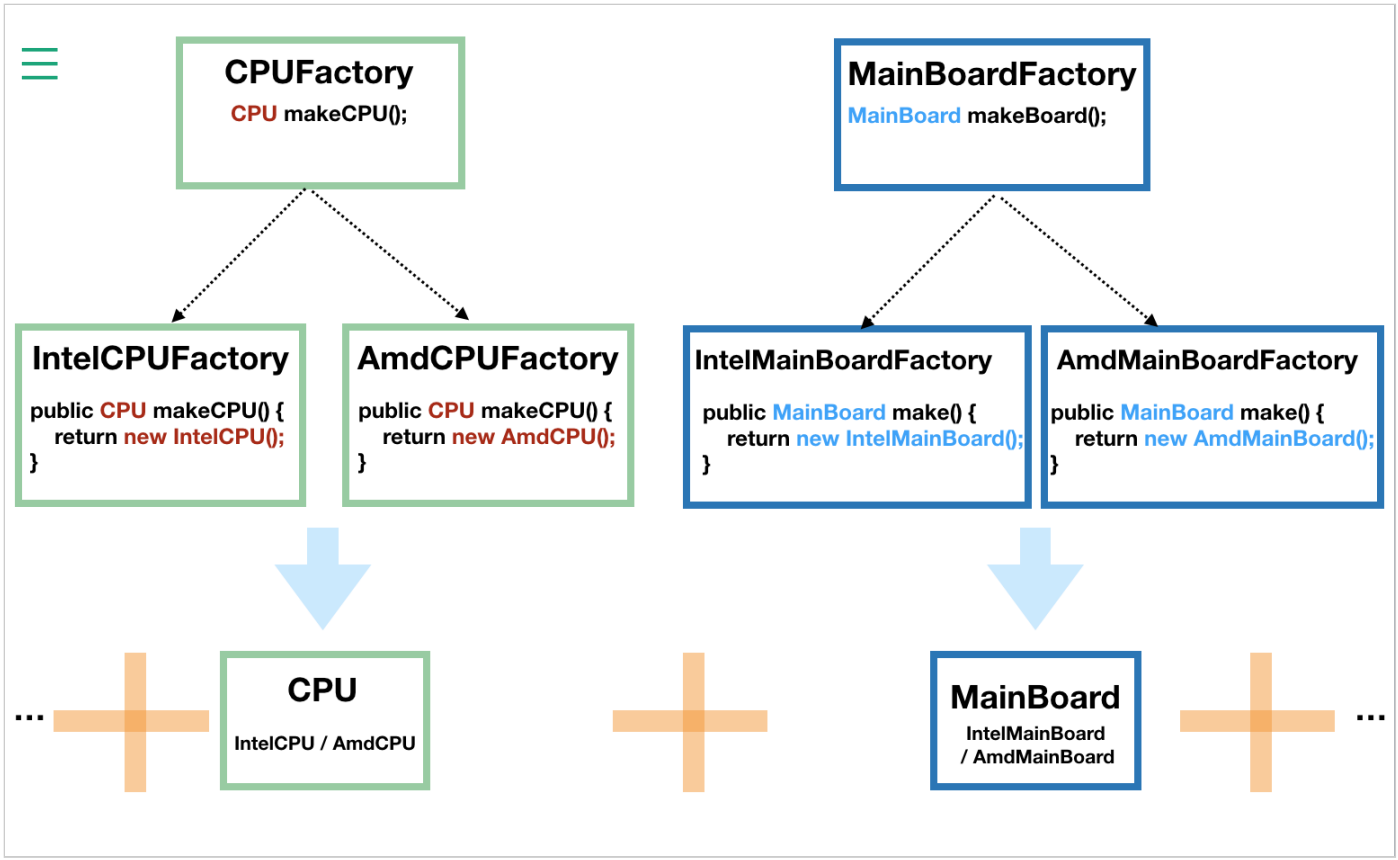
虽然简单，不过我也把所有的构件都画到一张图上，这样读者看着比较清晰：

抽象工厂模式

当涉及到**产品族**的时候，就需要引入抽象工厂模式了。

一个经典的例子是造一台电脑。我们先不引入抽象工厂模式，看看怎么实现。

因为电脑是由许多的构件组成的，我们将 CPU 和主板进行抽象，然后 CPU 由 CPUFactory 生产，主板由 MainBoardFactory 生产，然后，我们再将 CPU 和主板搭配起来组合在一起，如下图：



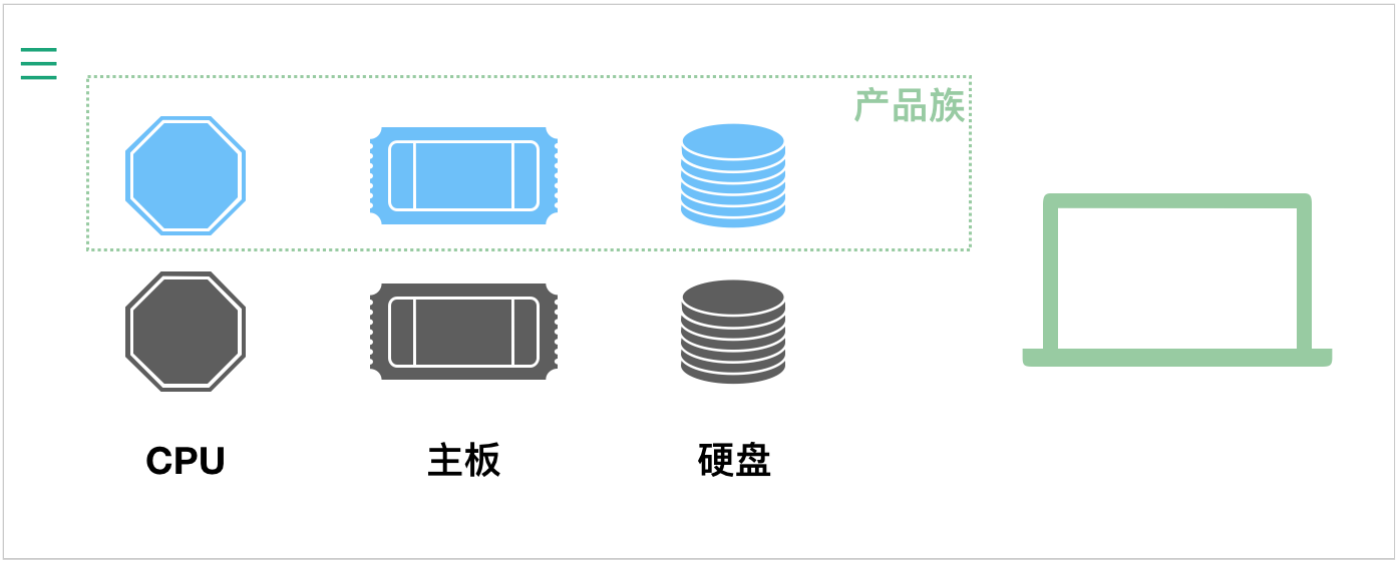
这个时候的客户端调用是这样的：

```
// 得到 Intel 的 CPU
CPUFactory cpuFactory = new IntelCPUFactory();
CPU cpu = intelCPUFactory.makeCPU();

// 得到 AMD 的主板
MainBoardFactory mainBoardFactory = new AmdMainBoardFactory();
MainBoard mainBoard = mainBoardFactory.make();

// 组装 CPU 和主板
Computer computer = new Computer(cpu, mainBoard);
```

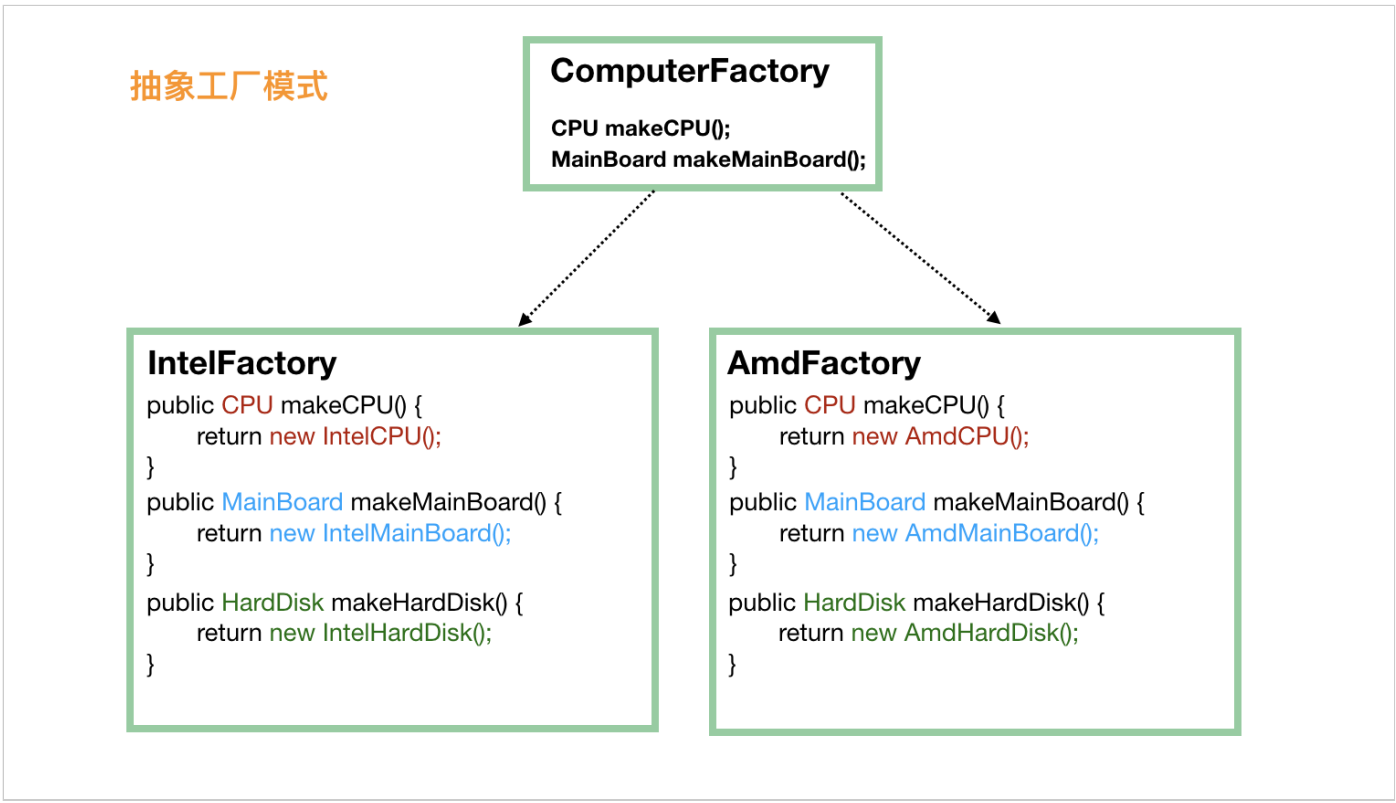
单独看 CPU 工厂和主板工厂，它们分别是前面我们说的**工厂模式**。这种方式也容易扩展，因为要给电脑加硬盘的话，只需要加一个 HardDiskFactory 和相应的实现即可，不需要修改现有的工厂。

