



## 写在前面

很高兴你能来阅读我写的《和 Lvgo 一起学习设计模式》，下面的内容是希望在读之前也让你能对我有一点点了解。

# 你能得到什么

1. 能够轻松的提高对设计模式的认知
2. 成为**星尘的一个朋友**, 从此一起交流学习

## 关于作者

首先, 我是一个有**热血的情感动物**, 并善于思考总结。非奴隶性动物, 自我驱动力很强, 很不喜欢被非同类人管!

其次, 我没有过人的履历 (这一点正在努力改变, 我想我不该如此), 只是一个普通的不能在普通的热爱技术的普通人。由于人太实在, 不会用些华丽的词语来“诱惑”你来阅读我的文章。我只是简简单单的热爱技术, 热爱分享而已。

## 关于学习

对于技术我一直保持着求是精神, 在看其他大佬的文章时, 无论对付是谁, 从来都是“谨慎”的阅读, 毕竟智者千虑必有一失, 我也不相信谁的就是完全正确的, 我更愿意用合理的解释去接受一些主观的表述。所以这个时候就需要有大量的资料做对比参考。

## 声明

除设计模式背景或概念性内容外, 其余全部内容均为自己深入理解消化各位前辈内容原创输出。

## 源码

扫描二维码关注 **星尘的一个朋友**, 回复 “源码” 获取。亦或繁星、亦或尘埃。星尘**梦**, 为了梦想, 学习技术, 不要抱怨、坚持下去**梦**。



## 参考资料

非常感谢以下平台或书籍或个人的无私贡献知识资源, 才能够让我有更多机会去学习和了解各类知识。

- <https://bugstack.cn/itstack/itstack-demo-design.html>
- [http://c.biancheng.net/design\\_pattern/](http://c.biancheng.net/design_pattern/)
- <https://refactoringguru.cn/design-patterns>
- <https://www.journaldev.com/31902/gangs-of-four-gof-design-patterns>
- <https://www.runoob.com/design-pattern/design-pattern-tutorial.html>
- [可复用面向对象软件的基础](#)
- [大话设计模式](#)
- [设计模式之禅 \(第2版\)](#)
- [人人都懂设计模式：从生活中领悟设计模式：Python实现](#)
- [设计模式 \(Java版\)](#)

# 软件设计模式概述

当我们学习一门技术或者一类知识的时候，先去了解学习它的背景，会对我们接下来的学习和理解产生一些潜移默化的影响和帮助

这个背景千篇一律，事实就是如此。我用自己的话在总结一下。

## “设计模式”最初的提出，是在建筑领域。

1977 年被美国的建筑师  克里斯托夫·亚历山大 (Christopher Alexander) 在他的著作  《建筑模式语言：城镇、建筑、构造》(A Pattern Language: Towns Building Construction) 中描述了一些常见的建筑设计问题，并提出了一系列的解决方案，至此称为模式。

2年后，1979年，克里斯托夫·亚历山大在他的另一本著作  《建筑的永恒之道》(The Timeless Way of Building) 进一步强化了设计模式的思想。

直到 1990 年，也就是 13 年后。设计模式一词才到了软件工程界，同时为此开辟了专题研讨会。

## GOF

在“设计模式”进入软件行业后的第5年，也就是 1995 年，艾瑞克·伽马 (ErichGamma) 、理查德·海尔姆 (Richard Helm) 、拉尔夫·约翰森 (Ralph Johnson) 、约翰·威利斯迪斯 (John Vlissides)  等 4 位作者合作出版了  《设计模式：可复用面向对象软件的基础》(Design Patterns: Elements of Reusable Object-Oriented Software) 一书。

从此掀起了软件工程界的“设计模式”浪潮，使越来越多的开发者受益，同时涌现出了越来越多的设计模式。而这 4 位作者在软件领域小组名称 Gang of Four 四人组（四人帮），后来设计模式也以此匿名著称 GOF。



## 设计模式要做的事情

现如今的设计模式可远不止笔者这里收录整理学习的 23 种，而要比这多太多太多，但这么多的设计模式希望做的事情确实相同的。都是为了能够被反复利用，解决不断重复出现的问题而存在的，就像当初 GOF 写的那本书一样，‘可复用面向对象的基础’。

一种方法，解决n种问题。这就是模式，它要解决的，就是拥有共性的问题。

## 设计模式基本原则

基本原则摘自 [http://c.biancheng.net/design\\_pattern/](http://c.biancheng.net/design_pattern/)

当问题的解决方案有很多时，该怎么权衡哪一个方案可以成为是模式呢？这就有了原则性的约束。为了提高软件系统的可维护性和可复用性，增加软件的可扩展性和灵活性，我们应当要尽量根据⑦条原则来开发程序，从而提高软件开发效率、节约软件开发成本和维护成本。

- ① 开闭原则 OCP，1988年勃兰特·梅耶（Bertrand Meyer）在其著作《面向对象软件构造》中提出：软件实体应当对扩展开放，对修改关闭。
- ② 里式替换原则 LSP，1987年里斯科夫（Liskov）女士的“面向对象技术的高峰会议”（OOPSLA）上发表的一篇文章《数据抽象和层次》提出：继承必须确保超类所拥有的性质在子类中仍然成立。
- ③ 依赖倒置原则 DIP，1996年Object Mentor公司总裁罗伯特·马丁（Robert C.Martin）在C++ Report上发表的文章：  
高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象（要面向接口编程，不要面向实现编程。）
- ④ 单一职责原则 SRP，罗伯特·C·马丁（与DIP原则同一人）（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。这里的职责是指类变化的原因，单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分
- ⑤ 接口隔离原则 ISP，2002年罗伯特·C·马丁给“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法。该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上。以上两个定义的含义是：  
要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。
- ⑥ 迪米特法则又叫作最少知识原则 LOD/LKP，1987年美国东北大学的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被UML创始人之一布奇（Booch）普及，后来又在经典著作《程序员修炼之道》中提及，从而传播开来。原则定义：只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。
- ⑦ 合成复用原则 CRP 又叫组合/聚合复用原则，提倡软件复用过程中，优先使用组合复用，其次考虑继承，（如果使用继承，必须遵循里式替换原则），它与里式替换原则相辅相成。

对于原则的定义和约束，在多个设计模式中会有具体体现及说明。

## 设计模式分类

摘自 GOF 设计模式一书中文版译文，可能有内容有出入，还请参考原著辅助阅读。

根据模式是用来完成什么工作来划分，这种方式可分为创建型模式、结构型模式和行为型模式3种。

每种类型在对应部分中会再进行着重说明

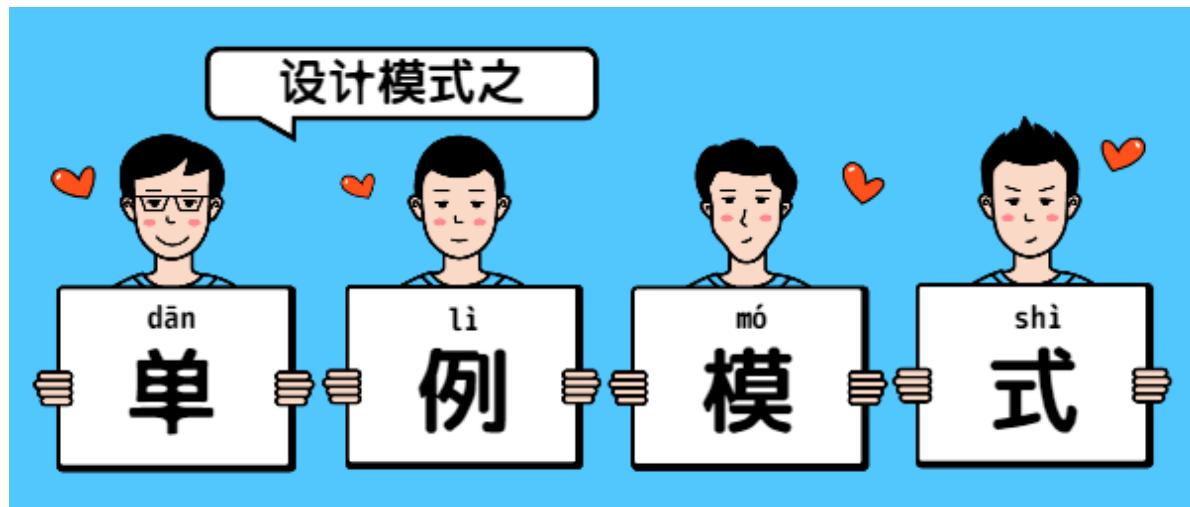
1. 创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。GoF中提供了单例、原型、工厂方法、抽象工厂、建造者等5种创建型模式。
2. 结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，GoF中提供了代理、适配器、桥接、装饰、外观、享元、组合等7种结构型模式。
3. 行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象无法单独完成的任务，以及怎样分配职责。GoF中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等11种行为型模式。

以上我们已经对设计模式有一个比较全面的简单了解了。那么现在，开始吧！

## 创建型(5)

用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

### 单例模式



前排提醒：学习设计模式的时候，千万不要咬文嚼字。因为模式本身就是一种思想，我们将其思想领悟，然后活学活用，而不是为了学语文。为了技术，放下语文。因为你接触的模式越多，越会发现很多模式的定义的界线开始模糊，这就是模式本身。一生万物，万物归一。

单例模式，顾名思义。单独的实例模式，其表达的意思即字面意思。它要解决的问题就是全局只能存在一个这样的类，或者说存在一个这样的类就可以满足业务需要了。

### 思考以下问题 ☺

你现在在做一个游戏，游戏中需要用到一个动作的音效，而这个音效的大小有 50M（假设），加载一次需要 2s 左右的时间。你在测试的过程中发现，每次人物做出动作 2s 左右之后才会出现之前动作音效，现在你需要如何解决这个问题？

### 资源的合理分配和利用

说到资源的合理分配和利用我想到了缓存，现在基本上大部分系统都会配备缓存，那为什么要配备缓存呢？扯远了，我们要谈的还是单例的问题。缓存的内容其实就是对资源的合理利用，比如我们将一些大对象或者频繁的 IO 操作内容保存（缓存）在一个对象中。而单例模式要解决的一个问题，就是这个问题，资源的合理分配和利用问题

### 你一直都在用的 singleton 模式

其实你如果细心一些你会发现，单例模式你每天都在使用

相信我们每位小伙伴都写过一个类，叫做 GlobalConstants（全局常量）而这个类中定义的所有变量（variable）都是 static final 的，大家肯定都知道其中的原因，有 2 个原因：

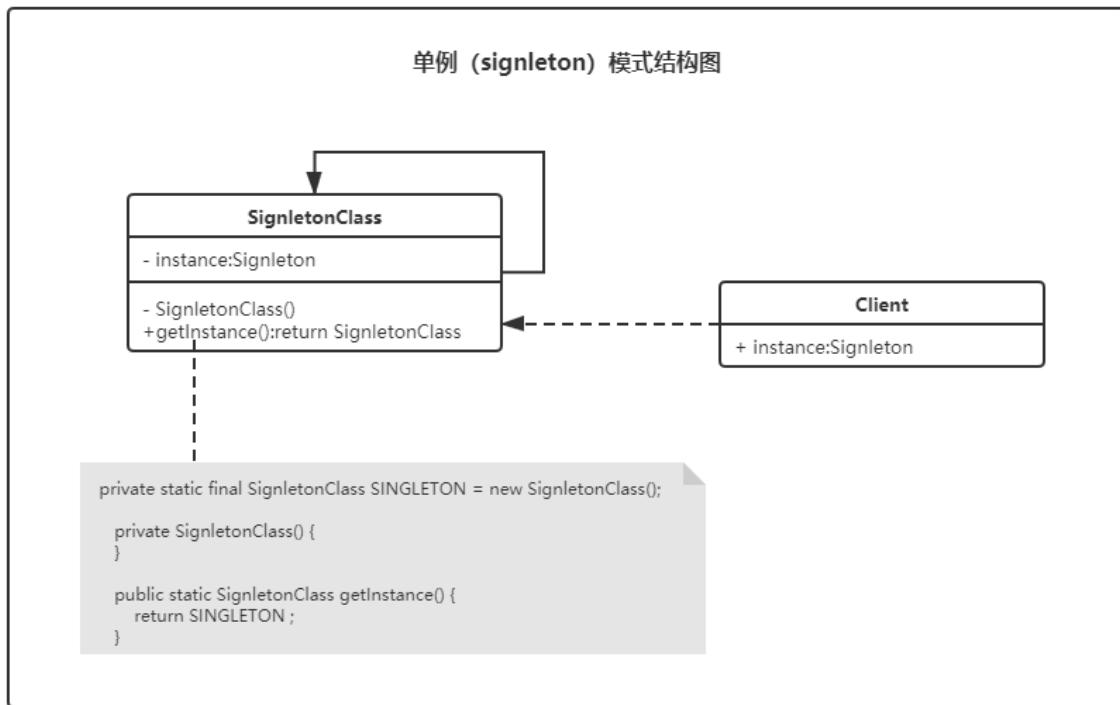
1. 静态类变量全局访问
2. final 修饰使其内容不可变

哦，顺带一提，java 中 String 类也是 singleton 模式的一种体现，当然，这不绝对

- 常量

- 数据库连接池
- Spring ApplicationContext
- JSP Application
- .....

## 单例模式结构



### 要点

1. 对象有本身自行创建，即构造函数私有化。
2. 全局统一访问，实例可被反复访问，即实例为静态实例。

## 单例模式实现的6种方式

关注回复“源码”获取全部单例模式完成代码

对于单例模式的实现方法有很多种，也看到大家的思维很活跃。但我脑子也不好使。我只想解决一些根本问题（使用单例模式），所以，我们每种方法都讨论一下。

**划重点：!! 单例模式的创建只能自己完成**

### 1. 延迟加载方式1（懒汉式）(线程不安全?)

```

public class DelayLoadSingleton1 {

    private static DelayLoadSingleton1 delayLoadSingleton1;

    private DelayLoadSingleton1() {
    }

    public static DelayLoadSingleton1 getInstance() {
        if (delayLoadSingleton1 == null) {
            delayLoadSingleton1 = new DelayLoadSingleton1();
        }
        return delayLoadSingleton1;
    }
}

```

```
}
```

对于**延迟加载（懒汉式）** 单例模式的线程不安全其实说的就线程对共享数据的使用而言，但就具体问题而言，单例模式本身就没有线程安全与不安全之分。之所以考虑到线程安全不安全，其实是对上面所说的资源的合理分配和利用，这种方式很明显没有做到资源的合理分配和利用。如果在多线程场景下很可能造成资源的浪费。

单例模式，根本就没有线程安全与不安全，是错误的使用导致它有了这个问题。

lvgo语录：学东西要知其然而知其所以然，即使千年流传的东西，你都应该保持着一颗质疑的心。?  
❤️

## 2. 延迟加载方式2（懒汉式）

```
public class DelayLoadSingleton2 {  
  
    /**  
     * 增加 volatile 修饰，解决变量可见性问题  
     */  
    private static volatile DelayLoadSingleton2 delayLoadSingleton1;  
  
    private DelayLoadSingleton2() {  
    }  
  
    /**  
     * 方法使用同步锁，同时只能有一个客户端来请求该方法，去创建实例。  
     * <p>  
     * 如果不使用同步方法，可能会出现两个以上线程同时创建了多个对象，破坏了单例模式，至于线程安  
     全，其实也是说对资源的合理利用。拒绝了重复创建  
     */  
    public static synchronized DelayLoadSingleton2 getInstance() {  
        if (delayLoadSingleton1 == null) {  
            delayLoadSingleton1 = new DelayLoadSingleton2();  
        }  
        return delayLoadSingleton1;  
    }  
}
```

关于 volatile 的更多内容欢迎在个人博客搜索关键字 "volatile"

通过使用同步锁与 volatile 使得单例模式变得安全资源合理的分配和利用，但每次调用都要同步，岂不是另外一种资源的浪费体现？?

## 3. 双重检查锁（DCL）

既然要合理利用资源，又要保证调用方法本身不产生资源浪费。这样就促成了 DCL 双重检查锁 方式。（技术人的思维就是这么活，一个单例模式被实现的五花八门。害的我们这些设计模式学徒从入门到放弃越来越快）

前面说了 DCL 是为了解决资源的合理分配和利用，那我们一起来看看 DCL 是如何工作的

```
public class DCLSingleton {  
    private static volatile DCLSingleton dclSingleton;  
  
    private DCLSingleton() {}  
  
    public static DCLSingleton getInstance() {
```

```

// 定义这个局部变量可以提高大约25%的性能💡 依据:Joshua Bloch "Effective Java,
Second Edition", p. 283-284
DCLSingleton current = dclSingleton;
// ① 第一次检查
if (dclSingleton == null) {
    // 🗑此时为了保证线程安全,我们不清楚其他线程是否已经实例化该对象,所以将类上锁达到
互斥效果
    synchronized (DCLSingleton.class) {
        /*
         * 再次将实例分配给局部变量并检查它是否由其他某个线程初始化
         * 当前线程被阻止进入锁定区域。如果它已初始化,那么我们可以
         * 返回先前创建的实例,就像上面检查对象是否为空一样。
         */
        current = dclSingleton;
    }
    // ② 第二次检查
    if (dclSingleton == null) {
        // 如果此时该类还没有被实例化,那么我们就可以安全的实例化一个单例的该对象
        current = dclSingleton = new DCLSingleton();
    }
}
return current;
}

```

💡笔记: DCL 方式是为了解决延迟加载 (懒汉式) 中的资源合理分配和利用问题。

当然,以上3种方式我,注意是我!全不推荐使用!! 😊

#### 4. 立即加载方式 (饿汉式)

```

public class Straightwaysingleton {
    private static final Straightwaysingleton straightwaysingleton = new
    Straightwaysingleton();

    private Straightwaysingleton() {
    }

    public static Straightwaysingleton getInstance() {
        return straightwaysingleton;
    }
}

```

立即加载方式是通过 classloader 来完成单例的创建,即当类第一次被主动调用初始化的时候。即使该类你不会使用(但是不用你还要设计成单例,我觉得这种方式已经可以满足一般的业务场景了)

##### Runtime.java 中使用该种方式实现

拓展类的装载过程: 加载 - 验证 - 准备 - 解析 - 初始化 - 使用 - 卸载

## 5. 内部类（推荐使用）

```
public class InnerClassSingleton {  
  
    private InnerClassSingleton() {  
    }  
  
    public static InnerClassSingleton getInstance() {  
        return InnerClassSingletonBuild.innerClassSingleton;  
    }  
  
    private static class InnerClassSingletonBuild {  
        private static final InnerClassSingleton innerClassSingleton = new  
InnerClassSingleton();  
    }  
  
}
```

这种方式综合使用了Java的类级内部类和多线程缺省同步锁的知识JVM来保证资源不会被浪费，巧妙地同时实现了延迟加载和线程安全，比起花里胡哨的DCL，这种方式更好的解决了实质的问题，并且没有了DCL的副作用，同时不受jdk版本的影响。

**当你的业务场景，很明确系统启动不需要的时候，以后也不知道需不需要，不用怀疑，用它！稳！资源控制的死死的**

一般我们默认会选择这种方式来实现单例模式，简单、好用、强大。

关于内部类的一些拓展，更多关于内部类内容查看我的[CSDN博客](#)

### 内部类分为对象级别和类级别

- 类级内部类指的是，有static修饰的成员变量的内部类。
- 如果没有static修饰的成员变量的内部类被称为对象级内部类。

类级内部类相当于其外部类的static成员，它的对象与外部类对象间不存在依赖关系，相互独立，因此可直接创建。

对象级内部类的实例，是必须绑定在外部对象实例上的。

类级内部类只有在第一次被使用的时候才被装载。

## 6. 枚举（推荐使用）

```
public enum EnumIvoryTower {  
  
    /**  
     * 实例  
     */  
    INSTANCE  
}
```

这种方法是一个叫做Joshua Bloch的人提出的，对于学习这种单例模式，我觉得更有必要带大家认识一下这个人。待会介绍。先说这种设计方式。

**当Joshua Bloch推荐的一种单例方式，与立即加载方式有过之而无不及。**

**简单、大方、得体、完美**

对于用枚举来实现单例模式近乎完美。因为它完完全全的由虚拟机来完成单例创建，这种方式是不是想到了和我们上面讲到的一个方式有点类似，没错，就是我们的立即加载方式（饿汉式），但是它相比立即加载方式却多了很多内容。就是我们接下来要说的保护单例模式。同时少了一点点东西——继承。

## 保(po)护(huai)单例模式

### 破坏单例

有时我们使用了以上的方式创建单例对象，同样会有两种方式来破坏单例对象（除枚举方式外）

1. 通过反射破坏单例
2. 通过序列化破坏单例

### 保护单例

上面的两点对于枚举来说，不存在。但我们自己写的方法如何规避这两点呢？

1. 调整私有构造函数，阻止反射调用单例。
2. 重写 `readResolve()` 方法。

#### 调整私有构造函数，阻止反射调用单例

```
// 解决反射创建对象破解单例模式
if (dclsingleton != null) {
    throw new IllegalStateException("Already initialized");
}
```

#### 重写 `readResolve()` 方法

```
/**
 * 解决反序列化创建对象破坏单例模式
 */
private Object readResolve() {
    return straightwaysingleton;
}
```

## 单例模式总结

### ☞要点

1. 对象有本身自行创建，即构造函数私有化。
2. 全局统一访问，实例可被反复访问，即实例为静态实例。

### 实现方式的选择

内部类 > 枚举 > 立即加载

### 优缺点

优点：资源的合理分配和利用

缺点：违反了单一职责原则

## 简单说说 Joshua Bloch

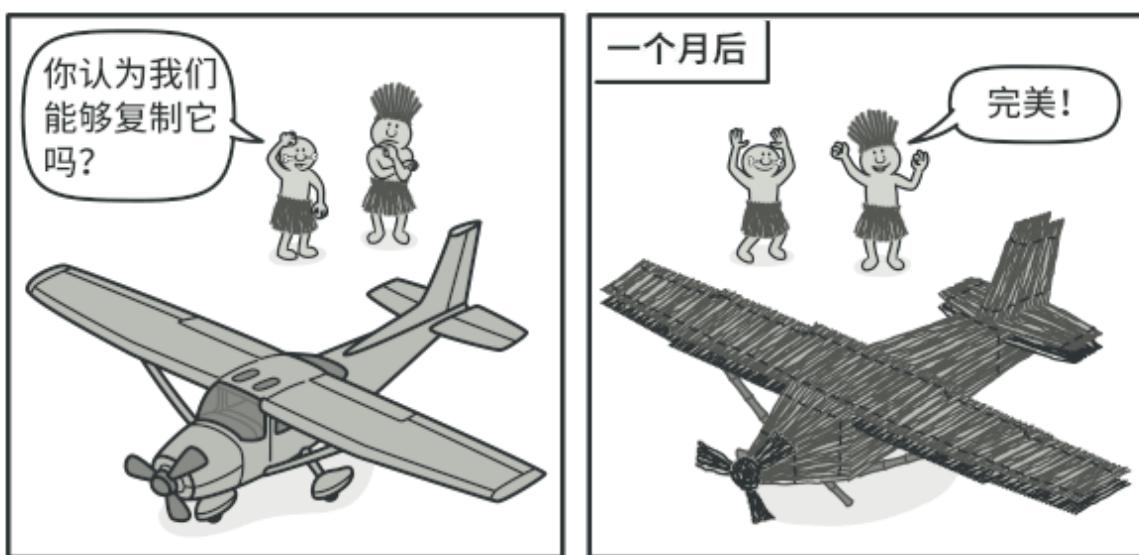
《Effective Java》、Effective 系列图书作者。同时我们每天都在使用着他写的代码，集合框架，它们的位置位于 `java.util.*`。

```
* @param <E> the type of elements in this collection
*
* @author Josh Bloch
* @author Neal Gafter
* @see Set
* @see List
* @see Map
* @see SortedSet
* @see SortedMap
* @see HashSet
* @see TreeSet
* @see ArrayList
* @see LinkedList
* @see Vector
* @see Collections
* @see Arrays
* @see AbstractCollection
* @since 1.2
*/
public interface Collection<E> extends Iterable<E> {}
```

老爷子的github：<https://github.com/jbloch>

## 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和原型相同或相似的新对象。



图片来源：<https://refactoringguru.cn/design-patterns/prototype>

## 月饼？盗文章？



每年中秋节的时候，大家都会吃到自己心仪口味的样式各异的月饼，但是他是怎么生产出来的呢，我猜它应该是有一个模板，比如花边图案的月饼



他会创造出来一个月饼原型，当你想吃五仁的时候，就把里面的馅改成五仁的，当你想吃蛋黄的（自己准备鸡蛋），就把馅改成蛋黄的，这样做不仅提高了生产效率，而且还节省了一部分再创建一个月饼的时间。

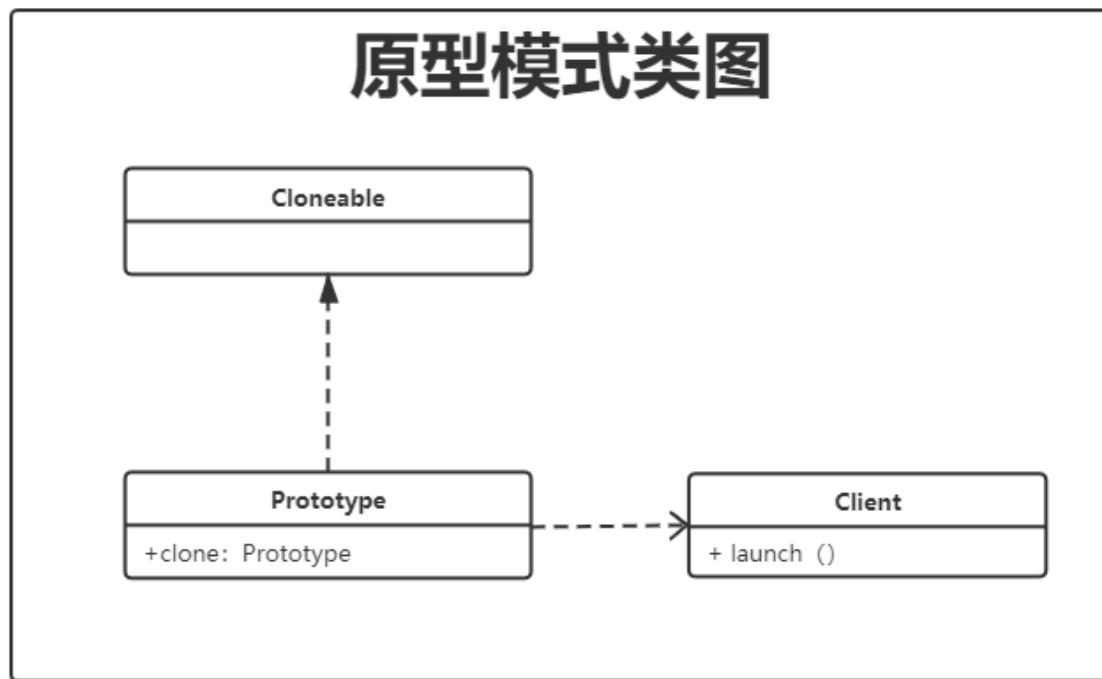
再比如说可恶的盗图、盗文章、盗视频的人，他们把原创内容拿回去改个名字，去掉水印，随便改改内容，就成了自己的了！？



上面说的两个案例的行为都是在节省了创建时间，同时达到了自己的一些目的。而这在设计模式中，就叫做**原型模式**，为了解决一个对象的创建而出现的一种设计模式，归类在了创建型模式中。

注意：在行为型模式中有着与之类似的一种模式——模版方法模式，是为了制定解决一件事情中的一系列操作约束而存在的一种模式，二者的区别在于，原型模式用于对象创建，模板方法模式用于事件行为约束。

## 原型模式类图



## 来看代码

关注回复“源码”获取。

java 的 JDK 中，提供一个标识接口 `Cloneable`，我们将需要定义成原型的类实现这个接口就可以完成复制了。

### 通过 `cloneable` 接口实现原型模式

```
public class Graphics implements Cloneable {
    private final String color;
    private final String shape;

    public Graphics(String color, String shape) {
        this.color = color;
        this.shape = shape;
    }

    @Override
    protected Graphics clone() throws CloneNotSupportedException {
        return (Graphics) super.clone();
    }

    .... set/get/toString
}
```

在上面的例子中，类中的对象类型都是基本类型，如果出现引用类型的时候，就会引发一个问题“浅克隆”，这会导致我们克隆出来的类会受原型中引用的类型影响，那我们如何才能规避这个问题做到“深克隆”呢？

## 浅克隆？Clone 深克隆？Clone

浅、深指的是对对象的占有权利。比如我借给你一个手机，那你只能使用这个手机里现有的东西，我如果删除了一个软件，那你自然也就没有这个软件了。假如我送给你一个手机，那你就可以随便的使用，不用担心我会做什么了，因为这个手机就是你的了。

那在 java 代码中，我们怎么理解浅克隆 Clone、深克隆 Clone 呢？

实现了 cloneable 接口，可以克隆一个区别于当前对象的另外一个新的对象，但对于对象中的引用，却不能进行克隆，你虽得到了他的人，但你却得不到他的心，如果想要得到他的心怎么办？拿钱砸他！非也，你只要把他的心也克隆一份就可以了。但是前提是他的心允许克隆（实现了 cloneable 接口）。

### 浅克隆代码

```
public class Graphics implements cloneable {  
  
    private final String color;  
    private final String shape;  
    // 引用类型没有实现 cloneable 接口  
    private final Size size;  
  
    public Graphics(String color, String shape, Size size) {  
        this.color = color;  
        this.shape = shape;  
        this.size = size;  
    }  
  
    @Override  
    protected Graphics clone() throws CloneNotSupportedException {  
        return (Graphics) super.clone();  
    }  
}
```

引用类型没有实现 cloneable 接口

```
// 引用类型没有实现 cloneable 接口  
public class Size {  
    public int width;  
    public int height;  
  
    public Size(int width, int height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    @Override  
    public String toString() {
```

```
        return "Size(" + width + ", " + height + ")";
    }
}
```

测试结果

```
class GraphicsTest {
    @Test
    void graphicsTest() throws CloneNotSupportedException {
        Size size = new Size(1, 2);
        Graphics graphics = new Graphics("red", "circular", size);
        Graphics clone = graphics.clone();
        size.height = 3;
        size.width = 5;
        System.out.println("graphics = " + graphics);
        // 判断两个对象是否不同
        Assertions.assertNotSame(graphics, clone);
        clone.setColor("blue");
        clone.setShape("square");
        System.out.println("clone = " + clone);
    }
}
```

注意此时的引用对象 size 的值

修改引用类型内容导致 clone 类的内容也跟着发生了变化

```
graphics = Graphics[color='red', shape='circular', size=Size(5, 3)]
clone = Graphics[color='blue', shape='square', size=Size(5, 3)]
```

## 深克隆：引用类型也实现 Cloneable 接口

```
// 引用类型实现了 cloneable 接口
public class Size implements Cloneable {
    public int width;
    public int height;

    public Size(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    protected Size clone() throws CloneNotSupportedException {
        return (Size) super.clone();
    }

    @Override
    public String toString() {
        return "Size(" + width + ", " + height + ")";
    }
}
```

在原型类中调整 clone 方法

```
@Override  
protected Graphics clone() throws CloneNotSupportedException {  
    Graphics clone = (Graphics) super.clone();  
    clone.size = size.clone();  
    return clone;  
}
```

测试结果

```
class GraphicsTest {  
    @Test  
    void graphicsTest() throws CloneNotSupportedException {  
        Size size = new Size(1, 2);  
        Graphics graphics = new Graphics("red", "circular", size);  
        Graphics clone = graphics.clone();  
        // 修改引用类型内容  
        size.height = 3;  
        size.width = 5;  
        System.out.println("graphics = " + graphics);  
        // 判断两个对象是否不同  
        Assertions.assertNotSame(graphics, clone);  
        clone.setColor("blue");  
        clone.setShape("square");  
        System.out.println("clone = " + clone);  
    }  
}
```

注意此时的引用对象 size 的值

```
graphics = Graphics[color='red', shape='circular', size=Size@5, 3]  
clone = Graphics[color='blue', shape='square', size=Size@1, 2]
```

关注回复“源码”获取。

## 原型模式自身有什么优势和问题呢? 🤔

优势:

1. JDK 的 cloneable 接口是基于内存数据的直接复制，速度相较于 new 关键字创建对象更加快速；同时简化了创建过程（不会执行构造方法）。
2. 通过深克隆来保存一个对象某一时刻的状态，便于还原，实现撤销操作；

问题:

1. 需要为每个类重写 #clone 方法；
2. 深克隆需要将每个对象都维护一个 cloneable 接口；
3. 构造方法中的代码不会执行；

## 总结

当我们需要频繁使用一些类似的对象的时候，可以考虑使用原型模式来降低资源的开销，使资源得到合理的分配和使用。而对于原型模式的深克隆带来的弊端，就显得那么的不重要了。

1. 类似的对象使用频繁，考虑原型模式
2. 深克隆时注意类中的引用类型是否实现了 cloneable 接口
3. 注意构造函数中是否有必要代码要执行，可以考虑放到 #clone 方法中执行

## 工厂模式



定义一个用于创建产品的接口，由子类决定生产什么产品。

大家可能都知道工厂模式，可真正理解应用的又有多少呢？此文本着能让大家彻底了解和何时适合使用工厂模式的原则来书写，希望能对你有所帮助，点个关注，一起开启新的思维来学习设计模式。

### 概念 ⑥

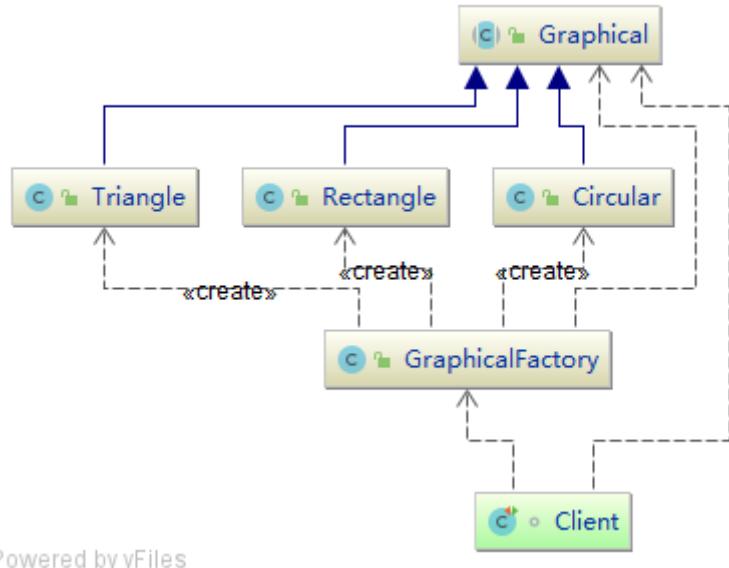
首先我们要知道的是，工厂模式是**创建型**设计模式分类下的一种，用来创建对象时选择使用。而还有一种看似功能一样，但实际的实现却大相径庭的方式叫做**简单工厂模式或（静态工厂模式）**。要注意这两者区别。即使他们完成的工作都是**创建对象**

- 简单工厂模式（静态工厂模式）：通过一个工厂类完成所有对象的创建工作；
- 工厂方法模式：如引用说明**定义一个用于创建产品的接口，由子类决定生产什么产品**；

### 简单工厂模式 😊

让我们先来看一下简单工厂模式，这个模式‘人如其名’，非常简单。

#### 类图 📋



## 具体代码

```

/**
 * 抽象图形类
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 21:33
 * @since 1.0.0
 */
public abstract class AbstractGraphical {
    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }
}

```

```

/**
 * 圆形
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 21:55
 * @since 1.0.0
 */
public class Circular extends AbstractGraphical {
}

```

```

/**
 * 矩形
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 22:10
 * @since 1.0.0
 */
public class Rectangle extends AbstractGraphical {
}

```

```

/**
 * 三角形
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 22:10
 * @since 1.0.0
 */
public class Triangle extends AbstractGraphical {
}

```

```

/**
 * 图形工厂
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 22:05
 * @since 1.0.0
 */
public class GraphicalFactory {

    public static final int CIRCULAR = 0;
    public static final int RECTANGLE = 1;
    public static final int TRIANGLE = 2;

    public static AbstractGraphical create(int type) {
        switch (type) {
            case CIRCULAR:
                return new Circular();
            case RECTANGLE:
                return new Rectangle();
            case TRIANGLE:
                return new Triangle();
            default:
                throw new IllegalStateException("please check param, range 0 - 2");
        }
    }
}

```

## 使用时机

当我们所要创建的对象个数较少，创建过程较复杂，使用较频繁 可以通过简单工厂模式将创建对象的过程封装起来，这样可以提高代码可读性，业务代码更专注于业务本身（当然案例代码中没有模拟构建复杂对象的情景）同时为了便于使用，将方法定义为静态。故也称之为静态工厂模式。

这里在强调一下，同时解释一下使用时机

1. 对象个数少：指的是需要通过这种方式创建的对象个数，通常为不变个数。因为如果对象个数迭代频繁，个数较多，在这种方法的维护上会出现一个很大的问题，即每新增加一个 class （一种图形，比如在增加一个正方形）就要调整一次 GraphicalFactory 类的代码。同样，即违反了开闭原则。
2. 创建过程较复杂：通过反向推理可知，如果创建对象过程不复杂，我选择直接 new。
3. 使用较频繁：同上可得，如果使用不频繁，我选择直接 new。不会考虑相对较复杂的设计模式。

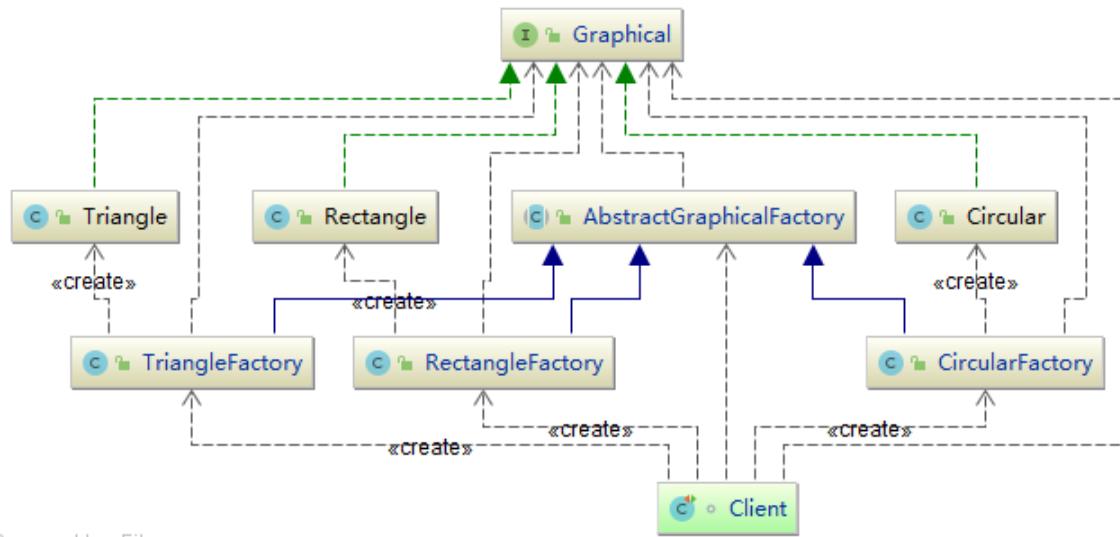
# 工厂方法模式

当我们将上面的简单工厂模式中的创建图形的方法抽象出来，将创建的过程延迟到子类中。满足了开闭原则的时候，那这就是工厂方法模式了。

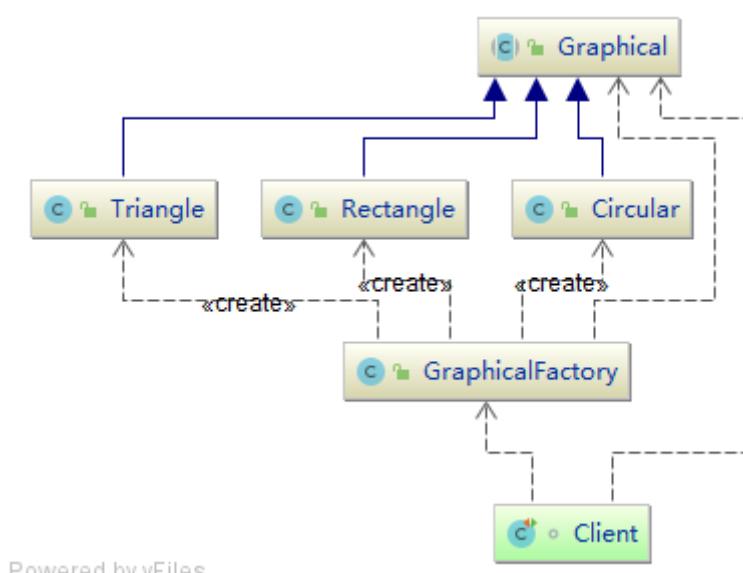
工厂方法模式怎么理解呢，顾名思义，通过工厂的方法来创建对象，每个对象都由一个工厂来创建，怎么创建这个工厂说了算。理解工厂方法模式对后面的抽象工厂理解可以说是“很精彩”

## 类图

工厂方法模式的类图



简单工厂模式的类图



通过类图的比较我们发现。简单工厂的工厂类抽象成了一个抽象工厂，而工厂方法模式中多了3个工厂。这就是工厂模式的定义诠释

定义一个用于创建产品的接口，由子类决定生产什么产品

## 具体代码

关注公众号 星尘的一个朋友，回复“源码”获取

```
/***
 * 图形类接口
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 21:33
 * @since 1.0.0
 */
public interface Graphical {

    /**
     * 图形描述
     */
    void description();
}
```

```
/***
 * 圆形
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 21:55
 * @since 1.0.0
 */
public class circular implements Graphical {

    @Override
    public void description() {
        LOGGER.info("circular");
    }
}
```

```
/***
 * 抽象工厂
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/8 23:16
 * @since 1.0.0
 */
public abstract class AbstractGraphicalFactory {

    /**
     * 创建一个图形
     *
     * @return 具体图形
     */
    public abstract Graphical creat();
}
```

```
/***
 * 圆形工厂
 *
 * @author lvgorice@gmail.com
 *
```

```

* @date 2020/10/8 23:27
* @since 1.0.0
*/
public class CircularFactory extends AbstractGraphicalFactory {
    /**
     * 将创建复杂的圆形过程封装到工厂里。
     * 1. 选定圆形位置;
     * 2. 指定圆形半径;
     * 3. 设置绘制图形所用的画笔;
     * 4. 选择图形的颜色;
     * 5. .....
     *
     * @return 一个复杂的圆形
    */
    @Override
    public Graphical create() {
        return new Circular();
    }
}

```

## 使用时机

其实这里我们通过与上面的简单工厂模式比较就可以看出，工厂方法模式适合在对象可能存在新增的情况，而且数量不定。创建对象过程复杂，使用频繁的场景。

## JDK中的工厂设计模式示例

案例来源：<https://www.journaldev.com/1392/factory-design-pattern-in-java>

1. `java.util.Calendar`, `ResourceBundle`和`NumberFormat getInstance()`方法使用`Factory`模式。
2. `valueOf()` 包装器类（例如`Boolean`, `Integer`等）中的方法。

## 总结

当我们所要创建的对象个数较少且不会新增，创建过程较复杂，使用较频繁可以通过简单工厂模式将创建对象。如不满足以上 3 种情况，建议直接 `new`。

**当我们所需要创建的对象使用频繁，创建过程较复杂，可能增加对象个数时，这无疑选择使用工厂方法模式。**

当我们试图用上面的3个原则去选择使用工厂模式的时候应该要思考几个问题。如

1. 对象个数很少，创建不复杂。（`new`关键字）
2. 创建过程虽然复杂，但是很少使用。（建造者模式）
3. 使用虽然很频繁，但只有1个对象就满足了需要。（单例模式）

