第2章 简单工厂

在java编程中，出现只知道接口而不知实现，该怎么办?

解决方案：使用简单工厂来解决问题

> 提供一个创建对象实例的功能，而无需关心其具体实现。被创建实例的类型可以是接口，抽象类，也可以是具体的类。

简单工厂的优点

- 帮助封装
- 解耦

简单工厂的缺点

- 可能增加客户端的复杂度
- 不方便扩展子工厂

简单工厂的本质是：选择实现

何时选用简单工厂

> 想要完全封装隔离具体实现，让外部只能通过接口来操作封装体。
>
> 想要把对外创建对象的职责集中管理和控制，可以选用简单工厂。

第3章 外观模式

如何能让子系统外部的客户端在使用子系统的时候，既能简单地使用这些子系统内部的模块功能，而又不用客户端去与子系统内部的多个模块交互呢?

解决方案是使用**外观模式**来解决问题

外观模式：

> 为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

外观模式的目的

> 不是给子系统添加新的功能接口，而是为了让外部减少与子系统内多个模块的交互，松散耦合，从而让外部能够更简单地使用子系统。

外观模式的优点

- 松散耦合
- 简单易用
- 更好地划分访问的层次

外观模式有如下缺点

过多的或者是不太合理的Facade也容易让人迷惑。到底是调用Facade好呢，还是直接调用模块好。

外观模式的本质：封装交互，简化调用

外观模式很好地体现了"最少知识原则"

何时用外观模式：

- 为一个复杂的子系统提供一个简单接口的时候；
- 想要让客户程序和抽象类的实现部分松散耦合，可以考虑使用外观模式。
- 构建多层结构的系统，可以考虑使用外观模式。

第4章 适配器模式

优缺点：

- 更好的复用性
- 更好的可扩展性
- 过多的使用适配器会让系统非常零乱，不容易整体进行把握。

适配器的本质是：转换匹配，复用功能。

何时选用适配器：

- 如果你想要用一个已经存在的类，但是它的接口不符合你的需求，此时可以用。
- 如果你想创建一个可以复用的类，这个类可能和一些不兼容的类一起工作，此时可以用
- 如果你想用一些已经存在的子类，但是不可能对每一个子类都进行适配，此时可以使用适配器。直接适配这些子类的父类就可以了。

第5章 单例模式(Singleton)

保证一个类只有一个实例，并提供一个访问它的全局访问点。

单例模式分为两种：一种是懒汉式，一种是饿汉式。

单例模式是用来保证这个类在运行期间只会被创建一个类实例。

Java中一种更好的单例实现方式

既能够实现延迟加载，又能够实现线程安全。

这个解决方案被称为Lazy initialization holder class模式，这个模式综合使用了Java的类级内部类和多线程缺省同步锁的知识。

```
public class SingletonTaste {

    private static class SingletonHolder {
        private static SingletonTaste singletonTaste = new
SingletonTaste();
    }

    private SingletonTaste() {}

    public static SingletonTaste getInstance() {
        return SingletonHolder.singletonTaste;
    }
}
```

思考单例模式

单例模式的本质：**控制实例数目**

何时选用单例模式

当需要控制一个类的实例只能有一个，而且客户只能从一个全局访问点访问它时，可以选用单例模式。

相关模式

比如抽象工厂方法中的具体工厂类就通常是一个单例。

第6章 工厂方法模式(Factory Method)

定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method使一个类的实例化延迟到子类。

模式讲解

1.工厂方法模式的功能

工厂方法模式的主要功能是让父类在不知道具体实现的情况下，完成自身的功能调用；而具体的实现延迟到子类来实现。

2.实现成抽象类

工厂方法的实现中，通常父类会是一个抽象类，里面包含创建所需对象的抽象方法，这些抽象方法就是工厂方法。

这里有一点需要注意的是，子类在实现这些抽象方法的时候，通常并不是真正地由子类来实现具体的功能，而是在子类的方法里面做选择，选择具体的产品实现对象。

3.实现成具体的类

也可以把父类实现成为一个具体的类。这种情况下，通常是在父类中提供获取所需对象的默认实现方法，这样即使没有具体的子类，也能够运行。

通常这种情况还是需要具体子类来决定具体要如何创建父类所需要的对象。也把这种情况称为工厂方法为子类提供了挂钩。通过工厂方法，可以让子类对象来覆盖父类的实现，从而提供更好的灵活性。