但是，这种方式有一个问题，那就是如果 **Intel 家产的 CPU 和 AMD 产的主板不能兼容使用**，那么这代码就容易出错，因为客户端并不知道它们不兼容，也就会错误地出现随意组合。

下面就是我们要说的**产品族**的概念，它代表了组成某个产品的一系列附件的集合：



当涉及到这种产品族的问题的时候，就需要抽象工厂模式来支持了。我们不再定义 CPU 工厂、主板工厂、硬盘工厂、显示屏工厂等等，我们直接定义电脑工厂，每个电脑工厂负责生产所有的设备，这样能保证肯定不存在兼容问题。



这个时候，对于客户端来说，不再需要单独挑选 CPU厂商、主板厂商、硬盘厂商等，直接选择一家品牌工厂，品牌工厂会负责生产所有的东西，而且能保证肯定是兼容可用的。

```
public static void main(String[] args) {
    // 第一步就要选定一个“大厂”
    ComputerFactory cf = new AmdFactory();
    // 从这个大厂造 CPU
    CPU cpu = cf.makeCPU();
```

```
        // 从这个大厂造主板
        MainBoard board = cf.makeMainBoard();
        // 从这个大厂造硬盘
        HardDisk hardDisk = cf.makeHardDisk();

        // 将同一个厂子出来的 CPU、主板、硬盘组装在一起
        Computer result = new Computer(cpu, board, hardDisk);
    }
```

当然，抽象工厂的问题也是显而易见的，比如我们要加个显示器，就需要修改所有的工厂，给所有的工厂都加上制造显示器的方法。这有点违反了**对修改关闭，对扩展开放**这个设计原则。

单例模式

单例模式用得最多，错得最多。

饿汉模式最简单：

```
public class Singleton {
    // 首先，将 new Singleton() 堵死
    private Singleton() {};
    // 创建私有静态实例，意味着这个类第一次使用的时候就会进行创建
    private static Singleton instance = new Singleton();

    public static Singleton getInstance() {
        return instance;
    }
    // 瞎写一个静态方法。这里想说的是，如果我们只是要调用 Singleton.getDate(...)，
    // 本来是不想要生成 Singleton 实例的，不过没办法，已经生成了
    public static Date getDate(String mode) {return new Date();}
}
```

> 很多人都能说出饿汉模式的缺点，可是我觉得生产过程中，很少碰到这种情况：你定义了一个单例的类，不需要其实例，可是你却把一个或几个你会用到的静态方法塞到这个类中。

饱汉模式最容易出错：

```
public class Singleton {
    // 首先，也是先堵死 new Singleton() 这条路
```

```
    private Singleton() {}
    // 和饿汉模式相比，这边不需要先实例化出来，注意这里的 volatile，它是必须的
    private static volatile Singleton instance = null;

    public static Singleton getInstance() {
        if (instance == null) {
            // 加锁
            synchronized (Singleton.class) {
                // 这一次判断也是必须的，不然会有并发问题
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

> 双重检查，指的是两次检查 instance 是否为 null。
>
> volatile 在这里是需要的，希望能引起读者的关注。
>
> 很多人不知道怎么写，直接就在 getInstance() 方法签名上加上 synchronized，这就不多说了，性能太差。

嵌套类最经典，以后大家就用它吧：

```
public class Singleton3 {

    private Singleton3() {}
    // 主要是使用了 嵌套类可以访问外部类的静态属性和静态方法 的特性
    private static class Holder {
        private static Singleton3 instance = new Singleton3();
    }
    public static Singleton3 getInstance() {
        return Holder.instance;
    }
}
```

> 注意，很多人都会把这个**嵌套类**说成是**静态内部类**，严格地说，内部类和嵌套类是不一样的，它们能访问的外部类权限也是不一样的。

最后，我们说一下枚举，枚举很特殊，它在类加载的时候会初始化里面的所有的实例，而且 JVM 保证了它们不会再被实例化，所以它天生就是单例的。

虽然我们平时很少看到用枚举来实现单例，但是在 RxJava 的源码中，有很多地方都用了枚举来实现单例。

建造者模式

经常碰见的 XxxBuilder 的类，通常都是建造者模式的产物。建造者模式其实有很多的变种，但是对于客户端来说，我们的使用通常都是一个模式的：

```
Food food = new FoodBuilder().a().b().c().build();
Food food = Food.builder().a().b().c().build();
```

套路就是先 new 一个 Builder，然后可以链式地调用一堆方法，最后再调用一次 build() 方法，我们需要的对象就有了。

来一个中规中矩的建造者模式：

```
class User {
    // 下面是“一堆”的属性
    private String name;
    private String password;
    private String nickName;
    private int age;

    // 构造方法私有化，不然客户端就会直接调用构造方法了
    private User(String name, String password, String nickName, int age) {
        this.name = name;
        this.password = password;
        this.nickName = nickName;
        this.age = age;
    }
    // 静态方法，用于生成一个 Builder，这个不一定要有，不过写这个方法是一个很好的习惯，
    // 有些代码要求别人写 new User.UserBuilder().a()...build() 看上去就没那么好
    public static UserBuilder builder() {
```

```
        return new UserBuilder();
    }

    public static class UserBuilder {
        // 下面是和 User 一模一样的一堆属性
        private String  name;
        private String password;
        private String nickName;
        private int age;

        private UserBuilder() {
        }

        // 链式调用设置各个属性值，返回 this，即 UserBuilder
        public UserBuilder name(String name) {
            this.name = name;
            return this;
        }

        public UserBuilder password(String password) {
            this.password = password;
            return this;
        }

        public UserBuilder nickName(String nickName) {
            this.nickName = nickName;
            return this;
        }

        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }

        // build() 方法负责将 UserBuilder 中设置好的属性“复制”到 User 中。
        // 当然，可以在 “复制” 之前做点检验
        public User build() {
            if (name == null || password == null) {
```

```
                throw new RuntimeException("用户名和密码必填");
            }
            if (age <= 0 || age >= 150) {
                throw new RuntimeException("年龄不合法");
            }
            // 还可以做赋予"默认值"的功能
              if (nickName == null) {
                nickName = name;
            }
            return new User(name, password, nickName, age);
        }
    }
}
```

核心是：先把所有的属性都设置给 Builder，然后 build() 方法的时候，将这些属性**复制**给实际产生的对象。

看看客户端的调用：

```
public class APP {
    public static void main(String[] args) {
        User d = User.builder()
                .name("foo")
                .password("pAss12345")
                .age(25)
                .build();
    }
}
```

说实话，建造者模式的**链式**写法很吸引人，但是，多写了很多“无用”的 builder 的代码，感觉这个模式没什么用。不过，当属性很多，而且有些必填，有些选填的时候，这个模式会使代码清晰很多。我们可以在 **Builder 的构造方法**中强制让调用者提供必填字段，还有，在 build() 方法中校验各个参数比在 User 的构造方法中校验，代码要优雅一些。

> 题外话，强烈建议读者使用 lombok，用了 lombok 以后，上面的一大堆代码会变成如下这样:

```
@Builder
class User {
```

```
    private String  name;
    private String password;
    private String nickName;
    private int age;
}
```

> 怎么样，省下来的时间是不是又可以干点别的了。

当然，如果你只是想要链式写法，不想要建造者模式，有个很简单的办法，User 的 getter 方法不变，所有的 setter 方法都让其 **return this** 就可以了，然后就可以像下面这样调用：

```
User user = new User().setName("").setPassword("").setAge(20);
```

> 很多人是这么用的，但是笔者觉得其实这种写法非常地不优雅，不是很推荐使用。

原型模式

这是我要说的创建型模式的最后一个设计模式了。

原型模式很简单：有一个原型**实例**，基于这个原型实例产生新的实例，也就是“克隆”了。

Object 类中有一个 clone() 方法，它用于生成一个新的对象，当然，如果我们要调用这个方法，java 要求我们的类必须先**实现 Cloneable 接口**，此接口没有定义任何方法，但是不这么做的话，在 clone() 的时候，会抛出 CloneNotSupportedException 异常。

```
protected native Object clone() throws CloneNotSupportedException;
```

> java 的克隆是浅克隆，碰到对象引用的时候，克隆出来的对象和原对象中的引用将指向同一个对象。通常实现深克隆的方法是将对象进行序列化，然后再进行反序列化。

原型模式了解到这里我觉得就够了，各种变着法子说这种代码或那种代码是原型模式，没什么意义。

创建型模式总结

创建型模式总体上比较简单，它们的作用就是为了产生实例对象，算是各种工作的第一步了，因为我们写的是**面向对象**的代码，所以我们第一步当然是需要创建一个对象了。

简单工厂模式最简单；工厂模式在简单工厂模式的基础上增加了选择工厂的维度，需要第一步选择合适的工厂；抽象工厂模式有产品族的概念，如果各个产品是存在兼容性问题的，就要用抽象工厂模式。单例模式就不说了，为了保证全局使用的是同一对象，一方面是安全性考虑，一方面是为了节省资源；建造者模式专门对付属性很多的那种类，为了让代码更优美；原型模式用得最少，了解和 Object 类中的 clone() 方法相关的知识即可。

结构型模式

前面创建型模式介绍了创建对象的一些设计模式，这节介绍的结构型模式旨在通过改变代码结构来达到解耦的目的，使得我们的代码容易维护和扩展。

代理模式

第一个要介绍的代理模式是最常使用的模式之一了，用一个代理来隐藏具体实现类的实现细节，通常还用于在真实的实现的前后添加一部分逻辑。

既然说是**代理**，那就要对客户端隐藏真实实现，由代理来负责客户端的所有请求。当然，代理只是个代理，它不会完成实际的业务逻辑，而是一层皮而已，但是对于客户端来说，它必须表现得就是客户端需要的真实实现。

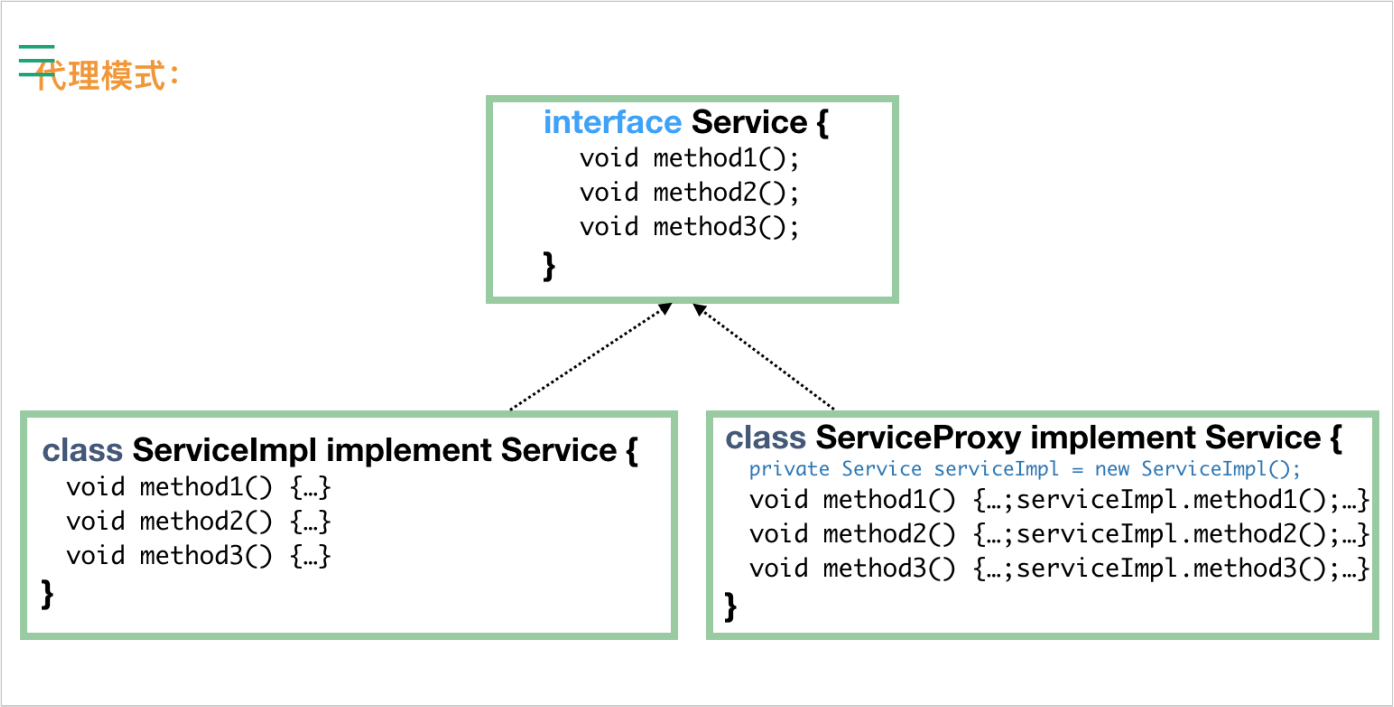
> 理解**代理**这个词，这个模式其实就简单了。