等等诸如以上对象与使用使机的权衡都需要我们自己去仔细的设计和衡量，设计模式只提供了一种思想，你可以将一些思想整合使用，也可以使用一个方法来解决你的所有问题。

以上的几个问题，分别可以考虑单例模式和后面要讲到的建造者模式来实现，并不一定非要用工厂模式，活学活用才是我们的宗旨。

千万不要搞骚操作，为了用设计模式而用，否则岂不是 `new` 个 `String` 对象也要工厂来创建？

## 抽象工厂模式



| 提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

## 概念理解（重要！！！）

特别强调了一下抽象工厂模式的概念理解部分我觉得是非常有必要的，当然我在写下这篇文章之前看过很多优秀的博文、书籍、视频等资料对抽象工厂模式的讲解和代码示例等内容，但我发现。抽象工厂的概念被一次又一次的刷新，所以我也想表达一下自己对抽象工厂的理解。如果你和我持不同的意见，可以继续往下看，我很愿意和你一起讨论这个问题。

看我过之前的文章应该知道了我写的工厂模式的概念和代码实现，以及使用的时机。而抽象工厂模式的实现，等于工厂方法模式的实现。

那为什么会有两个模式的定义出现呢？这个问题解决了，那我们的概念就捋清楚了。我们一起来回顾一下这两个模式的定义：

1. 工厂（Factory）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。
2. 抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

我们将上面的两个模式的定义放在一起总结一下，是不是可以认为是，首先定义一个工厂接口，由子类去实现具体的工厂。如果我总结的定义你可以认可，那继续往下看。不认可忍一忍，看完再喷。让我们通过代码来理解一下。

下面内容很关键，希望你能认真看完。当然，不建议死扣字眼和代码，还是最初的那个誓言，学习设计模式的思想。而不是学语文。

## 一行代码！

### 工厂方法模式的伪代码

```
/**  
 * 电子工厂  
 */  
  
public interface ElectronicsFactory {  
  
    /**  
     * 生产一个手机  
     */  
    Phone creatPhone();  
}
```

```
/***
 * 苹果手机电子工厂
 */
public class IphoneElectronicsFactory implements ElectronicsFactory{

    /**
     * 生产一个苹果手机
     */
    Phone creatPhone() {
        return new IPhone();
    }
}
```

```
/***
 * 小米手机电子工厂
 */
public class MiPhoneElectronicsFactory implements ElectronicsFactory{

    /**
     * 生产一个小米手机
     */
    Phone creatPhone() {
        return new MiPhone();
    }
}
```

## 让我们在看一下抽象工厂模式的伪代码

```
/***
 * 电子工厂
 */
public interface ElectronicsFactory {

    /**
     * 生产一个手机
     */
    Phone creatPhone();

    /**
     * 生产一个电脑
     */
    Computer creatComputer();
}
```

```
/***
 * 苹果电子工厂
 */
public class AppleElectronicsFactory implements ElectronicsFactory{

    /**
     * 生产一个苹果手机
     */
    Phone creatPhone() {
        return new IPhone();
    }
}
```

```

    /**
     * 生产一个苹果电脑
     */
    Computer creatComputer() {
        return new MACBook();
    }
}

```

```

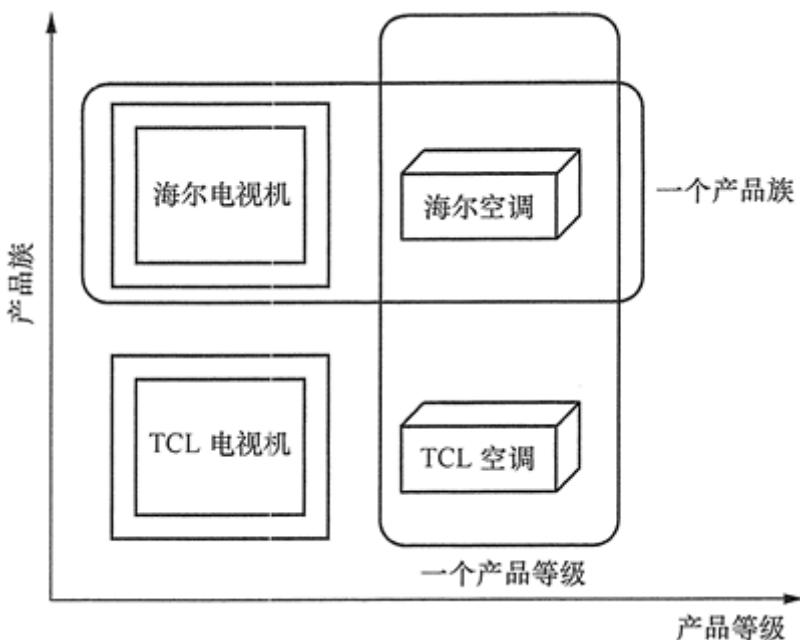
    /**
     * 小米电子工厂
     */
    public class MiElectronicsFactory implements ElectronicsFactory{

        /**
         * 生产一个小米手机
         */
        Phone creatPhone() {
            return new MiPhone();
        }

        /**
         * 生产一个小米电脑
         */
        Computer creatComputer() {
            return new MiComputer();
        }
    }
}

```

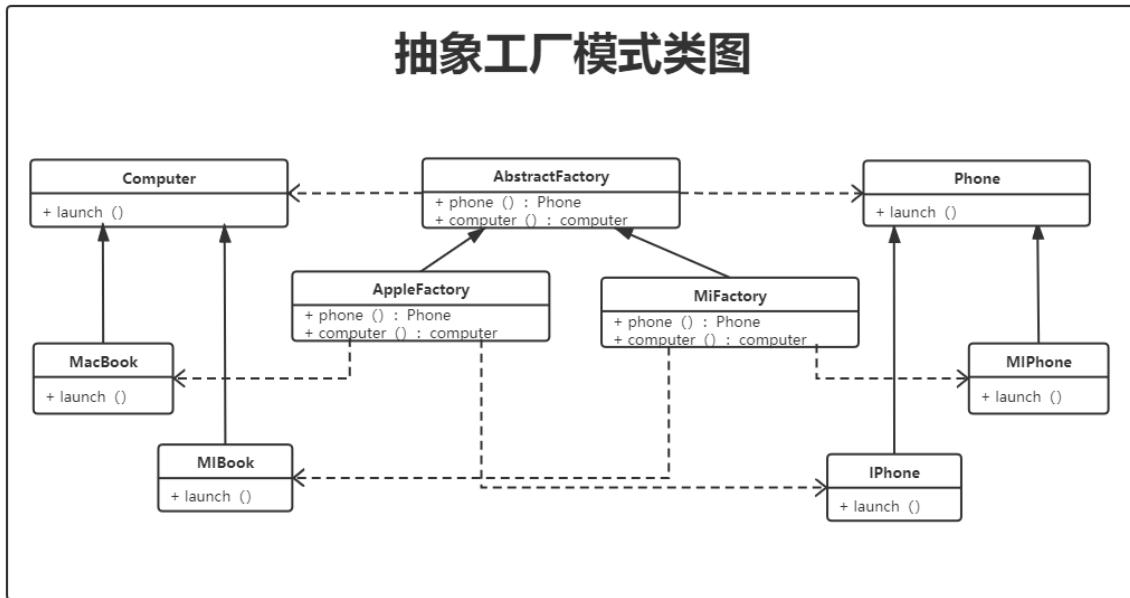
我们通过工厂方法模式，可以得到各种各样的同类型产品（都是手机），但我们如果通过抽象工厂模式，就可以得到各种各样同个产品族的产品（一个品牌的所有产品）**而这一切的内容，仅仅相差了一行代码**。同样的，当抽象工厂中只有一个工厂时，它与工厂模式，没有什么不同。



工厂方法模式只考虑生产同等级的产品，抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族（品牌）

来源：<http://c.biancheng.net/view/1351.html>

# 抽象工厂类图



## 具体代码

完整代码关注回复“源码”获取。

抽象工厂与工厂方法模式为同一种设计思想，他们不同于简单工厂。因为看了很多资料，对这部分内容的描述各有千秋。所以我在这里也表达了自己的一些看法。参考内容感兴趣的小伙伴可以看一下，我们一起讨论一下是极好的

- <http://c.biancheng.net/view/1351.html>
- <https://www.journaldev.com/1392/factory-design-pattern-in-java>
- <https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java>

再回到上面的两个定义：

1. 工厂 (Factory) 模式：**定义一个用于创建产品的接口，由子类决定生产什么产品。**
2. 抽象工厂 (AbstractFactory) 模式：**提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。**

抽象工厂，比如‘富士康’，细品一下，他就有很多个产品族，此时你应该明白了抽象工厂的概念和与工厂方法模式的区别（相差一行代码，相差一个产品族），如果被我说晕了，我真的很抱歉，愿意的话可以与我私聊。

当然相差一行代码是为了表达两者直接的关系，在实际应用情况下还是遵循标准的命名规范。避免产生歧义，出现理解误差。

文末的JDK中的抽象工厂设计模式示例生产的就是一个系列，所以上面也提到了抽象工厂与工厂本身并无大差别，当你一个工厂可以生产出多个系列的产品的时候，其实他就是抽象工厂了。比如看过我上一篇工厂模式文章的小伙伴就会发现，文末给出的JDK例子使用的是静态工厂模式。这一次给出的是抽象工厂（工厂方法模式）。抽象工厂与工厂方法本是同根生。我知道我把你说绕了，但是我的初衷是让你清楚这两者（抽象工厂模式与所谓的工厂方法模式），本就是一个思想。

⌚如果觉得我没说明白的请联系我，非常乐意被打扰

如果上面星尘表述的内容没能讲清楚抽象工厂的概念，大家不要急。继续往下看。如果说的还不明白，给我个机会，加我微信（1vgocc）或者公众号内私聊，直到聊清楚为止。你若不会，我愿受累，为了你，我愿意执着。

## 使用时机

- 当你想要管理多个系列产品的时候，比如多个套餐？多种策略组合？看需求，合理使用，总之多系列就用它！

例如你有一套方法，在不同的操作系统需要使用不同的实现，那这个时候你就可以使用抽象工厂，可以让它在不同的操作系统下发挥不同的功能。

## JDK中的抽象工厂设计模式示例

案例来源：<https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java>

- javax.xml.parsers.DocumentBuilderFactory # newInstance ()
- javax.xml.transform.TransformerFactory # newInstance ()
- javax.xml.xpath.XPathFactory # newInstance ()

## 建造者模式



指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示

根据建造者模式的定义，我们可以先简单的了解一下建造者模式要解决的问题，它是指将一个复杂对象的构建与它的表示分离，这句话的意思是指一个对象的构建过程与表示不再绝对。即一个构建过程对应多个结果，这取决于客户端如果指挥构建者进行对象的构建。这里的构建者就是我们接下来要讲的建造者模式内容。

## 理解程序中的建造

对于建造这个词语没什么好说的，在软件程序中建造是什么呢？我相信看到下面这个例子你应该就已经掌握了什么是建造者模式，当然这还不够，让我们慢慢来。

## JDK 中的 StringBuilder

```
public class StringBuilderTest {  
    @Test  
    void test(){  
        StringBuilder stringBuilder = new StringBuilder();  
        stringBuilder.append(1).append("个张三，和").append(4).append("个李四");  
        System.out.println(stringBuilder.toString());  
    }  
}
```

```
1个张三，和4个李四  
Process finished with exit code 0
```

上面的例子是妇孺皆知的 JDK 中提供的一个为了解决复杂 String 对象的 String 对象生成器。它还有个孪生姐夫 `StringBuffer` 用在并发环境下。

## Netty 中的 ServerBootstrap

再比如这个，netty 的启动器

```
ServerBootstrap bootstrap = new ServerBootstrap();  
bootstrap.group(parentGroup, childGroup)  
    .channel(NioServerSocketChannel.class)  
    .option(ChannelOption.SO_BACKLOG, 128)  
    .childHandler(new NettyProtobufChannelInitializer());  
  
try {  
    ChannelFuture sync = bootstrap.bind(2333).sync();  
    sync.channel().closeFuture().sync();  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

通过建造者 `ServerBootstrap` 来完成一个启动器的构建，同一个构造过程，却有着千差万别的结果。

## Ivgo 的 Slient 并发任务处理器

如果建造者被我们自己应用的话，我个人将它使用到了程序插拔配置上了，就像 netty 的启动器一样。

```
new TaskHandler<String>(testData) {  
    @Override  
    public void run(String s) {  
        try {  
            Thread.sleep(1000L);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        log.info("第" + s + "个任务" + Thread.currentThread());  
    }  
}.sync(false).overRun(() -> {  
    log.debug("我所有的任务执行结束了");  
}).execute(10);
```

上面我写的这个组件已发布到 maven 仓库；

```

<dependency>
  <groupId>org.lvgo</groupId>
  <artifactId>silent</artifactId>
  <version>1.0</version>
</dependency>

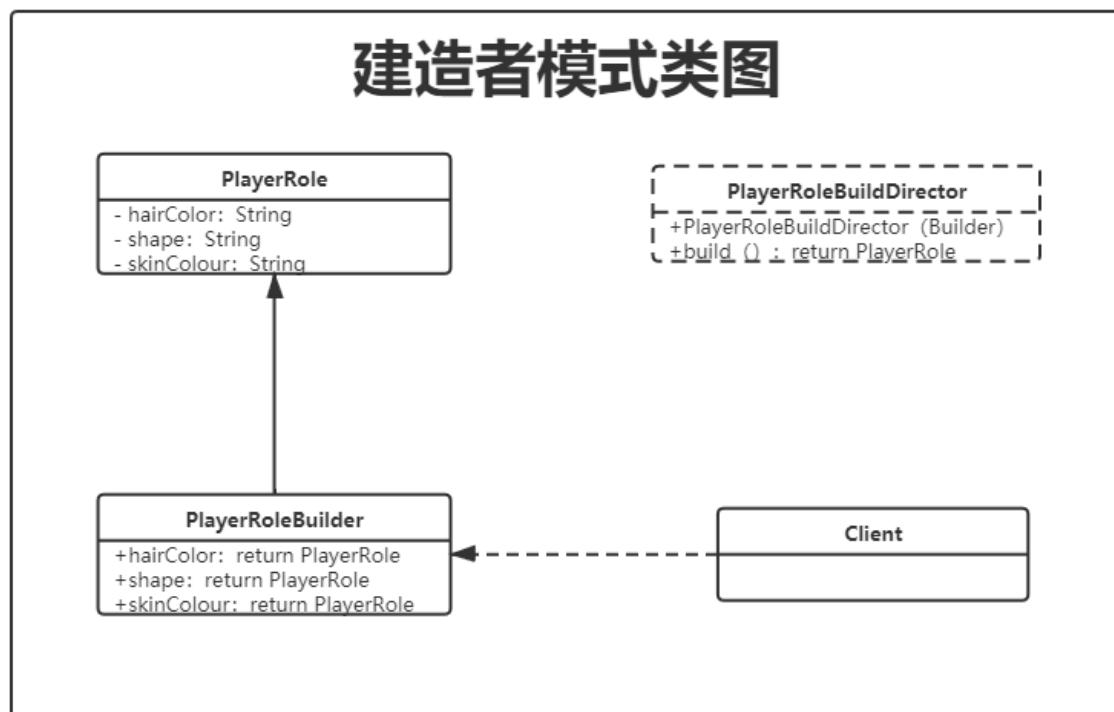
```

通过上面的几个例子我们大概知道什么是建造者模式了，它可以通过同一个构造过程来创建出不同的表示对象，比如

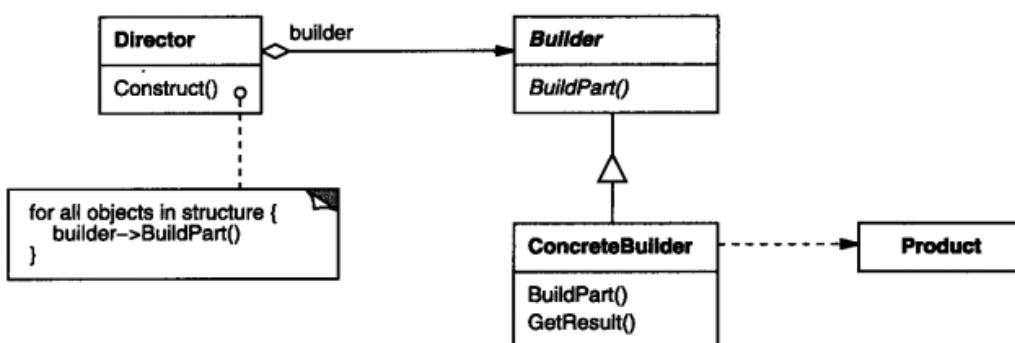
- `StringBuilder` 通过不同的参数传入 `append` 方法，得到结果是不一样的；
- `ServerBootstrap` 的不同参数设置，启动出来的 `netty` 服务端有着不同作用；
- `TaskHandler` 通过指定不同的参数，可以使任务的执行效果产生不同的变化；

## 建造者模式类图

注：在 GOF 的原著中对建造模式的理解与本篇差别较大。所以类图看起来会和很多资料中有所差异，望读者知悉。



GOF 中定义的建造者类图



这里说明一下，GOF的建造者模式中体现的是将要构建的对象、构建者、具体构建者、指挥者4部分独立开来，可以做很好的水平扩展。而lvgo将建造者的抽象类与具体的构建者合成一个，通过参数的方式将具体的构建者体现出来。将指挥者的角色延迟到了客户端，直接由客户端client来代替Director的工作。

## 代码

关注回复“源码”获取

为了能够更好的去理解lvgo与GOF中类图的差异，这里准备了两种写法，以诠释上面的说明。

### GOF类图的实现

```
public abstract class Builder {  
  
    protected PlayerRole playerRole = new PlayerRole();  
  
    abstract void setHairColor();  
    abstract void setShape();  
    abstract void setSkinColour();  
  
    PlayerRole build() {  
        return playerRole;  
    }  
}
```

```
public class Director {  
  
    private final Builder builder;  
  
    public Director(Builder builder) {  
        this.builder = builder;  
    }  
  
    public PlayerRole construct() {  
        builder.setHairColor();  
        builder.setShape();  
        builder.setSkinColour();  
        return builder.build();  
    }  
    public PlayerRole construct2() {  
        builder.setHairColor();  
        return builder.build();  
    }  
    public PlayerRole construct3() {  
        builder.setSkinColour();  
        return builder.build();  
    }  
}
```

```
public class PlayerRoleBuilder extends Builder {  
    @Override  
    void setHairColor() {  
        playerRole.setHairColor("褐色");  
    }  
}
```

```
@Override  
void setShape() {  
    playerRole.setShape("健硕");  
}  
  
@Override  
void setSkinColour() {  
    playerRole.setSkinColour("古铜色");  
}  
}
```

```
public class PlayerRole {  
  
    private String hairColor;  
    private String shape;  
    private String skinColour;  
}
```

## 测试类

```
void build() {  
    Builder playerRoleBuilder = new PlayerRoleBuilder();  
    Director playerRoleBuildDirector = new Director(playerRoleBuilder);  
    PlayerRole construct = playerRoleBuildDirector.construct();  
}
```

## 结果

```
construct = PlayerRole{hairColor='褐色', shape='健硕', skinColour='古铜色'}
```

个人觉得这种写法稍为复杂，不过他的水平扩展性和隔离性都比较好。

## Ivgo 整理的写法如下

```
public class PlayerRole {  
  
    private String hairColor;  
    private String shape;  
    private String skinColour;  
}  
  
public class PlayerRoleBuilder {  
  
    private final PlayerRole playerRole = new PlayerRole();  
  
    PlayerRoleBuilder hairColor(String color) {  
        playerRole.setHairColor(color);  
        return this;  
    }  
  
    PlayerRoleBuilder shape(String shape) {  
        playerRole.setShape(shape);  
        return this;  
    }  
}
```

```
PlayerRoleBuilder skinColour(String skinColour) {  
    playerRole.setSkinColour(skinColour);  
    return this;  
}  
  
PlayerRole build() {  
    return playerRole;  
}  
}
```

测试

```
@Test  
void test() {  
    PlayerRoleBuilder playerRoleBuilder = new PlayerRoleBuilder();  
    playerRoleBuilder.hairColor("红色").shape("健硕").skinColour("古铜色");  
    PlayerRole build = playerRoleBuilder.build();  
    System.out.println("build = " + build);  
}
```

结果

```
build = PlayerRole{hairColor='红色', shape='健硕', skinColour='古铜色'}
```

## 总结

**相同的资源，不同的结果**是我对建造者模式创建对象的理解。就像建造我们的人生，提供了相同的世界，相同的空气，每个个体的表现均不同。

通过使用建造者模式，我们可以更加灵活的去处理一个构建过程复杂的对象。将它的构建过程与表示分离开。例如如果你正在为**一长串的 set 方法**苦恼的时候可以考虑一下建造者模式。它使代码更整洁，可读性更好。

```
xxx.setA();  
xxx.setB();  
xxx.setC();  
xxx.setD();  
xxx.setE();
```

```
xxx.A().B().C().D().E().build();
```

当你想要给一个对象组装一个特有的结果的时候，不妨试试 GOF 的思路，很不错的。

### 缺点：

建造者模式因为需要维护一个单独的建造者类，同时要为每个属性单独维护一个方法，当类中有属性调整的时候，要一起调整对应建造者中的方法，这也是随它的优势而带来的一些副作用。**但如果有需要它的地方尽管去用。没有什么比混沌的代码更糟糕的事情了。**

## 案例应用

这里为了应读者要求，想有对应的案例可以参考，不然不清楚设计模式到底在什么地方用。

1. 餐饮系统有23道素材、18道荤菜，老板今天推出 8 种 2素 1 荤套餐，你如何实现？
2. 试想一下在我的世界 (mc) 中，提供了各种不同的道具，相同的道具组合，你做出来的房子和我做出来的房子看起来不那么一样。
3. 在塔防类游戏中，同一个射手，每次攒钱给他升级，最终有的变成了单体攻击高的神射手，有的变成散射群里攻击低的散箭手。

## § 结构型(7)

用于描述如何将类或对象按某种布局组成更大的结构，GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

### 代理模式



为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。

代理模式是为了解决**对象的访问控制**，特别是当你的目标对象不可改变的时候。效果更佳明显。

先来几段对话，简单感受一下这个“代理”

- “这 google 好慢啊，搞个代理”
- “您好，请问明年3月-12月有档期吗？” “和我经纪人联系”
- “我要告你” “好啊，有什么问题跟我的律师说吧”
- “哦？ 你是海大富海公公” “这是皇上口谕” “啊？ （急忙下跪）”
- “您好，这里是 12345 市长热线”

“**控制一个实际的对象访问，同时可以达到一定的目的**”

# 生活中的比喻

希望可以用生活中的一些例子能让我更好的去表达和梳理代理模式

## 明星&经纪人

比如经纪人，他代理了明星（主体），负责主体功能以外的事情，主体可以进行商演，但在商演以前或以后的事情，都不需要主体去关系，全部都由经纪人去处理。

## 公园门&门禁

一个公园的门禁，通过门禁系统，代理了公园的入口，公园入口只负责放人进入公园，不关心什么时候什么情况，来人就表示可以进入。门禁系统则负责什么时候，什么情况可以进。

通过使用代理模式，可以让我们的业务代码结构更加完整清晰，而将一些控制和辅助型的逻辑处理交给代理类，这其中体现的就是单一职责原则与迪米特法则。

上面的两个例子，我所想要表达的意思就是说明代理模式是为了**控制一个实际对象的访问而存在的一种模式**。不知道我说清了没有。

## 程序中的例子

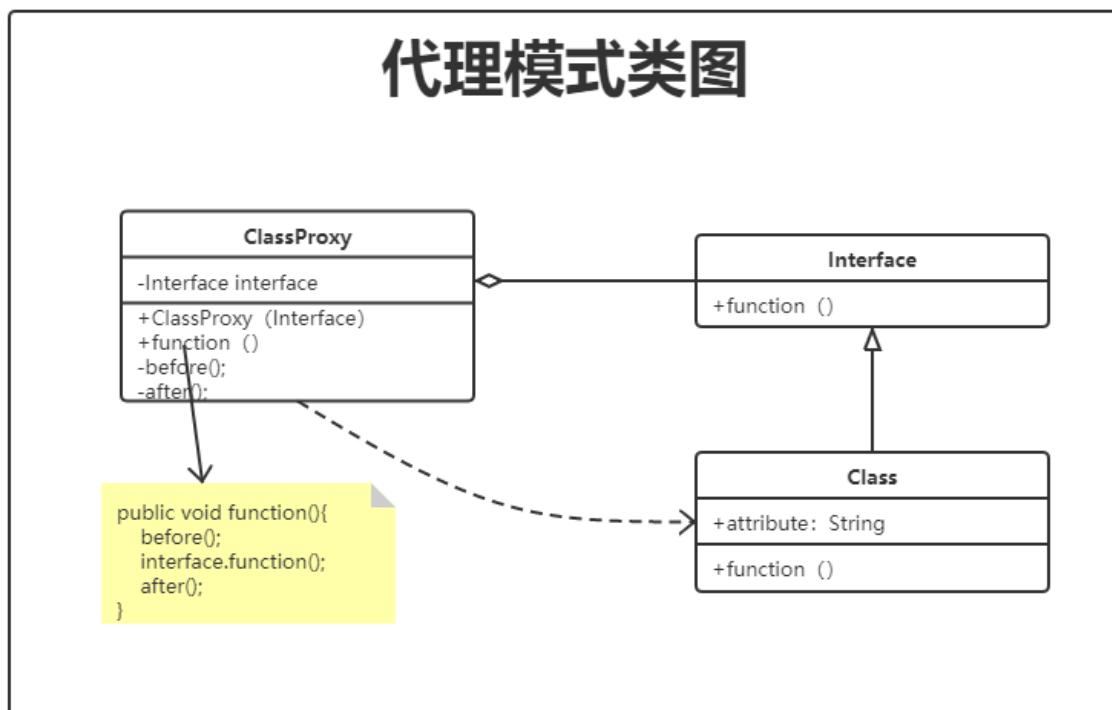
### mybatis 的数据访问接口

在代理模式中我们最常用到的应该就是 ORM 框架中的应用了，我们熟知的 mybatis 对**数据库的访问控制**就是使用了代理模式，通过一个接口的定义，来代理了数据库的访问操作。

在 mybatis 的代理模式处理思想如果翻译成白话：“你将你要执行的 SQL 告诉我在哪（mapper 映射，statement space），语句写好（xml）剩下的你都不用管”，这里的 mapper 接口即代理了数据库的访问工作。

**甚至 #{} \${}** 占位符，也是一种代理模式的体现，不一定非要有完整的接口，具体的实现类，代理类才是**代理模式**。可能这理解起来会让你觉得有点强词夺理。

## 代理模式类图



# 代码

完整代码获取关注公众号：星尘的一个朋友 回复“源码”

为了在深入的理解一下代理模式，我选择使用 mybatis 的代理模式实现原理伪代码。以及挖掘一下JDK 动态代理的一些细节内容，当然我不会去写源码的东西（毕竟道理大家都懂，不可能凭空 new 出来一个接口的实例，这当中定有蹊跷，我相信你在任何一篇博文中都能看到这部分内容，当然也欢迎加我微信（lvgocc）进群讨论）

## mybatis的核心代理模式伪代码

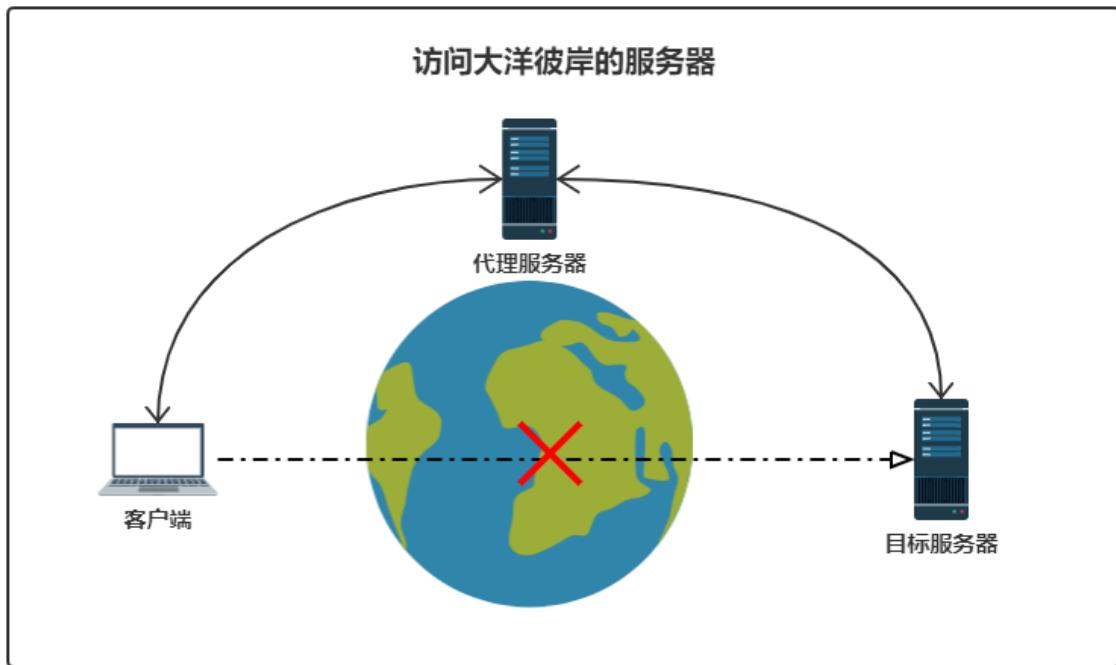
```
/**  
 * 用户接口  
 *  
 * @author lvgorice@gmail.com  
 * @date 2020/10/21 22:51  
 * @since 1.0.0  
 */  
public interface UserMapper {  
  
    /**  
     * 查询  
     *  
     * @param id 用户id  
     */  
    void selectById(int id);  
}
```

```
UserMapper o = (UserMapper) Proxy.newProxyInstance(  
    UserMapper.class.getClassLoader(),  
    new Class[]{UserMapper.class},  
    (proxy, method, arg) -> {  
        // 这里会执行具体的连接数据库执行 SQL 的操作 感兴趣可以查看 mybatis 源码继续了解。  
  
        // 打印参数  
        logger.info("statement position: {}, args: {}",  
            method.getDeclaringClass().getCanonicalName() + "#" + method.getName(),  
            Arrays.toString(arg));  
        return "用户id: " + arg[0] + "公众号: 星尘的一个朋友，加群一起学习设计模式";  
    });
```

```
14:25:43.966 [main] INFO io.github.lvgocc.App - Hello world!  
14:25:44.251 [main] INFO io.github.lvgocc.App - statement position:  
io.github.lvgocc.proxy.UserMapper#selectById, args: [2333]  
14:25:44.258 [main] INFO io.github.lvgocc.App - 查询结果: 用户id: 2333 公众号: 星尘的一个朋友，加群一起学习设计模式
```

mybatis 使用动态代理，让一个接口去代理了真实的数据库对象，当你需要的时候，再去建立连接、访问数据库、执行SQL、返回结果。如果之前有了解过 mybatis 的代理模式，这里应该不难理解。

下面再看一个简单的例子，当然用图说明可能会更容易



请忽略图中示意具体内容，只是借图表达代理的意义，控制对象的访问。

## 总结一下

代理模式为了解决对象的访问控制而存在。

- 当你想要抢一张回家的车票，你选择了候补，他选择了加速。此时12306或是第三方成了你的购票代理人。
- 当你来到一个陌生的小区，需要刷门禁卡才能进入。此时门禁成了小区的代理。

通过上面的总结，我知道

1. 当我选择了候补，我和购票解耦了✓，不需要等它的结果，等通知就行。但中间多了一个候补，链路更长✗了。
2. 我进门要刷卡，维护的对象多了✗，虽然系统变得复杂✗了，但小区更安全✓了，保护了小区。

## 适配器模式



# 什么是“榫”

嘶衣唔嗯ěn损，fao偻密，榫！

首先，让我们面向百度学习一波。

搜索框：榫

百度一下

Q 网页 资讯 视频 图片 知道 文库 贴吧 地图 采购 更多

百度为您找到相关结果约85,100,000个

搜索工具

**榫 - 百度汉语**

读音: [sǔn] ↗  
部首: 木 五笔: SWYF  
释义: 制木竹等器物时,为使两块材料接合所特制的凸凹部分。  
凸出的叫榫头,凹下的叫榫眼。

榫[sǔn]: 制木竹等器物时,为使两块材料接合所特制的凸凹部分。

感觉不够直观,找点图看看。



再来点



这里我借花献佛一下,榫说的是两块材料接合凸起的部分,凹进去的部分叫卯。

这东西是干什么用的呢,我不说大家也知道,是两块材料接合所用(切,这不废话吗,就是百度百科上说的么)。在中国建筑当中这个榫卯的用途可以说是处处皆是。现在已经火到了国外。

## 榫卯

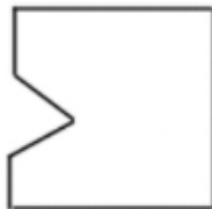
本来两种材料，一榫、一卯搭配的天衣无缝，怪就怪这建筑用的多了起来之后，各种各样的榫，各种各样的卯。也没办法统一，垄断法了解一下。

以下图片资源来自《设计模式之禅（第2版）》对其进行了一些简单的调整。

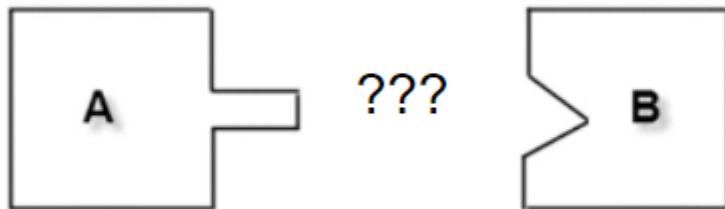
原来都是这样进行搭配



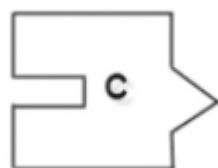
突然有一天给了我一个



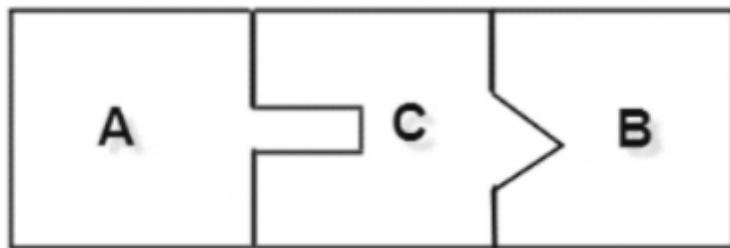
这两个材料怎么接，榫不对卯，卯不对榫，接不上啊。



这可难不倒伟大的工匠艺人们，他们做出来了一个榫卯，大概是这个样子。



这样他们就能完美的接在一起使用了。这就是榫卯。



榫卯使得本来不能直接接合的材料能够接合起来了。看到这里让我想到了插在大哥电脑上的扩展坞



这是华为的扩展坞，整个电脑上就两个外接口，一个充电口，一个这个扩展坞的口。想接 USB 设备必须  
要经过这个扩展坞才可以。

## 定义

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些  
类能一起工作。

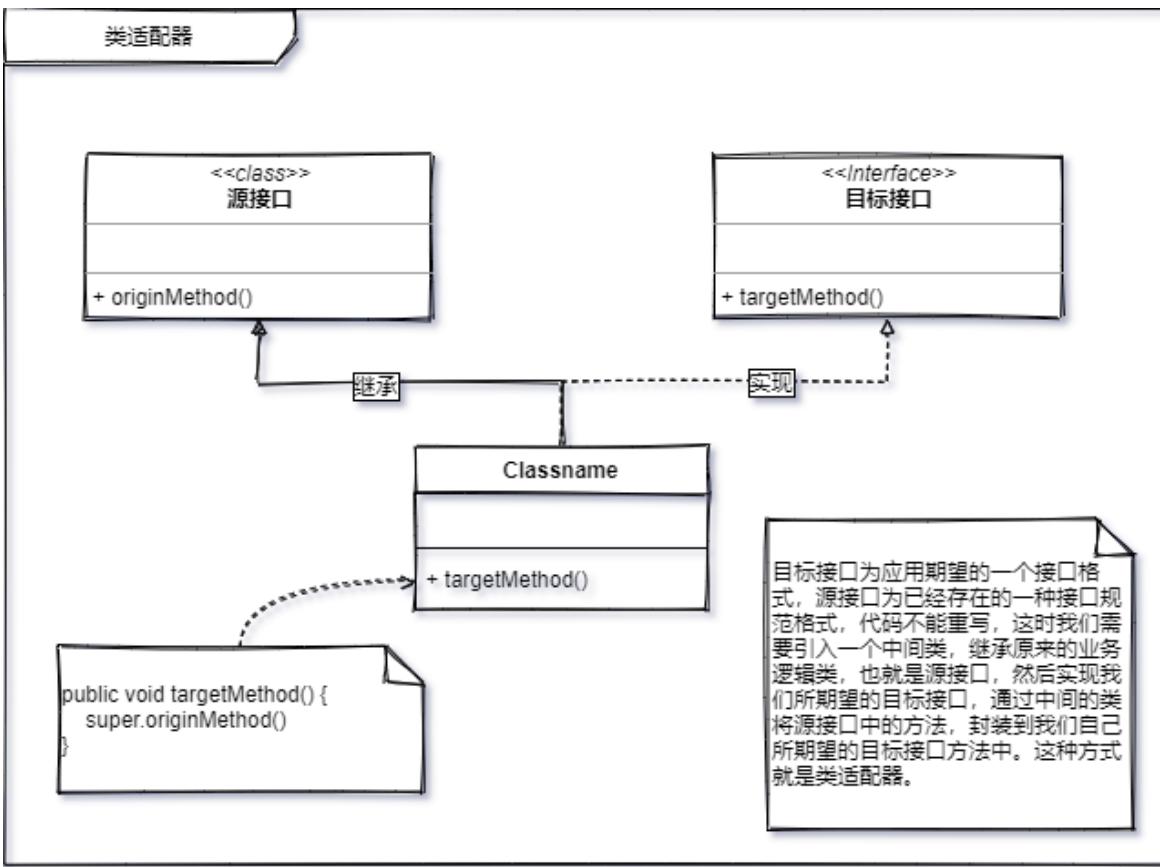
首先大家要知道一个可能一直被大家忽略的一个问题就是，适配器模式一共有两种，一种是类适配器，  
另一种叫做对象适配器。这两个是什么东西呢。

**类适配器：通过类的继承或者接口的实现来达到适配目的；**

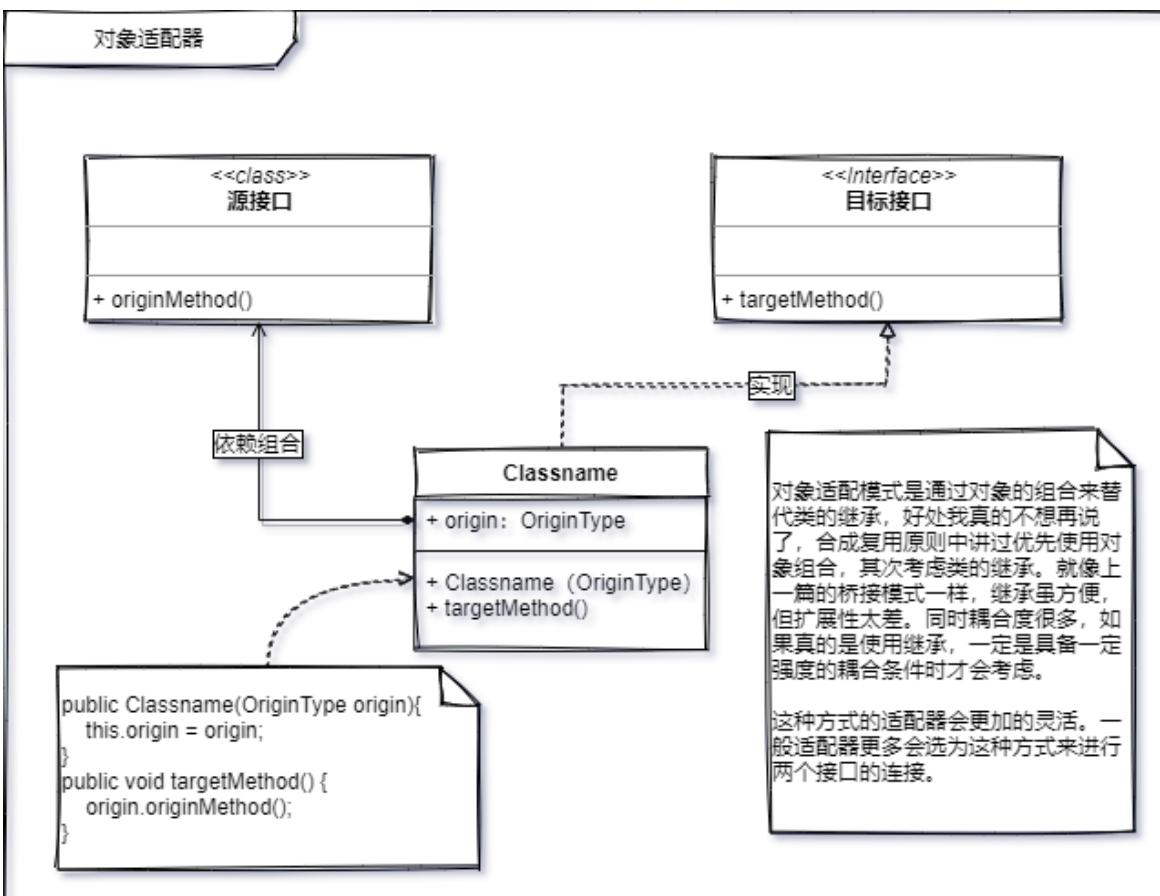
**对象适配器：通过对象的组合来达到适配目的；**

## 适配器模式类图

再来看看图，消化吸收一下。



目标接口为应用期望的一个接口格式，源接口为已经存在的一种接口规范格式。  
 代码不能重写。（工作量，系统稳定性，等等原因。）  
 这时我们需要引入一个中间类，继承原来的业务逻辑类，也就是源接口，然后实现我们所期望的目标接口，通过中间的类将源接口中的方法，封装到我们自己所期望的目标接口方法中。这种方式就是**类适配器**。



**对象适配模式**是通过对对象的组合来替代类的继承，好处我真的不想再说了，合成复用原则中讲过优先使用对象组合，其次考虑类的继承。

就像上一篇的桥接模式一样，继承虽方便，但扩展性太差。同时耦合度很多，如果真的是使用继承，一定是具备一定强度的耦合条件时才会考虑。

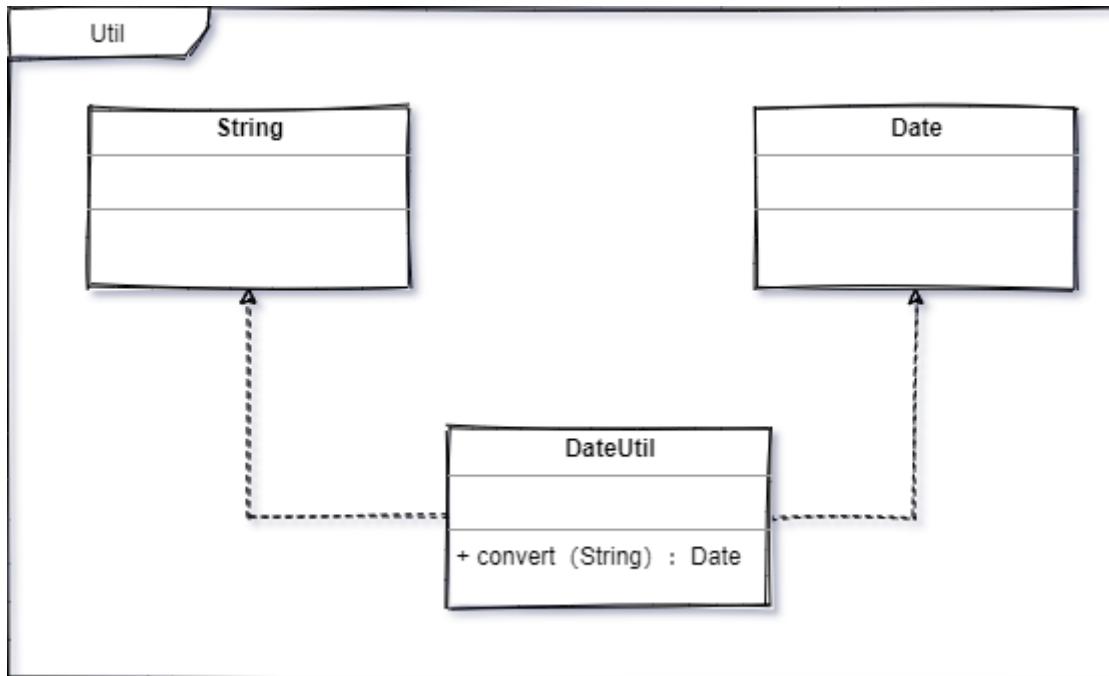
像对象四佩奇这种方式会更加的灵活。一般适配器更多会选为这种方式来进行两个接口的连接。

## 代码

学习适配器模式的时候，我没有再去找一些应用的案例来学习，因为我就一直在用这个模式。

```
public class DateUtil {  
  
    public final static String YYYY_MM_DD = "yyyy-MM-dd";  
    public final static String YYYY_MM_DD_TIGHT = "yyyyMMdd";  
  
    public static String getTightDate(Date date) {  
        Calendar calendar = new GregorianCalendar();  
        calendar.setTime(date);  
        SimpleDateFormat df = new SimpleDateFormat(YYYY_MM_DD_TIGHT,  
        Locale.CHINA);  
        return df.format(calendar.getTime());  
    }  
  
    public static String format(Date date) {  
        if (date == null) {  
            return null;  
        } else {  
            return new SimpleDateFormat(YYYY_MM_DD, Locale.CHINA).format(date);  
        }  
    }  
}
```

这个是日期工具类，我们经常会有这种场景，数据库存放的是 Date 类型，但接口给过来的是 String 类型，或者这两者相反，总之需要将其正常存储或返回，这个时候我们通常会采取使用一个日期工具类，将数据格式进行一个转换，这其中，工具类担任的角色，我认为就是适配器的一个职责，“将两个本不能直接结合的材料进行了接合”。这个看起来很简单，但这就是适配的模式的精髓，就是为了解决类似问题而存在的。如果把上面的程序用一个类图来表示的话，就是这个样子。



```

/**
 * 数组工具类
 *
 * 欢迎跟我一起学习，公众号搜索：星尘的一个朋友
 * 也可以加我微信（lvgoice）拉你进群
 *
 * @author lvgorice@gmail.com
 * @version 1.0
 * @blog @see http://lvgo.org
 * @CSDN @see https://blog.csdn.net/sinat_34344123
 * @date 2020/10/29
 */
public class ArraysUtil {

    public static <T> List<T> asList(T... a) {
        return new ArrayList<>(a);
    }

    private static class ArrayList<E> extends AbstractList<E>
        implements RandomAccess, java.io.Serializable
    {
        private static final long serialVersionUID = -2764017481108945198L;
        private final E[] a;

        ArrayList(E[] array) {
            a = Objects.requireNonNull(array);
        }

        .....
        .....
    }
}

```

一个数组工具类，大家应该都很熟悉，没错，这就是JDK中的Arrays工具类中的asList方法。记住哦，这种方式创建出来的List是不能够使用add方法的哦，因为此ArrayList（java.util.Arrays.ArrayList）非彼ArrayList（java.util.ArrayList），这个ArrayList里面没有重写add方法，这个知识点是送的，别客气。

在列举一下JDK中的适配器大家看一看，随便感受一下就好了。

```
163     * @since JAXB 2.0
164     */
165     public abstract class XmlAdapter<ValueType,BoundType> {
166
167         /**
168          * Do-nothing constructor for the derived classes.
169          */
170         @
171         protected XmlAdapter() {}
172
173         /**
174          * Convert a value type to a bound type.
175          *
176          * @param v
177          *      The value to be converted. Can be null.
178          * @throws Exception
179          *      if there's an error during the conversion. The caller is responsible for
180          *      reporting the error to the user through {@link javax.xml.bind.ValidationEventHandler}.
181         */
182         public abstract BoundType unmarshal(ValueType v) throws Exception;
183
184         /**
185          * Convert a bound type to a value type.
186          *
187          * @param v
188          *      The value to be converted. Can be null.
189          * @throws Exception
190          *      if there's an error during the conversion. The caller is responsible for
191          *      reporting the error to the user through {@link javax.xml.bind.ValidationEventHandler}.
192         */
193         public abstract ValueType marshal(BoundType v) throws Exception;
194     }
```

```
public final class NormalizedStringAdapter extends XmlAdapter<String, String> {
    /**
     * Replace any tab, CR, and LF by a whitespace character ' ',
     * as specified in <a href="http://www.w3.org/TR/xmlschema-2/#rf-whiteSpace">the whitespace facet 'replace'</a>
     */
    @
    public String unmarshal(String text) {
        if(text==null)      return null;      // be defensive

        int i=text.length()-1;

        // Look for the first whitespace char.
        while( i>=0 && !isWhiteSpaceExceptSpace(text.charAt(i)) )
            i--;

        if( i<0 )
            // no such whitespace. replace(text)==text.
            return text;

        // we now know that we need to modify the text.
        // allocate a char array to do it.
        char[] buf = text.toCharArray();

        buf[i--] = ' ';
        for( ; i>=0; i-- )
            if( isWhiteSpaceExceptSpace(buf[i]) )
                buf[i] = ' ';

        return new String(buf);
    }
}
```

```
public static final class ToStringAdapter extends XmlAdapter<String, Object> {
    public ToStringAdapter() {
    }

    public Object unmarshal(String s) { throw new UnsupportedOperationException(); }

    public String marshal(Object o) {
        return o == null ? null : o.toString();
    }
}
```

哦，这里还有群里小伙伴@ruize 提供的一个他写的适配器，一起和大佬学习一下



The screenshot shows a code editor with Java code. The code defines a class named DeviceBindAdapter with a static method adapter that takes a DeviceBindDTO parameter and returns a DeviceBindVO object. The DeviceBindVO class has a static builder method and a build method. The code is annotated with Javadoc-style comments for Params and Returns.