4.工厂方法的参数和返回

工厂方法的实现中，可能需要参数，以便决定到底选用哪一种具体的实现。也就是说通过在抽象方法里面传递参数，
在子类实现的时候根据参数进行选择，看看究竟应该创建哪一个具体的实现对象。
一般工厂方法返回得是创建对象的接口对象，当然也可以是抽象类或者一个具体的类的实例。

5.谁来使用工厂方法创建的对象

事实上，在工厂方法模式里面，应该是creator中的其他方法在使用工厂方法创建的对象，虽然

也可以把工厂方法创建的对象直接提供给creator外部使用，
但工厂方法模式的本意是由creator对象内部的方法来使用工厂方法创建的对象，也就是说，工厂方法一般不提供给creator外部使用。

客户端应该使用creator对象，或者使用由creator创建出来的对象。对于客户端使用creator对象，这个时候工厂方法创建的对象，是creator中的某些方法使用；
对于使用那些由creator创建出来的对象，这个时候工厂方法创建的对象，是构成客户端需要的对象的一部分。

工厂方法模式与IoC/DI

工厂方法模式和IoC/DI的关系

平行的类层次结构

参数化工厂方法

所谓参数化工厂方法指的就是：**通过给工厂方法传递参数，让工厂方法根据参数的不同来创建不同的产品对象**

当然工厂方法创建的不同的产品必须是同一个Product类型的。

工厂方法模式的优缺点

工厂方法模式的优点

- 可以在不知道具体的实现的情况下编程
- 更容易扩展对象的新版本
- 连接平行的类层次

工厂方法模式的缺点

- 具体产品对象和工厂方法的耦合性

思考工厂方法模式

工厂方法模式的本质

延迟到子类来选择实现。

对设计原则的体现

工厂方法模式很好体现了“依赖倒置原则”

依赖倒置原则告诉我们“要依赖抽象，不要依赖于具体类”。

何时选用工厂方法模式

- 如果一个类需要创建某个接口的对象，但是又不知道具体的实现，这种情况可以选用工厂方法模式，把创建对象的工作延迟到子类中去实现。
- 如果一个类本身就希望由它的子类来创建所需的对象的时候，应该使用工厂方法模式。

第7章 抽象工厂模式

此处举的实际例子是装机的例子，CPU和主板有关系的，需要互相匹配的。

虽然用简单工厂解决了具体和抽象的关系但是没有解决上面说的关系的维护。

使用抽象工厂模式来解决问题

抽象工厂模式的定义

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

这里要解决的问题是要创建一系列的产品对象，而且这一系列对象是构建新的对象所需要的组成部分，也就是这一系列被创建的对象相互之间是有约束的。

解决这个问题的一个解决方案就是抽象工厂模式，在这个模式里面会定义一个抽象工厂，在里面虚拟地创建客户端需要的这一系列对象。

所谓虚拟的就是定义创建这些对象的抽象方法，并不去真正的实现，然后由具体的抽象工厂的子类来提供这一系列对象的创建。

这样一来可以为同一个抽象工厂提供很多不同的实现，那么创建的这一系列对象也就不一样了，也就是说，抽象工厂在这里起到一个约束的作用，并提供所有子类的一个统一外观，来让客户端使用。

使用抽象工厂模式来重写示例，先来看看如何使用抽象工厂模式来解决前面提出的问题。

装机工程师要组装电脑对象，需要一系列的产品对象，比如CPU，主板等。于是创建一个抽象工厂给装机工程师使用，在这个抽象工厂里面定义抽象地创建CPU和主板的方法，这个抽象工厂就相当于一个抽象的装机方案，在这个装机方案里面，各个配件是能够相互匹配的。