```
public interface FoodService {
    Food makeChicken();
    Food makeNoodle();
}

public class FoodServiceImpl implements FoodService {
    public Food makeChicken() {
          Food f = new Chicken()
        f.setChicken("1kg");
          f.setSpicy("1g");
          f.setSalt("3g");
        return f;
    }
    public Food makeNoodle() {
        Food f = new Noodle();
        f.setNoodle("500g");
        f.setSalt("5g");
```

```
        return f;
    }
}

// 代理要表现得“就像是”真实实现类，所以需要实现 FoodService
public class FoodServiceProxy implements FoodService {

    // 内部一定要有一个真实的实现类，当然也可以通过构造方法注入
    private FoodService foodService = new FoodServiceImpl();

    public Food makeChicken() {
        System.out.println("我们马上要开始制作鸡肉了");

        // 如果我们定义这句为核心代码的话，那么，核心代码是真实实现类做的，
        // 代理只是在核心代码前后做些“无足轻重”的事情
        Food food = foodService.makeChicken();

        System.out.println("鸡肉制作完成啦，加点胡椒粉"); // 增强
          food.addCondiment("pepper");

        return food;
    }
    public Food makeNoodle() {
        System.out.println("准备制作拉面~");
        Food food = foodService.makeNoodle();
        System.out.println("制作完成啦")
        return food;
    }
}
```

客户端调用，注意，我们要用代理来实例化接口：

```
// 这里用代理类来实例化
FoodService foodService = new FoodServiceProxy();
foodService.makeChicken();
```



我们发现没有，代理模式说白了就是做“**方法包装**”或做“**方法增强**”。在面向切面编程中，其实就是动态代理的过程。比如 Spring 中，我们自己不定义代理类，但是 Spring 会帮我们动态来定义代理，然后把我们定义在 @Before、@After、@Around 中的代码逻辑动态添加到代理中。

说到动态代理，又可以展开说，Spring 中实现动态代理有两种，一种是如果我们的类定义了接口，如 UserService 接口和 UserServiceImpl 实现，那么采用 JDK 的动态代理，感兴趣的读者可以去看看 java.lang.reflect.Proxy 类的源码；另一种是我们自己没有定义接口的，Spring 会采用 CGLIB 进行动态代理，它是一个 jar 包，性能还不错。

适配器模式

说完代理模式，说适配器模式，是因为它们很相似，这里可以做个比较。

适配器模式做的就是，有一个接口需要实现，但是我们现成的对象都不满足，需要加一层适配器来进行适配。

适配器模式总体来说分三种：默认适配器模式、对象适配器模式、类适配器模式。先不急着分清楚这几个，先看看例子再说。

默认适配器模式

首先，我们先看看最简单的适配器模式**默认适配器模式(Default Adapter)**是怎么样的。

我们用 Appache commons-io 包中的 FileAlterationListener 做例子，此接口定义了很多的方法，用于对文件或文件夹进行监控，一旦发生了对应的操作，就会触发相应的方法。

```
public interface FileAlterationListener {
    void onStart(final FileAlterationObserver observer);
    void onDirectoryCreate(final File directory);
    void onDirectoryChange(final File directory);
    void onDirectoryDelete(final File directory);
    void onFileCreate(final File file);
    void onFileChange(final File file);
    void onFileDelete(final File file);
    void onStop(final FileAlterationObserver observer);
}
```

此接口的一大问题是抽象方法太多了，如果我们要用这个接口，意味着我们要实现每一个抽象方法，如果我们只是想要监控文件夹中的**文件创建**和**文件删除**事件，可是我们还是不得不实现所有的方法，很明显，这不是我们想要的。

所以，我们需要下面的一个**适配器**，它用于实现上面的接口，但是**所有的方法都是空方法**，这样，我们就可以转而定义自己的类来继承下面这个类即可。

```
public class FileAlterationListenerAdaptor implements FileAlterationListener {

    public void onStart(final FileAlterationObserver observer) {
    }

    public void onDirectoryCreate(final File directory) {

    }

    public void onDirectoryChange(final File directory) {
    }

    public void onDirectoryDelete(final File directory) {
    }

    public void onFileCreate(final File file) {
    }

    public void onFileChange(final File file) {
    }
```

```
    public void onFileDelete(final File file) {
    }

    public void onStop(final FileAlterationObserver observer) {
    }
}
```

比如我们可以定义以下类，我们仅仅需要实现我们想实现的方法就可以了：

```
public class FileMonitor extends FileAlterationListenerAdaptor {
    public void onFileCreate(final File file) {
        // 文件创建
        doSomething();
    }

    public void onFileDelete(final File file) {
        // 文件删除
        doSomething();
    }
}
```

当然，上面说的只是适配器模式的其中一种，也是最简单的一种，无需多言。下面，再介绍**“正统的”**适配器模式。

对象适配器模式

来看一个《Head First 设计模式》中的一个例子，我稍微修改了一下，看看怎么将鸡适配成鸭，这样鸡也能当鸭来用。因为，现在鸭这个接口，我们没有合适的实现类可以用，所以需要适配器。

```
public interface Duck {
    public void quack(); // 鸭的呱呱叫
    public void fly(); // 飞
}

public interface Cock {
    public void gobble(); // 鸡的咕咕叫
    public void fly(); // 飞
```

```
}

public class WildCock implements Cock {
    public void gobble() {
        System.out.println("咕咕叫");
    }
    public void fly() {
        System.out.println("鸡也会飞哦");
    }
}
```

鸭接口有 fly() 和 quare() 两个方法，鸡 Cock 如果要冒充鸭，fly() 方法是现成的，但是鸡不会鸭的呱呱叫，没有 quack() 方法。这个时候就需要适配了：

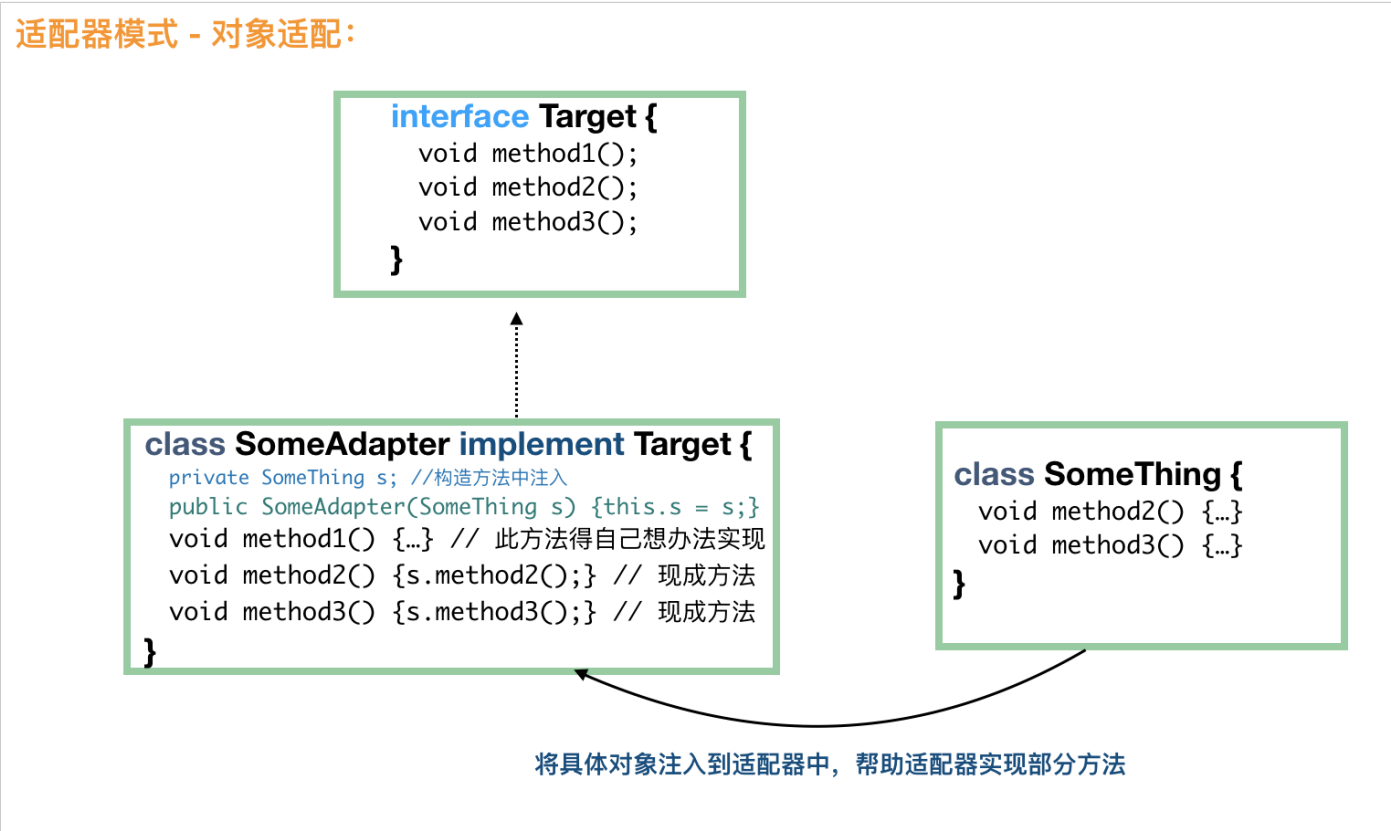
```
// 毫无疑问，首先，这个适配器肯定需要 implements Duck，这样才能当做鸭来用
public class CockAdapter implements Duck {