```
import com.leeann.swagger.app.vo.DeviceBindVO;
/*
 * 设备绑定适配 - adapter
 *
 * Author: ruize
 */
public class DeviceBindAdapter {
    /**
     * Params: bindDTO -
     * Returns:
     */
    public static DeviceBindVO adapter(DeviceBindDTO bindDTO) {
        return DeviceBindVO.builder()
            .deviceId(bindDTO.getDeviceId())
            .build();
    }
}
```

## 总结

适配器模式叫我总结，只有两字。“能用就行”，为什么用这两个字来总结，是因为适配器本身就是一个在出现问题以后用来补救的。像一个补丁一样。一般人不会上来就在系统初期写一些适配器在那里，如果是像上面列的一些对象转换的话合情合理。

适配器模式的出现是为了解决系统一些牵一发而动全身的事情，我们可以想象一下我的电脑没有 USB 接口（华为今年的最新款）难道我要把电脑拆了装个 USB 模组在里面吗？这显然是不可能的。所以有了扩展坞这东西。他就是为了解决这个问题的。（华为电脑这个是设计就没把外接设备接口留着，只留了一个充电的，和一个扩展坞的两个口）

1. 首先，适配器模式不会用在系统初期。
2. 其次，他是一个亡羊补牢一样的存在，你永远不知道系统的发展会遇到什么变故。只有出现需要的时候才会使用，不是故意设计的。就像你本来好好的接的 A 厂商接口，系统都开发完上线一年了，A 厂商说啥都不干了，你怎么办，现在有 B 厂商一样功能的接口，但接口规范不一样。这个时候就需要用适配器去补救了。
3. 代码复用，适配器模式可以充分的体现出代码复用。用一个适配类，解决修改老代码的尴尬局面。体现粗了老的代码可以完美的继续使（复）用。否则需要将老代码重构为新接口的规范，如果 B 厂商在换一次，估计开发人员头要炸了。

# 桥接模式



## 学习时间

2020年10月的某一天午饭后

“桥接模式？， 那是个啥” 心中突然蹦出这么一个想法。我心血来潮， 打开 Google， 输入 桥接模式， 回车走你， 等了半天。



## 未连接到互联网

代理服务器出现问题，或者地址有误。

请试试以下办法：

- 联系系统管理员
- 检查代理服务器地址
- 运行 Windows 网络诊断

ERR\_PROXY\_CONNECTION\_FAILED

这丝毫没有影响到我的情绪——(艹)， 随即我快速的切换搜索引擎视图忘掉刚刚发生的这一切。又是一记回车敲出，这次，它出现了

桥梁



百度一下

Q 网页 国 资讯 视频 图片 知道 文库 贴贴吧 地图 采购 更多

百度为您找到相关结果约100,000,000个

搜索工具

## 桥梁(使车辆行人等能顺利通行的构筑物)- 百度百科



桥梁，一般指架设在江河湖海上，使车辆行人等能顺利通行的构筑物。为适应现代高速发展的交通行业，桥梁亦引申为跨越山涧、不良地质或满足其他交通需要而架设的使通行更加便捷的建筑物。桥梁一般由上部构造、下部结构、支座和附属构造物组成，上部结构又称桥...

[概念定义](#) [历史发展](#) [中国历史](#) [桥梁类别](#) [巩固方法](#) [更多 >](#)

baike.baidu.com/

不知道是我手不行了，还是键盘要坏了，总之模式两字没带上，出来个桥梁，想着都差不多（呸，差不多个鬼）就看看吧，顺便学习了一下桥梁的专业释义（我就是这样东西越看越多，越看越杂的！龇牙咧嘴中！）。

不行，得回过神来，继续找桥梁模式去。这怎么都一样啊，抽象化、实现化、脱耦看不懂啊，然后就是那个到处都是，其实出自菜鸟教程的图形案例。

RUNOOB.COM

搜索.....

首页 笔记首页 ANDROID ES6 教程 排序算法 VERILOG 云服务器 程序员人生 编程技术

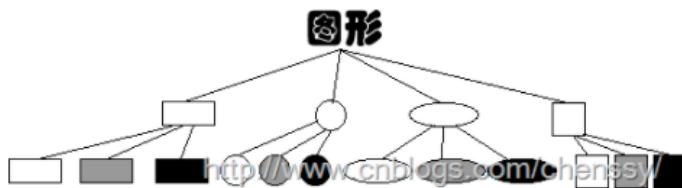
## 桥接模式

[分类](#) [编程技术](#)

在开始学Java的时候老师讲到继承的时候，总是喜欢用一个例子来讲解，那就是画图，这里有一个画笔，可以画正方形、长方形、圆形（这个大家都知道怎么做吧，我就不再解释了）。但是现在我们需要给这些形状进行上色，这里有三种颜色：白色、灰色、黑色。这里我们可以画出 $3 \times 3 = 9$ 种图形：白色正方形、白色长方形、白色圆形。……。到这里我们几乎知道了这里存在两种解决方案：

- 方案一：为每种形状都提供各种颜色的版本。
- 方案二：根据实际需要对颜色和形状进行组合。

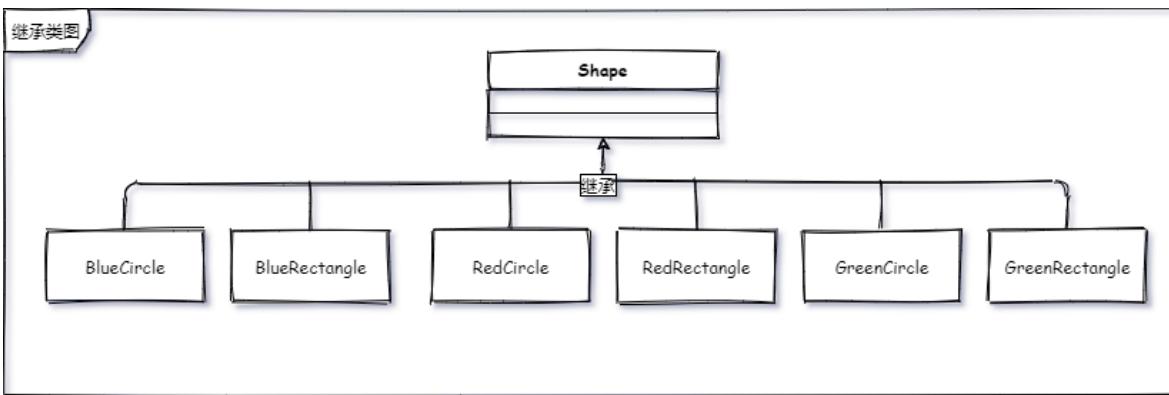
我们采用方案一来实现的话，我们是不是也可以这样来理解呢？为每种颜色都提供各种形状的版本呢？这个是完全的可以的。如下：



图片来源：<https://www.runoob.com/w3cnote/bridge-pattern2.html>

先看看问题吧，一个图形有2种形状（圆形、矩形）和2种颜色（红色、蓝色）的时候怎么去用类表示，我啥也不说，那肯定继承啊，我这封装、继承、多态老扎实了。

心里念着“首先有一个图形的基类，然后开始继承走起 红色的圆形、红色的矩形、蓝色的圆形、蓝色的矩形。”没毛病，一个抽象类，四个实现类，搞定。



代码写完，测一手。

```

@Test
void shape(){
    Shape blueCircle = new BlueCircle();
    Shape blueRectangle = new BlueRectangle();
    Shape redCircle = new RedCircle();
    Shape redRectangle = new RedRectangle();

    blueCircle.create();
    blueRectangle.create();
    redCircle.create();
    redRectangle.create();
}
  
```

蓝色の圆形  
蓝色の長方形  
红色の圆形  
红色の长方形

感觉还可以，这时坐在我边上的大哥说了句，如果再加一种形状呢？

我：“卧槽，你啥时候来的，想要偷窥我学习？”

大哥：“先回答问题，别转移话题”

我：“再加两个类不就行了”， **RedTriangle**、**BlueTriangle**，

大哥：“也还行，如果再这基础上再加一种绿颜色呢？”

我：“额。。。再加三个类 **GreenCircle**、**GreenRectangle**、**GreenTriangle**。。。 (开始声音微弱) ”

大哥：“再加一个椭圆呢”

“emm... 我刀呢！”

“老弟别激动，大哥帮你看看”

**大哥帮忙诊断代码**

大哥：“你这个是 **乱用继承** 导致的类爆炸晚期啊，要是不拔除对这种继承的理解，基本是废了啊”

我：“大哥我还不想放弃，救救我，咳...咳（一口老血咳出）”

大哥：“那你说说看，你都是什么时候用的继承？”

我：“多个类有共同特征的时候，会抽象出来特征，然后使用继承来扩展”

大哥：“嗯，看来你还有救，那你看你现在抽象出来的东西对吗？”

我小声嘀咕：“很多图形，抽象出来个图形，没问题啊”

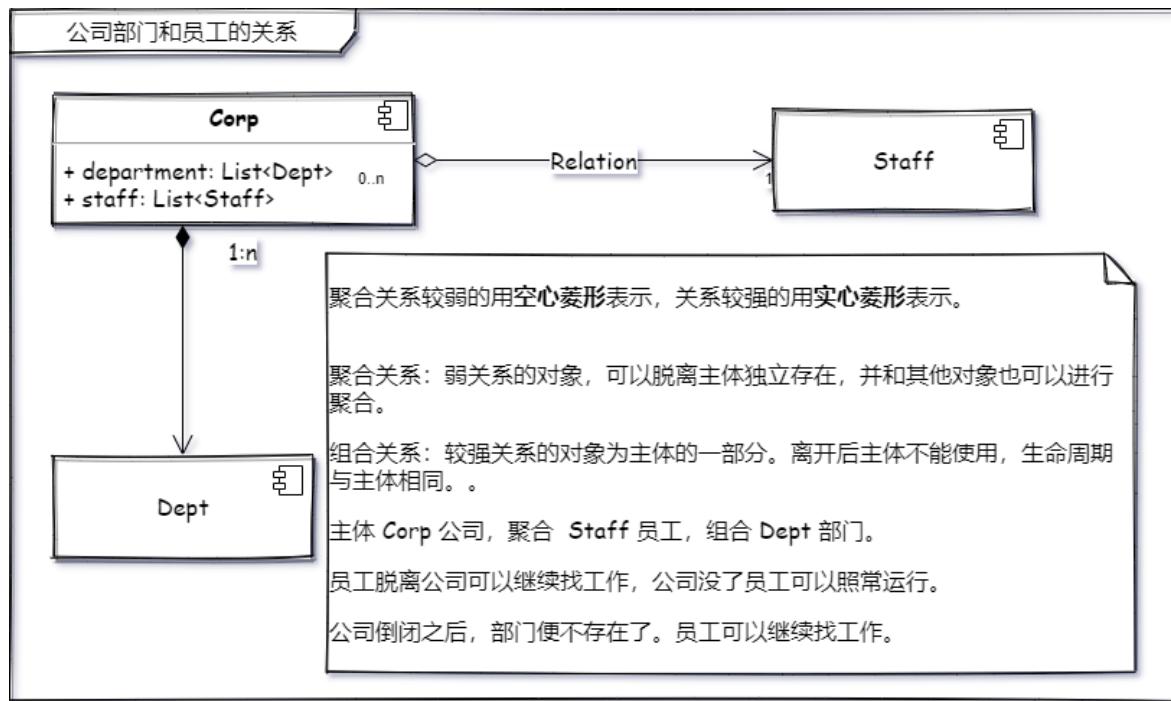
大哥：“那颜色呢？颜色和图形是什么关系？”

我：“emm....，什么什么关系啊？大哥，给点提示吧”

大哥：“UML中的聚合组合我没教你么？”

我：“这个真没有”

大哥：“那这个地方我再教你一次，记着点奥。咳咳！”



大哥：“这个就是组合和聚合的意思，同时他们与主体之间的关联关系的表达。”

大哥：“现在在看你的 **类爆炸** 知道怎么医治了么？”

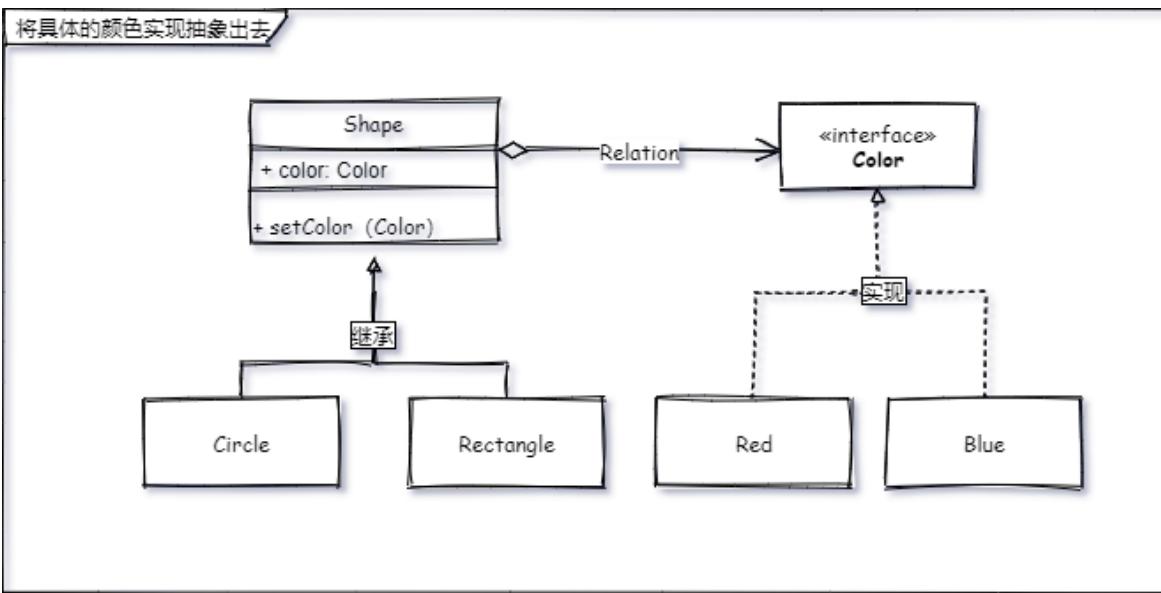
我：“我应该把颜色也抽象出来，然后使用聚合与图形进行关联！对不对！”

大哥：“还不赖嘛，你继续看吧，我忙我的去了”

## 重构代码

领悟了大哥的意思之后，我对代码进行了重构。

仍然将图形类抽象出来，同时将颜色作为一个接口引入，因为图形的形状和颜色本来就是两个不同的维度，所以它现在的类图应该是这个样子的。



有了类图，很快我就重构好了代码，测试一下。

完整代码关注公众号回复：“源码”获取

```

Test Results      73 ms
  ✓ ShapeTest    73 ms
    ✓ shape0     73 ms
      紅色の圆形
      紅色の長方形
      藍色の圆形
      藍色の長方形

Process finished with exit code 0
  
```

当我要新增一种图形或者一个颜色时，只需要增加一个类就可以了。真香。

## 定义

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

## 把这绕口的东西看清楚

将抽象部分与它的实现部分分离，使他们都可以独立地变化。这句话我不知道别人能不能读的懂，就我而言，刚看到这句话实在是没有搞清楚在表达什么，我猜想其中的原因，一个是因为设计模式是搞建筑的人提出来了，另一个原因是老外写的软件设计模式。翻译成中文为了达到统一的标准，所以很多知识变得晦涩难懂。

这里在顺带提一下所谓的统一的标准，就像开放平台的接口一样。他为了有更好的扩展性，定义了统一的对外接口，以后无论哪方想要对接，都需要适应我的标准，而不是给每个人都定制化一个接口。所以知识的传播也一样，要以一定的官方标准来定义和传播，不然可能传着传着就出现了歧义。这也就是复杂度守恒定律的根本，它本身其实真的并不复杂。以上个人见解，可以无视。

# 在看抽象化、实现化、脱。脱。脱你妹啊脱，解耦。

因为之前有大哥的帮助，所以很容易就理解了将抽象部分与它的实现部分分离，使它们都可以独立地变化。这句话。

就拿我刚刚学的图形的那个案例。

- 抽象部分就是图形的形状+颜色，图形它一定是有形状和颜色的。存在自身的两个不同的维度变化
- 实现部分就是具体的形状和颜色。形状和颜色一定有具体的体现。要么圆形红色，要么方形透明。而形状又是图形本身的一部分，所以可以跟在主体后通过继承进行变化。颜色可以独立出去进行单独的扩展。

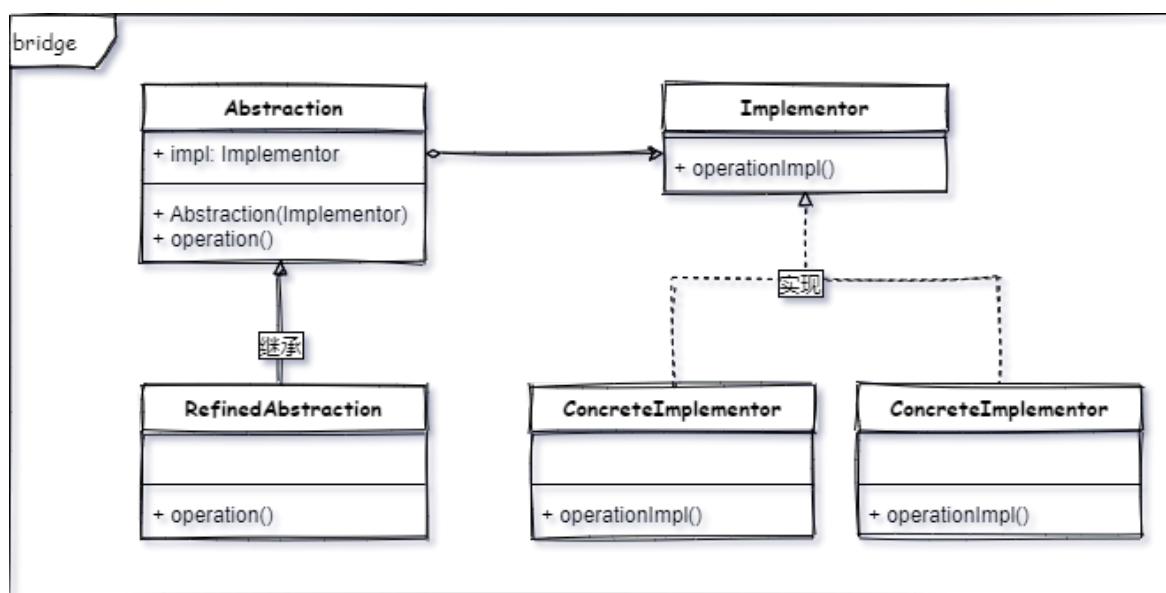
独立的变化就是讲到抽象部分和实现部分的两个实现

- 抽象部分的一个变化就是通过一个矩形类继承图形抽象类。同时完善一个构造函数，这是对抽象部分的矫正或者完备。
- 实现部分的变化遵循了里式替换与抽象部分的关联又根据依赖倒置原则设计。所以实现部分可以在自己的接口定义范畴内进行自由变化，同时又可以与抽象部分进行关联（桥接）

我试着把晦涩的东西简化一下

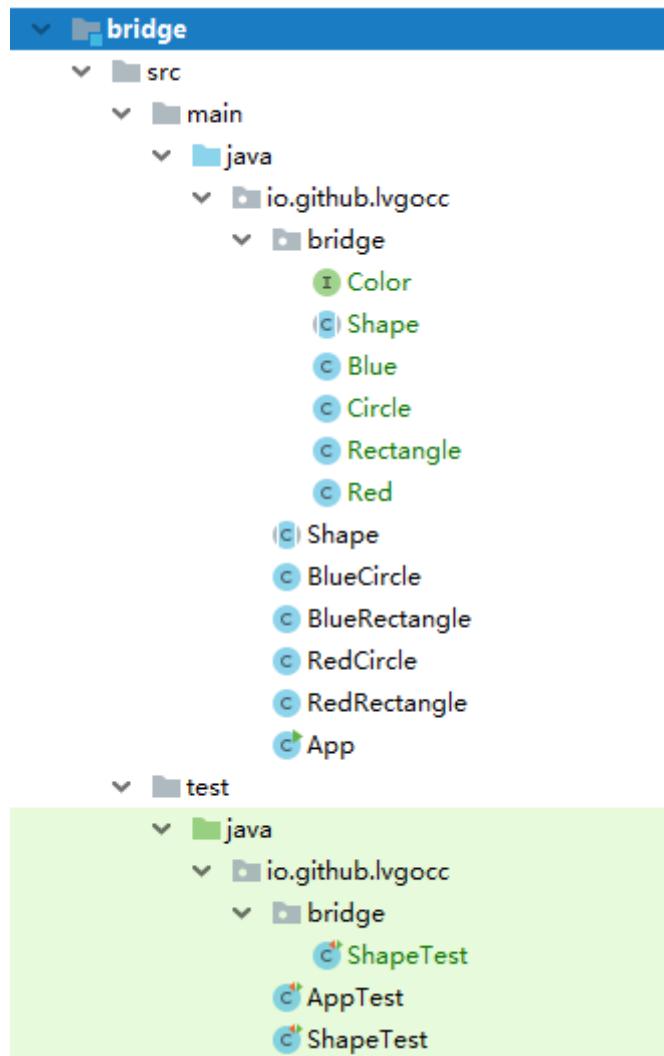
一个对象的多个维度状态独立变化时，将其通过类组合的方式进行关联，使其每个维度自由变化，降低与主体的耦合。

## 桥接模式类图



## 代码

完整代码关注公众号回复：“源码” 获取



## 总结

哎呀，这个桥接模式我是万万没想到它会是这个样子。同样又是学完不知道在哪用的一种模式，但这就是我放弃学习的理由？那可真是太可笑了。

- 当一个对象内存在多个维度多种状态时，可以使用桥接模式解耦，以防新增维度状态时导致 **类爆炸**
- 维度的体现可以延迟到使用阶段，比如上述例子，颜色被分离出去，当需要具体对象时，在通过 `set` 方法对维度赋值（回复源码，获取全部源码和文章原稿）

桥接模式的好处大家都看在眼里，记在心里。用了桥接模式首先解决的就是因为乱用继承导致的类爆炸问题，同时无论之后怎么扩展类，都只需要在对应维度维护新的实现就可以了，降低了对象间的耦合。

不好的地方，整个设计模式的缺点全都包含这一条：**增加了系统的复杂性，对系统设计的理解多了一层内容。维护的类变多了。** 这更能体现出一劳永逸的感觉，先吃苦，后舒坦。其实对于桥接模式还有一点，就是需要你能正确的去划分出一个对象的多维度状态，不然又成了“手里拿个锤子，看什么都像钉子”的感觉了。

---

## 装饰者模式



## 题外话

一直都有看到“包装者模式”出现在一些文章，甚至书中。它们被应用在装饰者模式和适配器模式中，这个原因笔者猜测源自 GOF 最早在书中给模式命名的时候提到了这两个模式的别名 wrapper 同时还有适配器也被成为 wrapper, 所以有人将这几个名称混来混去。后来 GOF 在结尾讲书的简史的时候有提到一些模式的名称变化，其中 glue 改成了 facade, wrapper 改为 decorator , walker 变成了 visitor 。

## 前言

前阵子出于自己学习使用的原因开发了一个 chrome extension , 这样我的 chrome 变得比以前更强了，我赋予了它一个可以保存某个页面的某个片段的位置，后续通过点击这个记录可以快速的回到并高亮当时浏览的记录。



Star Dust 星尘

## 和 lvgo 一起学设计模式 (四) 创建型之原型模式

“啥？盗图、盗文章的人居然用的是一种设计模式！叫原型模式？” by lvgo memo

2020年10月17日 设计模式

## 和 lvgo 一起学设计模式 (三) 创建型之抽象工厂模式

“抽象工厂模式和工厂模式有区别吗？”

我给 chrome 简单的装饰了一下（加了一个插件），它就变强了

插件开源，可以作为基础进行二次开发，想要开发 chrome 插件但是不知道如何开始的可以参考 [星尘的朋友](#) 公众号，回复源码获取

# 不知不觉你已经知道了装饰者的概念。

动态地给一个对象添加一些额外的职责



## # 题外话

一直都有看到“包装模式”出现在一些文章，甚至书中，它们被应用在装饰者模式和适配器模式中，这个原因源自 GOF 最早在书中给模式命名的时候遇到了这两个模式的别名：“wrapper”，所以有人将这几个名称混用。后来 GOF 在结尾讲书的历史的时候有提到，装饰者模式的名称由 glue 改成了 facade，wrapper 变为 decorator，walker 变成了 visitor，所以我们看到的“胶水模式”“包装模式”也就能理解了。

## # 前言

前阵子出于自己学习使用的原因为开发了一个 chrome extension，这样我的 chrome 变得比以前更强大了，我就可以一个可以保存某个页面的某个片段的位置，后端通过点击这个记录可以快速的回到并观看当时的记录。

!memo1!memo1.png  
!memo!memo0.png

\*\*我给 chrome 简单的装饰了一下（加了一个插件），它就变强了\*\*  
“插件开发，可以作为基础进行二次开发，想要开发 chrome 插件但是不知道如何开始的可以参考。关注\*\*\*先生的一个朋友\*\*\*”公众号，回复关键词获取”

## CSS

## 题外话

一直都有看到“包装模式”出现在一些文章，甚至书中，它们被应用在装饰者模式和适配器模式中，这个原因源自 GOF 最早在书中给模式命名的时候遇到了这两个模式的别名：“wrapper”，所以有人将这几个名称混用。后来 GOF 在结尾讲书的历史的时候有提到，装饰者模式的名称由 glue 改成了 facade，wrapper 变为 decorator，walker 变成了 visitor，所以我们看到的“胶水模式”“包装模式”也就能理解了。

## 前言

前阵子出于自己学习使用的原因为开发了一个 chrome extension，这样我的 chrome 变得比以前更强大了，我就可以一个可以保存某个页面的某个片段的位置，后端通过点击这个记录可以快速的回到并观看当时的记录。



Star Dust 星尘

## 和 logo 一起学设计模式（四）创建型之原型模式

【读了这篇文章的人还常用的另一种设计模式】向原型模式？ Go to next

2020年10月11日 读书俱乐部

上面两个举出的例子在实际过程中只要你想，你可以无限的装饰它，所以装饰者的类，可以一直嵌套下去。就像

```
InputStream in = new DataInputStream(new FileInputStream(new File("filePath")));
```

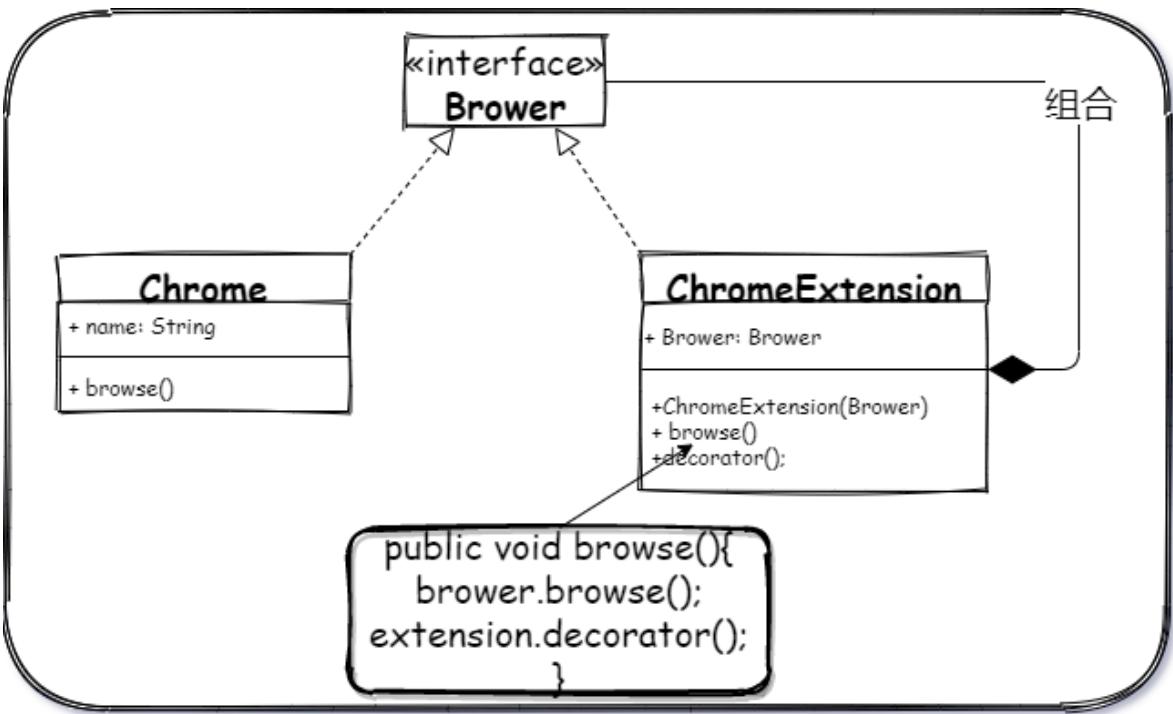
如果IO不熟，看得不理解，那再看看这个

```
List<Object> list = new ArrayList<>();  
list = Collections.synchronizedList(list);
```

其实，当我写到这里的时候已经很清楚装饰者的概念了，它就是通过“套娃”变强了 😊！

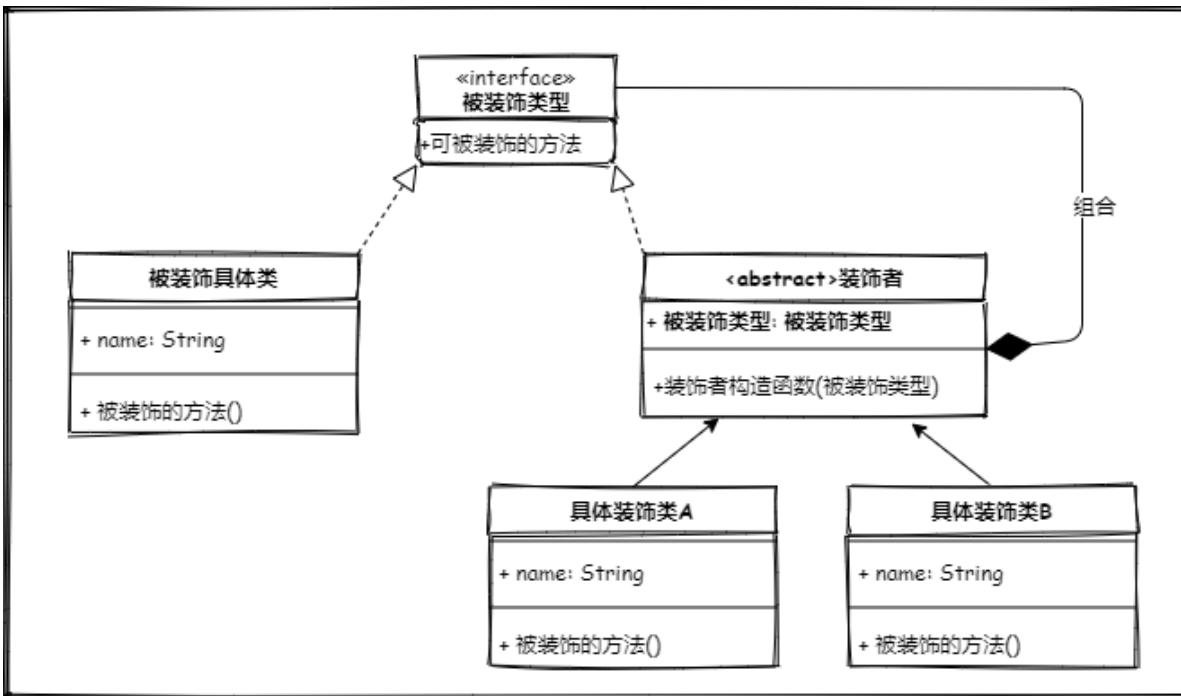
让我继续深入看看它的结构图@@。

## 装饰者模式类图



一个浏览器的接口，一个 Chrome 实现类，一个 ChromeExtension 插件的实现类（用以装饰 Chrome），扩展中的构造函数为 浏览器类型，在插件类中会对浏览器 Brower 的 browse 方法进行一层装饰（增强，或减弱），在不改变对象的情况下，对对象行为进行动态的改变。

上面的类图在抽象一层的话就变成了这个样子

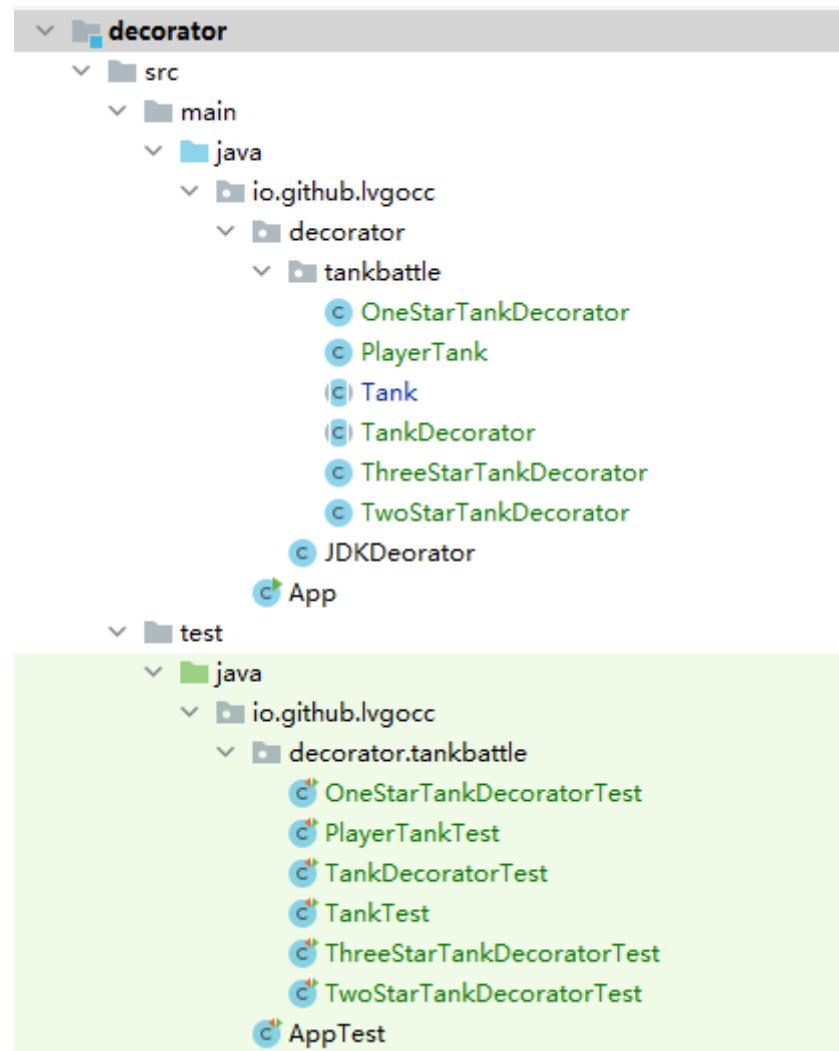


为了示意，名称使用了中文，当然如果能够让人清楚你在干什么，无所谓你怎么表达。

通过对象的组合来实现类的增强要比继承更加的灵活。这也是软件设计原则中的组合复用原则的一种体现，优先使用组合，然后考虑继承。

## 代码

关注公众号：星尘的一个朋友 回复：[源码](#)，获取全部代码和类图



代码演示通过一个游戏获取道具的方式来理解装饰者模式的具体实现；

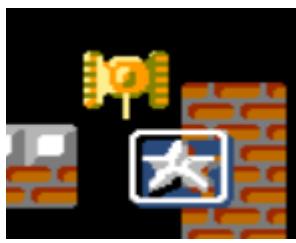
想来想去，我没有选择超级玛丽，图难找，代码不好表达

所以我选择了它 **Tank Battle**



很多回忆都在这里 而且好表达

在这游戏中，我们吃到一个星星☆的时候，就会变强，可以发射两发子弹，同时样子也会发生改变。



吃了这个星（装饰），我变得更强



## 结构组成

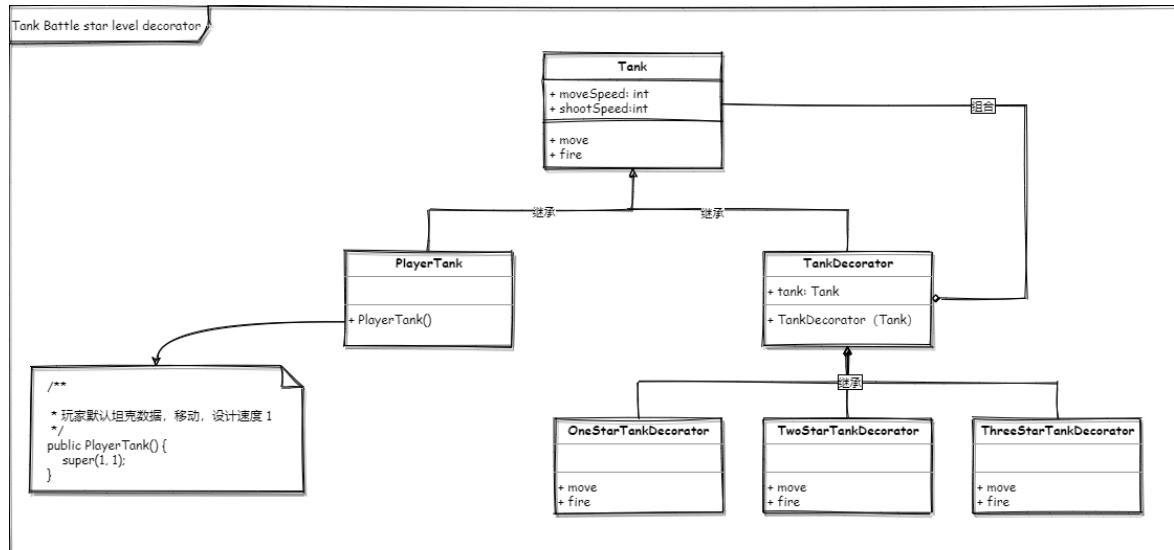
1. 被装饰者接口
2. 具体的装饰者类
3. 抽象装饰者
4. 具体的装饰者

因为星星强化一共有 3 个级别

- 移动、子弹加速
- 连发
- 打掉白色方块

所以我们应该有 3 个装饰者。类角色。

1. 被装饰者接口 -> 坦克 Tank
2. 具体的被装饰者 -> 玩家坦克 PlayerTank
3. 抽象装饰者 -> 用来抽象定义装饰者基本信息，如构造函数等 TankDecorator
4. 具体的装饰者 -> OneStarTankDecorator -> TwoStarTankDecorator -> ThreeStarTankDecorator



## 部分代码

关注公众号：「星尘的一个朋友」回复：「源码」，[获取全部代码和类图](#)

```
/**
 * 玩家坦克
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/25 11:40
 * @since 1.0.0
```

```
 */
public class PlayerTank extends Tank {

    /**
     * 玩家默认坦克数据，移动，设计速度 1
     */
    public PlayerTank() {
        super(1, 1);
    }
}
```

```
/**
 * 定义坦克抽象装饰者
 *
 * @author lvgorice@gmail.com
 * @date 2020/10/25 11:40
 * @since 1.0.0
 */
public abstract class TankDecorator extends Tank{

    protected Tank tank;

    public TankDecorator(Tank tank) {
        this.tank = tank;
    }

    @Override
    protected void move() {
        tank.move();
    }

    @Override
    protected void fire() {
        tank.fire();
    }
}
```

测试结果

```

    ✓ Test Results          14 ms
      ✓ TankDecoratorTest  14 ms
        ✓ tank()           11 ms
        ✓ oneStarTank()     1 ms
        ✓ twoStarTank()     1 ms
        ✓ threeStarTank()   1 ms

D:\java\jdk\bin\java.exe ...