每个装机客户会提出他们自己的具体装机方案，或者是选择已有的装机方案，相当于为抽象工厂提供了具体的子类，在这些具体的装机方案里面，会创建具体的CPU和主板实现对象。

结构示意图参见《研磨设计模式》P137

模式讲解

1. 抽象工厂模式的功能

   抽象工厂的功能是为了一系列相关对象或相互依赖的对象创建一个接口。一定要注意，这个接口内的方法不是任意堆砌的，而是一系列相关或相互依赖的方法

2. 实现成接口

   AbstractFactory在Java中通常实现成接口，大家不要被名称误导了，以为是实现成为抽象类。当然，如果需要为这个产品簇提供公共的功能，也不是不可以把AbstractFactory实现成为抽象类，但一般不这么做。

3. 使用工厂方法

   AbstractFactory定义了创建产品所需要的接口，具体的实现是在实现类里面，通常在实现类里面就需要选择多种更具体的实现。所以AbstractFactory定义的创建产品的方法可以看成是工厂方法，而这些工厂方法的具体实现就延迟到具体的工厂里面。也就是说使用工厂方法来实现抽象工厂。

4. 切换产品簇

   由于抽象工厂定义的一系列对象通常是相关或者相互依赖的，这些产品对象就构成了一个产品簇，也就是抽象工厂定义了一个产品簇。

   这就带来了非常大的灵活性，切换一个产品簇的时候，只要提供不同的抽象工厂实现就可以了，也就是说现在是以产品簇作为一个整体被切换。

5. 抽象工厂模式的调用顺序示意图

   首先是client创建一个具体工厂的示例，具体工厂这边返回抽象工厂类型的实例。然后客户端调用创建产品A的方法与抽象工厂交互此处是抽象工厂返回的产品A，然后客户端调用创建产品B的方法同样是抽象工厂返回的产品B，然后客户端调用产品A的方法，与具体的产品A交互了。然后客户端调用产品B的方法。

可扩展的工厂

抽象工厂为每一种它创建的产品对象定义了相应的方法，这种实现有一个麻烦，就是如果在产品簇中要增加一种产品，所有的具体工厂实现都要发生变化，如此就非常不灵活。

现在有一种相对灵活，但不太安全的改进方式可以解决这个问题：抽象工厂不需要定义那么多方法，定义一个方法就足够了，给这个方法设置一个参数，通过这个参数来判断具体创建什么产品对象；由于只有一个方法，在返回类型上就不能是具体的某个产品类型了，只能是所有的产品对象都继承或者实现的这么一个类型，比如让所有的产品都实现某个接口，或者干脆使用object类型。

抽象工厂模式和DAO

DAO：数据访问对象，Data Access Object 是JEE中的一个标准模式，通过它来解决访问数据对象所面临的一系列问题，比如数据源不同，存储类型不同，访问方式不同，供应商不同，版本不同等。

对于需要进行数据访问的逻辑层而言，它可不想面对这么多不同，也不想处理这么多差异，它希望能以一个统一的方式来访问数据。

也就是说，DAO需要抽象和封装所有的对数据的访问，DAO承担和数据仓库交互的指责，这也意味着，访问数据所面临的所有问题，都需要DAO在内部来自行解决。

DAO和抽象工厂的关系

在实现DAO模式的时候，最常见的实现策略就是使用工厂的策略，而且多是通过抽象工厂模式来实现，当然在使用抽象工厂模式来实现的时候，可以结合工厂方法模式。

抽象工厂模式的优缺点

抽象工厂模式的优点

- 分离接口和实现

  客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已。也就是说客户端从具体的产品实现中解耦。

- 使得切换产品簇变得容易

抽象工厂模式的缺点

- 不太容易扩展新的产品
- 容易造成类层次复杂

思考抽象工厂模式

本质是：选择产品簇的实现。

工厂方法是选择单个产品的实现，抽象工厂着重的是为一个产品簇选择实现，定义在抽象工厂里面的方法通常是有联系的，它们都是产品的某一部分或者是相互依赖的。

如果抽象工厂里面只定义一个方法，直接创建产品，那么就退化成为了工厂方法了。

何时选用抽象工厂模式

- 如果希望一个系统独立于它的产品的创建，组合和表示的时候，换句话说，希望一个系统只是知道产品的接口，而不关系实现的时候。
- 如果一个系统要由多个产品系列中的一个来配置的时候。换句话说，就是可以动态地切换产品簇的时候。
- 如果要强调一系列相关产品的接口，以便联合使用它们的时候。