    Cock cock;
    // 构造方法中需要一个鸡的实例，此类就是将这只鸡适配成鸭来用
      public CockAdapter(Cock cock) {
        this.cock = cock;
    }

    // 实现鸭的呱呱叫方法
    @Override
      public void quack() {
        // 内部其实是一只鸡的咕咕叫
        cock.gobble();
    }

      @Override
      public void fly() {
        cock.fly();
    }
}
```

客户端调用很简单了：

```
public static void main(String[] args) {
    // 有一只野鸡
    Cock wildCock = new WildCock();
    // 成功将野鸡适配成鸭
    Duck duck = new CockAdapter(wildCock);
    ...
}
```

到这里，大家也就知道了适配器模式是怎么回事了。无非是我们需要一只鸭，但是我们只有一只鸡，这个时候就需要定义一个适配器，由这个适配器来充当鸭，但是适配器里面的方法还是由鸡来实现的。

我们用一个图来简单说明下：

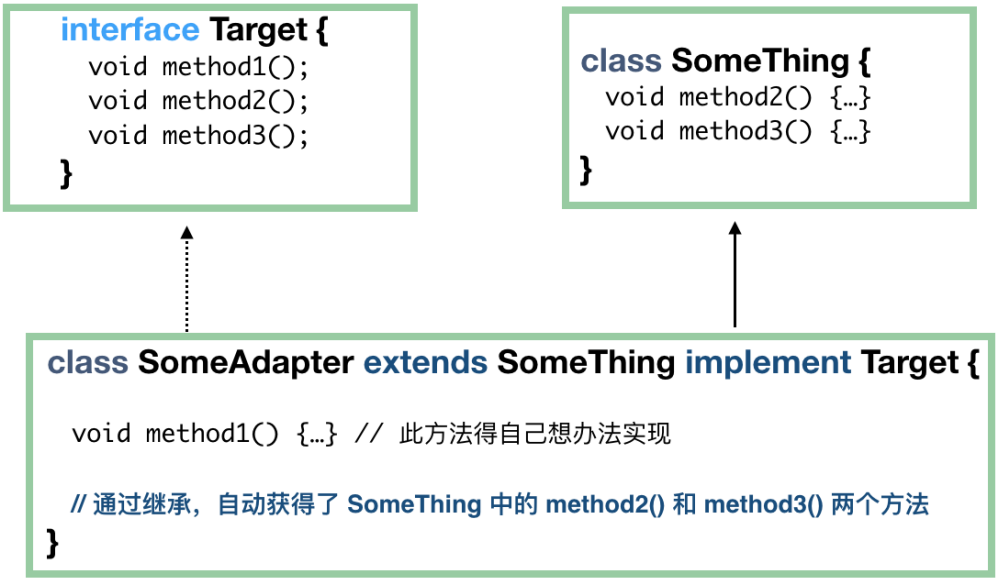


上图应该还是很容易理解的，我就不做更多的解释了。下面，我们看看类适配模式怎么样的。

类适配器模式

废话少说，直接上图：

适配器模式 - 类继承：



看到这个图，大家应该很容易理解的吧，通过继承的方法，适配器自动获得了所需要的大部分方法。这个时候，客户端使用更加简单，直接 `Target t = new SomeAdapter();` 就可以了。

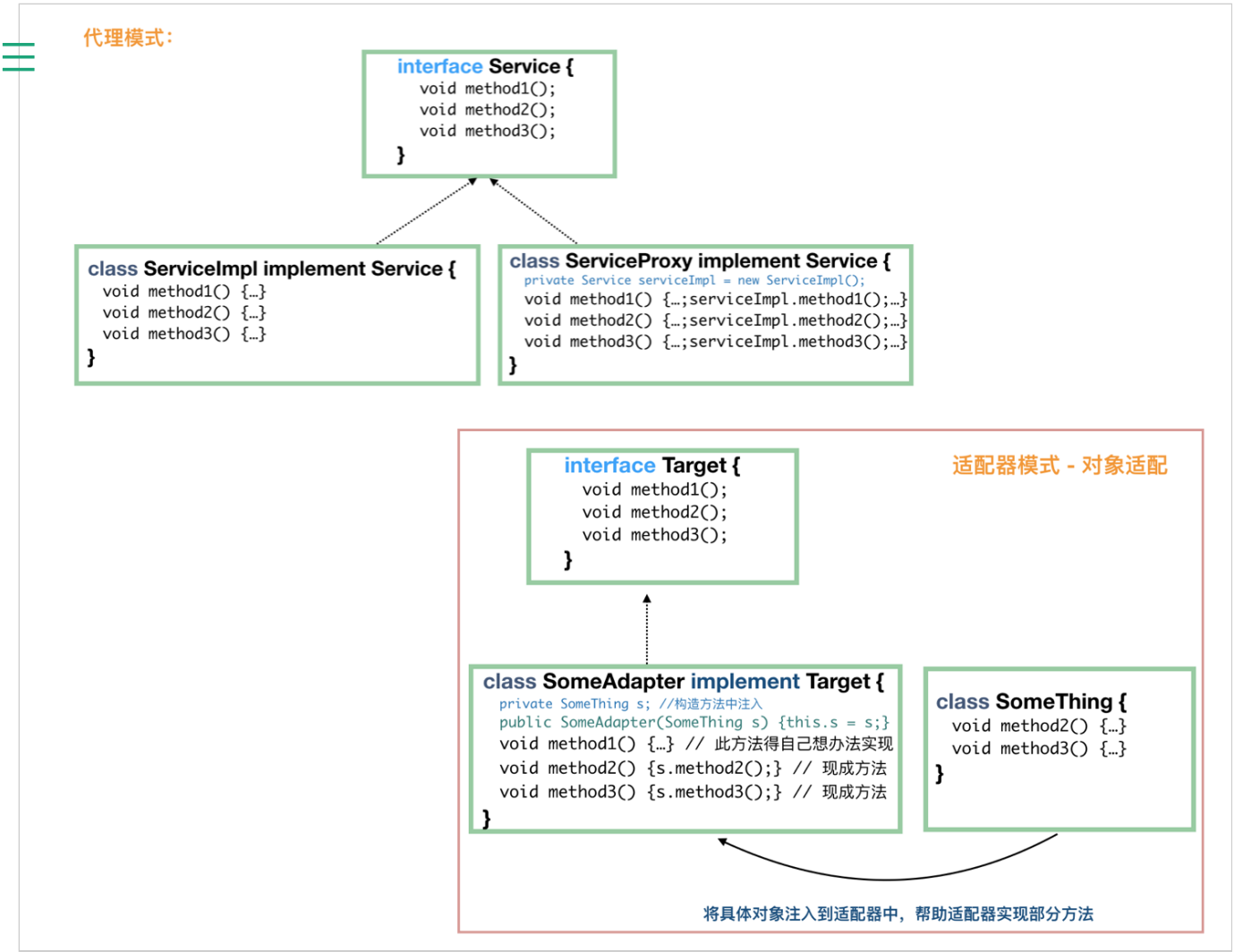
适配器模式总结

- 类适配和对象适配的异同

> 一个采用继承，一个采用组合；
>
> 类适配属于静态实现，对象适配属于组合的动态实现，对象适配需要多实例化一个对象。
>
> 总体来说，对象适配用得比较多。

- 适配器模式和代理模式的异同

比较这两种模式，其实是比较对象适配器模式和代理模式，在代码结构上，它们很相似，都需要一个具体的实现类的实例。但是它们的目的不一样，代理模式做的是增强原方法的活；适配器做的是适配的活，为的是提供“把鸡包装成鸭，然后当做鸭来使用”，而鸡和鸭它们之间原本没有继承关系。



桥梁模式

理解桥梁模式，其实就是理解代码抽象和解耦。

我们首先需要一个桥梁，它是一个接口，定义提供的接口方法。

```
public interface DrawAPI {
   public void draw(int radius, int x, int y);
}
```

然后是一系列实现类：

```
public class RedPen implements DrawAPI {
    @Override
    public void draw(int radius, int x, int y) {
        System.out.println("用红色笔画图, radius:" + radius + ", x:" + x + ", y:"
    }
}
```

```
public class GreenPen implements DrawAPI {
    @Override
    public void draw(int radius, int x, int y) {
        System.out.println("用绿色笔画图，radius:" + radius + ", x:" + x + ", y:"
    }
}
public class BluePen implements DrawAPI {
    @Override
    public void draw(int radius, int x, int y) {
        System.out.println("用蓝色笔画图，radius:" + radius + ", x:" + x + ", y:"