玩家坦克
它移动了，当前移动速度：1
它开火了，当前射击速度：1

吃到一颗星后
它移动了，当前移动速度：2
它开火了，当前射击速度：2
它开火了，当前射击速度：2

吃到两颗星后
它移动了，当前移动速度：2
它开火了，当前射击速度：2
它开火了，当前射击速度：2

吃到三颗星后
它移动了，当前移动速度：2
它开火了，当前射击速度：2
它开火了，当前射击速度：2
白色砖头击碎

```

## 总结

坦克增强的过程是一颗星一颗星获取的一个过程，一直在动态的增强。这个案例中只是一个维度，坦克吃星星。如果在增加一些额外的功能时，比如坦克变身，进化等等，不断的增加装饰时，就可以体会到装饰者模式组合的可扩展性。当然使用继承来实现的话，如果是单一不变的多种状况是很好的，比如说我的玩家坦克的选择不同的外观，可通过不同的子类来确定下来，但如果动态的想要增加一个类的时候，继承就显得非常的困难。

装饰者模式在不改变原对象的情况下，动态的增强具有较好的可扩展性。这也体现了开闭原则。但我们发现，如果你不合理的使用装饰者模式，类的数量会变的更多，且多重装饰使一个对象的维护变的更加复杂。所以，就像前面说的，具体的特性就完全可以用继承来实现而非装饰者模式，装饰者模式一定是使用在想要动态的给对象增加一些功能的时候使用。

- 比如JDK中对IO的操作有一个read()操作，对它进行装饰之后就变成了readLine()。

```
public void io() throws FileNotFoundException {
    InputStream in = new DataInputStream(new FileInputStream(new File( pathname: "filePath")));
}
```

```
  c DataInputStream.java x  c FileInputStream.java x  c JDKDeorator.java x
```

```

204             * @exception IOException if an I/O error occurs
205             */
206             @Range(from = -1, to = 255)
207             public int read() throws IOException {
208                 return read0();
}

```

```

500     @Deprecated
501     public final String readLine() throws IOException {
502         char buf[] = lineBuffer;
503
504         if (buf == null) {
505             buf = lineBuffer = new char[128];
506         }
507
508         int room = buf.length;
509         int offset = 0;
510         int c;
511
512         loop: while (true) {
513             switch (c = in.read()) {
514                 case -1:
515                 break loop;
516             }
517             if (c == '\n') {
518                 if (offset > 0)
519                     return new String(buf, 0, offset);
520                 else
521                     return "";
522             }
523             if (room-- < 0)
524                 resize();
525             buf[offset++] = (char) c;
526         }
527     }

```

- 再比如JDK中的Collections工具类，通过对集合类的装饰，使其变得线程安全，而对象本身却没有发生改变

```

public void collection() {
    List<Object> list = new ArrayList<>();
    list = Collections.synchronizedList(list);
}

```

```

2374     * @return a synchronized view of the specified list.
2375     */
2376     @NotNull
2377     public static <T> List<T> synchronizedList( @NotNull List<T> list) {
2378         return (list instanceof RandomAccess ?
2379             new SynchronizedRandomAccessList<>(list) :
2380             new SynchronizedList<>(list));
2381     }

```

仅仅是对原来的方法前面都加了 `synchronized` 关键字来对原对象做了增强

```

        ...
    }

    public int hashCode() { synchronized (mutex) {return list.hashCode();} }

    public E get(int index) { synchronized (mutex) {return list.get(index);} }
    public E set(int index, E element) { synchronized (mutex) {return list.set(index, element);} }
    public void add(int index, E element) { synchronized (mutex) {list.add(index, element);} }
    public E remove(int index) { synchronized (mutex) {return list.remove(index);} }

    public int indexOf(Object o) { synchronized (mutex) {return list.indexOf(o);} }
    public int lastIndexOf(Object o) { synchronized (mutex) {return list.lastIndexOf(o);} }

    public boolean addAll(int index, Collection<? extends E> c) {
        ...
    }
}

```

而 List 本身仍有更多的子类。Collections 工具类提供的就是对 List 对象做增强。

## 结尾

当我们明白了一件事物的本质之后，再去看表象会变的轻而易举。而这最关键的是要去亲自的操作它，看着再简单不过的东西，你第一次动手都会有很大的收获。这也让我想起了初中物理课本最常见的一句话“**动手动脑学物理**”



任何情况下，看会和听懂都不是掌握。再不济语文课也学过“书读百遍其义自见”也是要动动嘴的。加油！

## 外观模式

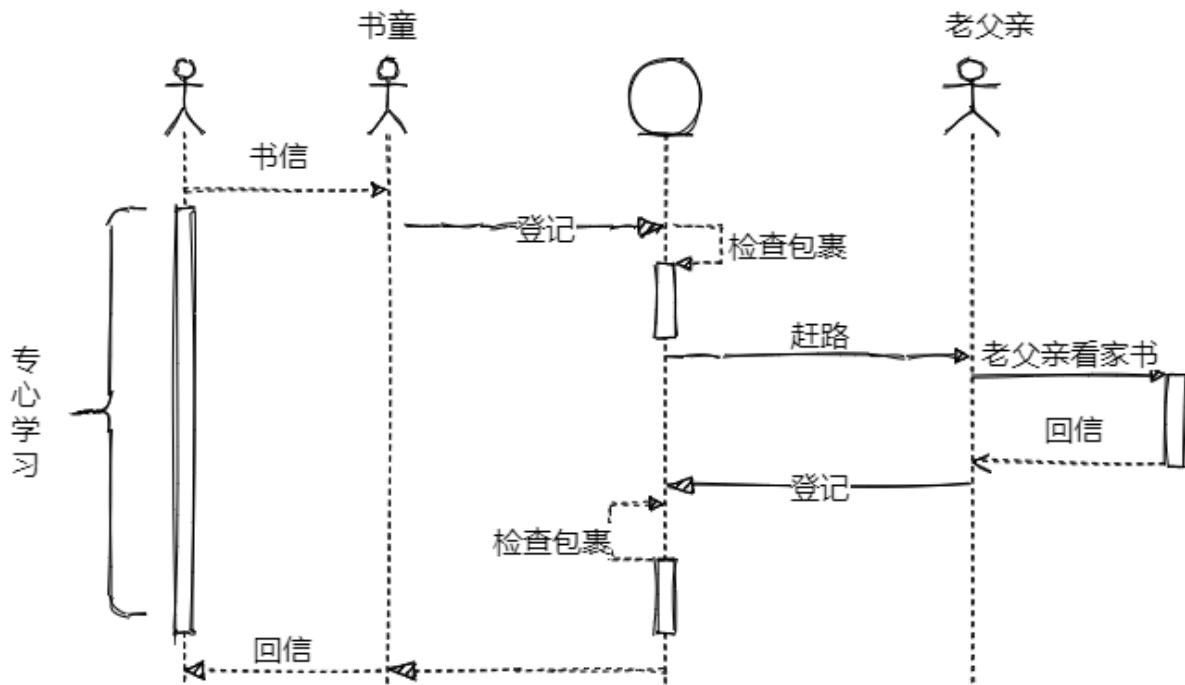


# 外觀模式

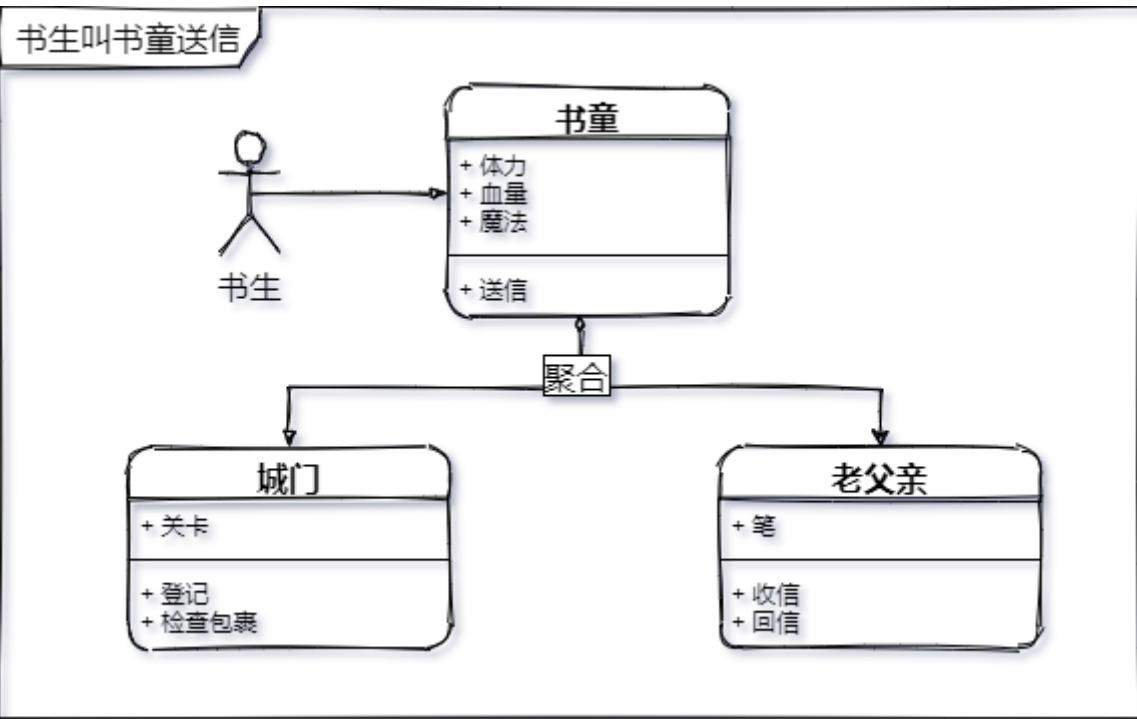
今天用5分23秒，搞懂外观模式，计时~开始！

## 书生的家书

从前，有一个书生，去到很远的地方读书。离开家里久了，难免会思念家乡，于是他便带着书童收拾好行囊，来到城门口登记——接收包裹检查——赶路——到家。几次折腾之后，书生的成绩下滑了，身体也吃不消了，家里觉得这也不是一个长期的办法，于是商量出来一个办法：想家的时候，他便写一封家书，叫自己的书童给他带到老父亲家里。这样一来，书童便拿着他的家书，在城门口进行登记、检查包裹、然后出了城赶路。这使得书生可以专心读书，传递家书的事情，都由书童来做。



书生再也不用为了每次登记检查包裹赶路这些事情费心了，可以**专心的学习做好自己的事情了**。即使哪一天不需要登记了，书生也不需要管，只需要努力学习和想家的时候写好家书送给书童就可以了。



书生为了使自己能够更好的学习，更加专注于自己的“业务”，将一些复杂的过程交给了书童去处理。至于书童要经历什么，书生并不关心，他只关心你给我提供传递书信的“服务”，我给你“信”，你再回给我“信”。

这让我想到了汽车加油，好多人他们只知道没有了去加油站加油就好了，但是并没有多少人清楚石油石化业的艰辛历程。扯远了，回来说我们的问题。

对于书童这种行为，慢慢的演变成了现在的邮局。

## 定义

为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。

## SLFJ

### 第一个想到的外观模式具体的应用.

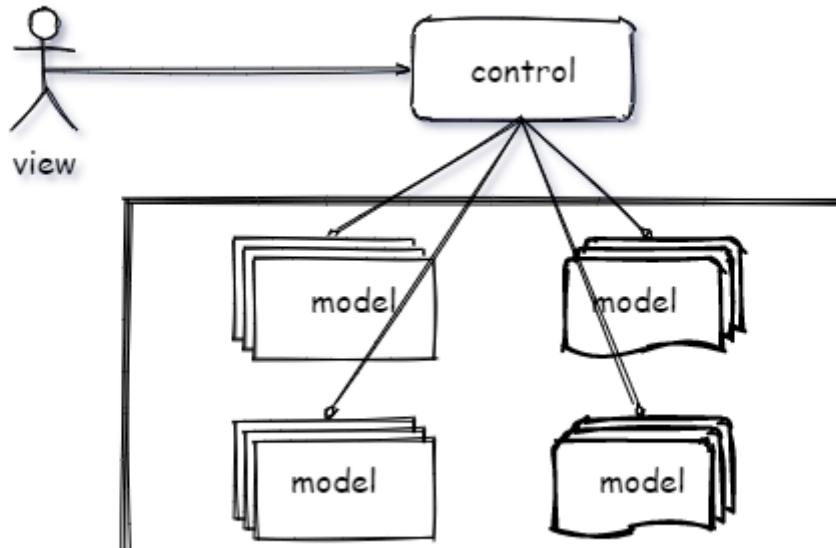
在我学习外观模式之前，我一直在想的事情就是我哪里用到了这个设计模式，脑子里第一个蹦出来的是 SL4J 这个日志 api 框架，他就是一个日志门面。主要的核心思想就是外观模式，他所负责的，就是书童干的事。提供你一个方法，你把参数给他，他给你返回一个记录日志的对象，至于这中间的复杂过程，你不需要知道。

赠送一个用不到的知识点：log4j、logback、sl4j 都是出自同一个作者，这也是为什么 logback 天然支持 sl4j 真香的原因之一。

## MVC

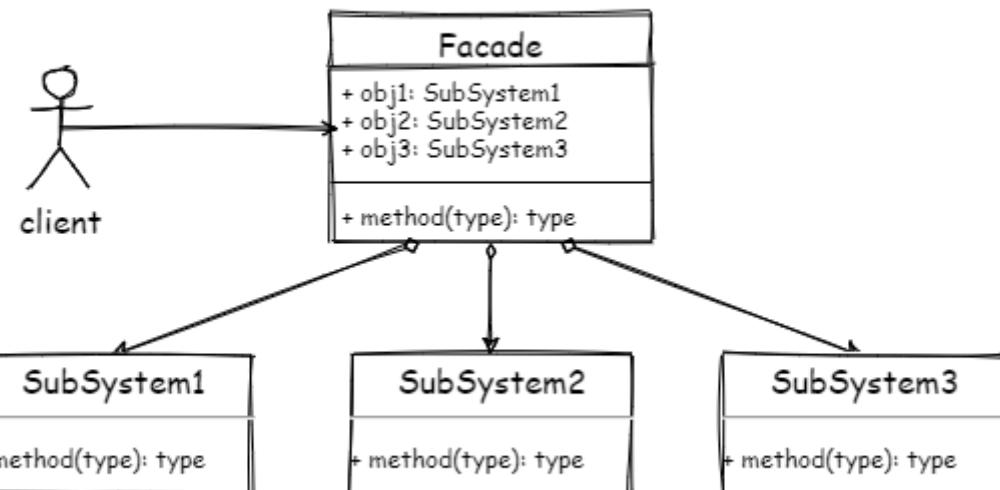
我想，基本上知道编程的人，应该都知道什么是 **MVC**，**model**、**view**、**control**。但大家仔细研究过这三者的关系吗？当然今天不是主要来介绍 **MVC** 的，而是通过 **MVC** 来认识外观模式。其中 **V** 就是书生，**C** 就是书童，**M** 就是具体的送信执行过程。**书生**（客户端 view）永远都不需要知道**书童**（服务端 control）是如何把**信送**（服务端 model 业务实现）**过去和拿回来的**。

## MVC



## 外观模式类图

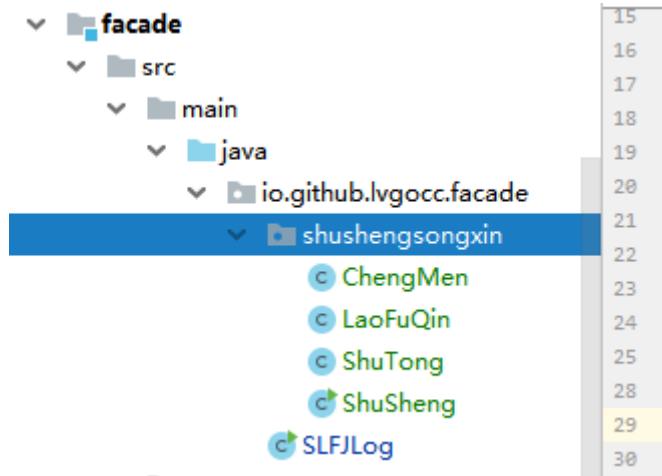
### 外观 (facade) 模式类图



## 代码

回复“源码”获取书童送信全部代码

我们通过代码在看一下书童是如何帮助书生完成送信的，看看他路上有没有偷懒。



```
/**  
 * 书生送信 - 书生  
 * <p>  
 * 欢迎跟我一起学习，公众号搜索：星尘的一个朋友  
 * 也可以加我微信（lvgocc）拉你进群  
 *  
 * @author lvgorice@gmail.com  
 * @version 1.0  
 * @blog @see http://lvg.org  
 * @CSDN @see https://blog.csdn.net/sinat_34344123  
 * @date 2020/11/2  
 */  
public class ShuSheng {  
    static final Logger LOGGER = LoggerFactory.getLogger(ShuSheng.class);  
  
    public static void main(String[] args) {  
        LOGGER.info("书生写好信给了书童");  
        ShuTong shuTong = new ShuTong();  
        shuTong.songXin();  
        LOGGER.info("书童拿回了信给了书生");  
    }  
}
```

.ShuSheng - 书生写好信给了书童  
.ChengMen - 登记中.....  
.ChengMen - 检查包裹中.....  
.ShuTong - 赶路中...  
.LaoFuQin - 老父亲看信中...  
.LaoFuQin - 老父亲写信中.....  
.ShuTong - 赶路中...  
.ChengMen - 登记中....  
.ChengMen - 检查包裹中....  
.ShuSheng - 书童拿回了信给了书生

我们通过代码可以看出，整个过程书生也没有参与送信的具体过程，这些全部都由书童（外观角色）来完成，这其中设计了2个子系统，城门和老父亲。

## 总结

迪米特法则又叫作最少知识原则 LOD/LKP，1987 年美国东北大学 (Northeastern University) 的一个名为迪米特 (Demeter) 的研究项目....(更多内容关注公众号点击“设计模式”专题序章查看);

外观模式是一个遵循了迪米特法则的一种设计模式。书生只知道送信找书童，但他不需要知道还要登记、检查等等其他的事情。

外观模式又称作门面模式，理解成门面感觉会更好的理解是不是。至少我认为更好理解一些。就像人的脸面嘛，比如有一件事叫我做，你就跟我说嘛，你也不知道我怎么想的，怎么做的，我给你反馈结果就好了。当然例子可能不是很恰当，希望我要表达的意思能让你看的清楚。

外观模式主要特征

1. ✓ 降低了系统间的耦合度，子系统的变化不会影响高层模块的调用。
2. ✓ 提高了高层模块的使用理解。我只想送信，给我接口。
3. ✗ 违背了开闭原则，其实你只要没有遵循依赖倒置，就一定会在功能增加或变更的时候违反开闭原则。如果想要不违背开闭原则，那就需要将依赖的具体的类，转成依赖抽象的类或接口。

**外观模式的扩展：**上面的案例还可以可以使用一个接口来代替书童的位置，然后让书童来实现这个接口，后面如果想要新增一种送信的工具人实现或者继承送信的接口就可以了。

## 写在最后

一转眼设计模式已经学了 10 种了。时间也过去 3 周了，有些内容可能已经慢慢的开始忘记了，我偶尔也会打开之前的文章在看上一下，如果第一遍第二遍仔细的看过，那后面再看就可以很快的复习一遍。学习任何知识也是如此，如果第一遍第二遍都是走马观花的看，后面每次看都会觉得很陌生，如果前面可以仔细的看过，然后定期的复习，很多东西都可以很容易做到，只是时间的问题。当然我这种都是比较笨的学习方法，但是还是蛮有效的。也特别希望大家能够在群里一起讨论一起学习复习，因为每次交流都是一次加深印象的时刻。

**真正掌握一个知识的时候，便是你能把它教给别人的时候。**

---

## 享元模式



运用共享技术来有效地支持大量细粒度对象的复用。

这个设计模式在 GOF 的书中是用 flyweight 这个词来定义这种模式的，然后翻译成中文就叫 享元 了，讲真挺不自在，首先这个词是一个自造词（享元）

抱歉，没有找到与享元相关的结果。

建议您尝试变换检索词，或者去[百度一下>>](#)

再者就是这个词语我认为用轻量化的解释更合适，不过现在被翻译成享元肯定是有他的原因的，至于为什么翻译享元已经不重要了，这都不会影响我们学习的对不对！

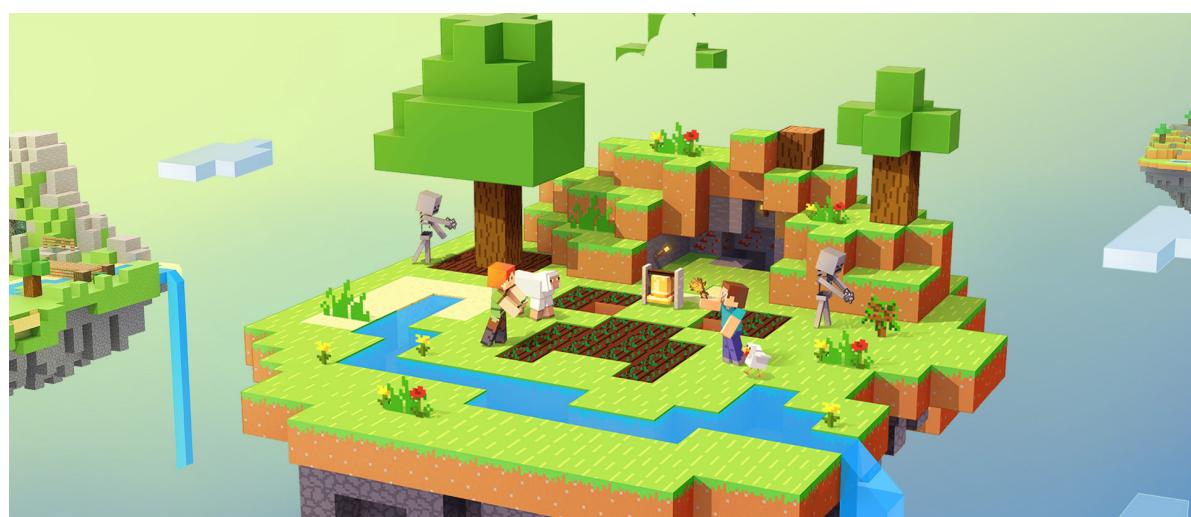
## 如何理解

我们先一起来理解一下这个词的意思，然后再说这个模式解决的问题，希望我的解释能帮你快速的了解这个设计模式的意图。

我是这样理解的。享元，共享单元。什么意思呢，将一些资源共享，以减少一些不必要的资源消耗。我接着举几个例子说明一下；为了代入感更强，我就拿游戏举例了。

**声明：以下内容只为学习类比使用，并不代表游戏设计方案，游戏如何设计实现，我未参与，也未研究，感兴趣的可自行了解。**

### 1. 我的世界



游戏地图

我们都知道我的世界是一个自由度超高的沙盒游戏。进到游戏之后我们应该会看到一个画面，就是地图在不断的渲染。这里可能以前玩的时候大家都沒有注意过，只是觉得游戏好大，但是不怎么卡。不卡的原因有很多。我们今天要说的就是如何通过享元模式来减少资源负担。

假如我的世界地图中每个单位格子的内容大小为1kb，粗略估计一个画面内格子的数量为1,000,000，此时加载地图需要1GB的内存，如果每个格子2kb则2GB。如果一个单元格内容所用的贴画是10kb呢。目前来看10G内存也都能接受，可这款游戏放在当年的话，估计不会有人玩了。

### 如何解决

其实这个方案非常的正常，也非常的简单。首先我们可以这样做，事先将需要用到的格子贴画统计好，然后一次加载到内存中，记录一下内存的地址，需要用的时候，直接取出来渲染就好了。他们的样子都差不多，只是摆放的位置不同。还有一种方式呢，就是我用一个先去我的[资源库](#)找，找不到就创建一个放到资源库中，如果能够找到，就直接返回。这两种方式都可以。第一种方式将压力放在了启动过程，第二种的方式将压力放在第一次渲染的过程。而一般情况下，游戏的开发都是用第一种方式，也就是我

们所说的“过图”，“地图加载”。这个时候去做的，因为一次卡顿加载完和你走着走着卡一下当然第一种更容易接受。

## 2.英雄联盟



英雄联盟这款游戏大家应该并不陌生，S10刚刚结束（10月31日全球总决赛），SN来年再战，加油。

### “兵线”

游戏中一共有3路兵线，每次出现几只我不清楚，8只好。3路乘以2（双方）然后在乘以8，这应该是48个对象。而且他们还包含各自的动作，比如魔法兵吐得“口水”，炮车喝奶茶吐的“珍珠”等等，如果你在开发兵线系统的时候，内存爆炸了，比如有的玩家搞怪，不杀小兵，积攒了很多小兵，然后他卡了，说你游戏垃圾。你该如何去做呢。

其实我们分析下来的话，这里只会出现三种不同的兵种，步兵、魔法兵、炮车。然后再分为红蓝两方。在加上两个子弹。是不是就只有这8个对象呢，至于他们的轨迹，那些是每个对象的“外部状态”

## 如何构成

知道了这种设计模式思路，就要继续了解一下享元模式具体的构成角色都有哪些了。比如以英雄联盟的兵线为例吧。

### 客户端

首先有一个客户端，负责获取对象，然后渲染，这里我们通过#get、#draw(x,y)来表示获取和画来代替这步动作，(x, y) 表示渲染出来的对象坐标。

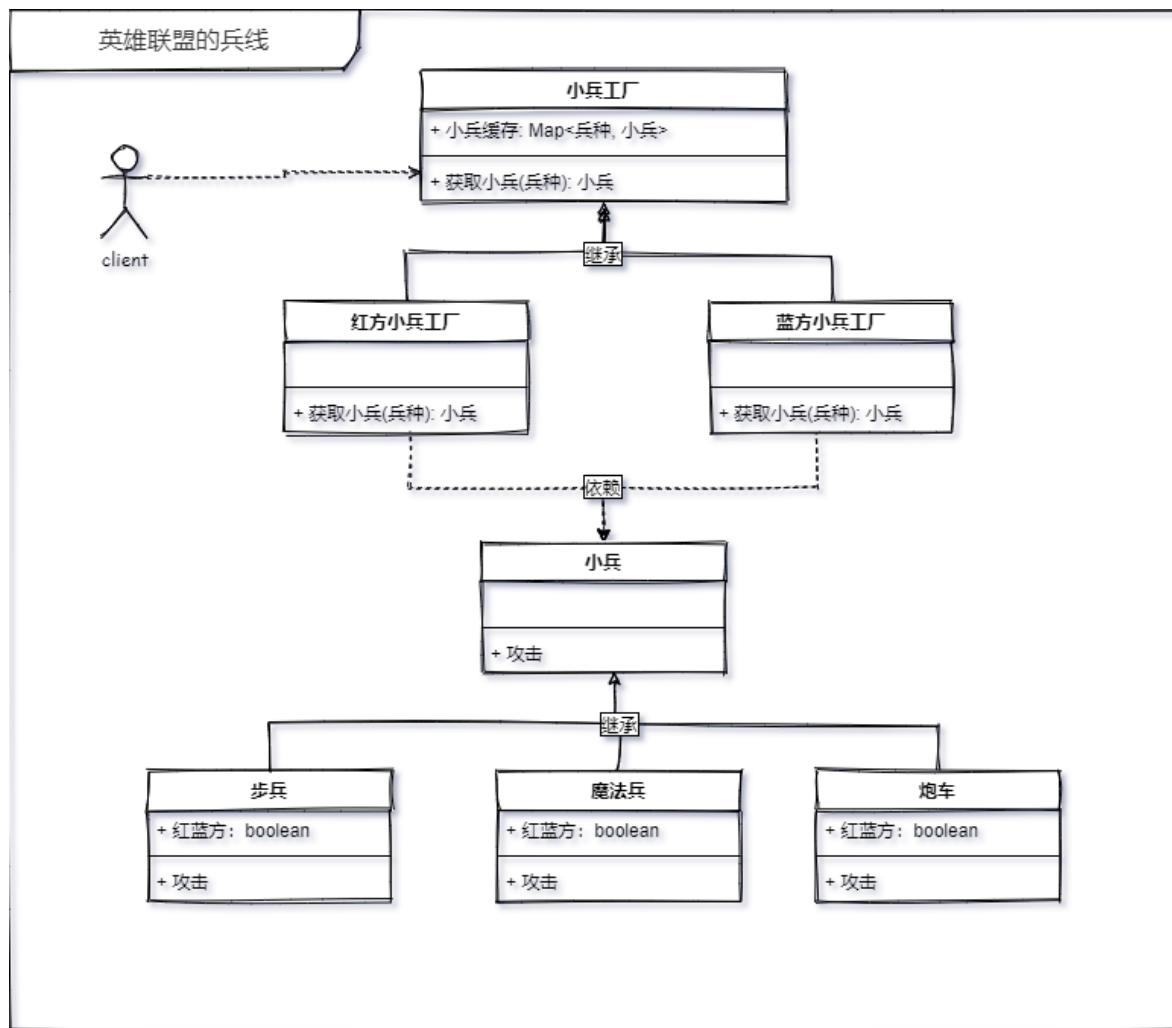
### 享元工厂

然后还有一个为我们提供小兵的统一接口，这里使用的就是我们前面学习的工厂方法，小兵工厂。这里顺便复习一下之前的工厂和抽象工厂两个设计模式。如果我通过一个工厂来实现小兵对象的创建，那么就是一个工厂模式，但是我现在想在应用的时候，在灵活一些，我们可以从小兵身上抽取特征，比如步兵、魔法兵、炮车、这是小兵类别，但我们有两个不同的作战方，红方和蓝方，所以此时可以使用抽象工厂模式来生产小兵，红方小兵工厂生产出来的都是红方的步兵、魔法兵、炮车。蓝方生产出来的就是蓝方的步兵、魔法兵、炮车。

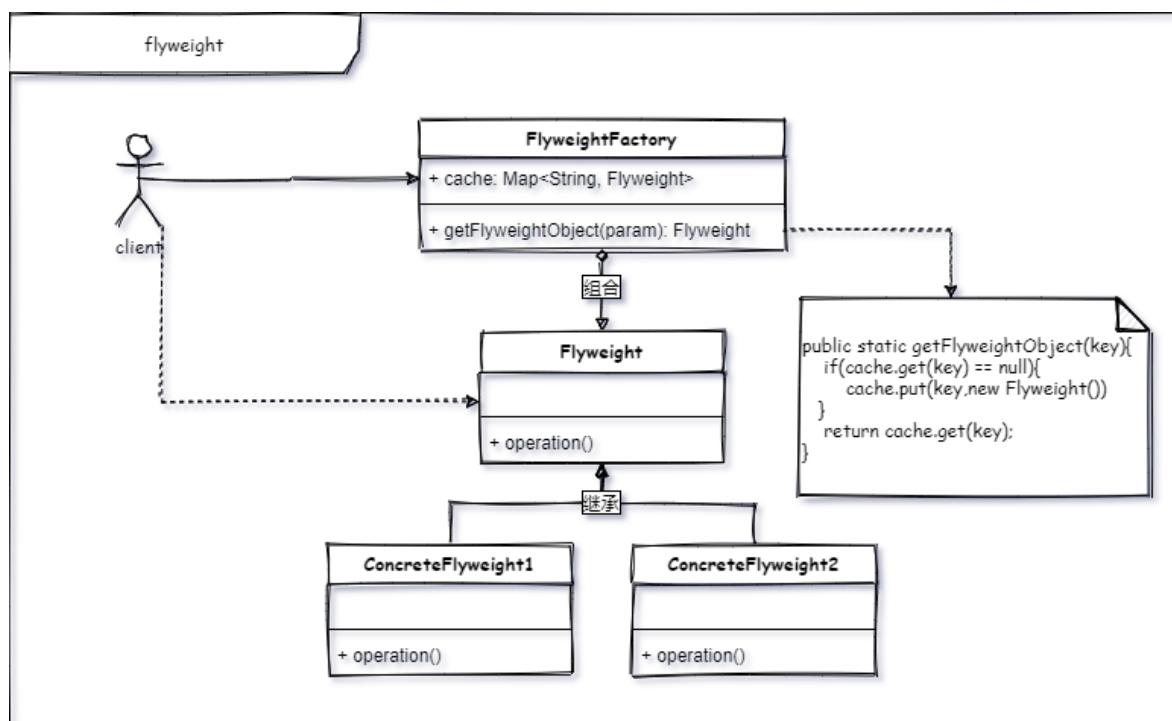
### 享元对象

有了工厂之后，我们就要有具体的共享对象了，共享对象就是我们上面所说的那8个。

下面这个类图顺便复习了一下 **抽象工厂模式**。

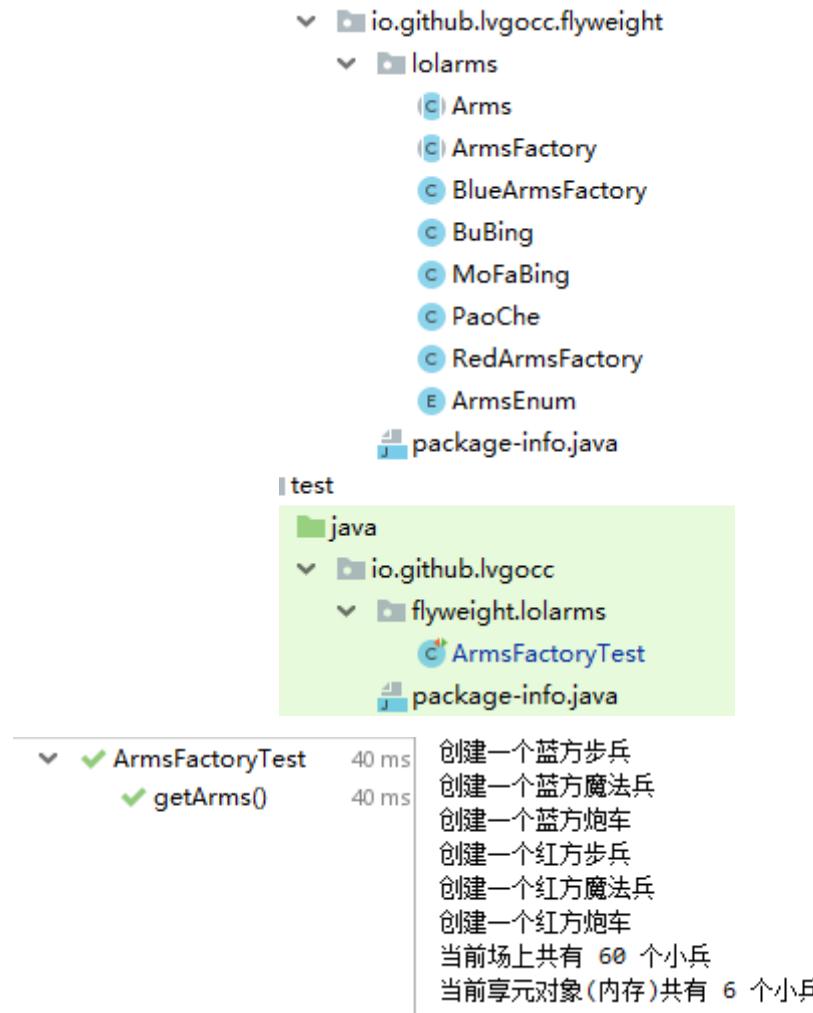


## 享元模式类图



## 代码

下面就使用 **享元模式** 来模拟一下英雄联盟的兵线的开发。



关注回复“源码” 获取享元模式创建LOL兵线代码。

## 总结

- 通过享元模式可以让我们用更小的空间来构造一个更大对象。这也是利用了池技术来实现的。
- 使用享元模式可以有效的缓解内存使用的问题。
- 你会发现，当你有外部状态的时候（具体体现在红蓝两方在创建小兵对象的时候，需要指定颜色），享元模式会变得稍显复杂。

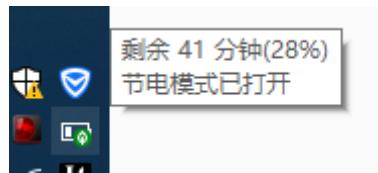
## 组合模式



将对象组合成树形结构以表示“部分 - 整体”的层次结构。组合使得用户对单个对象和组合对象的使用具有一致性

今天要学习的东西有点意思，因为我看到了很多有意思的内容，也在这里记录分享给现在正在阅读的你。

被拉来图书馆，电脑马上扛不住了，搞快。



## 如何理解

上面的定义是 **Gang Of Four** 在《可复用面向对象软件的基础》中对组合模式的意图阐述的内容。

不兜圈子，我先说下我是怎么理解这个设计模式的：**把对象以树形结构放在一起，想要用的时候，操作组合（抽象）对象和操作任意一个对象是一样一样的。**

在学习组合模式之前，我认为它就是把多个对象组装放在一起变成一个更大的对象，这就是我对组合模式最初的理解。但当我自己亲手使用组合模式来编写一段代码的时候我发现，实际情况和我所理解的还是有所差别的。当然这其中的差别只是在于具体的实现上，如果你对组合模式的理解停留在  $A + B = C$ ，我觉得这也是正确的。不过还有一个关键的内容，就是 **操作组合对象和操作任意一个对象是一样一样的** 的。

在这过程中，我问了身边的几个小伙伴，大家都说了组合模式是一个树结构。这也说明了  $A + B = C$  的形式体现是以树结构形式体现的，后来又在攀谈中聊到了具体的应用，比如：应用的菜单、组织架构，还有 pom（这里的 pom 要站在 maven 角度来看，而不是站在 xml 这种树结构来看）。

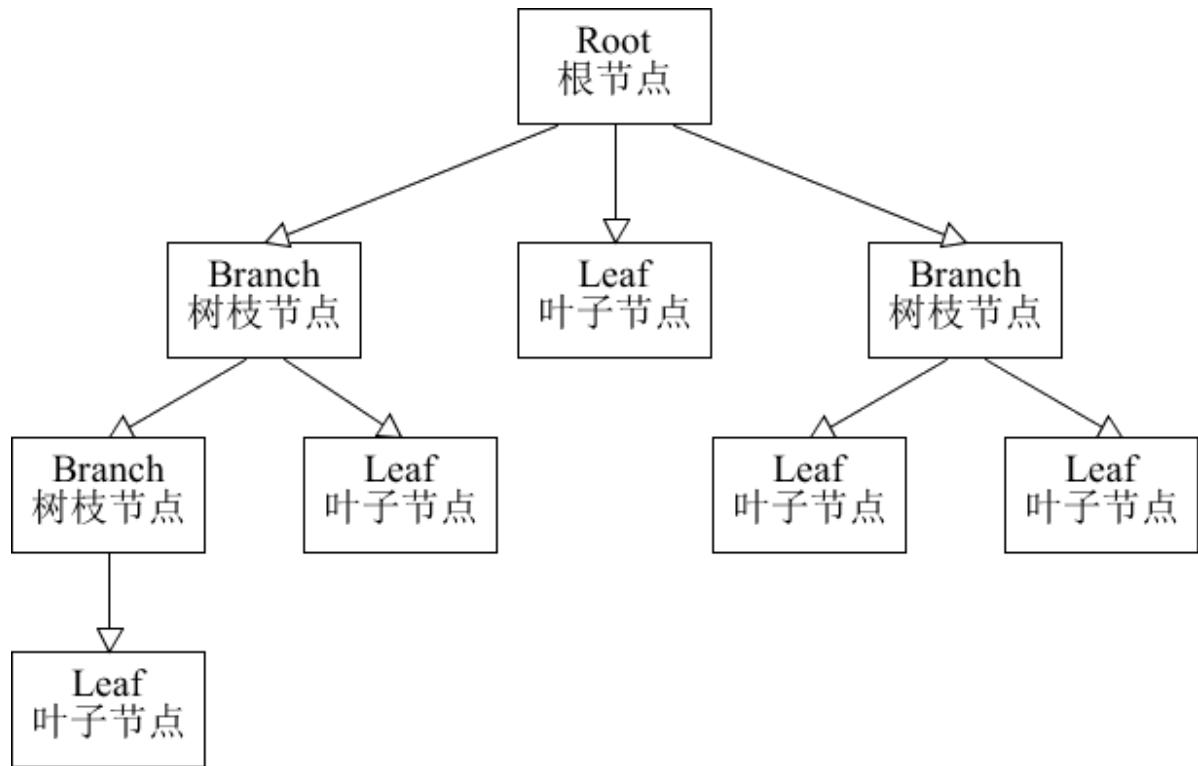
## 树形结构

上面的内容说到了 **组合** 模式的一个关键的定义内容，就是它的表现形式是以树形结构来呈现的，这里还想在墨迹一点东西就是组合模式只是利用了树结构这种形式的结构。

## 一致的访问

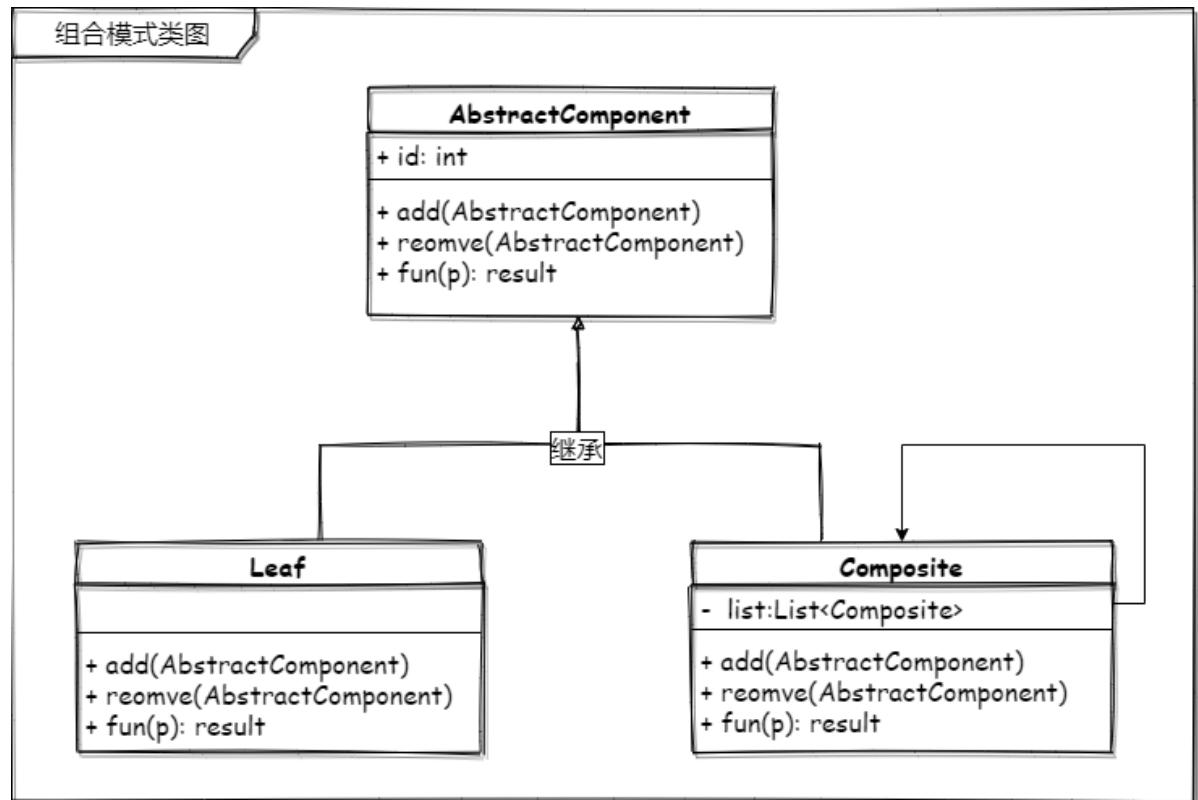
至于后半段的定义，**对单个对象和组合对象的使用具有一致性** 理解成对树形结构当中的根节点、子节点、叶节点的访问方式都是一样的。

放一张图



图片来源 <http://c.biancheng.net/view/1373.html>

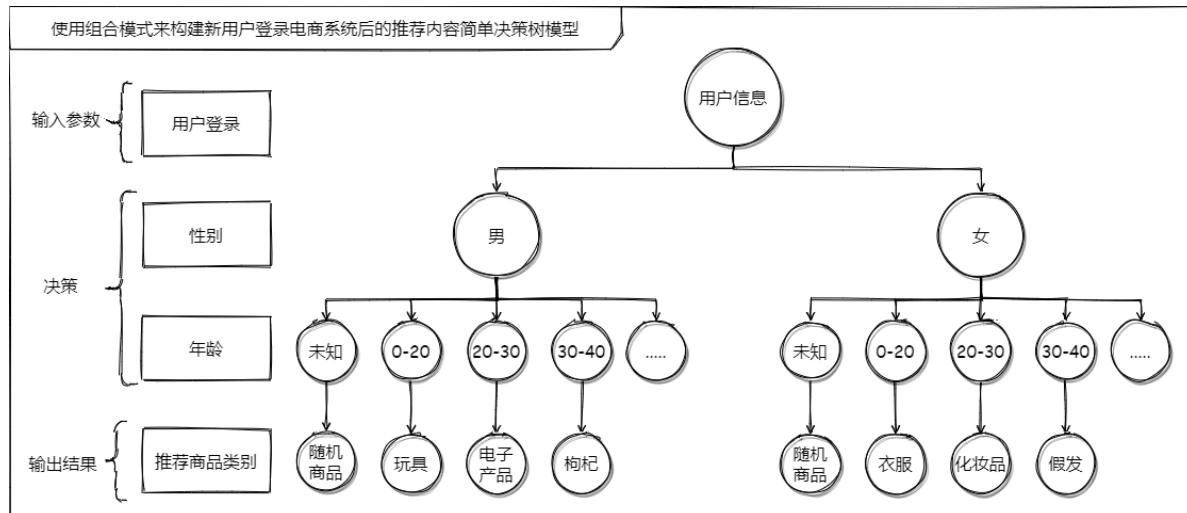
## 组合模式类图



# 让我们一起利用它做点事

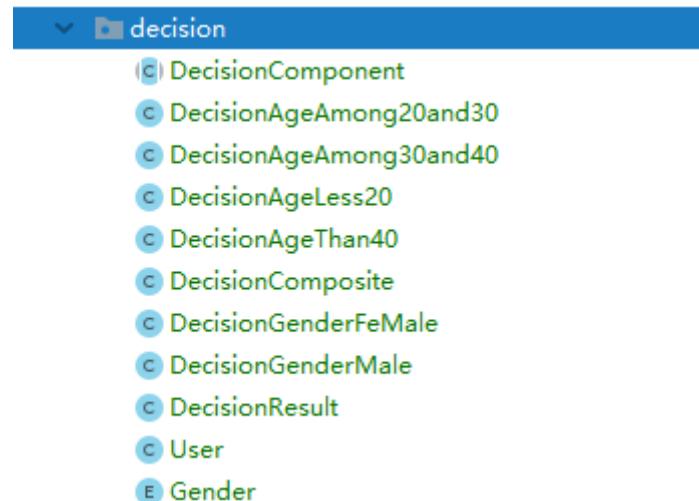
这个案例的想法来自 傅哥 火遍全网的《重学设计模式》中组合模式的案例

现在有一个需求，新注册的用户要进行关键信息的填写，性别、年龄。然后在首页根据用户信息进行一些商品的推送。拿到这个需求的时候是不是想着一顿 if else 猛如虎的操作来完成呢？当然我一开始也是这样想的哦，但谁知道产品经理哪天头皮发痒再给我来一个职业、地区、消费能力。。。为了满足产品未来的欲望，我想到了这个。



通过代码实现以上结构后，

篇幅原因完整代码关注回复“源码”获取



核心代码