第8章 生成器模式

生成器模式的定义：将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

模式讲解

生成器模式的重心在于分离构建算法和具体的构造实现，从而使得构建算法可以重用。

不管如何变化，Builder模式都存在这么两个部分，一部分是部件构造和产品装配，另一个部分是整体构建的算法。这个模式强调的是固定整体构建的算法，而灵活扩展和切换部件的具体构造和产品装配的方式，所以要严格区分这两个部分。

生成器模式的使用

可以让客户端创造Director，在Director里面封装整体构建算法，然后让Director去调用Builder，让Builder来封装具体部件的构建功能。

还有一种退化的情况，就是让客户端和Director融合起来，让客户端直接去操作Builder，就好像是指导者自己想要给自己构建产品一样。

生成器模式的实现

关于被构建的产品的接口

在使用生成器模式的时候，大多数情况下是不知道最终构建出来的产品是什么样的，所以在标准的生成器模式里面，一般是不需要对产品定义抽象接口的，因为最终构造的产品千差万别，给这些产品定义公共接口几乎是没有意义的。

生成器模式的优点

- 松散耦合
- 可以很容易地改变产品的内部表示
- 更好的复用性

思考生成器模式

生成器模式的本质

分离整体构建算法和部件构造。

构建一个复杂的对象，本来就有构建的过程，以及构建过程中具体的实现。

生成器模式就是用来分离这两个部分，从而使得程序结构更松散，扩展更容易，复用性更好，同样也会使得代码更清晰，意图更明确。

生成器模式的重心还是在于分离整体构建算法和部件构造，而分步骤构建对象不过是整体构建算法的一个简单表现，或者说是一个附带产物。

何时选用生成器模式

- 如果创建对象的算法，应该独立于该对象的组成部分以及它们的装配方式时。
- 如果同一个构建过程有着不同的表示时。

第9章 原型模式

实际例子举的是订单处理系统。订单数量的拆分。

已经有了某个对象实例后，如何能够快速简单地创建处更多的这种对象

原型模式的定义

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

原型模式会要求对象实现一个可以“克隆”自身的接口，这样就可以通过拷贝或克隆一个实例对象本身来创建一个新的实例。

模式讲解

- 原型模式的功能

  原型模式的功能实际上包含两个方面：

  1.一个是通过克隆来创建新的对象实例；

  2.另一个是为克隆出来的新的对象实例复制原型实例属性的值；

- 原型与new
- 原型实例和克隆的实例

原型管理器

如果一个系统中原型的数目不固定，比如系统中的原型可以被动态地创建和销毁，那么就需要在系统中维护一个当前可用的原型的注册表，这个注册表就叫做原型管理器。

原型模式的优缺点

原型模式的优点

- 对客户端隐藏具体的实现类型
- 在运行时动态改变具体的实现类型

原型模式的缺点

每个原型的子类都必须实现clone操作，尤其在包含引用类型的对象时，clone方法会比较麻烦，必须要能够递归的让所有相关对象都要正确地实现克隆。

思考原型模式

原型模式的本质：克隆生成对象

原型模式创建出来的对象，其属性的值是否一定要和原型对象属性的值完全一样，这个并没有强制规定，可以通过克隆来创造值不一样的实例，但是类型必须一样。

何时选用原型模式

- 如果一个系统想要独立于它想要使用的对象时，可以使用原型模式。
- 如果需要实例化的类是在运行时刻动态指定的，可以使用原型模式，通过克隆原型来得到需要的实例。

第10章 中介者模式

中介者模式的定义

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

模式讲解

中介者的功能非常简单，就是封装对象之间的交互。如果一个对象的操作会引起其他相关对象的变化，或者是某个操作需要引起其他对象的后续或者连带操作，而这个对象又不希望自己来处理这些关系，那么就可以找中介者，把所有的麻烦都扔给它，只在需要的时候通知中介者，其他的就让中介者去处理就可以了。