    }
}
```

定义一个抽象类，此类的实现类都需要使用 DrawAPI：

```
public abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI) {
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}
```

定义抽象类的子类：

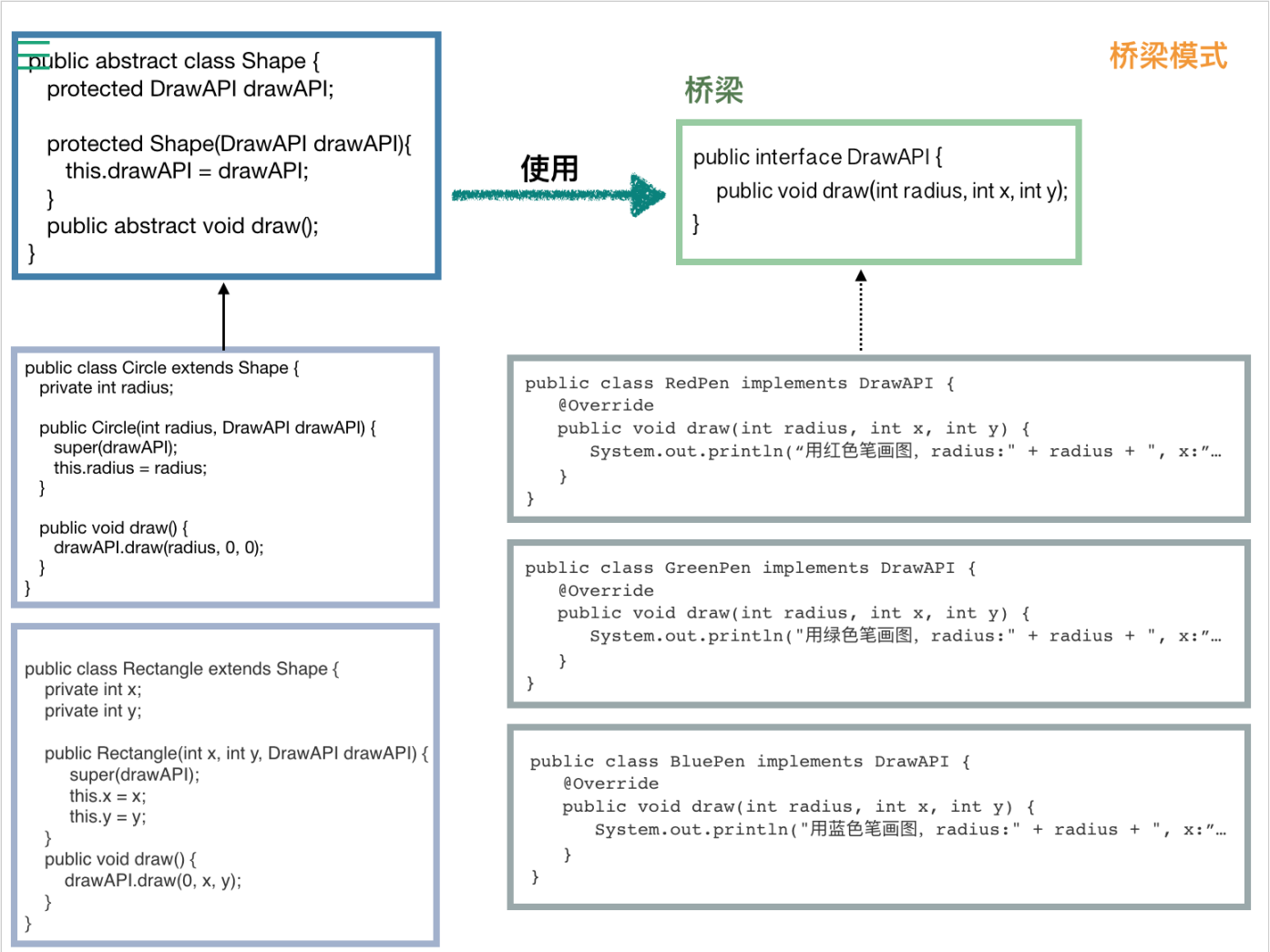
```
// 圆形
public class Circle extends Shape {
    private int radius;
    public Circle(int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.radius = radius;
    }
    public void draw() {
        drawAPI.draw(radius, 0, 0);
    }
}
```

```
// 长方形
public class Rectangle extends Shape {
    private int x;
    private int y;
    public Rectangle(int x, int y, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
    }
    public void draw() {
        drawAPI.draw(0, x, y);
    }
}
```

最后，我们来看客户端演示：

```
public static void main(String[] args) {
    Shape greenCircle = new Circle(10, new GreenPen());
    Shape redRectangle = new Rectangle(4, 8, new RedPen());
    greenCircle.draw();
    redRectangle.draw();
}
```

可能大家看上面一步步还不是特别清晰，我把所有的东西整合到一张图上：



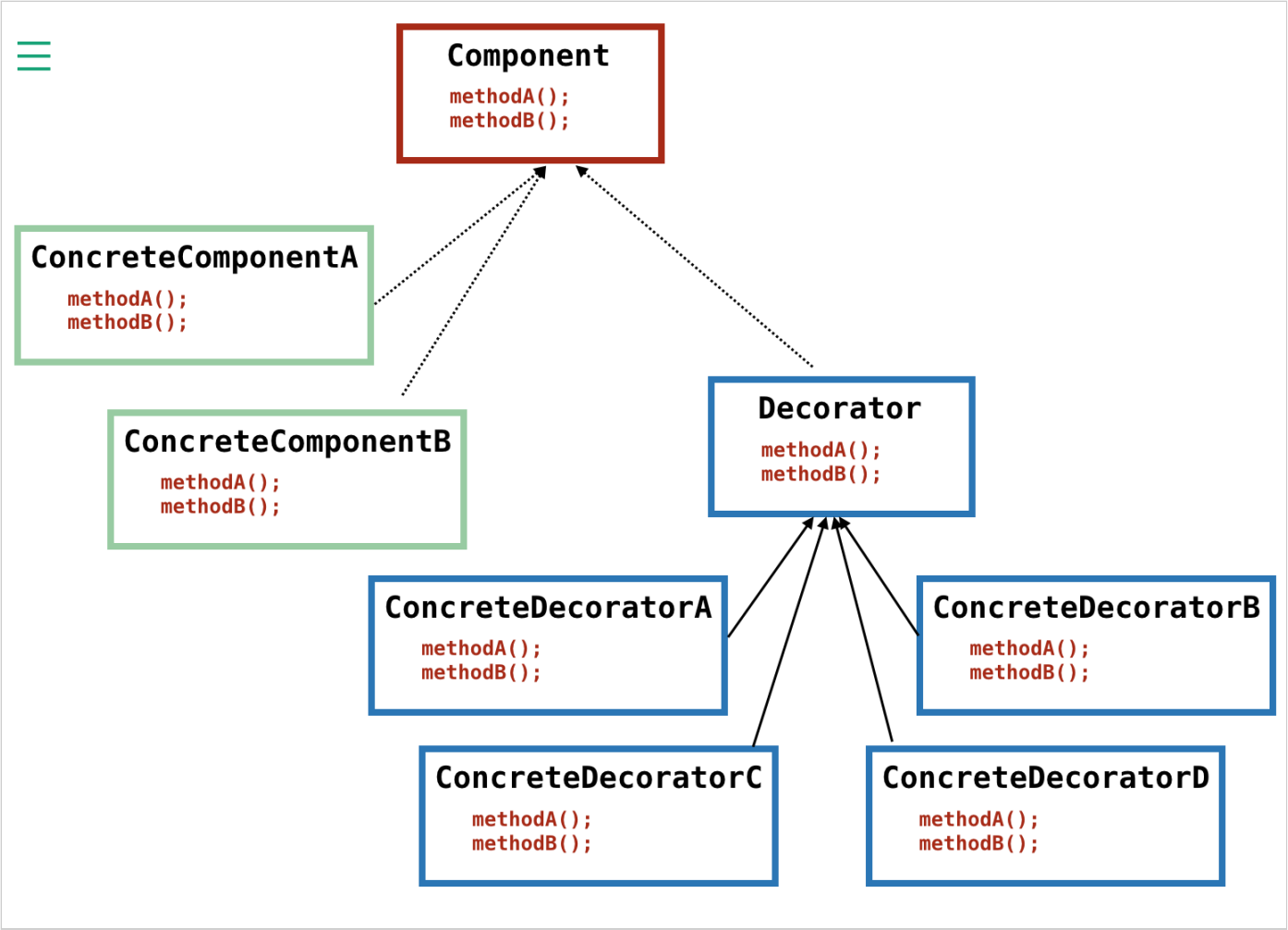
这回大家应该就知道抽象在哪里，怎么解耦了吧。桥梁模式的优点也是显而易见的，就是非常容易进行扩展。

> 本节引用了这里的例子，并对其进行了修改。

装饰模式

要把装饰模式说清楚明白，不是件容易的事情。也许读者知道 Java IO 中的几个类是典型的装饰模式的应用，但是读者不一定清楚其中的关系，也许看完就忘了，希望看完这节后，读者可以对其有更深的感悟。

首先，我们先看一个简单的图，看这个图的时候，了解下层次结构就可以了：



我们来说说装饰模式的出发点，从图中可以看到，接口 Component 其实已经有了 ConcreteComponentA 和 ConcreteComponentB 两个实现类了，但是，如果我们要**增强**这两个实现类的话，我们就可以采用装饰模式，用具体的装饰器来**装饰**实现类，以达到增强的目的。

> 从名字来简单解释下装饰器。既然说是装饰，那么往往就是**添加小功能**这种，而且，我们要满足可以添加多个小功能。最简单的，代理模式就可以实现功能的增强，但是代理不容易实现多个功能的增强，当然你可以说用代理包装代理的多层包装方式，但是那样的话代码就复杂了。

首先明白一些简单的概念，从图中我们看到，所有的具体装饰者们 ConcreteDecorator* 都可以作为 Component 来使用，因为它们都实现了 Component 中的所有接口。它们和 Component 实现类 ConcreteComponent* 的区别是，它们只是装饰者，起**装饰**作用，也就是即使它们看上去牛逼轰轰，但是它们都只是在具体的实现中**加了层皮来装饰**而已。

> 注意这段话中混杂在各个名词中的 Component 和 Decorator，别搞混了。

下面来看看一个例子，先把装饰模式弄清楚，然后再介绍下 java io 中的装饰模式的应用。

最近大街上流行起来了“快乐柠檬”，我们把快乐柠檬的饮料分为三类：红茶、绿茶、咖啡，在这三大类的基础上，又增加了许多的口味，什么金桔柠檬红茶、金桔柠檬珍珠绿茶、芒果红茶、芒果绿

茶、芒果珍珠红茶、烤珍珠红茶、烤珍珠芒果绿茶、椰香胚芽咖啡、焦糖可可咖啡等等，每家店都有很长的菜单，但是仔细看下，其实原料也没几样，但是可以搭配出很多组合，如果顾客需要，很多没出现在菜单中的饮料他们也是可以做的。

在这个例子中，红茶、绿茶、咖啡是最基础的饮料，其他的像金桔柠檬、芒果、珍珠、椰果、焦糖等都属于装饰用的。当然，在开发中，我们确实可以像门店一样，开发这些类：LemonBlackTea、LemonGreenTea、MangoBlackTea、MangoLemonGreenTea.....但是，很快我们就发现，这样子干肯定是不行的，这会导致我们需要组合出所有的可能，而且如果客人需要在红茶中加双份柠檬怎么办？三份柠檬怎么办？

不说废话了，上代码。

首先，定义饮料抽象基类：

```
public abstract class Beverage {
      // 返回描述
      public abstract String getDescription();
      // 返回价格
      public abstract double cost();
}
```

然后是三个基础饮料实现类，红茶、绿茶和咖啡：

```
public class BlackTea extends Beverage {
      public String getDescription() {
        return "红茶";
    }
      public double cost() {
        return 10;
    }
}
public class GreenTea extends Beverage {
    public String getDescription() {
        return "绿茶";
    }
      public double cost() {
        return 11;
    }
```

```
}
...// 咖啡省略
```

定义调料，也就是装饰者的基类，此类必须继承自 Beverage：

```
// 调料
public abstract class Condiment extends Beverage {

}
```

然后我们来定义柠檬、芒果等具体的调料，它们属于装饰者，毫无疑问，这些调料肯定都需要继承调料 Condiment 类：

```
public class Lemon extends Condiment {
    private Beverage bevarage;
    // 这里很关键，需要传入具体的饮料，如需要传入没有被装饰的红茶或绿茶，
    // 当然也可以传入已经装饰好的芒果绿茶，这样可以做芒果柠檬绿茶
    public Lemon(Beverage bevarage) {
        this.bevarage = bevarage;
    }
    public String getDescription() {
        // 装饰
        return bevarage.getDescription() + ", 加柠檬";
    }
    public double cost() {
        // 装饰
        return beverage.cost() + 2; // 加柠檬需要 2 元
    }
}

public class Mango extends Condiment {
    private Beverage bevarage;
    public Mango(Beverage bevarage) {
        this.bevarage = bevarage;
    }
    public String getDescription() {
        return bevarage.getDescription() + ", 加芒果";
```