```
/**  
 * 决策  
 *  
 * @param user 用户信息  
 * @return 决策结果  
 */  
protected DecisionComponent decision(User user) {  
    if (judge(user)) {  
        logger.info("进入 {} 决策分支", getName());  
        for (DecisionComponent decisionComponent : decisionComponents) {  
            if (decisionComponent.judge(user)) {  
                return decisionComponent.decision(user);  
            }  
        }  
    }  
}
```

```

        }
    }
}
return null;
}

```

输入参数：男性、35岁

输出结果：

```

[main] INFO io.github.lvgocc.composite.decision.DecisionComposite - 进入 用户信息 决策分支
[main] INFO io.github.lvgocc.composite.decision.DecisionGenderMale - 进入 性别男 决策分支
[main] INFO io.github.lvgocc.composite.decision.DecisionAgeAmong30and40 - 进入 30-40年龄之间 决策分支
[main] INFO io.github.lvgocc.composite.decision.DecisionCompositeTest - 决策结果: DecisionResult[type='枸杞']

```

## 还有个内容要知道

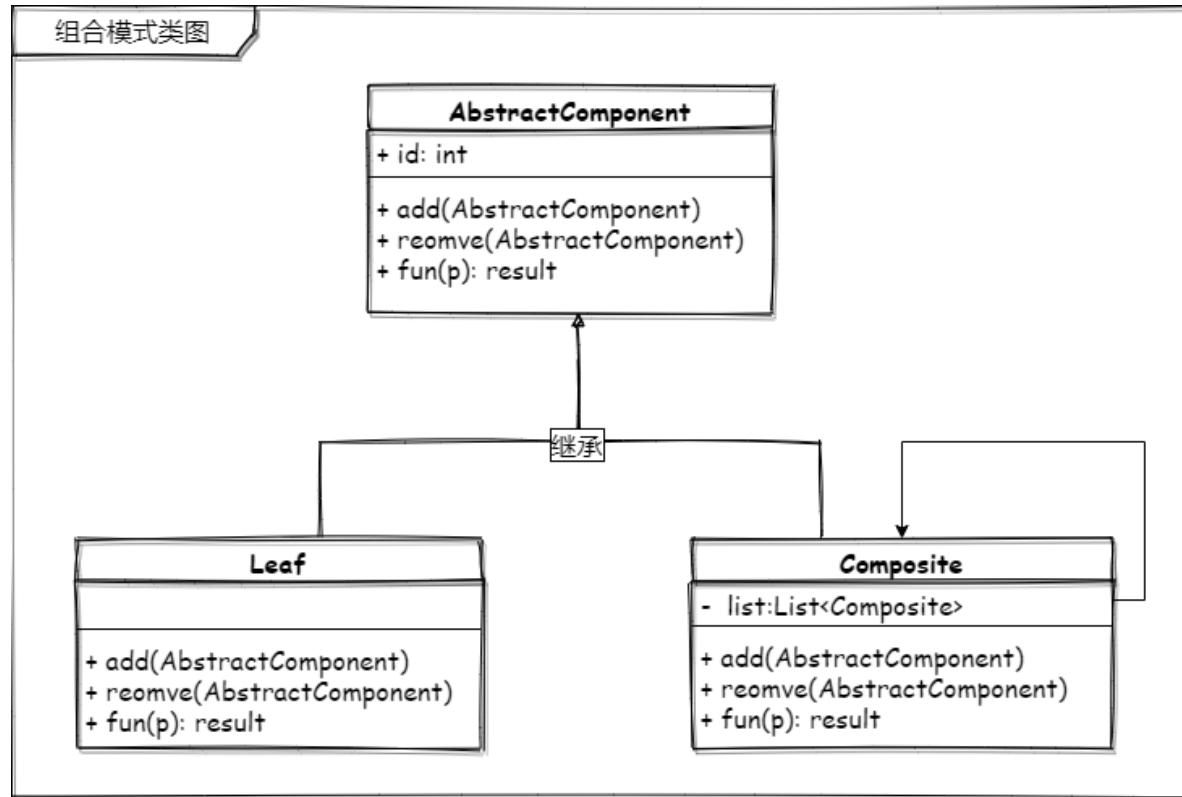
### 透明方式和已知（安全）方式

关于组合模式除了树形结构、一致的访问，还有一个就是它具体的呈现方式，这个呈现方式指的是对于 **客户端** 也就是高层模块，呈现方式有两种

- 透明的，高层模块不需要去区分是子节点还是叶子节点，一样的去使用，但是对于叶子节点，某些功能可能会失效或出现一些特殊的情况
- 已知（安全）的，需要高层模块自己对子节点或是叶子节点的使用进行选择。

对于透明和已知再通过一个 UML 类图和上面的类图对比加以说明

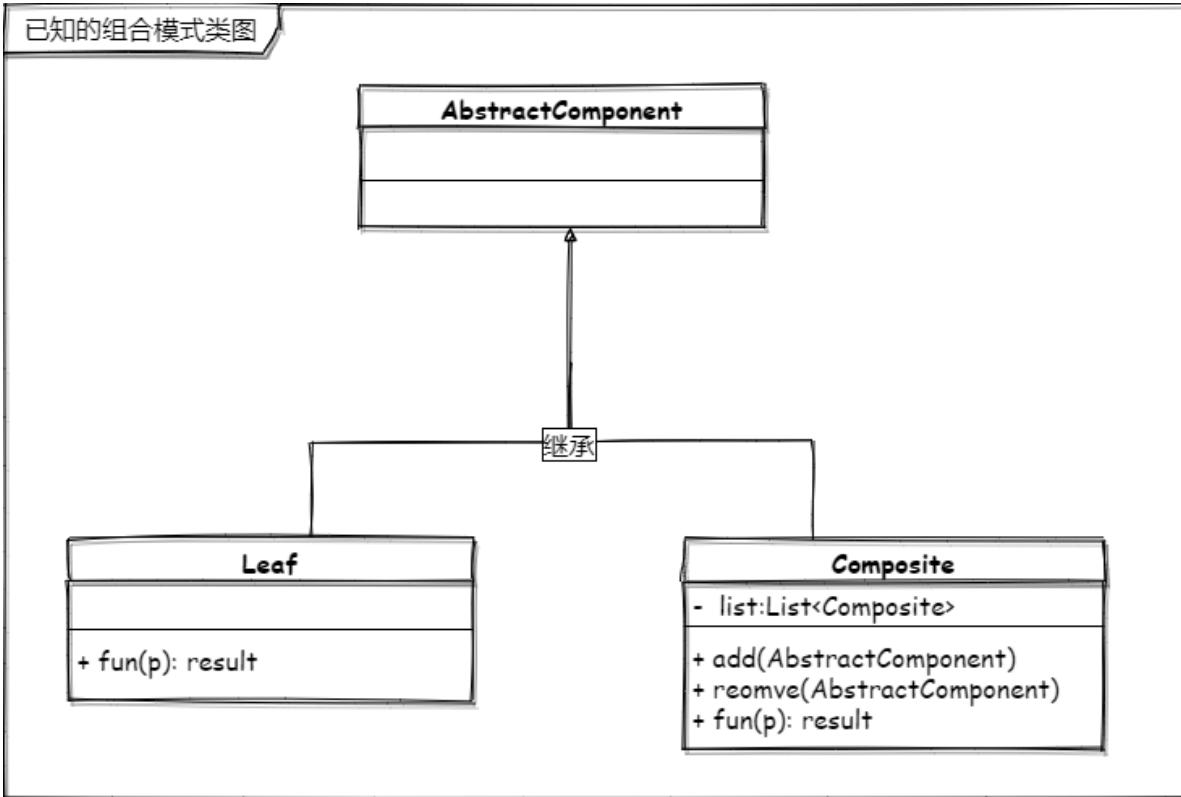
### 透明的组合模式类图



透明的组合模式希望各个节点（子节点、叶节点）行为与抽象节点一致，这样即高层模块无需关心是否是子节点还是叶节点，方法一样的使用，但是对于子节点，因为其没有继续的分支，所以一些方法是没有具体的实现的，这就导致这些“空方法”高层模块是不知情的，所以称为透明的。

### 已知的组合模式类图

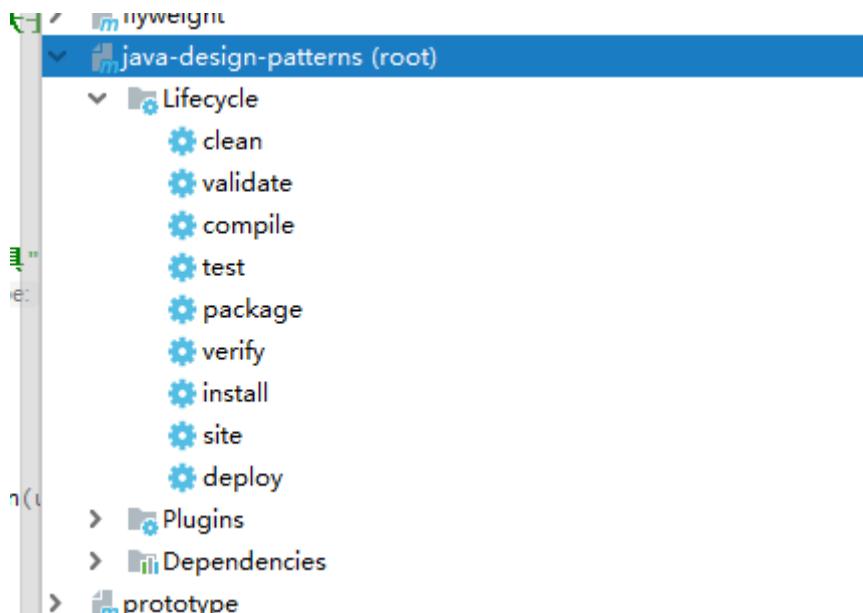
### 已知的组合模式类图



这个已知的名字是我起的，书上大多说的是安全方式。要表达的意思就是高层模块需要知道自己调用的节点是子节点还是叶子节点。

## 发现身边的组合模式

相信大家都用过 maven 来管理多模块项目，maven的结构主要分为三类，继承、聚合、依赖，以下这些命令在 root 模块执行的时候，就可以将整个项目完成对应的操作，当你在单个模块中使用的时候，他也只会影响单个模块或该模块以下的模块。



## 行为型(11)

用于描述类或对象之间怎样相互协作共同完成单个对象无法单独完成的任务，以及怎样分配职责。GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

## 模板方法模式



定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类在可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

### 冲啊！

最近经常看《四驱兄弟》，脑子已经被“冲啊”洗掉了。

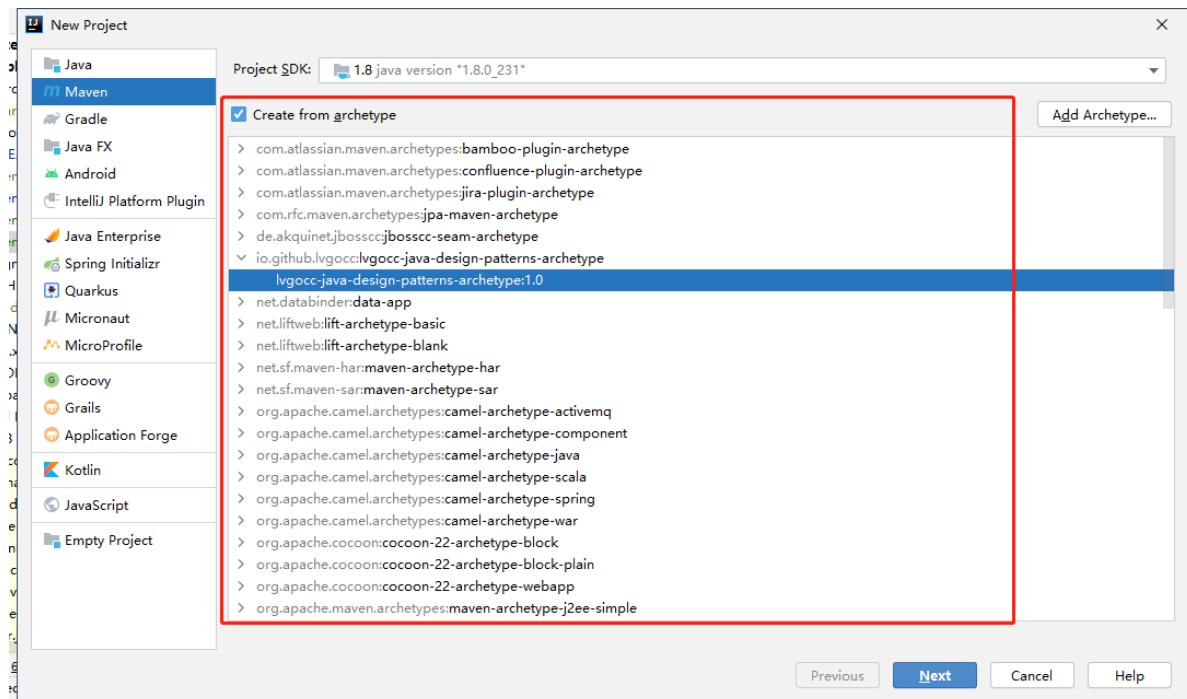
“冲啊，就让我们一路领先到底吧！”，“冲啊，看我的胜利冲锋！”，“冲啊！先驱音速！”，“去吧！三角箭！”，“冲啊！疾速斧头！”，“冲啊！二郎丸特别号！”，“冲啊！”

终于来到了设计模式的“人性”部分，行为型模式，为什么说行为型模式是“人性”部分呢，因为行为型模式当中的 11 种设计模式对理解都非常的友好啊。所以接下来的内容可能让我学起来说不定更有趣些。

### 如何理解

定义一个操作中的算法骨架，说白了这不就是一个步骤约束吗？在看第二段，将算法的一些步骤延迟到子类中，意思就是步骤里的一部分留给你了，具体怎么做看你（子类）自己了。使得子类在可以不改变该算法结构的情况下重定义该算法的某些特定步骤，这句就更好理解了，有了步骤的约束，你负责执行具体的步骤，说白了，步骤只要执行就可以，不管你怎么做，所以也就有了不改变结构的情况下可以重新定义特定的步骤，这里的特定指的就是约束步骤里留给你的那部分步骤。

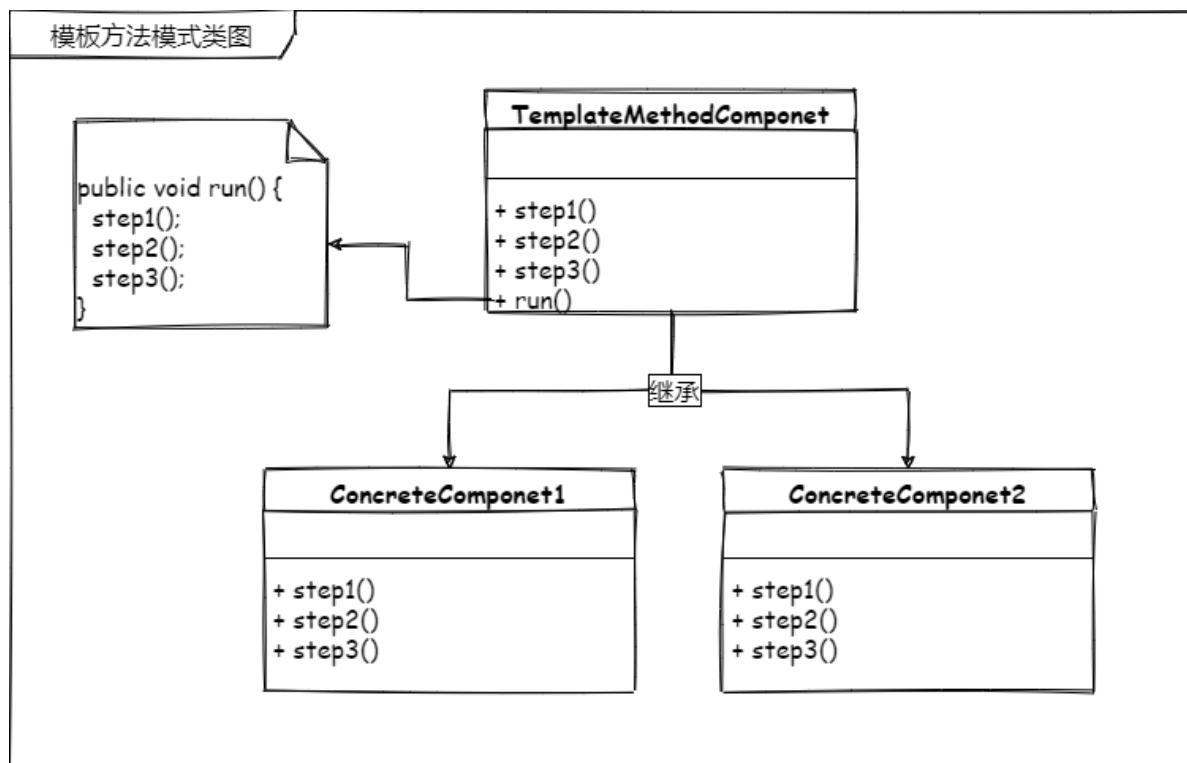
- 比如我们经常写 PPT 的小伙伴知道，在 PPT 中，点击新增一页 PPT，创建出来的页面，大部分的布局格式都是已经设置好的，我们只需要去填充里面的内容就可以了。
- 再比如经常写需求说明书的时候，会和同事要一份“模板”，这也是一种模板方法模式的体现。
- 还有如果你使用过 maven 创建过项目，那这个你一定不陌生



## 重新定义

模板方法模式就是：提供一个具体的步骤，1,2,3,4，现在1,2,4都写好了，步骤3的实现由具体的执行者说了算，只要步骤符合要求，随你发挥。

## 模板方法模式类图



看类图就是抽象和子类的样子，而实际模板方法模式就是利用了“抽象”。是一个完全遵守开闭原则的设计模式。可以说，搞懂了模板方法模式，开闭原则基本就通了。

**注意：**模板方法模式中与我们平时继承抽象类有一个关键性的区别，就是入口方法，正常抽象类继承是不需要有这个所谓的入口方法，可以通过入口方法来确定算法的执行顺序，即算法骨架。

# 我怎么用模板方法模式

模板方法模式可以说是非常简单的一种设计模式了，虽然简单，但它的作用却很大。比如我们经常使用的lock 锁，它的实现就利用了 AQS，而 AQS 就是使用 **模板方法模式** 维护的一个锁框架，通过它可以快速的开发出一个锁。这步可以结合 AQS 的代码来看一看。

第一步：通过 Lock 接口来约束一个锁所需要的几个关键方法（其实这也可以说是一种模板，只是约束力很小）

```
public class MutexLock implements Lock {  
  
    private final Sync sync = new Sync();  
  
    @Override  
    public void lock() {  
        sync.acquire(1);  
    }  
  
    @Override  
    public void lockInterruptibly() throws InterruptedException {  
        sync.acquireInterruptibly(1);  
    }  
  
    @Override  
    public boolean tryLock() {  
        return sync.tryAcquire(1);  
    }  
  
    @Override  
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {  
        return false;  
    }  
  
    @Override  
    public void unlock() {  
        sync.release(0);  
    }  
  
    @Override  
    public Condition newCondition() {  
        return sync.newCondition();  
    }  
}
```

## 核心代码

第二步：具体的锁实现，这个类的约束力比较强，因为我们想偷懒，借助 AQS 来实现一个锁，所以就要按照它所提供的模板要求来完成对应步骤的代码逻辑，也就是上面提到的（使得子类在可以不改变该算法结构的情况下重定义该算法的某些特定步骤。）这些需要我们去写的步骤就是 AQS 留给我们的“特殊步骤”

```
final static class Sync extends AbstractQueuedSynchronizer {  
    @Override  
    protected boolean tryAcquire(int arg) {  
        if (compareAndSetState(0, arg)) {
```

```

        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    } else {
        return false;
    }
}

@Override
protected boolean tryRelease(int arg) {
    if (compareAndSetState(1, arg)) {
        setExclusiveOwnerThread(null);
        return true;
    } else {
        return false;
    }
}

Condition newCondition() {
    return new ConditionObject();
}
}

```

关于 AQS 留给我们的“特殊步骤”可以在源码中看到

我们自己定一个锁，然后调用 acquire 方法

```

public class MutexLock implements Lock {

    private final Sync sync = new Sync();

    @Override
    public void lock() {
        sync.acquire(1);
    }
}

```

之后 AQS 按照它的模板继续执行，在需要的时候（特殊步骤）会调用我们自己提供的方法，锁具体的实现要自行实现，模板类 AQS 不提供具体实现。

The screenshot shows the Java code for the `AbstractQueuedSynchronizer` class. It highlights several methods:

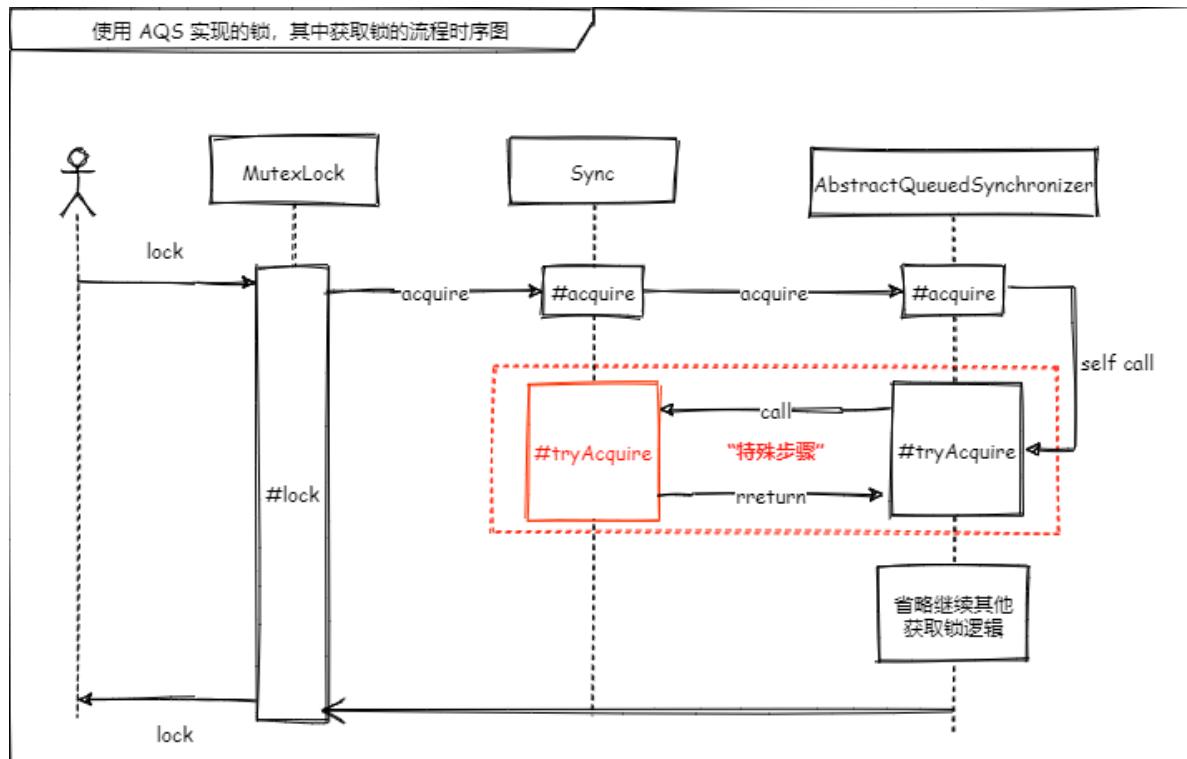
- `tryAcquire(int var1)`: This method is annotated with a yellow box and the note "这里是获取锁的实现" (This is the implementation of acquiring the lock).
- `tryRelease(int var1)`
- `tryAcquireShared(int var1)`
- `tryReleaseShared(int var1)`
- `isHeldExclusively()`
- `public final void acquire(int var1)`: This method is highlighted with a red box and the note "这里是获取锁的入口" (This is the entry point for acquiring the lock). It contains logic to check if `tryAcquire` failed and if so, add the current thread to a waiter queue.

```

// 重写 AQS 的特殊步骤，如果不写会抛出上述异常
@Override
protected boolean tryAcquire(int arg) {
    if (compareAndSetState(0, arg)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    } else {
        return false;
    }
}

```

通过一个图来理解这个过程



其中“特殊步骤”就是 AQS 模板留给我们要实现的地方。

最后，测试一下我们自己写的锁

```

class MutexLockTest {
    private static int f = 0;

    @Test
    void lock() throws InterruptedException {
        Lock lock = new MutexLock();
        int threadCount = Runtime.getRuntime().availableProcessors();
        CountDownLatch signal = new CountDownLatch(threadCount);
        int loop = 100000;
        for (int i = 0; i < threadCount; i++) {
            Thread thread = new Thread(() -> {
                int l = 0;
                while (l < loop) {
                    lock.lock();
                    try {
                        f++;
                    } catch (Exception e) {

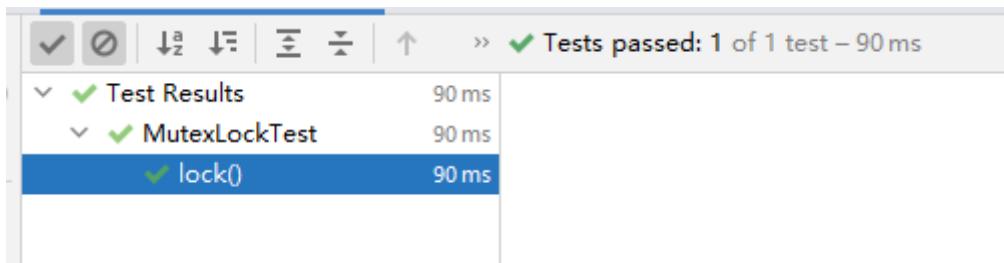
```

```

        } finally {
            lock.unlock();
        }
        l++;
    }
    signal.countDown();
});
thread.start();
}
signal.await();
Assertions.assertEquals(threadCount * loop, f);
}

```

## 运行结果



这里建议大家配合源码学习，同时也能学习一些与锁实现上的一些相关知识，如果有不清楚或觉得有疑问的地方，欢迎加我微信一起讨论（lvgocc）

## 总结

通过模版模式可以将一些我们想要约束的执行步骤固定下来，同时对于步骤中重复的内容可以进行抽象，这样就可以简化很多子类的操作，也避免了一些不必要的冗余代码产生。

使用模板方法模式可以非常简单的来约束一段逻辑的执行要求。在做程序扩展限制时，非常有用。定义好具体的逻辑流程抽象类，将公共部分代码写在抽象类中，然后将其中需要使用者自行实现的方法定义为抽象方法，这样当他继承这个类的时候，只需要将对应的抽象方法实现就好了，不需要关系其具体的执行顺序。

但同样的，这样会使**执行顺序对使用者完全透明**，如果抽象的方法较为复杂时，对于一个初次使用该逻辑的人来说，出现 bug 可能会使他很“难受”，因为他需要搞清楚整个抽象类中每个方法的执行顺序才能更好的去解决问题。这一点，**增加了系统的复杂性**。不过，设计模式的复杂性，是不可避免的。**在功能复用、提高生产力上来说，这点复杂性的代价，还是能够接受的。**

---

## 策略模式



策略模式最早的时候是在马老师的坦克大战看的，讲的很干，也挺清楚。现在回想起来更是记忆犹新。

说到策略模式，最应该关注的应该就是策略这个词语了吧。这个词我直接贴一段百度的翻译，大家看一下

[cè lüè] ↗

## 策略 (汉语词语)

编辑 讨论 (18) 上传视频

策略，指计策；谋略。一般是指：1. 可以实现目标的方案集合；2. 根据形势发展而制定的行动方针和斗争方法；3. 有斗争艺术，能注意方式方法。

提取和设计模式相关的两个含义

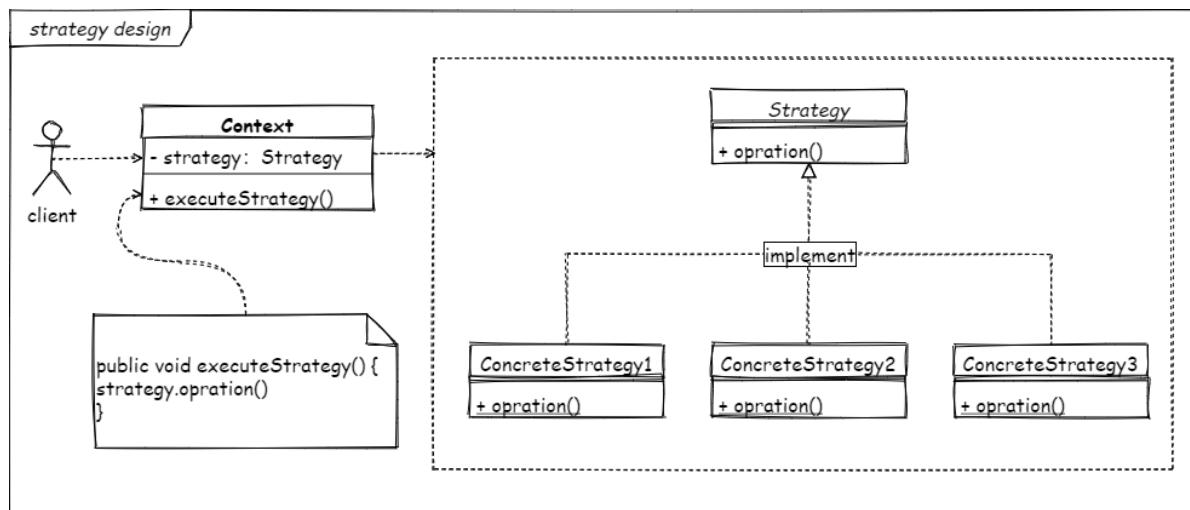
1. 可以实现目标的方案集合
2. 根据不同情况选择不同的策略

然后我们再看看标准的策略模式定义

定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

有了这个基本的清晰概念再学起来就很轻松了。

## 策略模式类图



定义了一个策略接口，然后将每种不同的策略实现同一个接口。这样各个策略之间就可以进行切换。在使用的过程中，可以将策略当成参数传到具体的方法中执行（这里要用函数接口），或者在客户端调用之前就将指定好具体的策略。

## 站点的主题切换

相信大家都使用过可以切换主题的站点，这次我也是尝试使用策略模式来实现这个功能。

**需求：**

1. 用户可以根据自己的喜欢自行选择预设的 3 个主题配色

**3 个主题配色：**

**1. 七彩斑斓的黑**

- 背景色：backgroundColor 黑色
- 字体颜色：fontColor 灰色

**2. 五颜六色的黑**

- 背景色：backgroundColor 黑灰色
- 字体颜色：fontColor 白色

**3. 绚烂多彩的黑**

- 背景色：backgroundColor 灰黑色
- 字体颜色：fontColor 黑色

**代码实现：**

```
/**
 * Theme 主题接口
 * <p>
 * 欢迎跟我一起学习，微信（lvgooc）公众号：星尘的一个朋友
 *
 * @author lvgorice@gmail.com
 * @version 1.0
 * @blog @see http://lvgo.org
 * @CSDN @see https://blog.csdn.net/sinat_34344123
 * @date 2020/11/21
 */
public interface Theme {

    void show();
}
```

```
/**
 * Context
 * <p>
 * 欢迎跟我一起学习，微信（lvgooc）公众号：星尘的一个朋友
 *
 * @author lvgorice@gmail.com
 * @version 1.0
 * @blog @see http://lvgo.org
 * @CSDN @see https://blog.csdn.net/sinat_34344123
 * @date 2020/11/21
 */
public class Context {
    private Theme theme;
```

```
public Theme getTheme() {
    return theme;
}

public void setTheme(Theme theme) {
    this.theme = theme;
}

public void show() {
    theme.show();
}
}
```

测试代码

```
/**
 * ThemeTest
 * <p>
 * 欢迎跟我一起学习，微信（lvgoice）公众号：星尘的一个朋友
 *
 * @author lvgorice@gmail.com
 * @version 1.0
 * @blog @see http://lvgo.org
 * @CSDN @see https://blog.csdn.net/sinat_34344123
 * @date 2020/11/21
 */
class ThemeTest {

    @Test
    void show() {
        Context context = new Context();
        System.out.println("七彩斑斓的黑");
        context.setTheme(new ColorfulBlack());
        context.show();

        System.out.println("五颜六色的黑");
        context.setTheme(new MotleyBlack());
        context.show();

        System.out.println("绚烂多彩的黑");
        context.setTheme(new SplendidBlack());
        context.show();
    }
}
```

测试结果

七彩斑斓的黑

- 背景色: `backgroundColor` 黑色
- 字体颜色: `fontColor` 灰色

五颜六色的黑

- 背景色: `backgroundColor` 黑灰色
- 字体颜色: `fontColor` 白色

绚烂多彩的黑

- 背景色: `backgroundColor` 灰黑色
- 字体颜色: `fontColor` 黑色

其实要说这个策略模式的实现，它本身就是一个非常简单的写法。只是可以通过更多的思想融入可以使其变得更加灵活好用，同时也会变得复杂起来。这里一起看看一个经典的策略模式的实现，就是 JDK 中的比较器。在 JDK 中就是不同的类型有不同的比较算法，这也是符合了策略模式的思想。我再把策略模式的定义拿过来看一看

定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

JDK 为每种不同的数据类型定义了一系列的算法，并将每个算法封装起来，使他们可以通过 `Comparator` 接口进行相互替换，对于客户端来讲，尽管调用比较方法就可以了，即使算法发生了改变（替换其他算法）也不会影响到客户端的使用。

## JDK 比较器

JDK 中的比较实现有两种，一种是直接通过实现 `Comparable` 接口，定义其他对象与自己的顺序（比较）。而另一种就是通过使用策略模式来实现的比较器 `Comparator` 接口。接下来就一起看看 JDK 是怎么运用策略模式设计的比较器。

首先是策略模式中的第一个关键的定义，定义一系列算法。根据开闭原则，这里定义了一个接口，然后将具体的一系列算法实现延迟到子类当中去。

比较器 - 策略接口

```
public interface Comparator<T>
```

一系列算法 - 具体策略

String 的比较算法

```
public static final Comparator<String> CASE_INSENSITIVE_ORDER
        = new CaseInsensitiveComparator();
private static class CaseInsensitiveComparator
    implements Comparator<String>, java.io.Serializable {
    // use serialVersionUID from JDK 1.2.2 for interoperability
    private static final long serialVersionUID = 8575799808933029326L;

    public int compare(String s1, String s2) {
        int n1 = s1.length();
        int n2 = s2.length();
        int min = Math.min(n1, n2);
        for (int i = 0; i < min; i++) {
            char c1 = s1.charAt(i);
```

```

        char c2 = s2.charAt(i);
        if (c1 != c2) {
            c1 = Character.toUpperCase(c1);
            c2 = Character.toUpperCase(c2);
            if (c1 != c2) {
                c1 = Character.toLowerCase(c1);
                c2 = Character.toLowerCase(c2);
                if (c1 != c2) {
                    // No overflow because of numeric promotion
                    return c1 - c2;
                }
            }
        }
    }
    return n1 - n2;
}

/** Replaces the de-serialized object. */
private Object readResolve() { return CASE_INSENSITIVE_ORDER; }
}

```

如果对象本身支持比较，即实现了 Comparable 接口的对象，可以使用 Comparator 提供的下面这个方法

```

public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
{
    return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
}

```

其中 Comparators 这个类是专门为 Comparator 接口创建比较算法使用的默认类，是一个同包访问权限的类

```

/**
 * Package private supporting class for {@link Comparator}.
 */
class Comparators {
    private Comparators() {
        throw new AssertionError("no instances");
    }

    /**
     * Compares {@link Comparable} objects in natural order.
     *
     * @see Comparable
     */
    enum NaturalOrderComparator implements Comparator<Comparable<Object>> {
        INSTANCE;

        @Override
        public int compare(Comparable<Object> c1, Comparable<Object> c2) {
            return c1.compareTo(c2);
        }

        @Override
        public Comparator<Comparable<Object>> reversed() {
            return Comparator.reverseOrder();
        }
    }
}

```

```
    }
}

....  
省略部分代码  
....  
}
```

其实对于比较这种算法，更多的是由使用者自己来决定谁大谁小，JDK 仅提供了一些基本的比较策略。比如如下几种比较策略

```
// 常规比较
static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {
    return NaturalOrderComparator.INSTANCE;
}

// 空值小于非空
static <T> Comparator<T> nullsFirst(Comparator<? super T> var0) {
    return new NullComparator(true, var0);
}

// 空值大于非空
static <T> Comparator<T> nullsLast(Comparator<? super T> var0) {
    return new NullComparator(false, var0);
}

// 等等，如果还想了解可以自行查看 java/util/Comparator.java
```

**注意：两个空值比较只是提供了当两个元素为空时的比较策略，当两个比较元素都不为空时需使用调用者提供的比较算法**

下面我们一起看下如何使用 JDK 的比较器来达到策略模式定义的第二点 它们可以相互替换，且算法的改变不会影响使用算法的客户。

## '你' 大还是 '好' 大

我们就拿 '你' 和 '好' 这两个汉字来尝试一下。

```
public class TestJDKComparator {

    public static void main(String[] args) {
        String you = "你";
        String fine = "好";
        // String 的比较器
        int ctic = you.compareToIgnoreCase(fine);
        System.out.println("compareToIgnoreCase:" + ctic);

        // JDK 提供的默认比较器
        Comparator<String> comparator = Comparator.naturalOrder();
        int compare = comparator.compare(you, fine);
        System.out.println("naturalOrder = " + compare);

        // 自定义比较器1
        Comparator<String> stringComparator1 = Comparator.nullsFirst((o1, o2) ->
    {
        byte[] bytes1 = o1.getBytes();
        byte[] bytes2 = o2.getBytes();
        return bytes1.length - bytes2.length;
    });
}
```

```

        int compare1 = stringComparator1.compare(you, fine);
        System.out.println("nullsFirst&customComparator1 = " + compare1);

        // 自定义比较器2
        Comparator<String> stringComparator2 = Comparator.nullsFirst(o1, o2) ->
    {
        int length1 = o1.length();
        int length2 = o2.length();
        int min = Math.min(length1, length2);
        for (int i = 0; i < min; i++) {
            char o1Char = o1.charAt(i);
            char o2Char = o2.charAt(i);
            if (o1Char != o2Char) {
                return o2Char - o1Char;
            }
        }
        return length2 - length1;
    );
    int compare2 = stringComparator2.compare(you, fine);
    System.out.println("nullsFirst&customComparator2 = " + compare2);
}
}

```

## 测试结果

```

compareToIgnoreCase:-2589
naturalOrder = -2589
nullsFirst&customComparator1 = 0
nullsFirst&customComparator2 = 2589

```

在 JDK 的比较器中有一个方法，可以让我们学习。就是 naturalOrder() 返回来的这个比较器。

```

// 常规比较
static <T extends Comparable<? super T>> Comparator<T> naturalOrder() {
    return NaturalOrderComparator.INSTANCE;
}

enum NaturalOrderComparator implements Comparator<Comparable<Object>> {
    INSTANCE;