在标准的中介者模式中，将使用中介者对象来交互的那些对象称为同事类。

在中介者模式中，要求这些类都要继承相同的类。也就是说，这些对象从某个角度讲是同一个类型，算是兄弟对象。

同事与中介者的关系：当一个同事对象发生了改变，需要主动通知中介者，让中介者去处理与其他同事对象相关的交互。

如何实现同事和中介者的通信

我自己首先想到的是观察者模式，正如书中所写作者说的第二种方案也就是观察这模式，把Mediator实现成为观察者，而各个同事类实现成为subject，这样同事类发生了改变，会通知Mediator。

作者所说的第一种实现方式是在Mediator接口中定义一个特殊的通知接口，作为一个通用的方法，让各个同事类来调用这个方法。

广义中介者

实际使用中介者模式的时候，会发现几个问题变得繁琐或困难。

one：是否有必要为同事对象定义一个公共的父类？因为Java是单继承的，继承它得不到多少好处。

two：同事类有必要持有中介者对象吗？

three：是否需要中介者接口？实际的开发中，很常见的情况是不需要中介者接口的，而且中介者也不需要创建很多个实例。因为中介者是用来封装和处理同事对象的关系的，它一般是没有状态需要维护的，因此中介者通常可以实现成单例。

four：中介者对象是否需要持有所有的同事？

基于上面的考虑，在实际应用开发中，经常会简化中介者模式，来使开发变得简单，比如有如下的简化：

- 通常会去掉同事对象的父类，这样可以让任意的对象，只要需要相互交互，就可以成为同事
- 通常不定义Mediator接口，把具体的中介者对象实现成为单例。
- 同事对象不再持有中介者，而是在需要的时候直接获取中介者对象并调用；中介者也不再持有同事对象，而是在具体处理方法里面去创建，或者获取，或者从参数传入需要的同事对象。

把这样经过简化变形使用的情况称为广义中介者。

中介者模式的优缺点

中介者模式的优点

- 松散耦合
- 集中控制交互
- 多对多变成一对多

中介者模式的缺点

过度集中化，会导致中介者对象变得很复杂。

中介者模式的本质：封装交互

何时选用中介者模式

- 如果一组对象之间的通信方式比较复杂，导致相互依赖，结构混乱，可以采用此模式，把这些对象的相互的交互管理起来，各个对象都只需要和中介者交互。
- 如果一个对象引用很多的对象，并直接跟这些对象交互，导致难以复用该对象，可以采用此模式。

第11章 代理模式

代理模式的定义

为其他对象提供一种代理以控制对这个对象的访问。

代理的分类

- 虚代理 根据需要来创建开销很大的对象，该对象只有在需要的时候才会被真正创建。
- 远程代理 用来在不同的地址空间上代表同一个对象，这个不同的地址空间可以是在本机，也可以在其他机器上。在Java里最典型的就是RMI技术。
- copy-on-write代理 在客户端操作的时候，只有对象确实改变了，才会真的拷贝一个目标对象，算是虚代理的一个分之。
- 保护代理 控制对原始对象的访问，如果有需要，可以给不同的用户提供不同的访问权限，以控制他们对原始对象的访问。
- cache代理 为那些昂贵的操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
- 防火墙代理 保护对象不被恶意用户访问和操作。
- 同步代理 使多个用户能够同时访问目标对象而没有冲突。
- 智能指引 在访问对象时执行一些附加操作。