```
    }
    public double cost() {
        return beverage.cost() + 3; // 加芒果需要 3 元
    }
}
...// 给每一种调料都加一个类
```

看客户端调用：

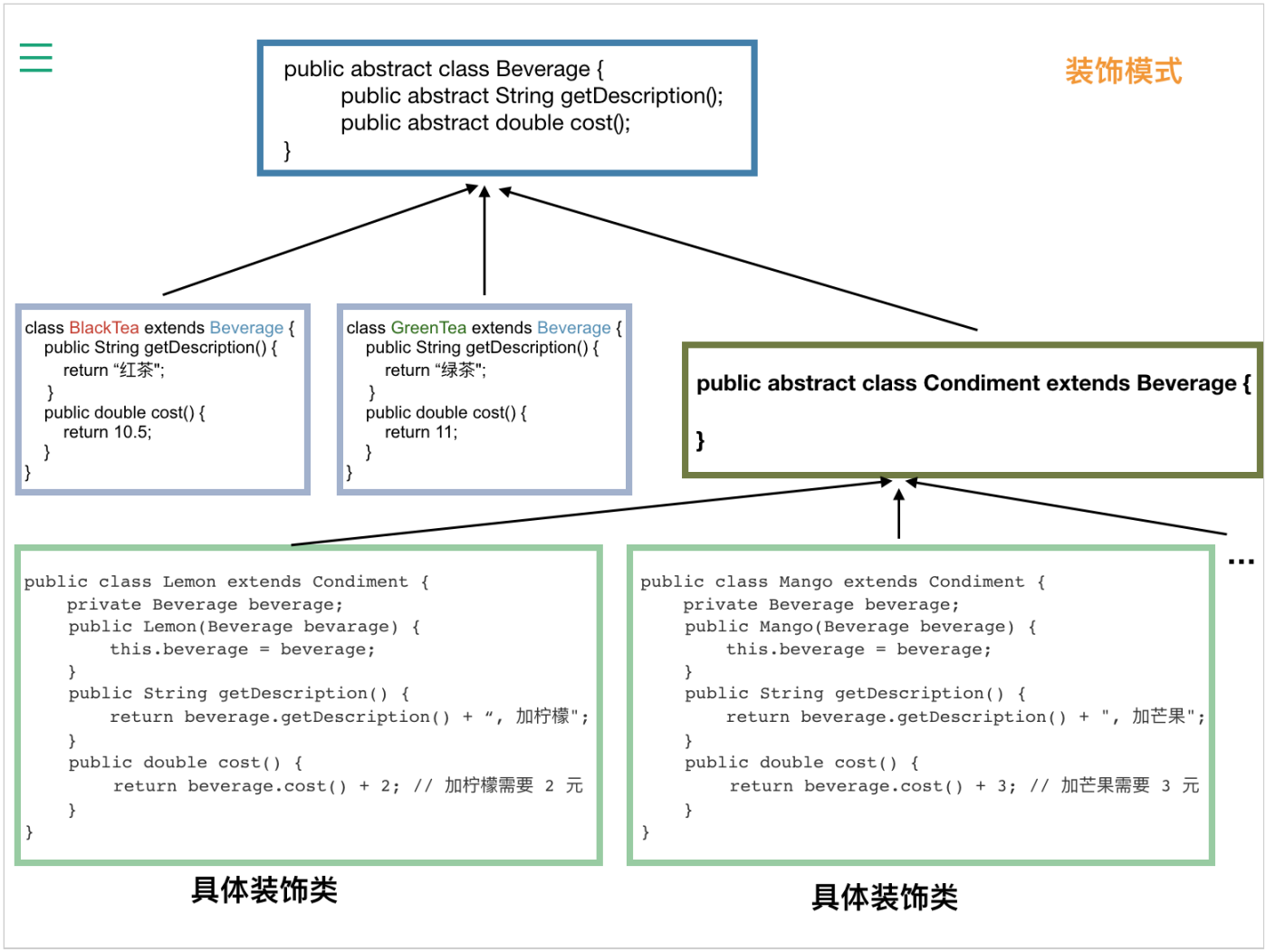
```
public static void main(String[] args) {
    // 首先，我们需要一个基础饮料，红茶、绿茶或咖啡
    Beverage beverage = new GreenTea();
    // 开始装饰
    beverage = new Lemon(beverage); // 先加一份柠檬
    beverage = new Mongo(beverage); // 再加一份芒果

    System.out.println(beverage.getDescription() + " 价格：￥" + beverage.cost()
    //"绿茶，加柠檬，加芒果 价格：￥16"
}
```

如果我们需要 **芒果-珍珠-双份柠檬-红茶**：

```
Beverage beverage = new Mongo(new Pearl(new Lemon(new Lemon(new BlackTea()))));
```

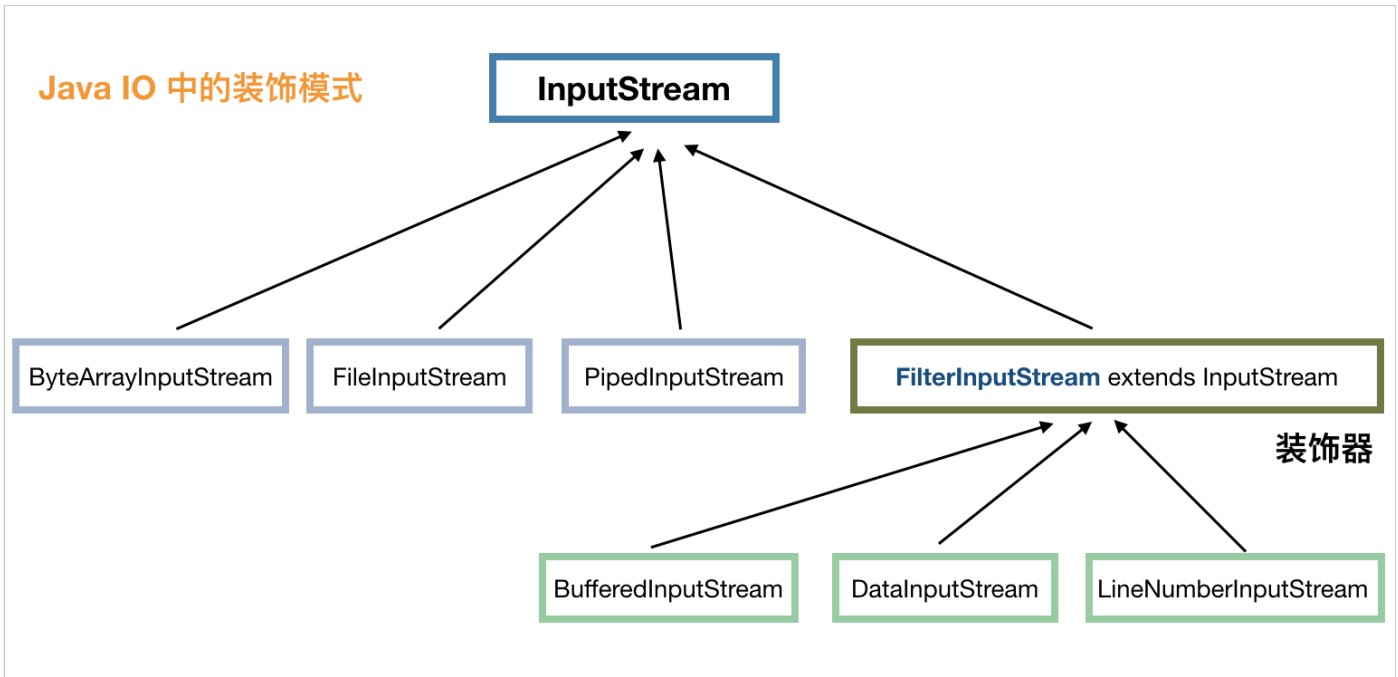
是不是很变态?

看看下图可能会清晰一些：



到这里，大家应该已经清楚装饰模式了吧。

下面，我们再来说说 java IO 中的装饰模式。看下图 InputStream 派生出来的部分类：



我们知道 InputStream 代表了输入流，具体的输入来源可以是文件（FileInputStream）、管道（PipedInputStream）、数组（ByteArrayInputStream）等，这些就像前面奶茶的例子中的红茶、

绿茶，属于基础输入流。

FilterInputStream 承接了装饰模式的关键节点，它的实现类是一系列装饰器，比如 BufferedInputStream 代表用缓冲来装饰，也就使得输入流具有了缓冲的功能，LineNumberInputStream 代表用行号来装饰，在操作的时候就可以取得行号了，DataInputStream 的装饰，使得我们可以从输入流转换为 java 中的基本类型值。

当然，在 java IO 中，如果我们使用装饰器的话，就不太适合面向接口编程了，如：

```
InputStream inputStream = new LineNumberInputStream(new BufferedInputStream(new
```

这样的结果是，InputStream 还是不具有读取行号的功能，因为读取行号的方法定义在 LineNumberInputStream 类中。

我们应该像下面这样使用：

```
DataInputStream is = new DataInputStream(
                                new BufferedInputStream(
                                    new FileInputStream("")));
```

> 所以说嘛，要找到纯的严格符合设计模式的代码还是比较难的。

门面模式

门面模式（也叫外观模式，Facade Pattern）在许多源码中有使用，比如 slf4j 就可以理解为是门面模式的应用。这是一个简单的设计模式，我们直接上代码再说吧。

首先，我们定义一个接口：

```
public interface Shape {
   void draw();
}
```

定义几个实现类：

```
public class Circle implements Shape {
    @Override
    public void draw() {
```

```
        System.out.println("Circle::draw()");
    }
}

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Rectangle::draw()");
    }
}
```

客户端调用：

```
public static void main(String[] args) {
    // 画一个圆形
      Shape circle = new Circle();
      circle.draw();

      // 画一个长方形
      Shape rectangle = new Rectangle();
      rectangle.draw();
}
```

以上是我们常写的代码，我们需要画圆就要先实例化圆，画长方形就需要先实例化一个长方形，然后再调用相应的 draw() 方法。

下面，我们看看怎么用门面模式来让客户端调用更加友好一些。

我们先定义一个门面：

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
```

```
        square = new Square();
    }

    /**
     * 下面定义一堆方法，具体应该调用什么方法，由这个门面来决定
     */

    public void drawCircle(){
        circle.draw();
    }
    public void drawRectangle(){
        rectangle.draw();
    }
    public void drawSquare(){
        square.draw();
    }
}
```