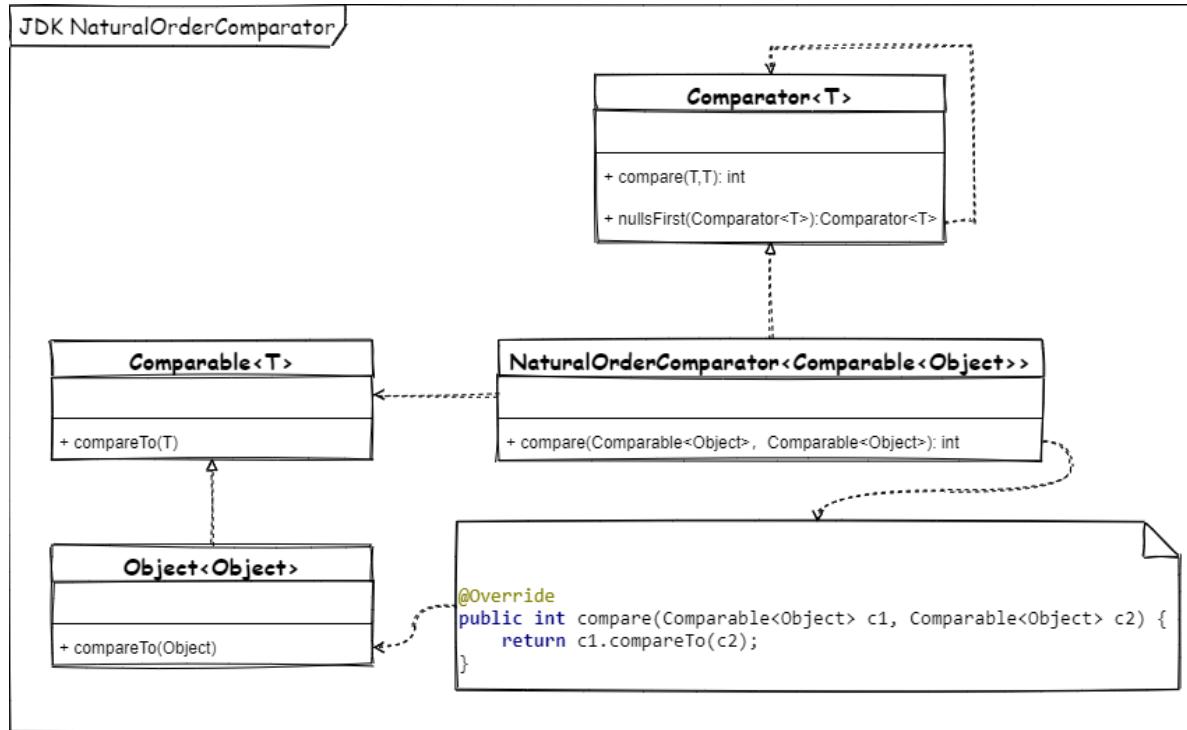
    // 实现比较器定义的抽象方法
    @Override
    public int compare(Comparable<Object> c1, Comparable<Object> c2) {
        // 使用参数自己的“策略”
        return c1.compareTo(c2);
    }

    @Override
    public Comparator<Comparable<Object>> reversed() {
        return Comparator.reverseOrder();
    }
}

```

这个比较器里有一个参数限制，而这个参数限制的就是必须是 Comparable 的实现类，同时是这个实现类的子类。其实，这个参数就是一个策略模式的“策略接口”，传入的参数就是具体的策略。因为这个传入的参数必须要实现 compareTo 这个方法，也就是实现 Comparable 接口的抽象方法。

```
public interface Comparable<T> {  
    int compareTo(T var1);  
}
```



在 JDK 中更灵活的使用比较器就是使用匿名类的写法

Choose Implementation of <b>Comparator</b> (101 found)	
AVAComparator (sun.security.x509)	Maven: org.junit.jupiter:junit-jupiter-params:5.8.1
Anonymous in AbstractInputValueSwitch (org.junit.jupiter.params.shadow.com.univocity.parsers.common.processor.core)	Maven: org.junit.jupiter:junit-jupiter-params:5.8.1
Anonymous in AnnotationHelper (org.junit.jupiter.params.shadow.com.univocity.parsers.annotations.helpers)	Maven: org.junit.jupiter:junit-jupiter-params:5.8.1
Anonymous in AreaOp (sun.awt.geom)	
Anonymous in AssertionSet (com.sun.xml.internal.ws.policy)	
Anonymous in Block (jdk.nashorn.internal.ir)	
Anonymous in CharsetMapping (sun.nio.cs)	
Anonymous in ClassHistogramElement (jdk.nashorn.internal.ir.debug)	
Anonymous in CleanupThread (com.sun.deploy.cache)	
Anonymous in Comparison in PolicyUtils (com.sun.xml.internal.ws.policy.privateutil)	
Anonymous in Compiler (jdk.nashorn.internal.codegen)	
Anonymous in DCmd (oracle.jrockit.jfr)	
Anonymous in DCmdCheck (oracle.jrockit.jfr)	
Anonymous in DateTimeFormatterBuilder (java.time.format)	
Anonymous in DateTimeTextProvider (java.time.format)	
Anonymous in FileFilterUtil (ch.qos.logback.core.rolling.helper)	Maven: ch.qos.logback:logback-classic:1.4.10
Anonymous in GraphicsPrimitiveMgr (sun.java2d.loops)	
Anonymous in INSURLOperationImpl (com.sun.corba.se.impl.resolver)	
Anonymous in JAXBContextImpl (com.sun.xml.internal.bind.v2.runtime)	
Anonymous in JREInfo (com.sun.deploy.config)	
Anonymous in JavaClass (sun.plugin.com)	
Anonymous in KeyTab (sun.security.krb5.internal.ktab)	
Anonymous in LinkerCallSite (jdk.nashorn.internal.runtime.linker)	
Anonymous in Lookup (org.junit.jupiter.params.shadow.com.univocity.parsers.fixed)	Maven: org.junit.jupiter:junit-jupiter-params:5.8.1
Anonymous in NativeArray (jdk.nashorn.internal.objects)	
Anonymous in NimbusStyle (javax.swing.plaf.nimbus)	
Anonymous in NumericShaper (java.awt.font)	
Anonymous in ObjectStreamClass (java.io)	
Anonymous in Observablelist (javafx.collections)	
Anonymous in OutputValueSwitch (org.junit.jupiter.params.shadow.com.univocity.parsers.common.processor)	Maven: org.junit.jupiter:junit-jupiter-params:5.8.1
Anonymous in Package (com.sun.java.util.jar.pack)	
Anonymous in PackageReader (com.sun.java.util.jar.pack)	
Anonymous in PackageWriter (com.sun.java.util.jar.pack)	
Anonymous in Packager (sun.plugin.com)	
Anonymous in Recording (oracle.jrockit.jfr)	
Anonymous in ShellFolder (sun.awt.shell)	
Anonymous in SnmpCachedData (sun.management.snmp.util)	
Anonymous in TimeBasedArchiveRemover (ch.qos.logback.core.rolling.helper)	Maven: ch.qos.logback:logback-classic:1.4.10
Anonymous in Trace (com.sun.deploy.trace)	

## 总结

其实抛开这个模式本身，我们在一些逻辑实现的时候也会使用这种写法，最简单的就是对一个接口方法的实现。使得他们可以在不同的情况下进行不同的切换。所以，在我们系统中，如果可能出现一些相同的操作，但是却会有很多不同的实现的时候，就是在使用这种“策略模式”来实现。每个具体的实现算法不同，但是他们的操作是相同。使用开闭原则来控制算法的入口，具体的实现延迟到子类。但当我们的具体算法变多的时候，使用起来可能会有一些副作用，所以这个时候可以考虑使用工厂模式来辅助策略模式变得更好用。

## 命令模式



将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。

## 前言

今天一大早就来了图书馆，刚坐下来就迫不及待的开始看命令模式的相关资料。不过这个模式跟我的之前的理解出入特别大。

最开始的时候，我以为的命令模式就是函数回调。但后来发现并不是，但他们两个确实是有关系，这一切的答案都藏在 GOF 的设计模式一书中。

## 开始学习

在软件设计模式之始 GOF 的原著中，命令模式的讲解还是在他们开发的那个编辑工具中，其用来讲解的案例就是我们日常编辑使用的编辑工具中，在工具栏有很多个功能按钮，或者菜单按钮。就比如编辑工具中的一个 新增文件 的按钮 NEW 吧。GOF 要表达的意思就是，这个 新增文件 对系统本身来讲就是给使用者提供的一个命令，我们在用的过程中可以给编辑器发送不同的命令，但是这个 新增文件 的操作并不是在这个按钮上实现的，同时对于我们发送命令的人来说，也不知道具体这个 新增文件 这个动作是由谁来执行、怎么执行，这对我们来讲完全是透明的。

我们先不讨论这样做的好处，先看下这里面要说的几个角色

1. 客户端应用
2. 新增文件 按钮（调用新增文件操作命令）
3. 操作命令
4. 操作接收（负责具体的操作执行）

我试着按照这个结构写了一下这个代码

```

public class Client {
    public static void main(String[] args) {
        FileReceiver fileReceiver = new FileReceiver();
        AddFileCommand addFileCommand = new AddFileCommand(fileReceiver);
        Invoker invoker = new Invoker();
        invoker.setCommand(addFileCommand);
        invoker.executeCommand();
    }
}

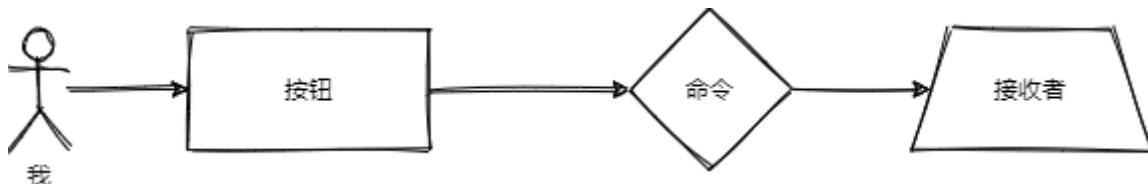
```

1. 客户端应用 `Client`
2. 新增文件 按钮（调用新增文件操作） `Invoker`
3. 操作命令 `AddFileCommand`
4. 操作接收者 `FileReceiver`

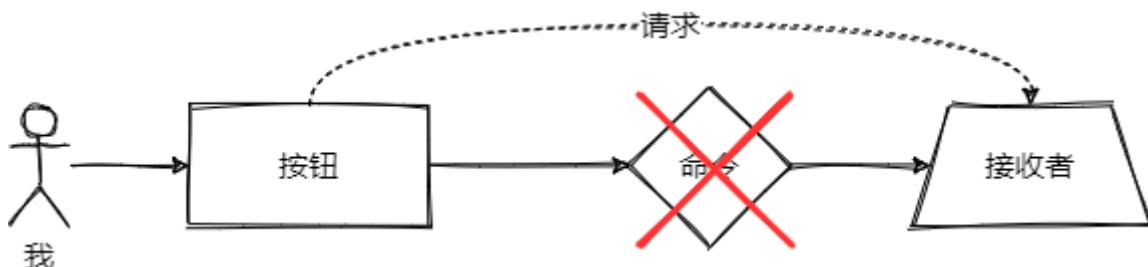
新增文件

## 关于‘命令’的疑惑 🤔

按照上面的方式实现下来，我有一种感觉，有种脱裤子放屁的感觉，我直接调用 `FileReceiver` 不香吗  
非要这样



我以为，使用者利用按钮直接调用对应的操作不就行了吗？就像我下面这样中间非要放一个命令对象（将具体的请求包装成了这个对象）？



## 解惑‘命令’ 🤔

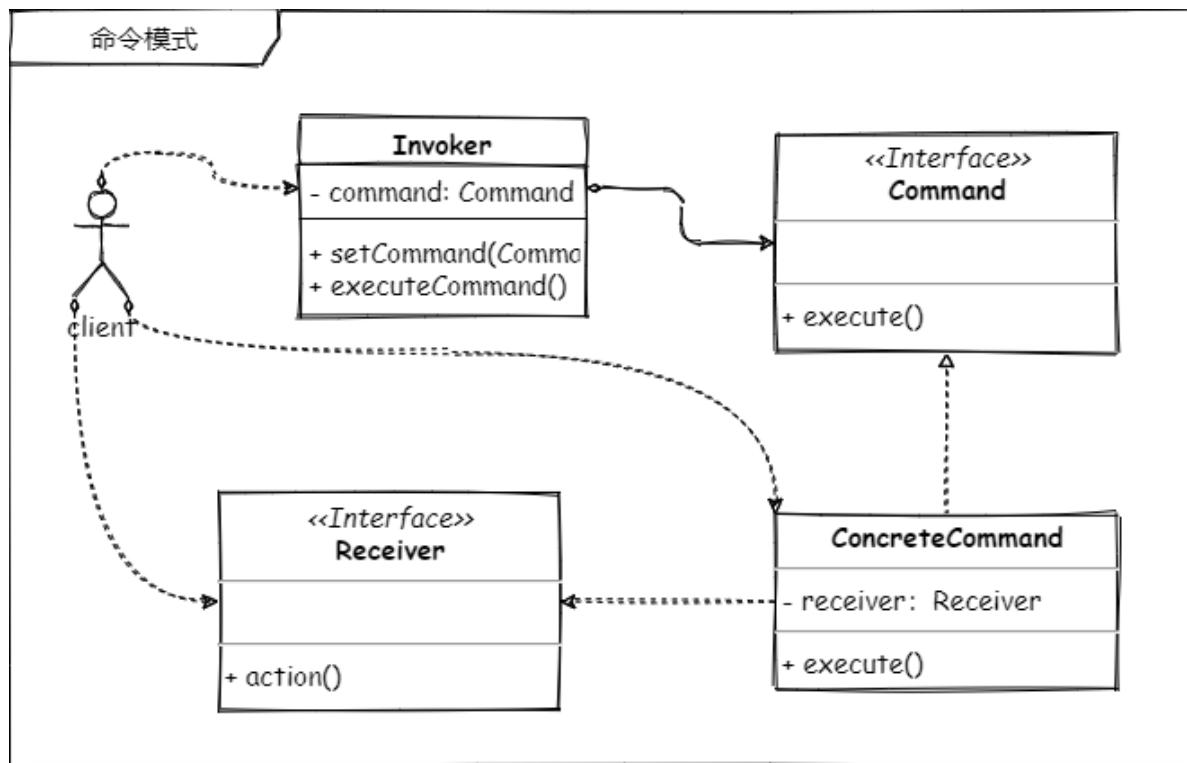
不过不久我就找到了答案💡

首先看下命令模式要解决的问题❓：对请求排队、下载或记录请求日志，以及支持可撤销的操作。

然后我们开始思考🤔如果没有中间这个“命令”角色，那这些功能做在哪里？只能做在接收者，也就是逻辑具体的实现里面，那这是不是违背了一个设计原则，叫做 `单一职责原则`？而且对这种“辅助型”的功能变多会导致逻辑实现类变得越来越“肿胀”，没错，就是“肿胀”！

并且这也使得调用者和实现者之间通过这个“命令”进行解耦，然后我们使用依赖倒置原则，将“命令”提取出来一个抽象类，这使得扩展请求也变得容易了。而且对于高层模块来说，自己完全不需要关心调用的时候具体的请求内容和实现内容，通过“命令”来完成自己的操作，比如点一个按钮、遥控器下的按键（从这里还可以看出，多个命令可以对应一个接受者，比如数字键的换台）、去餐厅点菜。这样一看，命令模式还真是符合这种设计思路的命名啊。

## 命令模式类图



### 主要结构

1. 调用者，也是暴露给客户端的对象 **Invoker**
2. 命令接口，**Command** (满足依赖倒置原则，便于扩展)
3. 具体的命令，这里要包含谁来接受这个命令的接受者对象 **ConcreteCommand**
4. 命令的接收者，这里没有列实现类是因为任何类都可以是接收者 **Receiver**

## 代码

命令模式这篇使用的是通用框架写了一个实现，在这基础上事实上我们可以做很多扩展，比如再 **Invoker** 类中将 `command` 换成 `List<Command>` 来实现请求的排队、撤销等操作。

The screenshot shows the file structure and code for the Command pattern implementation:

- File Structure:**
  - command**: A package containing **src**, **main**, and **java** folders.
  - java**: Contains a package **io.github.lvgocc** which contains a folder **command**. Inside **command**, there are several classes:
    - AddFileCommand**
    - Client**
    - Command** (marked with a green circle)
    - FileReceiver**
    - Invoker**
    - Receiver**
- Code Preview:**

```
13 */  
14 public class AddF  
15  
16     private final  
17  
18     public AddFil  
19         this.rece  
20     }  
21  
22     @Override  
23     public void e
```

## 总结

### 适用场景：

1. 需要记录请求记录；
2. 请求可以进行排队处理；
3. 请求可以进行撤销、重做；

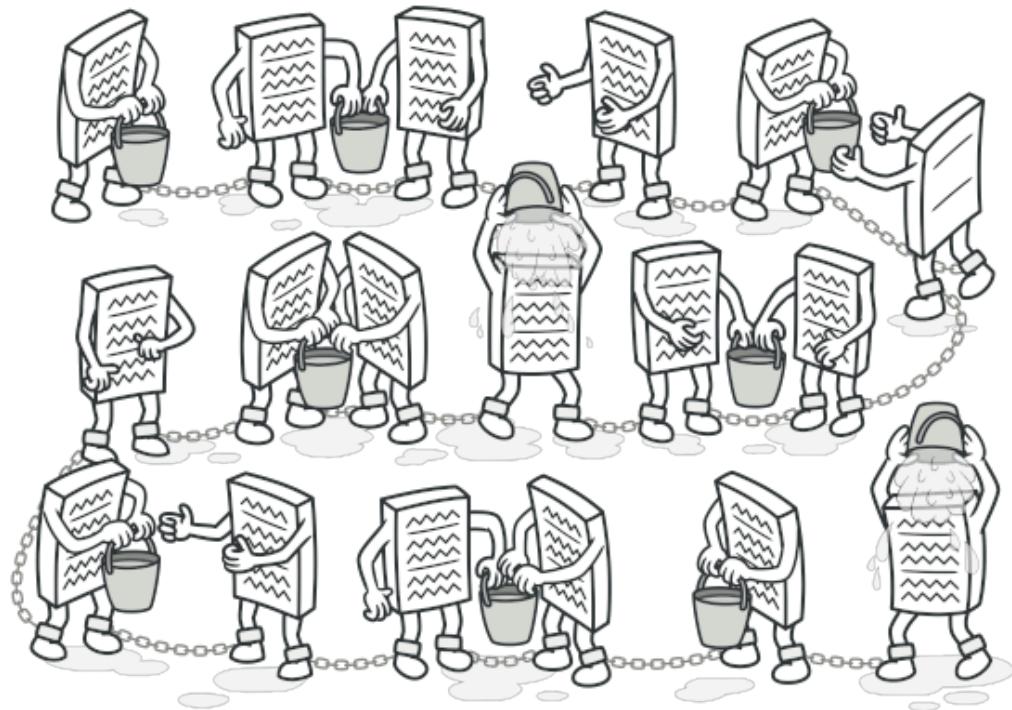
4. 具体接收者来决定请求是否执行（关于这一点，如果请求不是封装成一个对象的话，判断起来是比较困难的）

不过这种模式并不是一个常用的思想，一定是当你想要对请求做一些事情的时候才考虑，具体的事情就上面提到的 4 点，不然的话使用这种模式真的就是我上面说的，“脱裤子放屁了”。

最后再来一句话来总结一下命令模式，“张三，把门关一下”。这里我就是 Invoker，“把门关一下”就是 command（命令），“张三”是 receiver（接收者）。更多时候，我们实际开发中，“把门关一下”都是定义好的，“我”直接选就行了，就像遥控器上的按键一样。但切记这个模式的使用时机，别做“恶心”人的事！

如果哪里有问题或者有疑问，欢迎加我微信（lvgocc）讨论，或者直接进群交流！天凉了，进群一起取暖也好啊，等你~

## 责任链模式

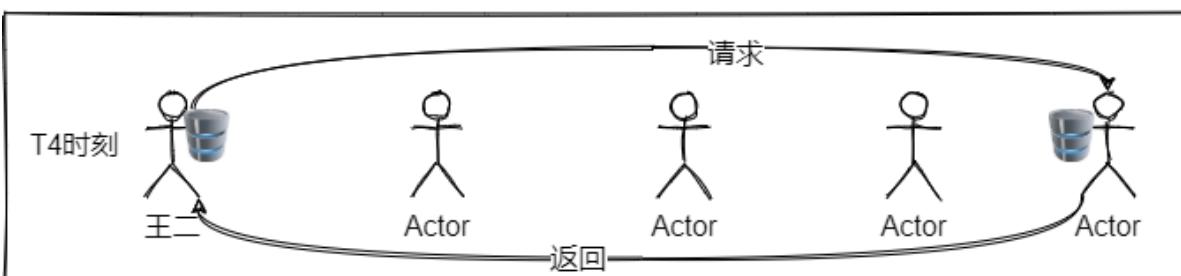
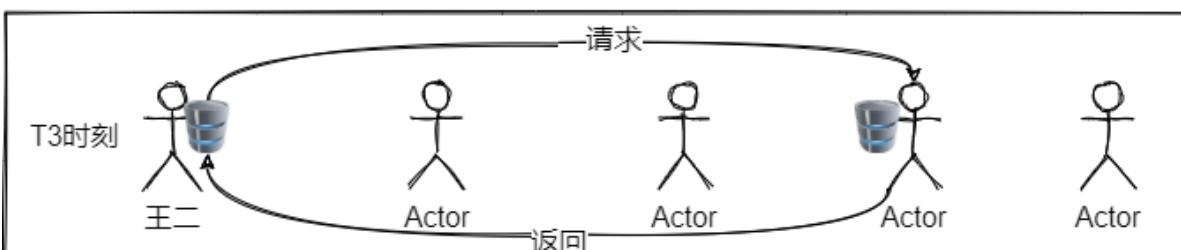
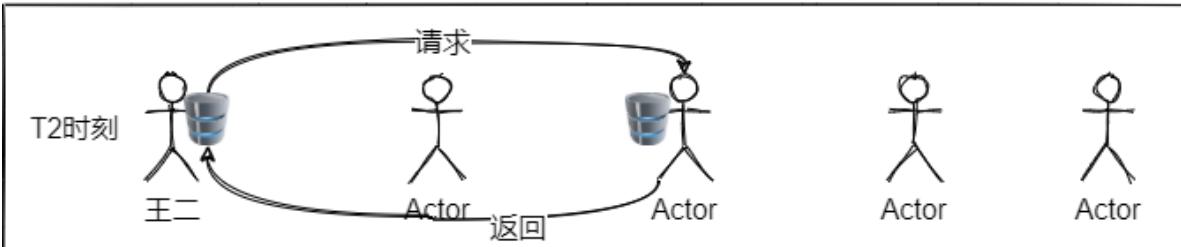
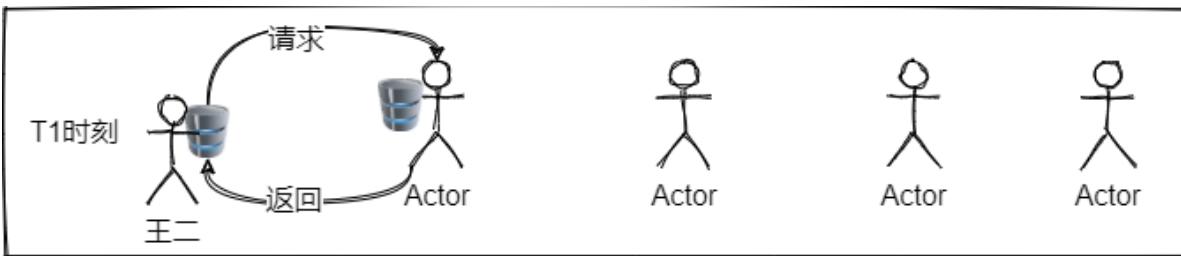


图片来源：<https://refactoringguru.cn/design-patterns/chain-of-responsibility>.

把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

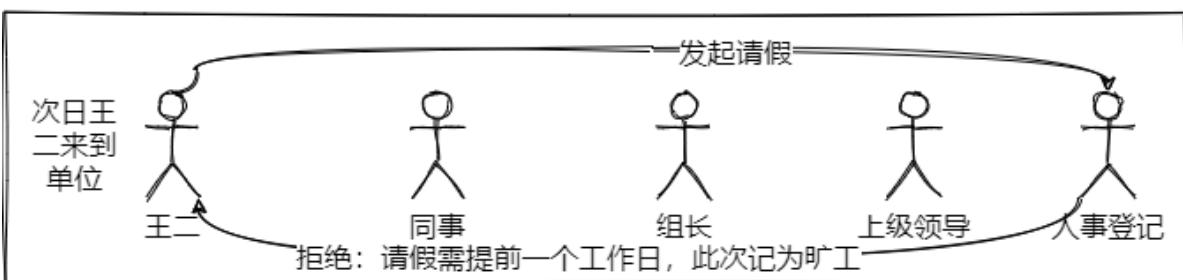
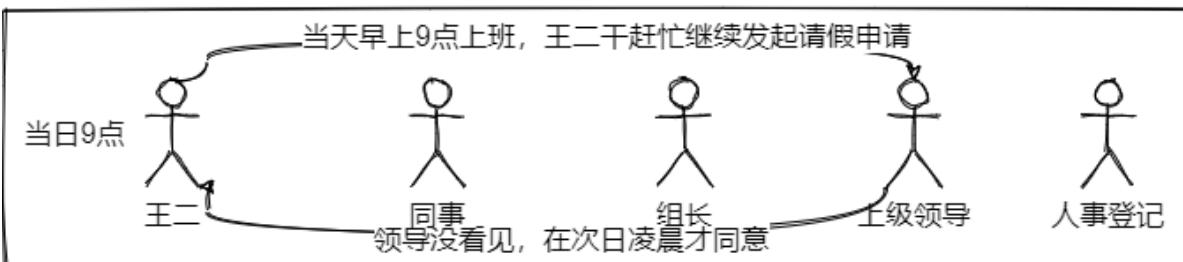
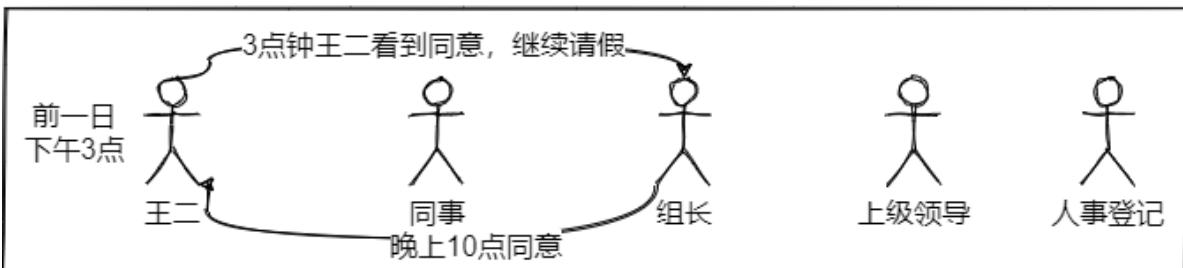
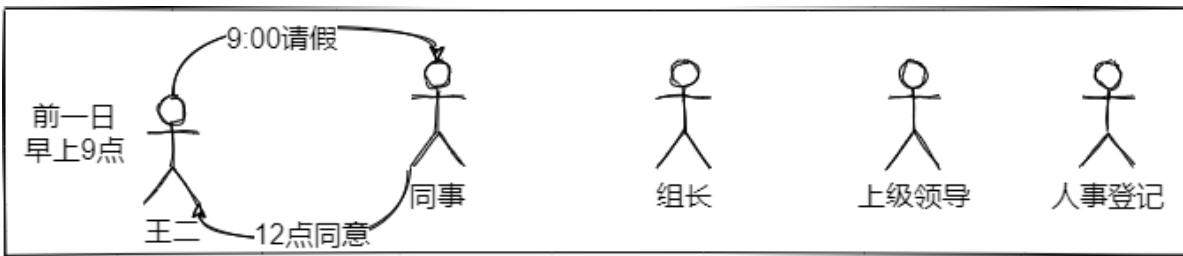
通过上面的图其实也能看个差不多了，在这幅图中水桶就是整个链路中被传递的对象。它可以在链路上的任意一个节点被消费，如果你觉得剩的水可以继续给下一个节点用，你甚至可以将它继续传递下去。这样设计的好处就行定义中说的那样，去除对象间的耦合。

假设这个业务场景需要请求的对象是存在联系的，比如具有一定顺序去消费同一个对象，又比如他们消费对象的方法相同，具体逻辑略有差异。此时如果这个水桶对象的传递不通过责任链这种模式，看看会有什么问题。



王二需要分为4个时刻与4个不同的对象进行交互，这无疑增加了系统的复杂性。并且这里其中任意一个请求目标发生变化，王二都必须要跟着调整。再比如下面这个生活中的例子。

王二因为一些原因不能上班，需要和领导请假，卑微的王二在单位的职位级别比较低，需要多级领导审批，甚至同事都是一个坎，让我们看看没有责任链模式介入时王二的请假过程。

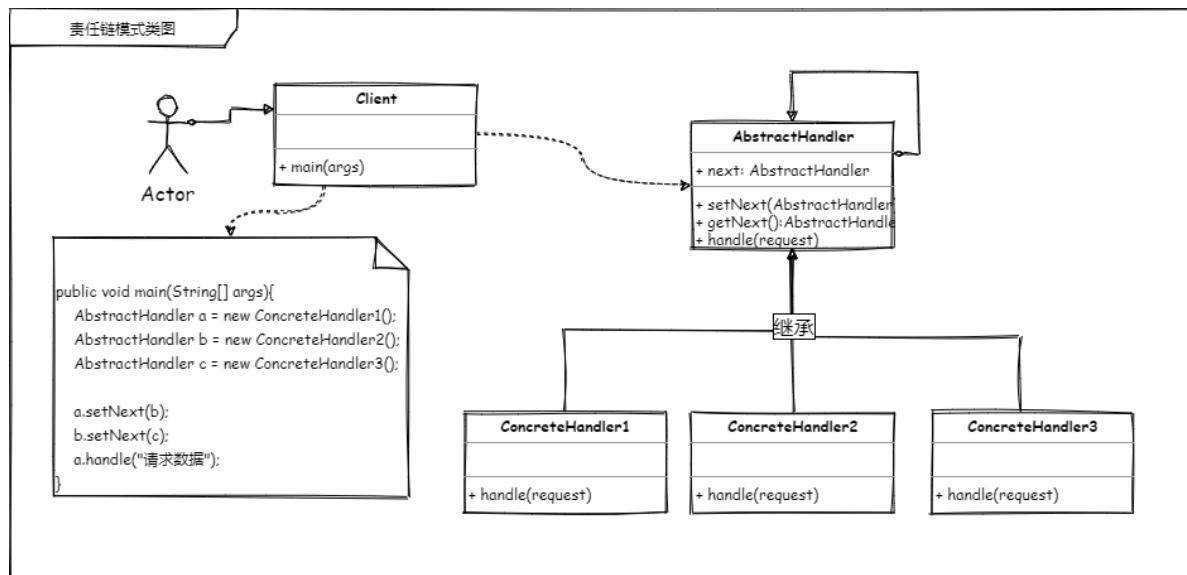


**王二卒。**

王二这件事被同事张三知道后，张三决定为了纪念王二的悲催经历。他决定向领导提出一个流程调整方案，具体的意见如下；

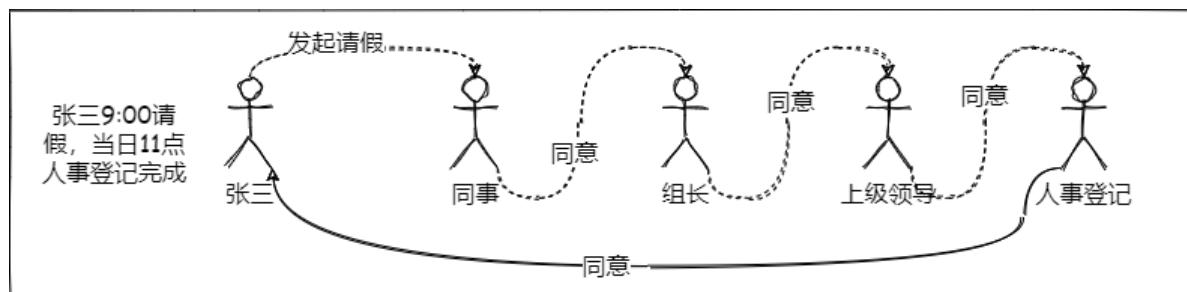
1. 请假时每个人的动作相同，均为审批。至于审批是同意或拒绝由审批者自己决定。
2. 对请假流程中涉及的人员使用链式传递。不得跨越（即每个人必须都需要经过处理后才能继续传递）。

## 责任链模式类图



## 张三请假

张三使用责任链模式请假流程示意图



抽象处理类，各个环节处理时统一标准。

```

public abstract class AbstractHandler {

    protected AbstractHandler next;

    public AbstractHandler getNext() {
        return next;
    }

    public void setNext(AbstractHandler next) {
        this.next = next;
    }

    protected void handle(String request) {
        conCreteHandle(request);
        if (getNext() == null) {
            System.out.println("流程结束");
        } else {
            getNext().handle(request);
        }
    }

    protected abstract void conCreteHandle(String request);
}
  
```

```

public class QingJia extends AbstractHandler{
    @Override
    protected void concreteHandle(String request) {
        System.out.println(request);
    }
}

class AbstractHandlerTest {
    @Test
    void handle() {
        AbstractHandler qingJia = new QingJia();
        AbstractHandler renShi = new RenShi();
        AbstractHandler shangjiLingdao = new ShangjiLingdao();
        AbstractHandler tongShi = new TongShi();
        AbstractHandler zuZhang = new ZuZhang();

        qingJia.setNext(tongShi);
        tongShi.setNext(zuZhang);
        zuZhang.setNext(shangjiLingdao);
        shangjiLingdao.setNext(renShi);

        qingJia.handle("张三请假");
    }
}

```

测试张三请假

```

张三请假
同事审批: 同意
组长审批: 同意
上级领导审批: 同意
人事审批: 同意
流程结束

```

完整代码文末关注，回复“源码”获取。

## 总结

使用责任链模式可以使原本的对象间耦合度降低。各个模块间功能更加具体专注。同时链上的处理也可以更加灵活，可以通过处理请求的时候进行判断来过滤自己关注的内容来处理，或者在生成链的时候将无关节点去掉。

同时可以配合创建型模式中的工厂模式，来封装链的维护，这样在链上节点发生变化时（算法实现发生改变、新增或删除）对于高层模块是没有感知的。扩展起来非常的方便。或使用建造者模式来更加灵活地创建这条“责任链”，以达到客户端的自定义目的。总之，责任链模式在处理链式问题是个利器。

## 状态模式

允许一个对象在其内部状态发生改变时改变其行为能力。

我刚开始看到这个模式的时候，没啥感觉，不知道这东西要说的是个啥，后来看了个案例，渐渐清楚了，这个模式本身还是比较简单的。

## 小菜的工作状态

这个案例出自程杰的《大话设计模式》，抽取案例模型，完整案例还请大家自行阅读

案例说的是主人公“小菜”上班写代码时的各种状态，上午的时候精神饱满，中午时有点萎靡，下午状态一般，晚上状态疲惫。然后这一天各个时段的写代码状态用程序表达出来是这个样子的；

```
public class Work {  
  
    static int clock;  
  
    public static void main(String[] args) {  
        // 上午 9 点  
        clock = 9;  
        writeCode();  
        // 中午 12 点  
        clock = 12;  
        writeCode();  
        // 下午 15 点  
        clock = 15;  
        writeCode();  
        // 晚上 21 点  
        clock = 21;  
        writeCode();  
    }  
  
    public static void writeCode() {  
        if (clock < 12) {  
            System.out.println("精神抖擞写代码");  
        } else if (clock < 13) {  
            System.out.println("饿了困了写代码");  
        } else if (clock < 17) {  
            System.out.println("状态一般写代码");  
        } else if (clock < 23) {  
            System.out.println("加班疲惫写代码");  
        }  
    }  
}
```

一段典型的面向过程编程代码，之后根据面向对象的思想来改了一版变成了这样。

```
public class OOPWork {  
  
    public static void main(String[] args) {  
        Working working = new Working();  
        // 上午 9 点  
        working.clock = 9;  
        working.writeCode();  
        // 中午 12 点  
        working.clock = 12;  
        working.writeCode();  
        // 下午 15 点  
        working.clock = 15;  
        working.writeCode();  
    }  
}
```

```

    // 晚上 21 点
    working.clock = 21;
    working.writeCode();
}

}

class Working {
    int clock;

    public void writeCode() {
        if (clock < 12) {
            new MorningState();
        } else if (clock < 13) {
            new NoonState();
        } else if (clock < 17) {
            new AfterNoonState();
        } else if (clock < 23) {
            new EveningState();
        }
    }
}

class MorningState {

}

class NoonState {

}

class AfterNoonState {

}

class EveningState {

}

```

其实写到这里我相信大家就算没看过这本书也能多少发现一点端倪，就是这个 Working 类是不是有点奇怪。每新增一个状态就要去改这个类，而且这里负责了全部的工作状态，还有最关键的就是这个 `if` `else` 是不是有点太长了？？？没错，其实这些都可以用 `状态模式` 来规避掉，并且这些在软件设计中也都违反了一些原则或建议。

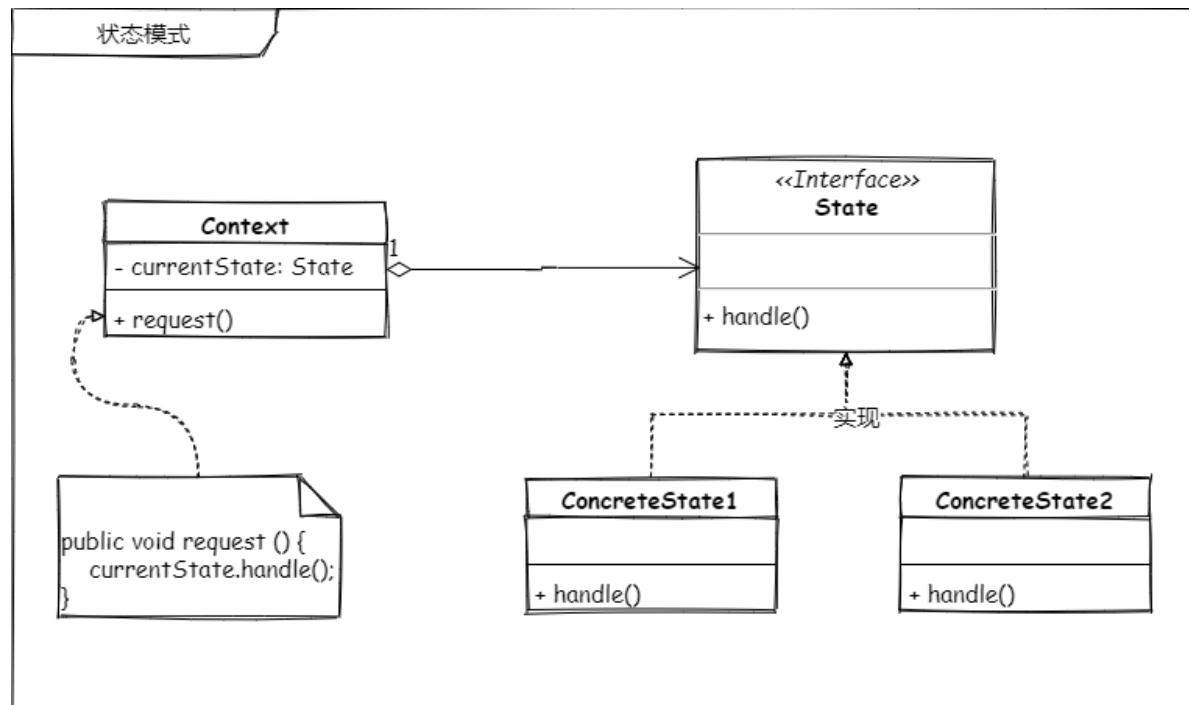
1. 每新增一个状态就要去改这个类（违反了 `开闭原则`）

2. 而且这里负责了全部的工作状态 (违反了 单一职责原则)
3. 还有最关键的就是这个 `if else` 是不是有点太长了？？？ (这是 重构 书中提到的 `long method` 的坏味道)

## 利用状态模式解决这 3 个问题

首先来看下状态模式的结构类图

### 状态模式类图



这里有几个关键的角色

1. 运行的上下文环境 `Context` 对应到程序中就是 `Working` 类
2. 状态接口 `State` 这是为了解决单一职责和开闭原则；
3. 具体的状态，也就是 `State` 接口的实现

### 状态模式代码

将上面的写法改成用状态模式的话就长这个样子

```

@Test
void writeCode() {
    Working working = new Working(new MorningState());
    // 手动模拟不同时刻
    working.setClock(9);
    working.writeCode();

    working.setClock(12);
    working.writeCode();

    working.setClock(15);
    working.writeCode();

    working.setClock(21);
    working.writeCode();
}
  
```

```
    working.setClock(24);
    working.writeCode();
}
```

```
public class Working {
    /**
     * 当前工作状态
     */
    private final WorkState concurrentState;