Java中的代理

Java提供了一个Proxy类和一个InVocationHandler接口。

代理模式的特点

客户直接使用代理，让代理来与被访问的对象进行交互。不同的代理类型，有不同的特点。

- 远程代理 隐藏了一个对象存在于不同的地址空间的事实，也即是客户通过远程代理去访问一个对象，根本就不关心这个对象在哪里，也不关心如何通过网络去访问到这个对象。从客户的角度来讲，它只是在使用代理对象而已。
- 虚代理 可以根据需要来创建“大”对象，只有到必须创建对象的时候，虚代理才会创建对象，从而大大加快程序运行速度，并节省资源。通过虚代理可以对系统进行优化。
- 保护代理 可以在访问一个对象的前后，执行很多附加的操作，除了进行权限控制之外还可以进行很多跟业务相关的处理，而不需要修改被代理的对象。也就是说，可以通过代理来给目标对象增加功能。
- 智能指引 和保护代理类似，也是允许在访问一个对象的前后，执行很多附加的操作，这样以来就可以做很多额外的事情，比如引用计数。

思考代理模式

代理模式的本质：控制访问对象

从实现上看，代理模式主要是使用对象的组合和委托。但是也可以采用对象继承的方式来实现代理，这种实现方法在某些情况下，比使用对象组合还要来的简单。

何时选用代理模式

- 需要为一个对象在不同的地址空间提供局部代表的时候，可以使用远程代理。
- 需要按照需要创建开销很大的对象的时候，可以使用虚代理。
- 需要控制对原始对象的访问的时候，可以使用保护代理。
- 需要在访问对象执行一些附加操作的时候，可以使用智能指引代理。

第12章 观察者模式

观察者模式的定义

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

现实例子是订阅报纸。

被观察的目标

报社对象是被观察的目标。

用观察者模式实现如下：

```
public interface Observer {
    public void update(Subject subject);
}


public class Subject {

    private List<Observer> readers = new ArrayList<>();

    public void attach(Observer reader) {
        readers.add(reader);
```

```
        readers.add(reader);
    }

    public void detach(Observer reader) {
        readers.remove(reader);
    }

    protected void notifyObservers() {
        for (Observer observer:readers) {
            observer.update(this);
        }
    }
}


public class NewsPaper extends Subject {

    private String content;

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
        notifyObservers();
    }
}

public class Reader implements Observer {

    private String name;

    @Override
    public void update(Subject subject) {
        System.out.println(name + " has received newspaper and content
is " + ((NewsPaper)subject).getContent());
    }

    public String getName() {
        return name;
    }

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

```
    }
}

public class Client {

    public static void main(String[] args) {
        NewsPaper newsPaper = new NewsPaper();
        Reader reader = new Reader();
        reader.setName("Smith");
        Reader copy = new Reader();
        copy.setName("Larry");

        newsPaper.attach(reader);
        newsPaper.attach(copy);

        newsPaper.setContent("this is washington newspaper and liberty
will be win all the time");
    }
}
```

运行结果：