看看现在客户端怎么调用：

```
public static void main(String[] args) {
    ShapeMaker shapeMaker = new ShapeMaker();

    // 客户端调用现在更加清晰了
    shapeMaker.drawCircle();
    shapeMaker.drawRectangle();
    shapeMaker.drawSquare();
}
```

门面模式的优点显而易见，客户端不再需要关注实例化时应该使用哪个实现类，直接调用门面提供的方法就可以了，因为门面类提供的方法的方法名对于客户端来说已经很友好了。

组合模式

组合模式用于表示具有层次结构的数据，使得我们对单个对象和组合对象的访问具有一致性。

直接看一个例子吧，每个员工都有姓名、部门、薪水这些属性，同时还有下属员工集合（虽然可能集合为空），而下属员工和自己的结构是一样的，也有姓名、部门这些属性，同时也有他们的下属

员工集合。

```
public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates; // 下属

   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }

   public String toString(){
      return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :"
   }
}
```

通常，这种类需要定义 add(node)、remove(node)、getChildren() 这些方法。

这说的其实就是组合模式，这种简单的模式我就不做过多介绍了，相信各位读者也不喜欢看我写废话。

享元模式

英文是 Flyweight Pattern，不知道是谁最先翻译的这个词，感觉这翻译真的不好理解，我们试着强行关联起来吧。Flyweight 是轻量级的意思，享元分开来说就是 共享 元器件，也就是复用已经生成的对象，这种做法当然也就是轻量级的了。

复用对象最简单的方式是，用一个 HashMap 来存放每次新生成的对象。每次需要一个对象的时候，先到 HashMap 中看看有没有，如果没有，再生成新的对象，然后将这个对象放入 HashMap 中。

这种简单的代码我就不演示了。

结构型模式总结

前面，我们说了代理模式、适配器模式、桥梁模式、装饰模式、门面模式、组合模式和享元模式。读者是否可以分别把这几个模式说清楚了呢？在说到这些模式的时候，心中是否有一个清晰的图或处理流程在脑海里呢？

代理模式是做方法增强的，适配器模式是把鸡包装成鸭这种用来适配接口的，桥梁模式做到了很好的解耦，装饰模式从名字上就看得出来，适合于装饰类或者说是增强类的场景，门面模式的优点是客户端不需要关心实例化过程，只要调用需要的方法即可，组合模式用于描述具有层次结构的数据，享元模式是为了在特定的场景中缓存已经创建的对象，用于提高性能。

行为型模式

行为型模式关注的是各个类之间的相互作用，将职责划分清楚，使得我们的代码更加地清晰。

策略模式

策略模式太常用了，所以把它放到最前面进行介绍。它比较简单，我就不废话，直接用代码说事吧。

下面设计的场景是，我们需要画一个图形，可选的策略就是用红色笔来画，还是绿色笔来画，或者蓝色笔来画。

首先，先定义一个策略接口：

```
public interface Strategy {
   public void draw(int radius, int x, int y);
}
```

然后我们定义具体的几个策略：

```
public class RedPen implements Strategy {
    @Override
    public void draw(int radius, int x, int y) {
        System.out.println("用红色笔画图, radius:" + radius + ", x:" + x + ", y:" +
    }
}
public class GreenPen implements Strategy {
    @Override
    public void draw(int radius, int x, int y) {
        System.out.println("用绿色笔画图, radius:" + radius + ", x:" + x + ", y:" +
    }
}
public class BluePen implements Strategy {
    @Override
    public void draw(int radius, int x, int y) {
        System.out.println("用蓝色笔画图, radius:" + radius + ", x:" + x + ", y:" +
    }
}
```

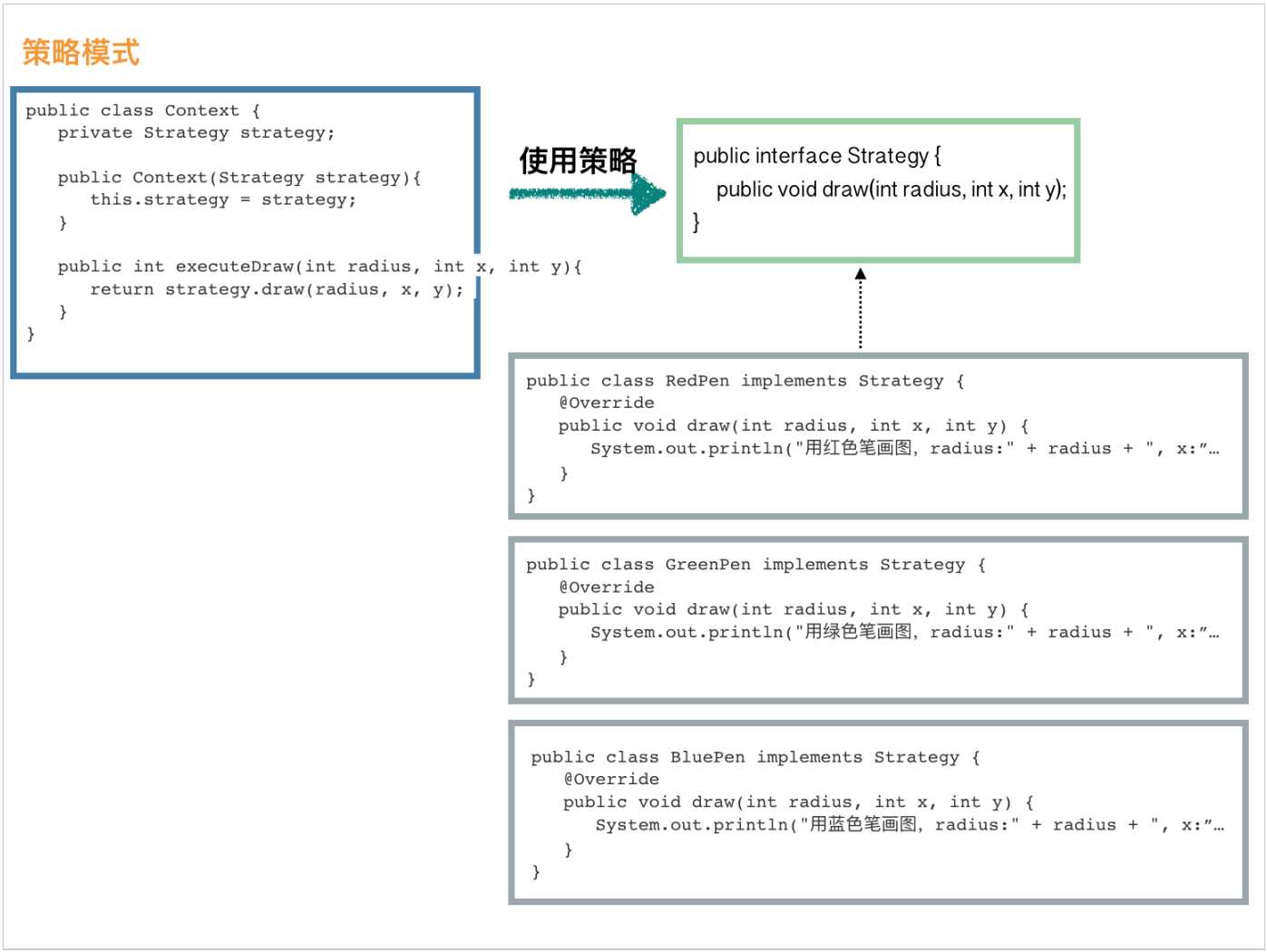
使用策略的类：

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeDraw(int radius, int x, int y){
        return strategy.draw(radius, x, y);
    }
}
```

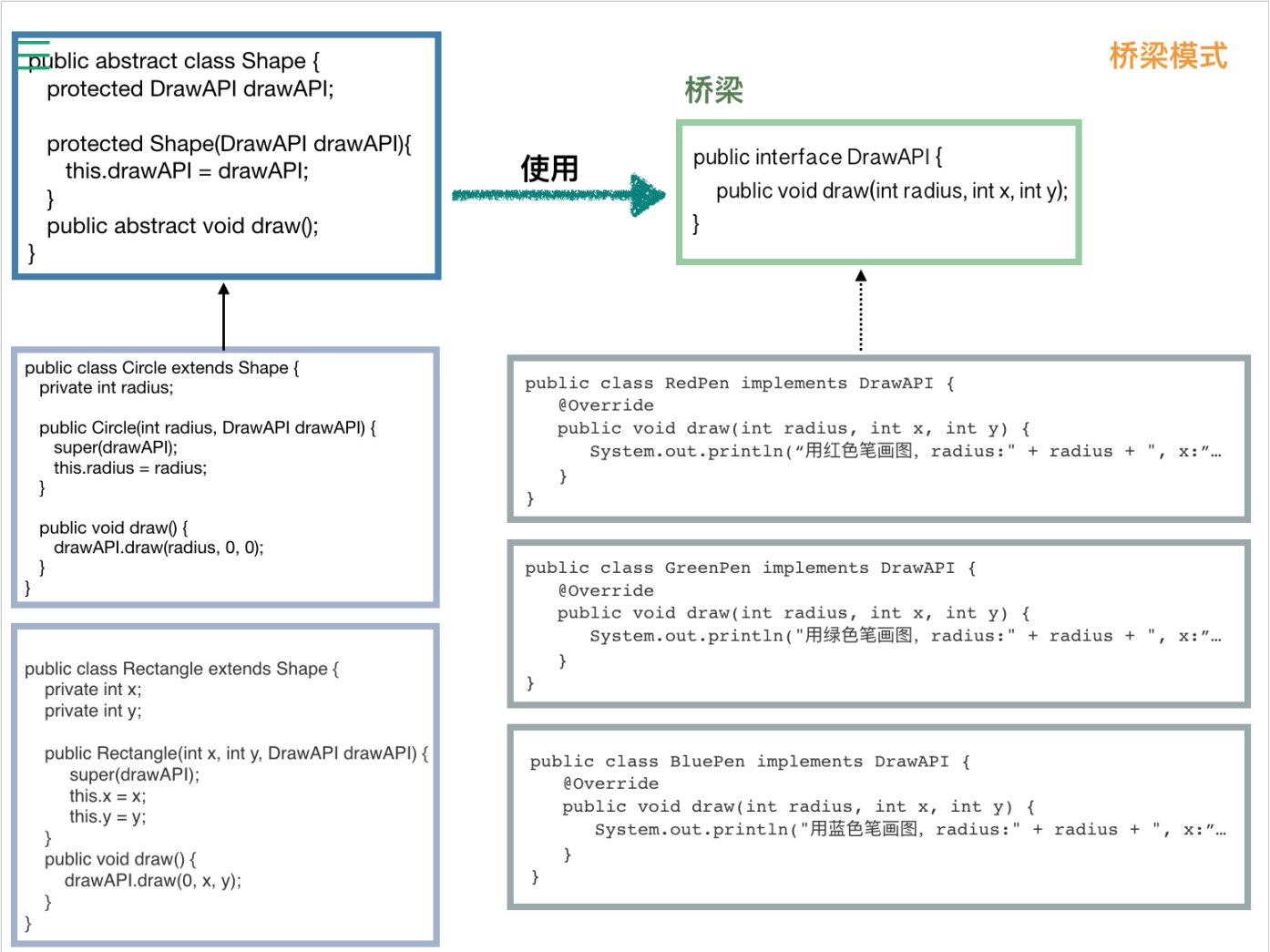
客户端演示：