    /**
     * 当前时刻
     */
    private int clock;

    public Working(WorkState concurrentState) {
        this.concurrentState = concurrentState;
    }

    public void writeCode() {
        concurrentState.handle(this);
    }

    ...
}

// 篇幅原因，完整代码关注回复“源码”获取。
}
```

## 测试结果

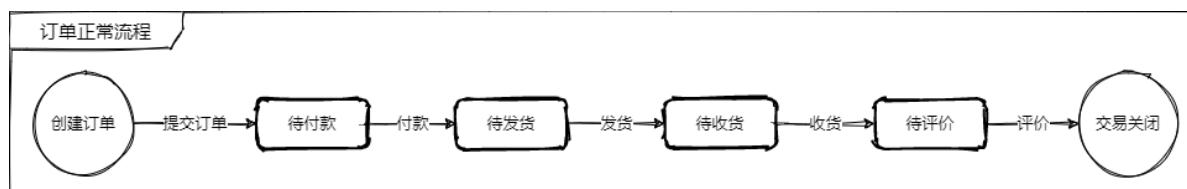
```
精神抖擞写代码
饿了困了写代码
状态一般写代码
加班疲惫写代码
别再写了，程序员回家了，明早再试吧㉚，现在都已经24点了。放过他吧
```

篇幅原因，完整代码关注回复“源码”获取。

这样不仅消除了 `if else` 的臃肿 `long method` 坏味道代码，同时 `Working` 类更专注 “写代码”，同时有 `State` 接口的出现，实现了开闭原则，让程序的扩展到了保障，并且一个关键的内容就是“在不同时刻，调用 `writeCode` 方法的结果是不同的。这也是状态模式的定义中提到的 一个对象在其内部状态发生改变时改变其行为能力。 改变对象的一个状态，使他的行为也发生了变化，这看起来就像我们对这个类的代码进行了修改一样。

## 状态模式延伸

如果大家有做过交易系统的订单的话，有一个东西应该不会陌生，叫做 `有限状态机` 也叫做 `状态机`。

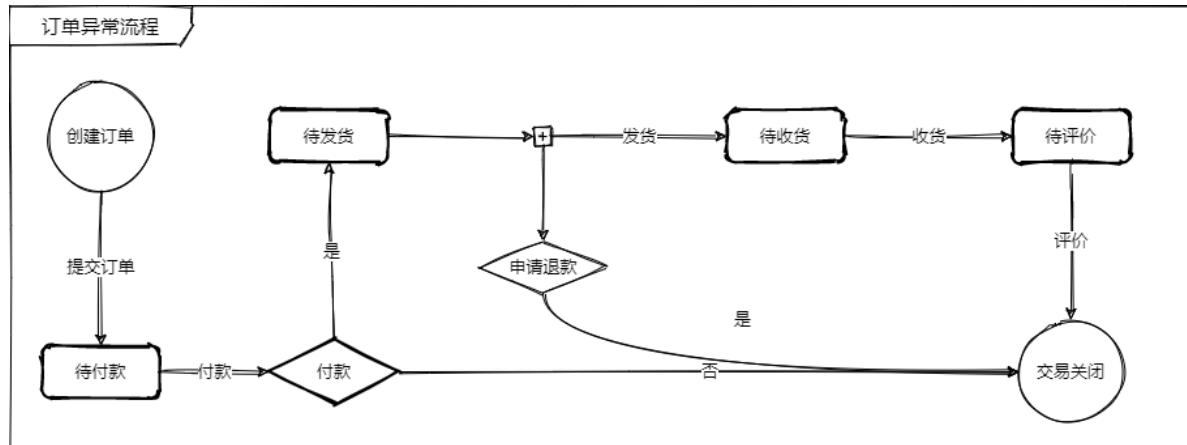


这是正常情况，也就是如果一个订单按照正常步骤来流转是会按照上图所示来进行。但是，真实的订单场景远比这个要复杂的多。

这里我们只看一些简单的几个场景：

1. 提交订单之后突然心思转变，不买了，那这个时候给客户就是 取消订单 和 付款 两种选择，这一步发生的时候，订单状态为 待付款；
2. 付款之后又看了看发现有更好的店铺选择，此时钱已经付了，所以此时的操作提供给客户的就是 申请退款，此时订单状态为 待发货；
3. 等等订单场景较为复杂，不过万变不离其宗，即 不同的状态，用户的操作行为和这笔订单之后的行为是有限的；

用一个图来看下上述几个问题场景



我们可以通过图可以看到，即使只有这简单的两种情况，整个订单的处理逻辑就已经开始变得复杂了起来，如果利用传统的面向过程编程或简单的面向对象编程思路来设计这个订单流程逻辑，我想，第一版，也就是正常的订单流程开发实现起来问题应该“不大”，无非代码啰嗦一点、判断逻辑多一点。

不过一旦产品经理提出在付款时可以进行撤单操作、在发货时可以进行退款、发货前可以修改接收地址又或者收货时可以拒收等等这些需求时，我觉得这个业务应该不会有人愿意去开发了。

不是开发人员不愿意做这个业务，而是不愿意在糟糕的代码上进行再次迭代（当然在现实情况，一个糟糕的团队确有可能继续在糟糕的代码上继续迭代，原因很简单，因为他们从一开始便能设计出来，领导能通过就说明这是一个没有技术沉淀的团队，他们很愿意将一坨坨代码“至死不渝”的一直维护下去）。

如果订单状态利用 状态模式 来设计，无论状态如何变化，高层模块也永远不需要关心，这也是开闭与单一职责这两个原则的很好体现。在哪个状态能做哪些事情，完全有对应的状态说了算，即使在复杂的业务，也会因为状态的区分而使业务颗粒变得很小（如果传统的 if else 到底，整个业务流程必须全部重新测试一遍，这不是谁说的，这是由高耦合紧密设计决定的，逃不开），这一点在开发和测试上，都会大幅提高开发和测试效率和节省成本。

好了，关于这一块内容确实大家可以看看 有限状态机，订单业务在实际情况是交由 状态机 来管理的。

## 总结

当你的应用程序可能会存在多种状态，而且每种状态的行为会随着状态的改变而改变，这时你可以考虑使用状态模式。通过状态模式不光可以使应用程序可以和应用状态可以很好的解耦，同时在状态的管理和扩展上都是非常的有帮助。

状态模式关键的几个点：

1. 应用环境的上下文，这个是用来作为状态模式的入口，他负责来调用当前状态的执行方法。
2. 状态接口，这个接口负责管理全部的状态，这里在使用的时候需要好好设计，可以将整个业务所有的状态方法全部维护好，在具体接口类中间放一个抽象状态类，如果当前状态不能操作这个方法可以放在一个抽象类中来实现一个空方法，而不是子类去实现全部的接口方法。

设计模式是一把双刃剑，在合理的时机使用可以达到很好的应用效果，反之亦然。大家要谨慎对待。不要变成手里只有一把锤子的人。

# 观察者模式



多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

提到观察者，就一定有“被观察者”。

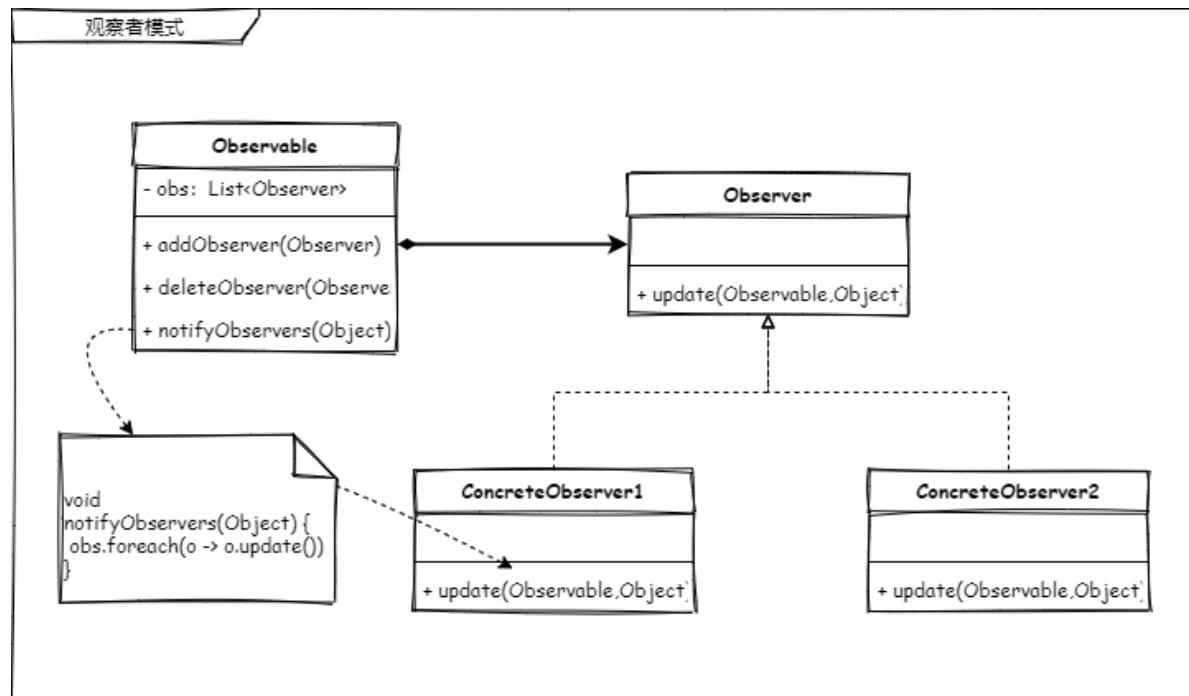
被观察者发生改变时，通知给每个观察者，这就是观察者模式。放到生活中的例子就是

- 天气预报的短信通知，在很早以前，我们的天气预报服务可以通过短信的信息订阅。在这个业务场景中，天气预报就是被观察者，每个付费的用户就是观察者，当有最新的天气预报的消息时，会以短信的形式通知给曾经付过费订阅的用户。如果不订阅是不会收到的。
- 你现在在使用的通讯软件中的‘群聊’功能，这个相对较为复杂一点，可以抽象的理解一下，用一个人去理解，群里的每个人都是被观察者，每个人都是观察者。当一个人发出消息的时候，所有在群里的人们都会收到消息也包括自己。反之，不在群里的人则收不到。
- 还记得小区单元门口一个一个小盒子吗？有的小盒子里是有当天的报纸的。甚至里面的报纸内容可能是不相同的，这里就涉及到了多个被观察者和观察者之间的关系，这个小盒子能收到自己订阅的报社报纸，当然你可以同时订阅多个，这样你就可以收到多个报社送过来的报纸了。

对于上面的例子，天气预报需要用户自己付费订阅、群聊需要先进群、在家看报需要联系报社订阅报纸。这里就能看出来。这种关系是一个一对多的关系。被观察者是同一个，而观察者却可以是很多个不同的对象。还有就是观察者需要自己主动的去找被观察者“提前”说明好，“一旦有消息，请通知我一声”。所有这里可以抽象出来几个角色和动作。

1. 被观察者（1个）
2. 观察者（n个）
3. 被观察者负责管理观察者对象
4. 观察者自己负责被观察者给予的通知内容

## 观察者模式类图



1. 被观察者 (Observable)
2. 观察者 (Observer)
3. 被观察者负责管理观察者对象(Observable.obs)
4. 观察者自己负责被观察者给予的通知内容(Observer.update)

如果对 JDK 熟悉的同学可能早已看穿，这个类图画的其实就是 JDK 提供的观察者框架，我们可以用它轻松的实现一个订阅通知功能。而这一功能在 JDK 1.0 的版本就已经存在了。

## 代码

JDK 源码，篇幅原因只保留了核心代码

```

package java.util;
// 观察者
public interface Observer {
    void update(Observable var1, Object var2);
}
  
```

JDK 源码，篇幅原因只保留了核心代码

```

package java.util;
// 被观察者
public class Observable {
    // 管理观察者对象
    private final Vector<Observer> obs = new Vector();

    public synchronized void addObserver(Observer var1) {
        if (!this.obs.contains(var1)) {
            this.obs.addElement(var1);
        }
    }
    // 通知给订阅的观察者
    public void notifyObservers(Object var1) {
        Object[] var2 = this.obs.toArray();
        for(int var3 = var2.length - 1; var3 >= 0; --var3) {
            ((Observer)var2[var3]).update(this, var1);
        }
    }
}
  
```

```
    }

}

}
```

自己实现部分

```
public class Producer extends Observable {

    @Override
    public synchronized void setChanged() {
        super.setChanged();
    }
}
```

```
public class Consumer1 implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println("我是 consumer1 我收到了" + o + "的通知, 通知内容: " +
                           arg);
    }
}
```

```
public class Consumer2 implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println("我是 consumer2 我收到了" + o + "的通知, 通知内容: " +
                           arg);
    }
}
```

```
public class Consumer3 implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        System.out.println("我是 consumer3 我收到了" + o + "的通知, 通知内容: " +
                           arg);
    }
}
```

测试，定义了 2 个被观察者（生产者），3 个观察者（消费者）来分别使用12生产者来发布消息。

```
@Test
void jdkob() {
    Producer producer1 = new Producer();
    producer1.setChanged();
    Producer producer2 = new Producer();
    producer2.setChanged();
    Consumer1 consumer1 = new Consumer1();
    Consumer2 consumer2 = new Consumer2();
    Consumer3 consumer3 = new Consumer3();

    producer1.addObserver(consumer1);
    producer1.addObserver(consumer2);
    producer1.addObserver(consumer3);
```

```
producer2.addObserver(consumer1);
producer2.addObserver(consumer2);
producer2.addObserver(consumer3);

producer1.notifyObservers("我是生产者1，我现在给你们通知一条消息，收到赶紧去消费掉");
producer2.notifyObservers("我是生产者2，我现在给你们通知一条消息，收到赶紧去消费掉");
}
```

## 测试结果

```
我是 consumer3 我收到了Producer@57cd70的通知，通知内容：我是生产者1，我现在给你们通知一条消息，收到赶紧去消费掉
我是 consumer2 我收到了Producer@57cd70的通知，通知内容：我是生产者1，我现在给你们通知一条消息，收到赶紧去消费掉
我是 consumer1 我收到了Producer@57cd70的通知，通知内容：我是生产者1，我现在给你们通知一条消息，收到赶紧去消费掉

我是 consumer3 我收到了Producer@1a7504c的通知，通知内容：我是生产者2，我现在给你们通知一条消息，收到赶紧去消费掉
我是 consumer2 我收到了Producer@1a7504c的通知，通知内容：我是生产者2，我现在给你们通知一条消息，收到赶紧去消费掉
我是 consumer1 我收到了Producer@1a7504c的通知，通知内容：我是生产者2，我现在给你们通知一条消息，收到赶紧去消费掉
```

## 总结

使用观察者模式需要注意的几个点

1. 观察者数量，如果一个被观察者被很多观察者观察（订阅）时，在通知时的时间将会变得漫长；
2. 不能出现被观察者和观察者之间存在循环观察情况，否则系统会直接崩溃；

观察者模式的代码虽然很简单，但是它所创造的价值却远不止这些。相信你同我一样，通过观察者模式联想到了消息通知、binlog订阅、注册中心等技术组件。其核心内容也只是在此简单的不能再简单的思想上去做更多更复杂的功能迭代。

万变不离其宗。在复杂的系统，在复杂的功能，都能找到其根本所在。知识，亦是如此。

当然，你也可以尝试在现有的代码中进行一些修改，比如通知的情况改为有新的观察者加入时？通知的数据变得更丰富一些？异步通知？等等等等。

如果你有更好的点子可以关注并分享给我们（欢迎加群）！

## 中介者模式

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

与其说中介者模式还不如说是软件设计原则的具体体现。这个原则就是——迪米特法则。

这里可以参考之前的系列文章《和 Ivg0 一起学习设计模式 - 序》中的软件设计基本原则 6 迪米特法则

- ⑥ 迪米特法则又叫作最少知识原则 LOD/LKP，1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被

UML 创始人之一布奇 (Booch) ~~就~~ 普及，后来又在经典著作《程序员修炼之道》中提及，从而传播开来。原则定义：只与你的直接朋友交谈，不跟“陌生人”说话 (Talk only to your immediate friends and not to strangers)。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。 其目的是降低类之间的耦合度，提高模块的相对独立性。

通过这个原则的核心内容我们知道，迪米特法则（中介者模式）要解决的问题就是提高软件程序的聚合度、降低对象之间的耦合。

## 要解决的问题

### 王二入职

前阵子王二刚刚毕业，入职了一家公司，报道的第一天，人事将王二带到部门后介绍了接头人就走了，王二坐在座位上等待人“接待”他，等了20分钟，没人管他，于是他起身去找了当时的部门接头人旺仔。

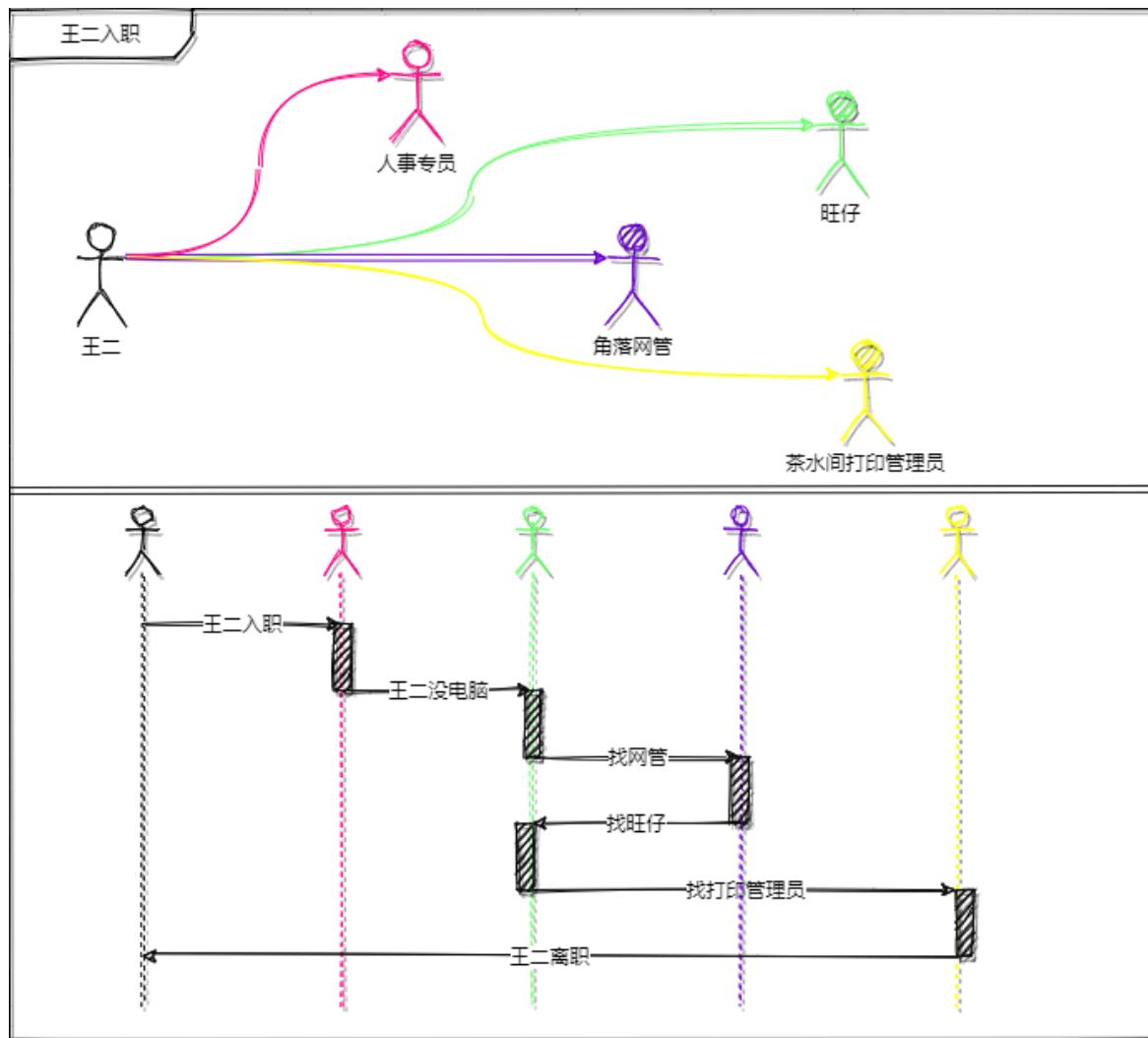
“仔哥，我~”，“啊，我知道，你新来的”。“我还没电脑”

“没电脑？你去找那个角落里，那是网管，问他们要一台”，“好的”

王二初来乍到，总觉得哪里不对，又不好意思说，就去角落里问：“您好，我是xxx部新来的，需要领台电脑”，“啊，去找你组长申请”。

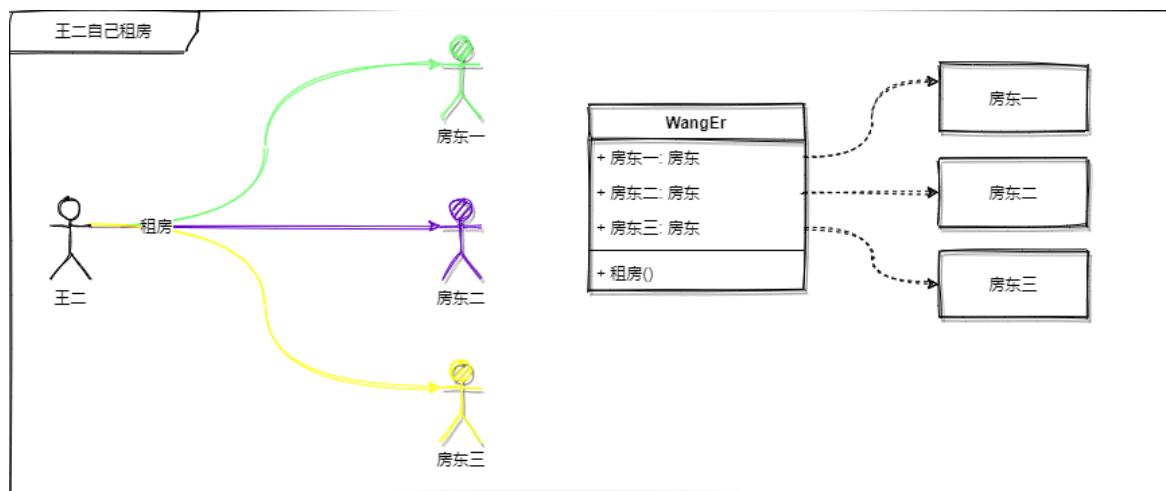
“仔哥，那面叫我来和组长申请。我组长是谁啊？”，“我啊，我就是你组长，跟我申请”。王二没说话。  
“把这个表格打出来，填一下，给我和部门经理签字”，“仔哥，去哪里打印？”，“茶水间边上的房间，找打印机管理员”。

王二蹑手捏脚的过去了，到了那里“您好，我想打份申请单。”“什么申请？”，“离职申请。”王二淡定的说。



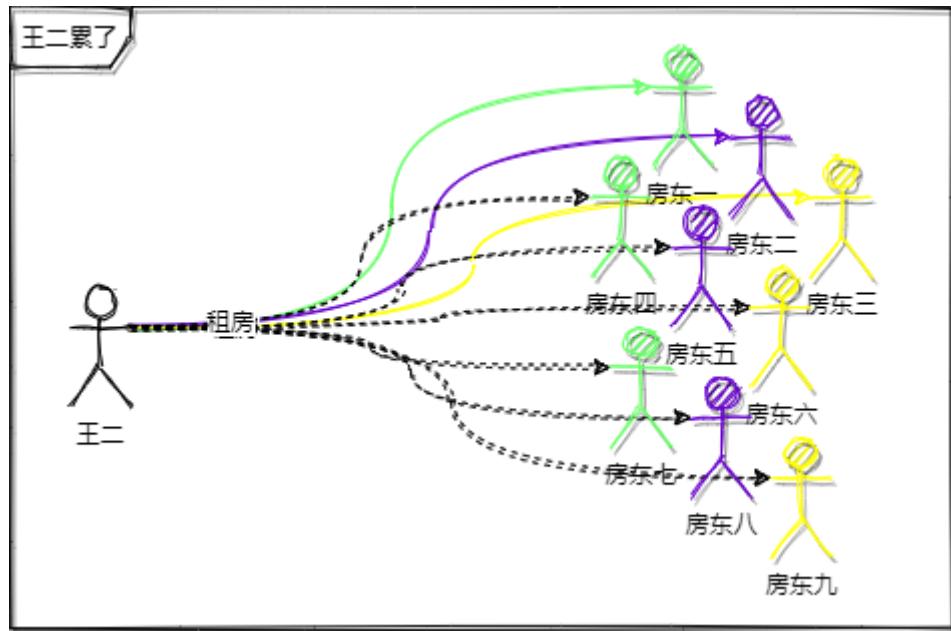
## 王二找房

王二最终找到了一个让他满意的地方，然后开始找地方住，所以他开始了繁忙的“找房之旅”，一开始，王二一口气就找到了三个房东



- 房东一 = 10平米, 无窗
- 房东二 = 20平米, 半个窗
- 房东三 = 1000平米, 108个窗

找了很多，王二也没找到自己心仪的，而此时的王二已经累了。



最后经历了第 10 个房东的时候，王二终于找到自己满意的房子了。

**通过这件事我们知道买房的重要性**

## 如何解决

### 王二入职

如果你是这家公司的人事，你会怎么来安排一个新同事的到来的各种事情呢，或者说怎么管理公司间同事的交叉问题呢？

如果是我，我会制定一个专员负责管理同事间的诉求，比如人事专员、行政专员等等，当然还可以设立其他专员来解决这种类似的问题。

有了专员之后，专员就负责在各种业务流程上起引导督促等作用。比如刚刚王二入职，人事专员需负责跟踪引导其完成入职知道可以开展工作之前的的相关事项。于此同时，王二只需要和人事专员进行通信，其中间的流程步骤变得清晰了很多。

王二：“你好，我这里没有电脑”

人事专员：“已经再给你申请了，稍等”

王二：“好的”

过了，20分钟

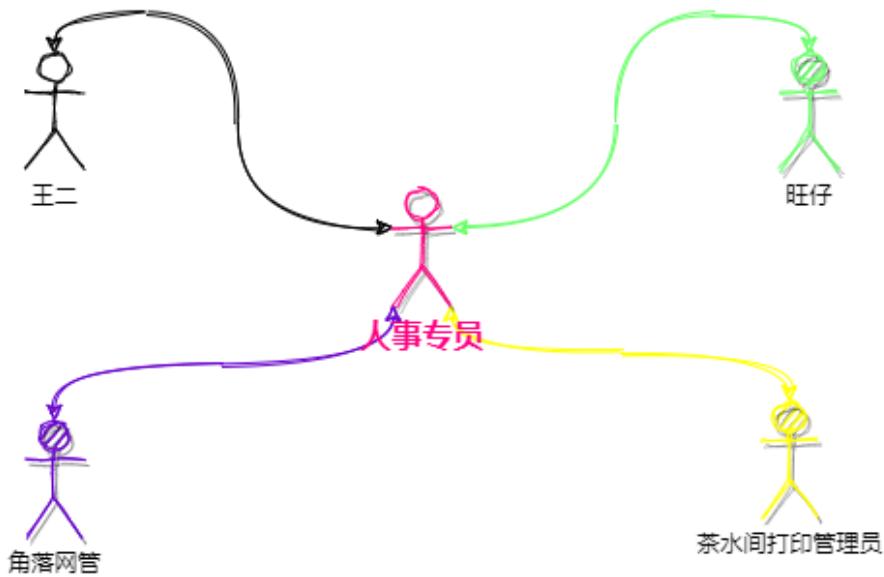
设备管理员：“电脑登记好了，给你吧”

人事专员：“好的，我给王二送过去，**你也不认识他**”；

人事专员：“王二，这是你的电脑，编号在背面，有什么问题再联系我”

通过专员的加入，解耦了各个同事之间的耦合，同时降低了各个同事间复杂的交互，也避免了不必要的“人才流失”

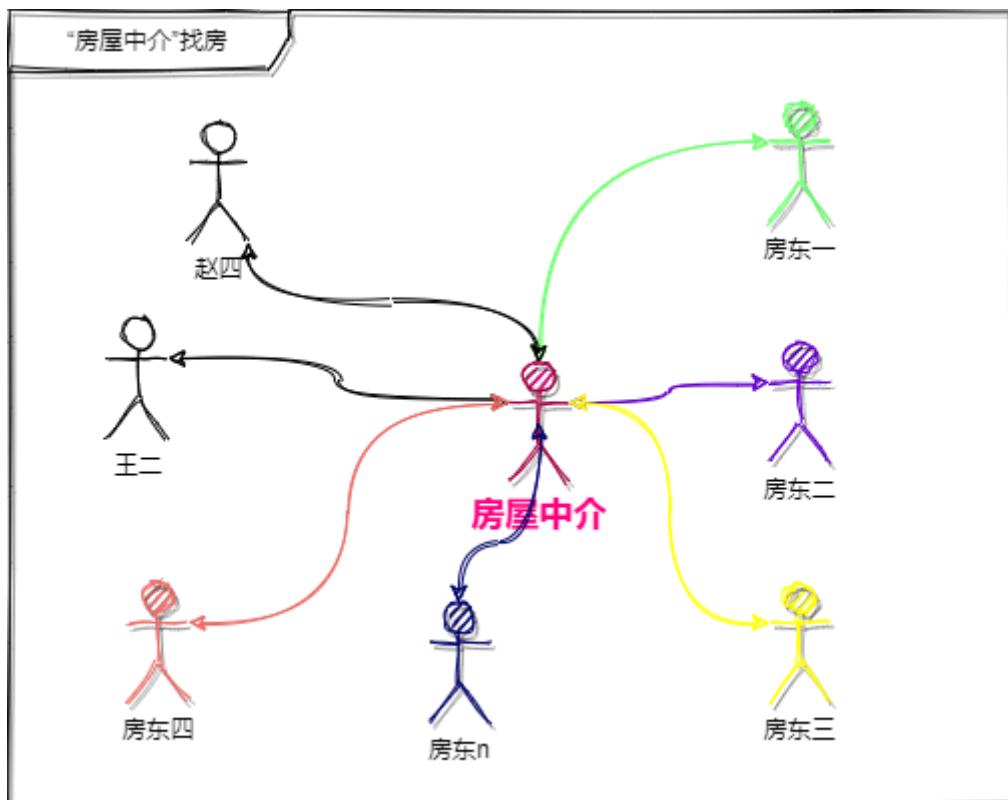
### 发挥人事专员的作用



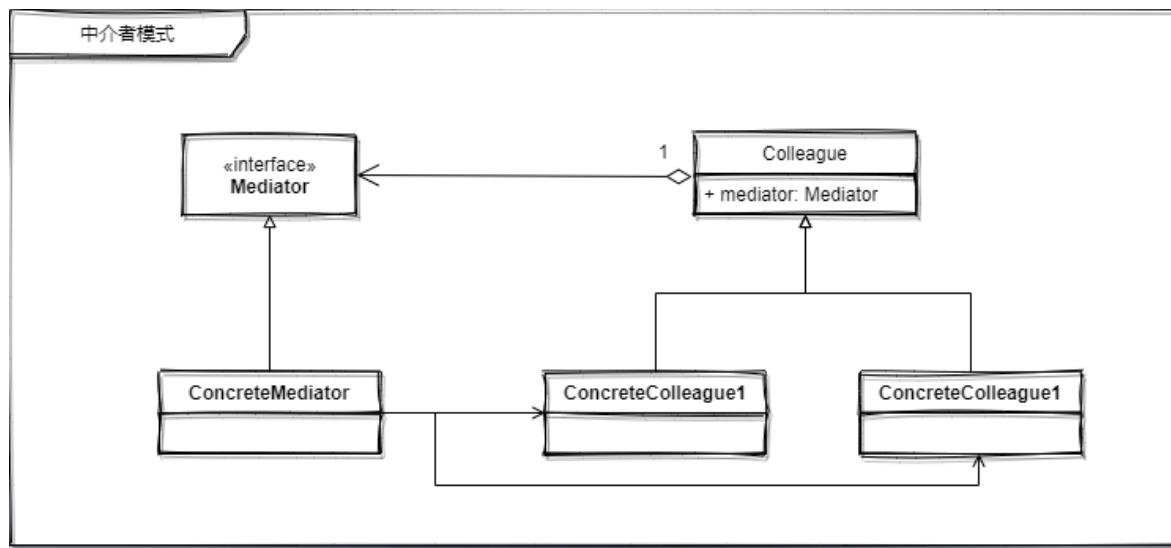
### 王二找房

因为有了输入就会有输出，王二给市场上输入了一种“找房”的需求，市场自然而然的反馈回来一个输出，那就是“房屋中介”

当有了“房屋中介”的加入，王二就没那么辛苦了，并且各个房东的资源因为在“房屋中介”这也会让“租房”变得高效起来。



## 中介者模式类图



模式结构：

1. 中介者接口 Mediator
2. 具体中介者 ConcreteMediator
3. 同事抽象类 Colleague
4. 具体同事 ConcreteColleague

多学一点：在设计模式中发现几乎所有的模式结构图中，都会有接口或者抽象类，这其实是依赖倒置的默认思路，任何的程序设计这一点原则都要优先考虑。

## 代码

篇幅和过长代码展示原因：完整代码关注回复“源码”获取。

这里通过王二入职案例的解决方案为背景，用程序来表示一下如何解决多个对象间复杂交错导致程序难以维护最终崩溃（离职）的问题。

```
class HRTTest {  
  
    @Test  
    void send() {  
        HR hr = new HR();  
        DeviceManager deviceManager = new DeviceManager(hr);  
        hr.addColleague(deviceManager);  
        WangEr wangEr = new WangEr(hr);  
        hr.addColleague(wangEr);  
        WangZai wangZai = new WangZai(hr);  
        hr.addColleague(wangZai);  
  
        wangEr.send("我没有电脑");  
        deviceManager.send("设备管理员下发一台电脑");  
    }  
}
```

设备管理员收到消息：我没有电脑

王二收到消息：设备管理员下发一台电脑

# 总结

## 解决的问题

- 在 1 对多对象关系中，可以通过 **中介者模式** 来解耦，达成 1 对 1 的松耦合关系。

## 存在的问题

- 中介者模式** 自身有个隐患问题，就是中介者自己本身知道了太多的内容。稍有不慎，就会导致 **同事** 的信息出现问题。
- 由于 **中介者模式** 本身的原因，这个 **中介者** 会变得特别复杂。（对象间的复杂转换成了类复杂，两权相害取其轻的道理。）
- 在 多对多 的问题上，最好仔细的考虑一下，甚至考虑要不要用这种模式。

# 迭代器模式



提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

在看迭代器模式之前，我觉得应该来研究一段代码开开胃先。

## Java 中的 List 集合遍历

```
public class Appetizer {  
  
    public static void main(String[] args) {  
        ArrayList<String> strings = new ArrayList<>();  
        for (int i = 1; i <= 10; i++) {  
            strings.add("第" + i + "个元素");  
        }  
        Iterator<String> iterator = strings.iterator();  
        while(iterator.hasNext()) {  
            iterator.next();  
        }  
    }  
}
```

这段代码很简单，我们在日常开发中可能也是经常使用到。有的人可能会说了，啊不对，我用的都是

```
for(int i = 0; i < strings.size(); i++)
```

还有的朋友说了，我直接用增强 for 循环啊

```
for(String s : strings)
```

是的，没错。在日常开发中，或多或少的人会用以上两种方式来进行一个列表的遍历。那这两者有什么区别呢？让我们通过编译出来的 class 文件来一探究竟吧。

这里使用三种不同的写法来遍历一个 list

### java 源码文件

```
// 1. 使用迭代器遍历
Iterator<String> iterator = strings.iterator();
while(iterator.hasNext())
    iterator.next()

// 2. jdk 8 提供的 lambda 写法
strings.forEach(System.out::println);

// 3. 增强 for 循环写法
for (String string : strings) {
    System.out.println(string);
}

// 4. 下标遍历
for (int i = 0; i < strings.size(); i++) {
    System.out.println(strings.get(i));
}
```

### class 反编译的 java 文件内容

```
// 1. 使用迭代器遍历
Iterator<String> iterator = strings.iterator();
while(iterator.hasNext()) {
    iterator.next();
}

// 2. jdk 8 提供的 lambda 写法
var10001 = System.out;
strings.forEach(var10001::println);
Iterator var3 = strings.iterator();

// 3. 增强 for 循环写法
while(var3.hasNext()) {
    String string = (String)var3.next();
    System.out.println(string);
}

// 4. 下标遍历
for(int i = 0; i < strings.size(); ++i) {
    System.out.println((String)strings.get(i));
}
```

第一种和第三种可以算为同一种，所以就只剩下三种迭代方式

```
// 1. 增强 for 循环（迭代器）
for(String s : strings)
// 2. JDK8 的 forEach 方法
strings.forEach()
// 3. 下标遍历
for(int i = 0; i < strings.size(); i++)
```

接下来我们用数据来看一下这几种方式的表现情况

## 第一次

测试方法: iterator

测试数据量: 1000000

花费时长 (ms) : 21

---

测试方法: forEach

测试数据量: 1000000

花费时长 (ms) : 132

---

测试方法: 增强 for 循环

测试数据量: 1000000

花费时长 (ms) : 18

---

测试方法: 下标遍历

测试数据量: 1000000

花费时长 (ms) : 1

## 第二次

测试方法: iterator

测试数据量: 1000000

花费时长 (ms) : 17

---

测试方法: forEach

测试数据量: 1000000

花费时长 (ms) : 123

---

测试方法: 增强 for 循环

测试数据量: 1000000

花费时长 (ms) : 12

---

测试方法: 下标遍历

测试数据量: 1000000

花费时长 (ms) : 3

## 第三次

测试方法: iterator

测试数据量: 1000000

花费时长 (ms) : 18

测试方法: **forEach**

测试数据量: 1000000

花费时长 (ms) : 119

测试方法: 增强 **for** 循环

测试数据量: 1000000

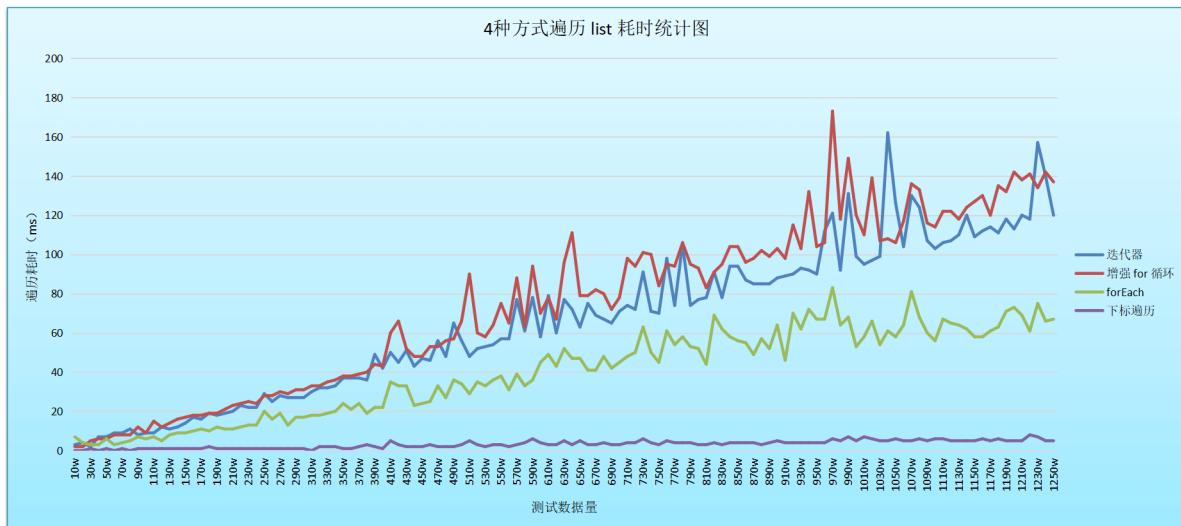
花费时长 (ms) : 14

测试方法: 下标遍历

测试数据量: 1000000

花费时长 (ms) : 2

为了更直观的展示, 我整理了一张统计图



通过数据也证实了 迭代器 和 增强 for 循环的写法效果相同。

## List 集合到底该如何遍历

其实乍一看数据, 应该用下标遍历这种方式啊, 当然, 正常是这样的, 这是因数组的下标索引决定的它的访问时间复杂度O(1), 同时 JDK 也为 `ArrayList` 增加了

```
public interface RandomAccess {  
}
```

标记。标记其为随机访问集合。

```

47     * provide asymptotically linear access times as they get huge, but constant
48     * access times in practice. Such a <tt>List</tt> implementation
49     * should generally implement this interface. As a rule of thumb, a
50     * <tt>List</tt> implementation should implement this interface if,
51     * for typical instances of the class, this loop:
52     * <pre>
53     *     for (int i=0, n=list.size(); i &lt; n; i++)
54     *         list.get(i);
55     * </pre>
56     * runs faster than this loop:
57     * <pre>
58     *     for (Iterator i=list.iterator(); i.hasNext(); )
59     *         i.next();
60     * </pre>
61     *
62     * <p>This interface is a member of the
63     * <a href="{@docRoot}/..//technotes/guides/collections/index.html">
64     * Java Collections Framework</a>.
65     *
66     * @since 1.4
67     */
68     public interface RandomAccess {
69 }

```

Java 官方给出的遍历说明：根据经验，下标要比迭代器更快。

当然，根据我们对数据的测试表现情况来看，当你的数据量低于30w时，这个时间差基本是没有任何影响的，我想，这一切应该得益于当下处理器的计算能力以及内存更高的数据交互速度吧。所以你用以上的 4 种方式都是没有问题的。但是如果你遍历的数据量大于100w时，一定要使用下标遍历了。

关于 List 集合的遍历，我们就讨论这么多，更主要的是我们要讨论一下上面提到的一个东西，“迭代器”

文末关注回复“源码”获取本文测试使用代码及图表数据

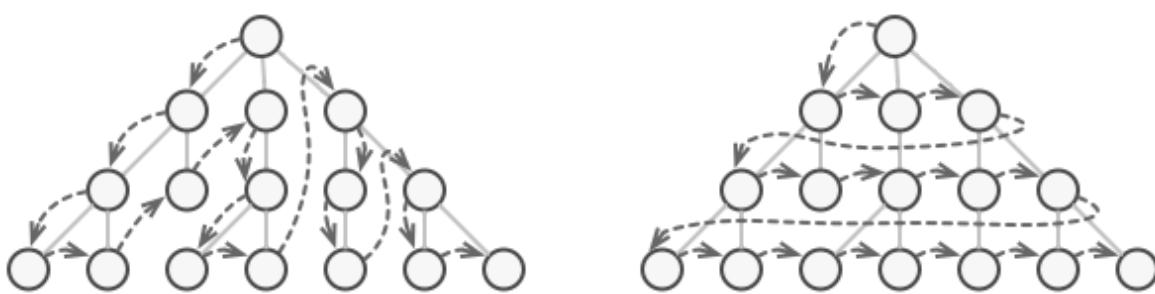
## 迭代器

我通过上面的开胃菜知道，迭代器是用来遍历集合的，或者说它是用来遍历的。

这个时候我们就想了，那刚刚的列表不用这个迭代器速度反而更快，用它还慢还麻烦，为什么要用它呢？

我们可以想象一下，如果此时的数据结构不是数组，而是链表、是树、是图呢？

集合本身的是存取，目的明确，但如果我们在集合本身增加了遍历操作的话，我们可以看看下图。



图片来源：<https://refactoringguru.cn/design-patterns/iterator>

假如对于一个链表，我们开始想要以 DFS 方式遍历，写好了一个算法在集合类中，后面发现有需要 BFS 方式遍历这个集合，以目前这种方式，就只能去修改集合类，再加一个方法。如果哪天发现这两个都不合适，又要加一个呢，慢慢的，集合本身的存取目的开始变得不明确，这其实是因为违反了单一职责原则。

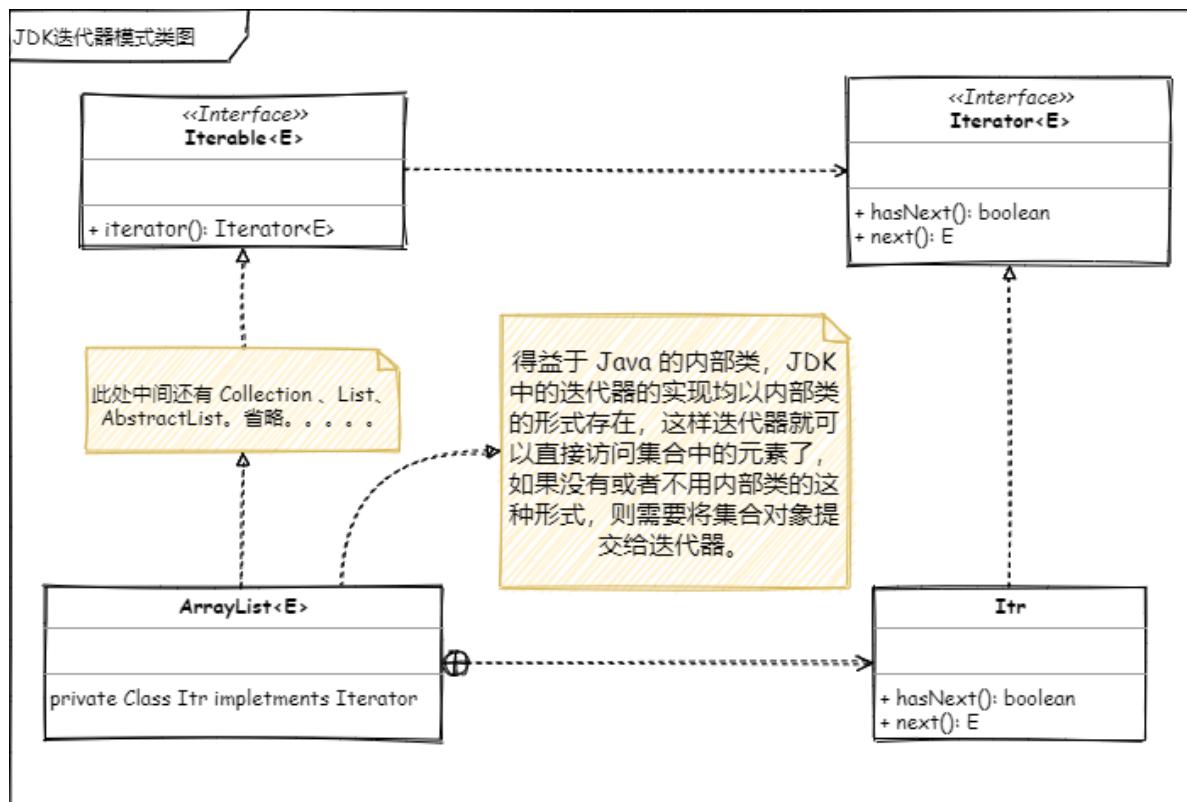
致使遍历访问的问题需要被单独解决。于是迭代器就出现了，它要解决的问题就是用来遍历集合，同时它并不需要去关注具体要遍历的集合是什么样的数据结构。这里我们可以回想一下刚刚测试遍历列表的操作，迭代器在迭代的时候，它知道遍历的具体的数据结构是什么吗？不知道，对于一个迭代器来讲，它只需要关注如何将集合的数据完整无缺的取出来就好了。

这样，迭代器的概念就捋清了，再看看迭代器模式的定义

提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

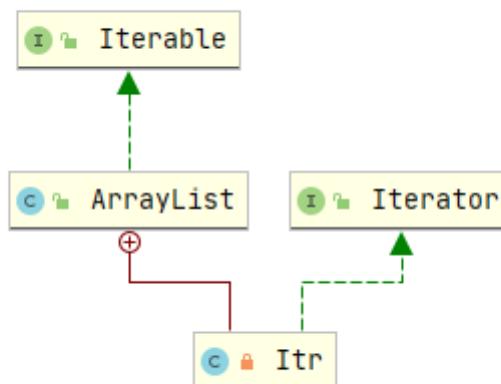
接下来我们就看看 JDK 中是怎么样用这个迭代器模式来设计集合遍历程序的。go！

## 迭代器模式类图



下面是根据 JDK 的类直接生成的 UML 类图

注意：并非全部类生成的 UML 类图，这里去掉了一些无关类。



## 代码

因为迭代器模式是围绕 JDK 的 iterator 来学习的，所以这里具体的迭代器模式的实现代码就没有自己实现，不过这里学习了 JDK 的代码。我就简单记录一下 JDK 的实现思路。

1. 定义一个可被迭代的约束类，表示其子类可以被迭代器迭代，这里用的是 Iterable 接口。
2. 定义迭代器接口，其子类为具体的迭代器实现。这里说的是 Iterator 接口。
3. 可被遍历的集合实现 Iterable 接口，比如 ArrayList。
4. 具体迭代器的实现 Iterator 接口，JDK 使用的 ArrayList 的内部类 Itr 来实现的这个接口。
5. 迭代器中的提供一个顺序访问的规则，然后通过顺位标识调用集合的“get”方法。

为了直观表示这个“get”方法，我贴几个迭代器的 next() 实现关键部分代码。

### ArrayList