```
Smith has received newspaper and content is this is washington
newspaper and liberty will be win all the time
Larry has received newspaper and content is this is washington
newspaper and liberty will be win all the time
```

认识观察者模式

1.目标和观察者之间的关系

按照模式的定义，目标和观察者之间是典型的一对多的关系。

但是要注意，如果观察者只有一个，也是可以的，这样就变相实现了目标和观察者之间一对一的关系，这也使得在处理一个对象的状态便会会影响到另一个对象的时候，也可以考虑使用观察者模式。

同样地，一个观察者也可以观察多个目标，如果观察者为多个目标定义的通知更新方法都是update方法的话，这会带来麻烦，因为需要接受多个目标的通知，如果是一个update的方法，那就需要在方法内部区分，到底这个更新的通知来自于哪一个目标，不同的目标有不同的后续操作。

一般情况下，观察者应该为不同的观察者目标定义不同的回调方法，这样实现最简单，不需要在update方法内部进行区分。

2.单向依赖

在观察者模式中，观察者和目标是单向依赖的，只有观察者依赖于目标，而目标是不会依赖于观察者的。

它们之间的主动权掌握在目标手中，只有目标知道什么时候需要通知观察者。在整个过程中，观察者始终是被动的，被动地等待目标的通知，等待目标传值给它。

对目标而言，所有的观察者都是一样的，目标会一视同仁的对待。

3.基本的实现说明

- 具体的目标实现对象要能维护观察者的注册信息，最简单的实现方案就如同前面的例子那样，采用一个集合来保存观察者的注册信息。
- 具体的目标实现对象需要维护引起通知的状态，一般情况下是目标自身的状态。变形使用的情况下，也可以是别的对象的状态。
- 具体的观察者实现对象需要能接受目标的通知，能接受目标传递的数据，或者是能够主动去获取目标的数据，并进行后续处理。
- 如果是一个观察者观察多个目标，那么在观察者的更新方法里面，需要去判断是来自哪一个目标的通知。一种简单的解决方案是扩展update方法，比如在方法里面多传一个参数进行区分等；还有一种更简单的方法，那就是干脆定义不同的回调方法。

4.命名建议

- 观察者模式又被称为发布----订阅模式。
- 目标接口的定义，建议在名称后面跟subject
- 观察者接口的定义，建议在名称后面跟Observer
- 观察者接口的更新方法，建议名称为update，当然方法的参数可以根据需要定义，参数个数不限，参数类型不限。

5.触发通知的时机

在实现观察者模式的时候，一定要注意触发通知的时机。一般情况下，是在完成了状态维护后触发，因为通知会传递数据，不能够先通知后改数据，这很容易出问题，会导致观察者和目标对象的状态不一致。

6.相互观察

在某些应用中，可能会出现目标和观察者相互观察的情况。要特别注意不要出现死循环。

7.通知的顺序

从理论上来说，当目标对象的状态变化后通知所有的观察者的时候，顺序是不确定的。因此观察者实现的功能，绝对不要依赖于通知的顺序。也就是说，多个观察者之间的功能是平行的，相互不应该有先后的依赖关系。

推模型和拉模型

在观察者模式的实现中，又分为推模型和拉模型两种方式。

推模型

目标对象主动向观察者推送目标的详细信息，不管观察者是否需要，推送的信息通常是目标对象的全部或者部分数据，相当于在广播通信。

拉模型

目标对象在通知观察者的时候，只传递少量信息，如果观察者需要更具体的信息，由观察者主动到目标对象中获取，相当于是观察者从目标对象中拉取数据。

一般这种模式的实现中，会把目标对象自身通过update方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

两种实现模型在开发的时候，究竟应该使用哪一种还应该具体问题具体分析，比较如下：

- 推模型是假定目标对象知道观察者需要的数据；而拉模型是目标对象不知道观察者具体需要什么数据，没有办法的情况下，干脆把自身传给观察者，让观察者自己去按需取值。
- 推模型可能会使得观察者对象难以复用，因为观察者定义的update方法是按需而定义的，可能无法兼顾没有考虑到的使用情况。这就意味着出现新情况的时候，就可能需要提供新的update方法，或者是干脆重新实现观察者。

观察者模式的优缺点

观察者模式有以下优点

- 实现了观察者和目标之间的抽象耦合
- 实现了动态联动
- 支持广播通信

观察者模式的缺点

- 可能会引起无谓的操作

思考观察者模式

观察者模式的本质：触发联动

何时选用观察者模式