```
public static void main(String[] args) {
    Context context = new Context(new BluePen()); // 使用绿色笔来画
      context.executeDraw(10, 0, 0);
}
```

放到一张图上，让大家看得清晰些：



这个时候，大家有没有联想到结构型模式中的桥梁模式，它们其实非常相似，我把桥梁模式的图拿过来大家对比下：



要我说的话，它们非常相似，桥梁模式在左侧加了一层抽象而已。桥梁模式的耦合更低，结构更复杂一些。

观察者模式

观察者模式对于我们来说，真是再简单不过了。无外乎两个操作，观察者订阅自己关心的主题和主题有数据变化后通知观察者们。

首先，需要定义主题，每个主题需要持有观察者列表的引用，用于在数据变更的时候通知各个观察者：

```
public class Subject {
    private List<Observer> observers = new ArrayList<Observer>();
    private int state;
    public int getState() {
        return state;
    }
    public void setState(int state) {
        this.state = state;
```

```
        // 数据已变更，通知观察者们
        notifyAllObservers();
    }
    // 注册观察者
    public void attach(Observer observer) {
        observers.add(observer);
    }
    // 通知观察者们
    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

定义观察者接口：

```
public abstract class Observer {
    protected Subject subject;
    public abstract void update();
}
```

其实如果只有一个观察者类的话，接口都不用定义了，不过，通常场景下，既然用到了观察者模式，我们就是希望一个事件出来了，会有多个不同的类需要处理相应的信息。比如，订单修改成功事件，我们希望发短信的类得到通知、发邮件的类得到通知、处理物流信息的类得到通知等。

我们来定义具体的几个观察者类：

```
public class BinaryObserver extends Observer {
    // 在构造方法中进行订阅主题
    public BinaryObserver(Subject subject) {
        this.subject = subject;
        // 通常在构造方法中将 this 发布出去的操作一定要小心
        this.subject.attach(this);
    }
    // 该方法由主题类在数据变更的时候进行调用
    @Override
    public void update() {
```

```
        String result = Integer.toBinaryString(subject.getState());
        System.out.println("订阅的数据发生变化，新的数据处理为二进制值为：" + result);
    }
}

public class HexaObserver extends Observer {
    public HexaObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }
    @Override
    public void update() {
        String result = Integer.toHexString(subject.getState()).toUpperCase();
        System.out.println("订阅的数据发生变化，新的数据处理为十六进制值为：" + result)


    }
}
```

客户端使用也非常简单：

```
public static void main(String[] args) {
    // 先定义一个主题
    Subject subject1 = new Subject();
    // 定义观察者
    new BinaryObserver(subject1);
    new HexaObserver(subject1);

    // 模拟数据变更，这个时候，观察者们的 update 方法将会被调用
    subject.setState(11);
}
```

output:

```
订阅的数据发生变化，新的数据处理为二进制值为：1011
订阅的数据发生变化，新的数据处理为十六进制值为：B
```

当然，jdk 也提供了相似的支持，具体的大家可以参考 java.util.Observable 和 java.util.Observer 这两个类。

实际生产过程中，观察者模式往往用消息中间件来实现，如果要实现单机观察者模式，笔者建议读者使用 Guava 中的 EventBus，它有同步实现也有异步实现，本文主要介绍设计模式，就不展开说了。

还有，即使是上面的这个代码，也会有很多变种，大家只要记住核心的部分，那就是一定有一个地方存放了所有的观察者，然后在事件发生的时候，遍历观察者，调用它们的回调函数。

责任链模式

责任链通常需要先建立一个单向链表，然后调用方只需要调用头部节点就可以了，后面会自动流转下去。比如流程审批就是一个很好的例子，只要终端用户提交申请，根据申请的内容信息，自动建立一条责任链，然后就可以开始流转了。

有这么一个场景，用户参加一个活动可以领取奖品，但是活动需要进行很多的规则校验然后才能放行，比如首先需要校验用户是否是新用户、今日参与人数是否有限额、全场参与人数是否有限额等等。设定的规则都通过后，才能让用户领走奖品。

> 如果产品给你这个需求的话，我想大部分人一开始肯定想的就是，用一个 List 来存放所有的规则，然后 foreach 执行一下每个规则就好了。不过，读者也先别急，看看责任链模式和我们说的这个有什么不一样?

首先，我们要定义流程上节点的基类：

```
public abstract class RuleHandler {
    // 后继节点
    protected RuleHandler successor;

    public abstract void apply(Context context);

    public void setSuccessor(RuleHandler successor) {
        this.successor = successor;
    }

    public RuleHandler getSuccessor() {
        return successor;
```

```
    }
}
```

接下来，我们需要定义具体的每个节点了。

校验用户是否是新用户：

```
public class NewUserRuleHandler extends RuleHandler {
    public void apply(Context context) {
        if (context.isNewUser()) {
            // 如果有后继节点的话，传递下去
            if (this.getSuccessor() != null) {
                this.getSuccessor().apply(context);
            }
        } else {
            throw new RuntimeException("该活动仅限新用户参与");
        }
    }
}
```

校验用户所在地区是否可以参与：

```
public class LocationRuleHandler extends RuleHandler {
    public void apply(Context context) {
        boolean allowed = activityService.isSupportedLocation(context.getLocati
        if (allowed) {
            if (this.getSuccessor() != null) {
                this.getSuccessor().apply(context);
            }
        } else {
            throw new RuntimeException("非常抱歉，您所在的地区无法参与本次活动");
        }
    }
}
```

校验奖品是否已领完：

```
public class LimitRuleHandler extends RuleHandler {
```

```
    public void apply(Context context) {
        int remainedTimes = activityService.queryRemainedTimes(context); // 查询
        if (remainedTimes > 0) {
            if (this.getSuccessor() != null) {
                this.getSuccessor().apply(userInfo);
            }
        } else {
            throw new RuntimeException("您来得太晚了，奖品被领完了");
        }
    }
}
```

客户端：

```
public static void main(String[] args) {
    RuleHandler newUserHandler = new NewUserRuleHandler();
    RuleHandler locationHandler = new LocationRuleHandler();
    RuleHandler limitHandler = new LimitRuleHandler();

    // 假设本次活动仅校验地区和奖品数量，不校验新老用户
    locationHandler.setSuccessor(limitHandler);

    locationHandler.apply(context);
}
```

代码其实很简单，就是先定义好一个链表，然后在通过任意一节点后，如果此节点有后继节点，那么传递下去。

至于它和我们前面说的用一个 List 存放需要执行的规则的做法有什么异同，留给读者自己琢磨吧。

模板方法模式

在含有继承结构的代码中，模板方法模式是非常常用的。

通常会有一个抽象类：

```
public abstract class AbstractTemplate {
    // 这就是模板方法
```

```
    public void templateMethod() {
        init();
        apply(); // 这个是重点
        end(); // 可以作为钩子方法
    }

    protected void init() {
        System.out.println("init 抽象层已经实现，子类也可以选择覆写");
    }

    // 留给子类实现
    protected abstract void apply();

    protected void end() {
    }
}
```

模板方法中调用了 3 个方法，其中 apply() 是抽象方法，子类必须实现它，其实模板方法中有几个抽象方法完全是自由的，我们也可以将三个方法都设置为抽象方法，让子类来实现。也就是说，模板方法只负责定义第一步应该要做什么，第二步应该做什么，第三步应该做什么，至于怎么做，由子类来实现。

我们写一个实现类：

```
public class ConcreteTemplate extends AbstractTemplate {
    public void apply() {
        System.out.println("子类实现抽象方法 apply");
    }

    public void end() {
        System.out.println("我们可以把 method3 当做钩子方法来使用，需要的时候覆写就可以
    }
}
```

客户端调用演示：

```
    public static void main(String[] args) {
        AbstractTemplate t = new ConcreteTemplate();
        // 调用模板方法
        t.templateMethod();
    }
```

代码其实很简单，基本上看到就懂了，关键是要学会用到自己的代码中。

状态模式

update: 2017-10-19

废话我就不说了，我们说一个简单的例子。商品库存中心有个最基本的需求是减库存和补库存，我们看看怎么用状态模式来写。

核心在于，我们的关注点不再是 Context 是该进行哪种操作，而是关注在这个 Context 会有哪些操作。

定义状态接口：

```
public interface State {
    public void doAction(Context context);
}
```

定义减库存的状态：

```
public class DeductState implements State {

    public void doAction(Context context) {
        System.out.println("商品卖出，准备减库存");
        context.setState(this);

        //... 执行减库存的具体操作
    }

    public String toString() {
        return "Deduct State";
```

```
    }
}
```

定义补库存状态：

```
public class RevertState implements State {

    public void doAction(Context context) {
        System.out.println("给此商品补库存");
        context.setState(this);

        //... 执行加库存的具体操作
    }

    public String toString() {
        return "Revert State";
    }
}
```

前面用到了 context.setState(this)，我们来看看怎么定义 Context 类：

```
public class Context {
    private State state;
      private String name;
      public Context(String name) {
        this.name = name;
    }

      public void setState(State state) {
        this.state = state;
    }
      public void getState() {
        return this.state;
    }
}
```

我们来看下客户端调用，大家就一清二楚了：

```
public static void main(String[] args) {
    // 我们需要操作的是 iPhone X
    Context context = new Context("iPhone X");

    // 看看怎么进行补库存操作
      State revertState = new RevertState();
      revertState.doAction(context);

    // 同样的，减库存操作也非常简单
      State deductState = new DeductState();
      deductState.doAction(context);

      // 如果需要我们可以获取当前的状态
    // context.getState().toString();
}
```

读者可能会发现，在上面这个例子中，如果我们不关心当前 context 处于什么状态，那么 Context 就可以不用维护 state 属性了，那样代码会简单很多。

不过，商品库存这个例子毕竟只是个例，我们还有很多实例是需要知道当前 context 处于什么状态的。

行为型模式总结

行为型模式部分介绍了策略模式、观察者模式、责任链模式、模板方法模式和状态模式，其实，经典的行为型模式还包括备忘录模式、命令模式等，但是它们的使用场景比较有限，而且本文篇幅也挺大了，我就不进行介绍了。

总结

学习设计模式的目的是为了让我们的代码更加的优雅、易维护、易扩展。这次整理这篇文章，让我重新审视了一下各个设计模式，对我自己而言收获还是挺大的。我想，文章的最大收益者一般都是作者本人，为了写一篇文章，需要巩固自己的知识，需要寻找各种资料，而且，自己写过的才最容易记住，也算是我给读者的建议吧。

（全文完）