```
public E next() {  
    // 这里的 elementData[i] 就是我上面说的 “get” 方法  
    return (E) elementData[lastRet = i];  
}
```

LinkedList(使用的是 AbstractList 的迭代器)

```
public E next() {  
    // 这里的 get 就是我上面说的 “get” 方法  
    E next = get(i);  
    return next;  
}
```

### HashSet/HashMap

```
public final K next() {  
    // 这里的 nextNode() 就是我上面说的 “get” 方法  
    return nextNode().key;  
}  
final Node<K,V> nextNode() {  
    Node<K,V>[] t;  
    Node<K,V> e = next;  
    if ((next = (current = e).next) == null && (t = table) != null) {  
        do {} while (index < t.length && (next = t[index++]) == null);  
    }  
    return e;  
}
```

以上内容可以配合 JDK 源码了解，下面列一些涉及类或方法位置

1. Iterable.java
2. Iterator.java
3. ArrayList.java
4. java/util/ArrayList.java:846 (jdk8)

## 总结

1. 迭代器模式主要解决的问题就是集合的遍历与集合访问要进行合理的划分职责，这满足了**单一职责原则**。集合类本身专注集合的存取，迭代器专注集合的遍历。
2. 同时迭代器在实现的过程中不需要关注待遍历集合的数据结构，因为它会使用目标集合的“get”方法来按序读取集合元素。所以这使得了同一个迭代器可以遍历不同的集合，同样的同一个集合也可以用不同的迭代器来进行遍历。
3. 因为有了迭代器接口和可被迭代的集合接口两个接口的设计方式可以在集合或迭代器的扩展上提供很好的支持，这也满足了**开闭原则**。
4. **这个模式基本不会使用。除非你有自己的数据结构和对他们的遍历规则时。**

## 访问者模式

将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。它将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

刚看到这个模式的时候，我人都傻了，完全不知道说的是啥，直到看了近5份资料！才搞清楚这个设计模式，不愧是最复杂的一种，我也这样觉得。不过千万别被复杂吓到，捋清了之后，还是比较简单的。

## 开门见山

访问者模式“人如其名”，就是说不同的访问者对同一个对象的访问结果不同。为什么会有不同呢？因为这个访问者是我们自己定义的，我们就想让他不同②。

而实际情况更是如此。我通过几份资料总结下来，这个访问者模式所谓的访问者其实是我们想要控制的访问权限一样。因为任何一个“访问者”都可以看到具体数据的全部内容，他只是选择性的“不看”，这样便区分开了“访问者”**关注的内容，或者限制了访问者的权限**。

可能我说的有点绕，有点抱歉，我再简化一下这个内容。

## 网络用语

### 抛开表象看本质

如果我们抛开访问者模式这些专业的定义，单纯的去理解这个访问者模式要表达的意思，我觉得用一个东西最合适不过。那就是“网络用语”；

不知道大家听没听过前阵子火了的百度广告《你说啥》单曲。歌曲中的朝阳大妈就是一个不知道关注点或者是被限制了访问权限的访问者，当然他歌曲中说的网络语有好多我也不知道是啥③。没听过的快去听吧。

还有最近的 **凡尔赛文学** 我不百度的时候以为是个地名，所以我的 **权限** 也被限制了。

正好提到这个了，那我们就拿 **凡尔赛文学** 这个网络语来学习一下访问者模式吧~④

## 凡尔赛文学

### 首先我们就要再一次抛开表象看本质⑤

下面是我搜集到有关凡尔赛的释义：

1. 凡尔赛是法国巴黎的卫星城以及伊夫林省省会，曾是法兰西王朝的行政中心。
2. 《凡尔赛》是皮埃尔·苏勒执导的剧情片。
3. 以法国路易十四为时代背景的电视剧。

4. 凡尔赛文学，网络热词，指通过先抑后扬、自问自答或第三人称视角，不经意间露出“贵族生活的线索”。
5. 啥？？？

对于凡尔赛一共有 5 种释义，他的结构应该是这样的

```
public class Versailles {

    private final String interpretation1 = "凡尔赛是法国巴黎的卫星城以及伊夫林省省会，曾是法兰西王朝的行政中心。";
    private final String interpretation2 = "《凡尔赛》是皮埃尔·苏勒执导的剧情片。";
    private final String interpretation3 = "以法国路易十四为时代背景的电视剧。";
    private final String interpretation4 = "凡尔赛文学，网络热词，指通过先抑后扬、自问自答或第三人称视角，不经意间露出\"贵族生活的线索\"。";
    private final String interpretation5 = "啥？？？";

}
```

因为我们还要对这个数据进行访问，所以还要给他加个访问的方法 #visit

```
public class Versailles {

    .....
    .....
    ...
    /*
     * 访问
     */
    public void visit(){

    }
}
```

既然要访问，肯定要有访问者啊，因为访问者挺多的，比如我、我的小伙伴、还有你，所以我们就使用依赖倒置原则来定义一个访问者接口 `Visitor` 然后有个访问方法，再把凡尔赛给访问者去让其自己访问，那代码实现起来应该是这样的。

`Visitor` 接口

```
public interface Visitor {
    void visit(Versailles versailles);
}
```

凡尔赛的访问方法调整一下，最终完整类如下

```
public class Versailles {

    private final String interpretation1 = "凡尔赛是法国巴黎的卫星城以及伊夫林省省会，曾是法兰西王朝的行政中心。";
    private final String interpretation2 = "《凡尔赛》是皮埃尔·苏勒执导的剧情片。";
    private final String interpretation3 = "以法国路易十四为时代背景的电视剧。";
    private final String interpretation4 = "凡尔赛文学，网络热词，指通过先抑后扬、自问自答或第三人称视角，不经意间露出\"贵族生活的线索\"。";
    private final String interpretation5 = "啥？？？";
```

```

/**
 * 将该对象提供给访问者访问
 * @param visitor 访问者
 * 方法名改成 accept 更好，表示这个类接受一个访问者来访问自己
 */
public void accept(Visitor visitor){
    visitor.visit(this);
}

```

接下来就是具体的访问者了，那我根据实际情况来定义一些访问者

1. I
2. MyFriend
3. You

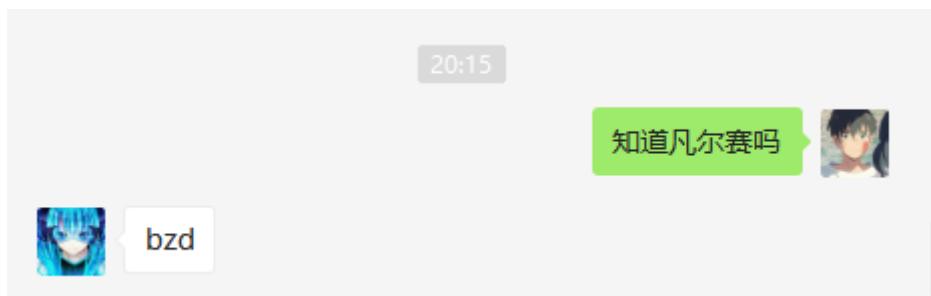
一共三个访问者

I (我自己)

我比较博学多识，我知道凡尔赛是地名、电影、电视剧三个

**MyFriend** (狗哥)

看他的样子应该是不知道



You (你呢？)

我就当你知道凡尔赛文学，已经领悟到了无形装逼的境界好了

看下这三个类的情况

```

/**
 * 我比较博学多识
 * <p>
 * 欢迎跟我一起学习，微信（lvgoice）公众号搜索：星尘的一个朋友
 *
 * @author lvgorice@gmail.com
 * @version 1.0
 * @blog @see http://lvgo.org
 * @CSDN @see https://blog.csdn.net/sinat_34344123
 * @date 2020/12/1
 */
public class I implements Visitor {
    @Override
    public void visit(Versailles versailles) {
        System.out.println(versailles.getInterpretation1());
        System.out.println(versailles.getInterpretation2());
        System.out.println(versailles.getInterpretation3());
    }
}

```

```
// 弱智狗哥
public class MyFriend implements Visitor {
    @Override
    public void visit(Versailles versailles) {
        System.out.println(versailles.getInterpretation5());
    }
}
// 网络达人
public class You implements Visitor {
    @Override
    public void visit(Versailles versailles) {
        System.out.println(versailles.getInterpretation4());
    }
}
```

最后我们在模拟一下运行起来的情况

结果，狗哥拉胯

1vgo 你知道凡尔赛吗?  
凡尔赛是法国巴黎的卫星城以及伊夫林省省会，曾是法兰西王朝的行政中心。  
《凡尔赛》是皮埃尔·苏勒执导的剧情片。  
以法国路易十四为时代背景的电视剧。

狗哥 你知道凡尔赛吗?  
啥？？？

你知道凡尔赛吗？  
凡尔赛文学，网络热词，指通过先抑后扬、自问自答或第三人称视角，不经意间露出“贵族生活的线索”。

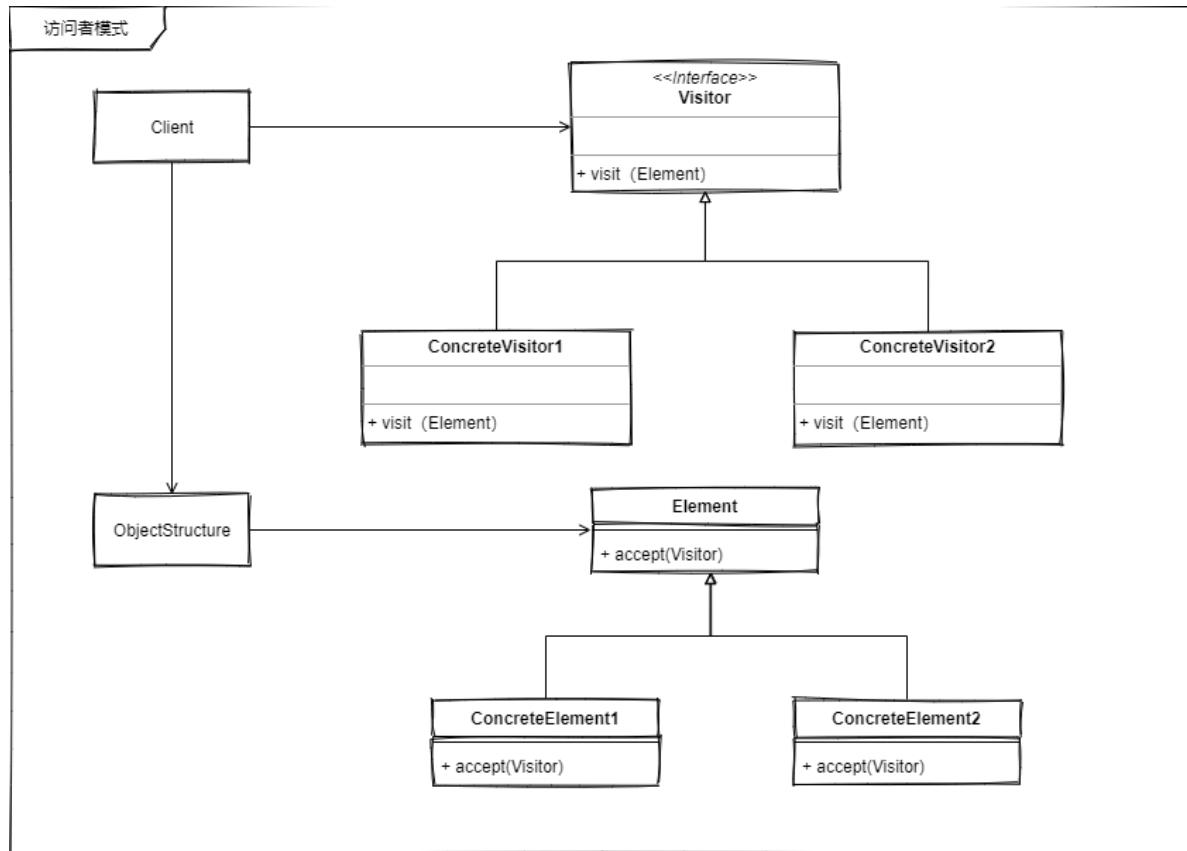
不同的访问者，看到数据结构中的结果不同。再来看下访问者的定义

在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。

虽然我们这里用的是一个对象，试着将它变成集合（多个网络语而已）吧。使用循环把每个元素都“送”给访问者，这个就留着给你动手试试吧，也留给自己以后回来看的时候能被逼动动脑😊。实在不想动，关注回复“源码”吧！😊

## 访问者模式类图 🎉

最后，我们来看下标准的访问者模式结构图



这个结构比较复杂

1. 客户端高层模块 `client`;
2. 访问者接口，依赖倒置接口 `visitor`;
3. 被访问的元素，`Element`;
4. 最后一个，`ObjectStructure` 对象结构;

这里唯一可能需要解释的就是这个 `ObjectStructure` 了，他即用于来定义管理 `Element` 的对象载体。它可以是我们业务场景中任何需要被访问元素的载体对象，比如上述例子中，我们想把这个结构放进去那我就可以定义一个词语类 `word`，里面可以有 `NetwordLanguage`, `Professional vocabulary` 等等对象。如下所示

```
public class Word {  
  
    /**  
     * 网络语  
     */  
    private final List<NetwordLanguage> networdLanguages = new ArrayList<>();  
  
    public void addword(NetwordLanguage networdLanguage) {  
        if (!networdLanguages.contains(networdLanguage)) {  
            networdLanguages.add(networdLanguage);  
        }  
    }  
}
```

访问者全部源代码关注回复“源码”获取

## 总结

访问者模式适合在**数据结构稳定的**系统中，即很少或不变的数据结构场景；

当你想要对一个数据集合增加一些不同的使用规则，或者是“权限”控制时，可以考虑使用访问者模式，并要一同考虑数据结构是否稳定（是否会在增加类），因为这会导致访问者需要“重构”。

**解决的问题：**

1. 访问者模式使数据结构与数据访问分离
2. 可以很灵活的增加不同的访问规则

**自身的问题：**

1. 一旦出现数据结构变更（新增类型），将会使访问者发生较大的修改，因为需要调整访问者接口！  
严重违反了开闭原则

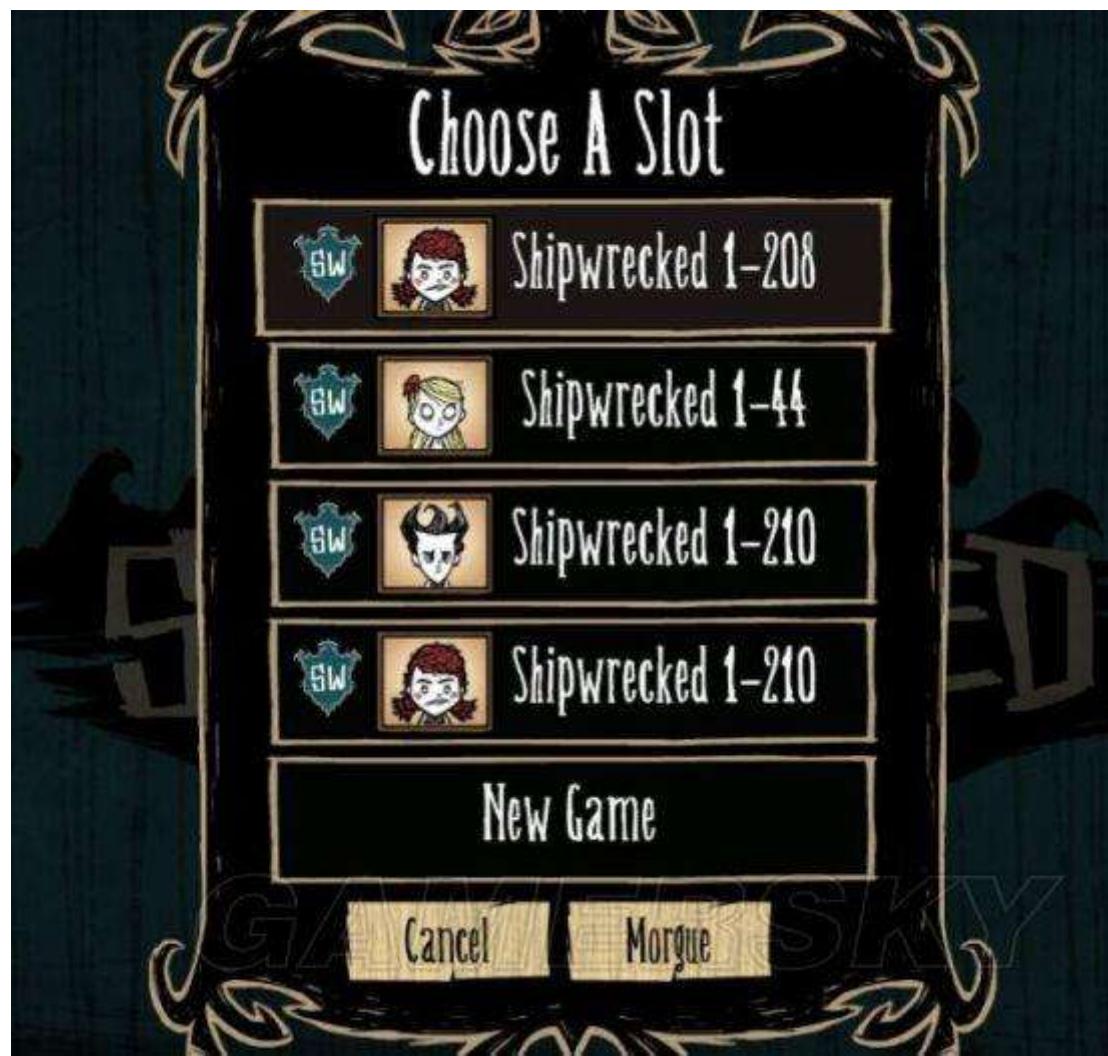
---

## 备忘录模式

在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

还记得那些年你的游戏存档吗？

**Don't Starve**



unascribed-game1



unascribed-game2



## 自己实现一个简易版的游戏存档功能

今天写一个游戏存档功能练习一下编码基本功。

### 需求分析

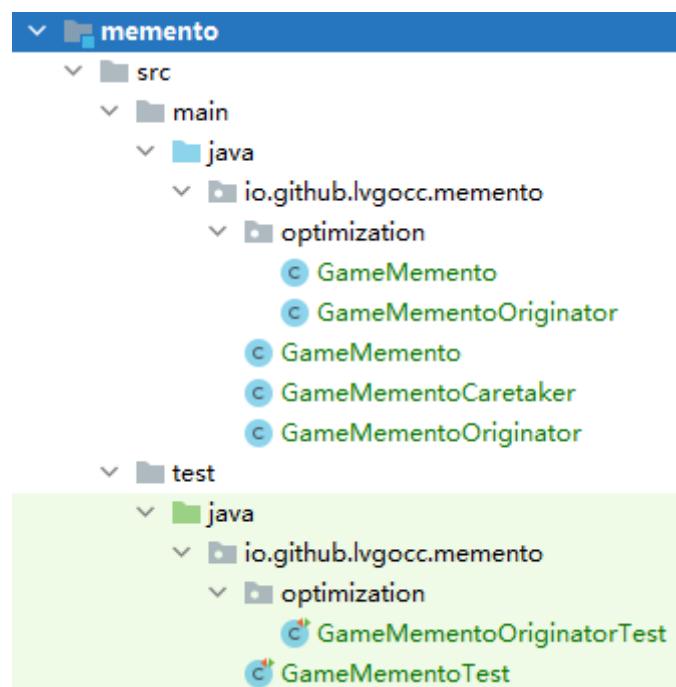
1. 需要有个管理存档的角色，那么多个存档总要有人来管啊，想存档或者读档需要它说了算。
2. 还需要有个具体的存档对象，方便我们管理呀。

### 需求设计

1. 存档对象的管理角色 Caretaker 管理存档对象
2. 存档对象 Memento 具体的存档对象
3. 存档的创建者 Originator 管理“存档”这件事，创建恢复存档。

## 代码

完整源码关注回复“源码获取”



测试结果

```
✓ Test Results 1s 160 ms "C:\Program Files (x86)\Java\jdk1.8.0_231\bin\java.exe" ...
  ✓ GameMementoTest 1s 160 ms
    ✓ testGameMement 1s 160 ms

      当前游戏状态:
      血量: 103
      体力: 33
      精神: 28
      保存游戏...
      存档成功!

      -----
      当前游戏状态:
      血量: 0
      体力: 14
      精神: 5
      you deaded!

      -----
      开始回档...
      读档成功!
      当前游戏状态:
      血量: 103
      体力: 33
      精神: 28
```

## 展示存档列表

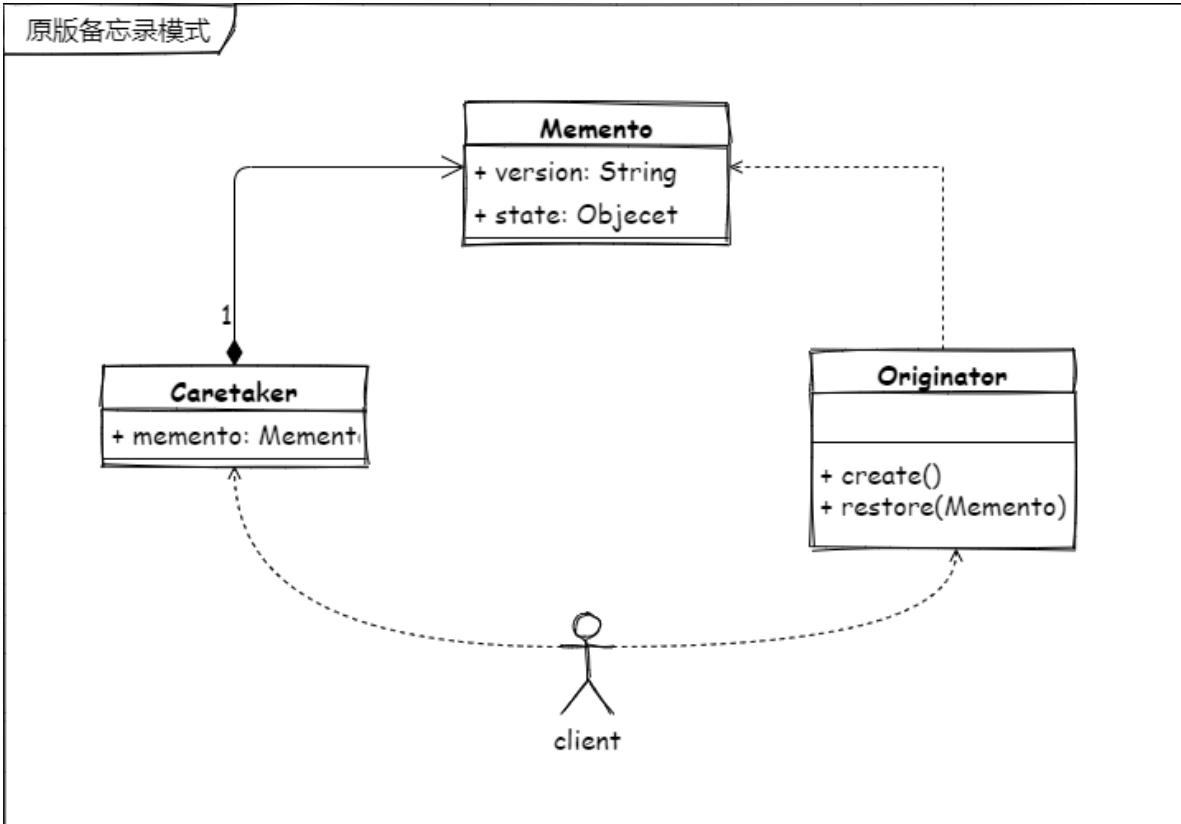
```
✓ Test Results 1s 351 ms
  ✓ GameMementoOrigi 1s 351 ms
    ✓ save0 1s 351 ms

      -----
      存档列表
      GameMemento{saveDate=Thu Nov 26 17:19:36 CST 2020, data='当前游戏状态:
      血量: 103
      体力: 33
      精神: 28'}
      GameMemento{saveDate=Thu Nov 26 17:19:36 CST 2020, data='当前游戏状态:
      血量: 110
      体力: 114
      精神: 53
      '}
      GameMemento{saveDate=Thu Nov 26 17:19:36 CST 2020, data='当前游戏状态:
      血量: 64
      体力: 74
      精神: 93
      '}

      -----
      选择第一个存档开始回档...
      读档成功!
      当前游戏状态:
      血量: 103
      体力: 33
      精神: 28
```

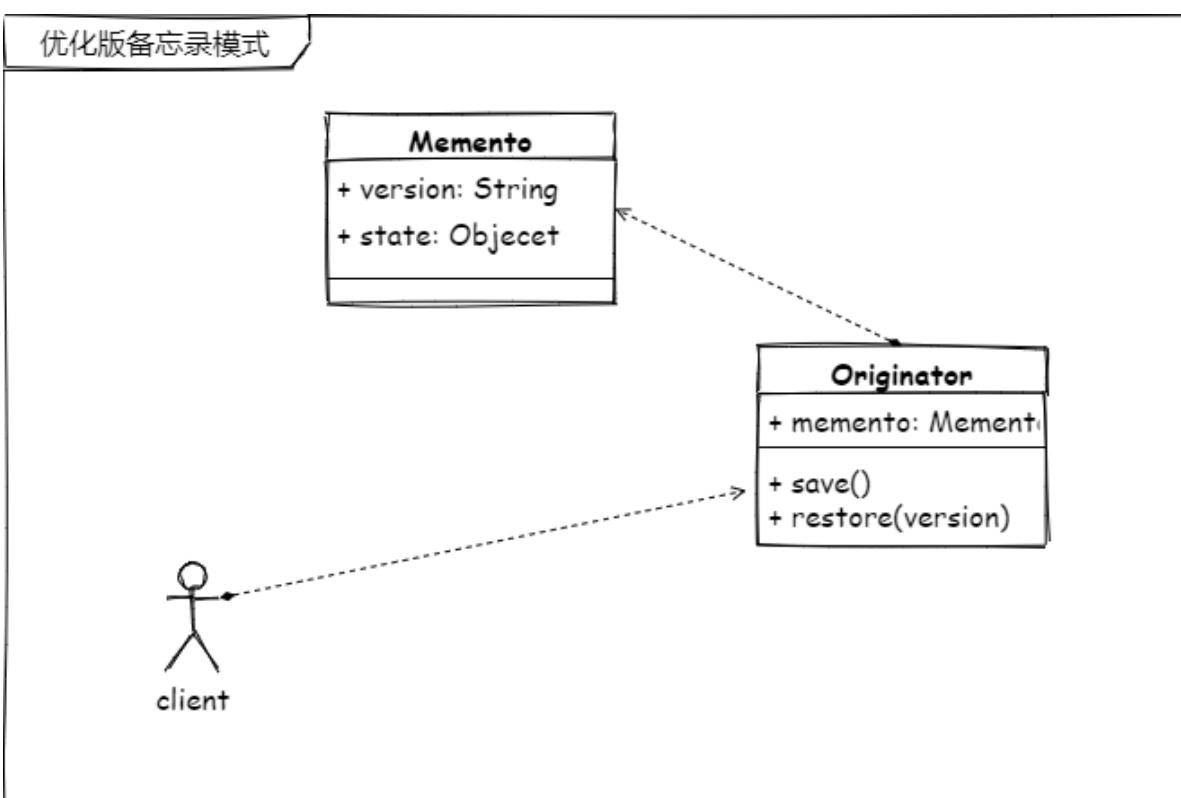
## 备忘录模式类图

### 原版备忘录模式



上面的代码是以这种类图方式实现的，不过这在 Java 中显然有可以优化的地方。即将 Caretaker 角色优化掉。优化后的类图

### 优化版备忘录模式



两个写法的完整源码关注回复“源码获取”，优化版代码在 optimization 目录中

## 总结

由于备忘录模式使用这种代码构建的方式在 Java 中并不多，这要得益于 Java 的 `Serializable` 接口，可以使用序列化来完成备份的操作。所以只是简单的使用一个游戏存档案例记录一下这个知识点，配合学习理解的最好的再就应该是文档编辑功能中的 `ctrl+z` 了。

备忘录模式可以使我们对一个对象的状态进行保存，在需要的时候快速恢复。不得不说的是备忘录模式如果保存的大量的对象时，内存的占用还是需要关注一下的。在备忘录模式中保存对象的时候，可以配合原型模式来一起使用。

最后，这个模式在 Java 中应该不太香，原因上面也说了。所以我对这个设计模式仅作为了解，真正要使用它的话我还是会尽可能的考虑使用 `Serializable`。

**优点：**无侵入备份/恢复对象状态（在 Java 里不是很香）。

**缺点：**GOF 中的写法在 Java 中不香，同时使用这种模式需考虑内存占用问题。

---

## 解释器模式

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

### 写在前面

这篇是《和 lvgo 一起学设计模式》系列的最后一个设计模式了，这篇就轻松一些吧。

因为时代的发展、技术的更替等等原因（你想做的解释器都有人做好了，且开源）吧，可能这个是我们很长一段时间都用不到的一种设计模式了。

### 你能看懂TA的“眼色”吗？

还记得那些年看过的影视剧吗？或是表情包吗？





你能看懂柯镇恶和“老婆”的眼色吗？

反正我是看不懂，单是看这情况，完全看不懂是什么意思。

但如果我提前给你说下规则呢？

### 柯镇恶图

1. 柯镇恶往左摆头，冲！
2. 柯镇恶往右摆头，撤！

### “老婆”图

1. “老婆”坐在坐垫上，生气！
2. “老婆”坐在摩的后面架子上，开心！

那这个时候再看他们的“眼色”，你能看懂了吗？如果有了上面的定义，我便知道了：

- 柯镇恶的意思是冲！（假设是往左摆头了）
- “老婆”很开心！

## 再谈解释器模式

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

定义一个语言：“眼色”

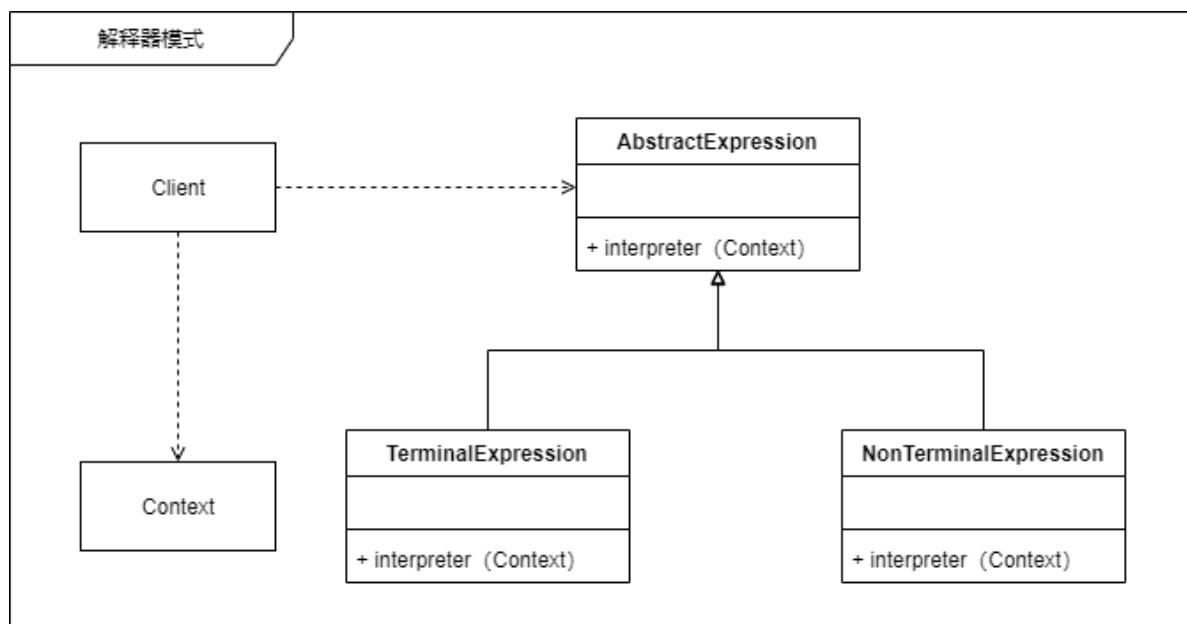
定义他的文法表示：“摆头”、“坐的位置”

定义解释器：“规则”

这样我们就可以通过这个解释器来了解TA了。

给定一个“眼色”，定义“摆头”或“坐的位置”，并定义一个规则，这样就可以解释图中的柯镇恶和“老婆”了。

## 解释器模式类图



这个结构比较简单，定义一个解释接口，然后就是两个具体的解释器

1. 最终解释器
2. 非最终解释器
3. 环境

这两个有点像组合模式中的子节点和叶节点的意思。这里的 `NonTerminalExpression` 是可以有多个的；

这里最麻烦的其实是 `Context` 环境。

## 代码

我们来看看代码来实现上面的“眼色”

完整代码关注回复“源码”获取。

```
@Test
void getType() {
    EyeColor eyeColor = null;
    Context context = new Context("柯镇恶往左摆头 | 老婆坐在了架子上");
    String content = context.getContent();
    String[] strings = content.split("\\|");
```

```
for (int i = 0; i < strings.length; i++) {
    String string = strings[i];
    context.setContent(string);
    if (string.contains("柯镇恶")) {
        eyeColor = new KeZhenE();
    } else if (string.contains("老婆")) {
        eyeColor = new Wife();
    }
    assert eyeColor != null;
    eyeColor.interpreter(context);
}
```

```
public class KeZhenE implements EyeColor {
    @Override
    public void interpreter(Context context) {
        if (context.getContent().contains("左摆头")) {
            System.out.println("冲！");
        } else if (context.getContent().contains("右摆头")) {
            System.out.println("撤！");
        }
    }
}
```

```
public class Wife implements EyeColor {
    @Override
    public void interpreter(Context context) {
        if (context.getContent().contains("座椅")) {
            System.out.println("生气！");
        } else if (context.getContent().contains("架子")) {
            System.out.println("开心！");
        }
    }
}
```

冲！  
开心！

## 总结

通过上面的内容我们了解到，解释器可以自己定义一些规则和对应的解释规则，来完成一些复杂的事情，这样就使得可以用一个简单的“动作”来达成一件复杂的事情。你看柯镇恶一个眼色，我就知道他想冲，他省去了复杂的“张嘴”过程。

其实解释器模式就像是我们现在用高级语言来开发软件程序一样，是怎么才能让计算机知道我们在说什么呢？其实这就是解释器的作用，我们按照一定规则（语法）来编写代码，然后解释器按照定义好的规则来将我们的代码翻译成机器认识的 01 代码。

对于解释器，它将复杂的事自己“包揽”了，但是一旦发生新的规则，你就不得不去修改“包揽”的复杂解析过程。

在今天，解释器模式应该很少会在我们的应用自己去设计了，毕竟这如同设计一门语言一样，过程很复杂，还记得我们正在用的正则表达式吗？他就是一个轻量级的语言，如果有能力有机会的时候，也可以挑战一下，开发一个自己的语言。

# 一个话痨想说的话（无关内容，可跳过）

为了不影响阅读，我把之前穿插写在文章中与其无关的内容单独的拿出来放到这最后。言论内容仅代表个人三观，以下内容选择性阅读。

## 设计模式的优缺点

我以为设计模式本身就是一种解决问题的办法，优缺点就本身而言有意义。相对来说即无意义。既然选择了这种模式，那固然有一定的道理。所以以后设计模式的优缺点也不会过多发表意见，没有最好，只有最合适。优缺点已经没有意义了。

## 返璞归真

我认为学习设计模式的过程就像拿到一台游戏机，玩到最后，我都会拆开看看里面是什么，而早已不关心游戏好不好玩了。

知识也是一样，知其然而知其所以然。

**我们为了学会使用某种东西看他的操作手册就可以了；**

**我们如果想要学会修某种东西就需要看他的设计手册；**

**当我们想要创造某种东西，你就需要掌握很多很多设计手册，将他们的经验进行吸收、消化、提炼。才能有更好的结果。**

当然，如果你只想会用，有一份差不多的操作指南也就够了。

抱歉这一篇说了一些题外话。

## 设计模式的学习问题!??

我一直都在想如何学会一个新东西，和掌握一个旧东西，有很多前辈给过一些建议和意见，（当然不是直接给我，都是看大佬的文章或者书籍当中。）先去用，再去学。嗯，我想是的，这样肯定是个很正常的学过程。但是对于一些你暂时无法使用的东西，你如何去掌握它呢？我认为应该抛开表象去了解本质，通过本质的类比去掌握那些暂时无法使用的东西。设计模式就是这种情况，在平常的开发中，常用的设计模式就那么几种，其它的那些没有机会去接触，干学，如何才能掌握呢？

我从小就是一个好为人师的家伙，然而自己学习却一直都不怎么样。人太实在，也不懂的包装，我只知道我学习这么差的能学会的东西，讲给别人应该差不哪去。

在学习算法的时候，学到一个特别巧妙而且很好用的解题思路或者说逻辑思维“分而治之”，还有之前和网友讨论的一个叫做“复杂度守恒定律”的东西。这两个东西放在一起，可以说是很“矛盾”了。

说了这么多我想说的就是，设计模式这个知识点，真的很简单又很难。简单是因为每种设计模式的定义拿出来都能看的懂，难是难在如何，何时的应用。而我希望我写的这个设计模式系列是去应用化的，就是单纯的把设计模式的思想记录下来。我所理解的设计模式就该如此。

GOF的《设计模式.可复用面向对象软件的基础》通过一个应用案例的实现，串起了23种设计模式，我想在系列更新完之后也更新一个应用，将23种设计模式尽可能的应用到一个应用案例中去，而不是每个设计模式都涉及具体的应用，因为我觉得这可能会带的人们更关注应用的实现，而不是设计模式的应用，这是我的一些想法。

**不要纠结每种模式的具体实现，把它们抽象出来，你能够清楚的描述每种模式是为了解决什么问题而存在的时候，就已经掌握它了，就可以把它用在任何当你需要的时候。而不是问什么时候需要它，这可能有点绕，不过它是真的。不要问我设计模式可以解决哪些问题，把问题给我，我告诉你用什么设计模式可以解决它！**

我发现我更适合写一些畅谈型的文章

## 再墨迹一点

设计模式这东西，我最早的理解就是武功秘籍，一招一式都是固定的。丢了一招半式可能就没用了，就像《武状元苏乞儿》中“星爷”饰演的苏乞儿最后来与赵无极那里，降龙十八掌少一掌都打不赢。而后又有无名的无名剑法、张三丰的太极剑法，万变不离其宗、无招胜有招。武学的真正奥义实为融会贯通，设计模式其便是技术这片江湖的一本绝世武功秘籍，幸运的是现在的武功秘籍人手一本。

## 科普闲聊

复杂度守恒定律由Larry Tesler于1984年提出，也称泰斯勒定律（Tesler's Law）。复杂度守恒定律（Law of conservation of complexity）由Larry Tesler于1984年提出，也称泰斯勒定律（Tesler's Law）。

根据复杂度守恒定律，每个应用程序都具有其内在的、无法简化的复杂度。无论在产品开发环节还是在用户与产品的交互环节，这一固有的复杂度都无法依照我们的意愿去除，只能设法调整、平衡。

这一观点主要被应用在交互设计领域。我们不得不面对的问题是，该由谁来为这一固有的复杂度埋单。打个比方，应该由软件开发工程师花费额外的时间来使软件变得更加简单好用，还是应该让用户自己去解决软件使用中可能存在的问题？

以上出自百度百科：[复杂度守恒定律 - 百度百科](#)

如上所述，复杂度守恒定律是一个规避不掉的东西，最早的时候我接触到这个词是发出的一个提问，当时有各种大佬出来解答，大家感兴趣可以去看看。

### [到底什么是RPC?远程调用有什么好处?](#)

迷惑不解，不知是何。

我了解了一下dubbo框架，很多的术语搞得是更加模糊不清。

顺便提一点，

为什么深奥的东西就是被人向往的？

将复杂的东西弄成粗浅易懂的这不是更好吗？

2018-01-15 09:04:03

但我一直以为，技术的东西，本就不复杂。让它变得复杂的是我那迫切想要得到结果的心。

学习从来都没有捷径，你只是想要速成。学的快慢是一个问题，学与不学是另一个问题，听懂掌声。

## 聊聊自己理想的“知识”

### 知识该怎么分享

周末的时候去了图书馆，去计算机技术区域想找一些书看，于是翻到了一本《零基础读懂云计算》，我发现我和作者的心态非常相似，他所谈及的就是因为“云计算”被太多太多的人去层层定义，结果导致人们对“云计算”这个词语已经开始有了一些丢失本质的理解了。他站在了“云计算”的本身出发，去掉了对它的层层包装，让读者真正的明白了什么是“云计算”（通过自己的思考），而不是将“云计算”给你定义一个什么什么高大上的名词来让你觉得很神奇的一种姿态来讲，字里行间也流露着他书名的意图。至少我看了之后，可以拨开很多营销或应用谈及的“云计算”虚伪的面纱，如果某人给我说哪个应用是怎么怎么利用“云计算”来完成的某个什么什么业务的时候，我也知道该怎么去追问他然后去判断到底他是不是应用了“云计算”。

我写下的，记录的这个设计模式系列文章《和 lvgo 一起学习设计模式》也是这个初衷。我希望能把每个设计模式也当成一个单纯的“套路”记录下来，然后希望自己以后复习和现在正在阅读的你都能自己去思考其中具体的内容。而不是走马观花或**强行应用某种模式**写一个案例来对自己甚至是正在阅读的你来一个“洗脑式”学习。

想要掌握一个知识，一门技术，一定要有自己的理解在其中，保持着怀疑的态度可以学到更多。因为你想知道的更多，随着你知道的越多，你不知道的就越多。**但要注意这个过程是一个体系的深入，而不是发散，不然你会发现自己“虚胖”。**



## 知识该怎么学习

网络中的知识多如牛毛，很多内容大多都是每篇文章作者自己的理解写出来的，还有一些利用“原型模式”写出来的，无关怎么写的，作为读者的我们都应该具备一些内容虚实辨别能力，多去抽象的理解，抛开层层包装，看其本质。而不是“双兔傍地走，安能辨我是雌雄”。举个不恰当就像你怎么辨别一个人是男人还是女人，无关他穿什么衣服，是否化妆，是否整容，是不是长发，等等等这些外在的包装。（如果是分辨正常人，一定有一个非常准确的办法，我不说你也知道）