- 当一个抽象模型有两个方面，其中一个方面的操作依赖于另一个方面的状态变化，那么就可以选用观察者模式，将这两者封装成观察者和目标对象，当目标对象变化的时候，依赖于它的观察者对象也会发生相应的变化，这样就把抽象模型的这两个方面分开了，使得它们可以独立地改变和复用。
- 如果在更改一个对象的时候，需要连带改变其他的对象，而且不知道究竟应该有多少对象需要被连带改变，这种情况可以选用观察者模式，被更改的那一个对象很明显就相当于是目标对象，而需要连带修改的多个其他对象，就作为多个观察者对象了。
- 当一个对象必须通知其他对象，但是你又希望这个对象和其他被它通知的对象是松散耦合的。也就是说这个对象其实不想知道具体被通知的对象。这种情况可以选用观察者模式，这个对象就相当于是目标对象，而被它通知的对象就是观察者对象了。

第13章 命令模式

对象只管发出命令，谁接收命令，谁实现命令，对象不关心。

命令模式的定义

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。

模式讲解

1.命令模式的关键

就是把请求封装成对象，也就是命令对象，并定义统一的执行操作的接口。

2.命令模式的组装和调试

该模式中经常会有一个命令的组装者，用它来维护命令的“虚”实现和真实实现之间的关系。如果是超级智能的命令，也就是说命令对象自己完全实现好了，不需要接收者，那就是命令模式的退化，不需要接收者，自然也不需要组装者了。

而真正的用户就是具体化请求的内容，然后提交请求进行触发就可以了。真正的用户会通过Invoker来触发命令。

在实际开发过程中，Client和Invoker可以融合在一起，由客户在使用命令模式的时候，先进行命令对象和接收者的组装，组装完成后，就可以调用命令执行请求。

3.命令模式的接收者

接收者可以是任意的类，对它没有什么特殊要求，这个对象知道如何真正的执行命令的操作，执行时从Command的实现类里面转调过来。

一个接收者对象可以处理多个命令，接收者和命令之间没有约定的对应关系。接收者提供的方法个数，名称，功能和命令中的可以不一样，只要能通过调用接收者的方法来实现命令对应的功能就可以了。

4.智能命令

标准的命令模式里，命令的实现类是没有真正实现命令要求的功能的，真正执行命令的功能的是接收者。

如果命令的实现对象比较智能，它自己就能真正地实现命令要求的功能，而不需要调用接收者，这就是智能命令。

也可以用半智能的命令，命令对象知道部分实现，其他的还是需要调用接收者来完成，也就是说命令的功能由命令对象和接收者共同完成。

5.发起请求的对象和真正实现的对象是解耦的

请求究竟由谁处理，如何处理，发起请求的对象是不知道的，也就是发起请求的对象和真正实现的对象是解耦的，发起请求的对象只管发出命令，其他的就不管了。

参数化配置

所谓命令模式的参数化配置，指的是：可以用不同的命令对象，去参数化配置客户的请求。

可撤销的操作

可撤销的操作意思就是：放弃该操作，回到未执行该操作前的状态。

有两种基本的思路来实现可撤销的操作，**一种是补偿式，又称反操作式**

另一种方式是**存储恢复式**，意思就是把操作之前的状态记录下来，然后要撤销操作的时候就直接恢复回去就可以了。

命令模式的优点

- 更松散的耦合
- 更动态的控制
- 很自然的复合命令
- 更好的扩展性

思考命令模式

1.命令模式的本质：封装请求

2.何时选用命令模式

- 如果需要抽象出需要执行的动作，并参数化这些对象，可以选用命令模式。将这些需要执行的动作抽象成为命令，然后实现命令的参数化配置。
- 如果需要在不同的时刻指定，排列和执行请求，可以选用命令模式。将这些请求封装成为命令对象，然后实现将请求队列化。
- 如果需要支持取消操作，可以选用命令模式，通过管理命令对象，能很容易地实现命令的恢复重做功能。
- 如果需要支持当系统崩溃时，能将系统的操作功能重新执行一遍，可以选用命令模式。将这些操作功能的请求封装成命令对象，然后实现日志命令，就可以在系统恢复后，通过日志获取命令对象列表，从而重新执行一遍功能。
- 在需要事务的系统中，可以选用命令模式。命令模式提供了对事务进行建模的方法。命令模式有一个别名就是Transaction。

第14章 迭代器模式

如何能以一个统一的方式来访问内部实现不同的聚合对象

使用迭代器模式：提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

所谓聚合是指一组对象的组合结构，比如Java中的集合，数组等。

模式讲解

1.迭代器模式的功能

主要在于提供对聚合对象的迭代访问，迭代器就围绕着这个访问做文章。

2.迭代器模式的关键思想

就是把对聚合对象的遍历和访问从聚合对象中分离出来，放入单独的迭代器中，这样聚合对象会变得简单一些。而且迭代器和聚合对象可以独立地变化和发展，会大大加强系统的灵活性。

迭代器模式的优点

- 更好的封装性
- 迭代器模式可以让你访问一个聚合对象的内容，而无须暴露该聚合对象的内部表示，从而提高聚合对象的封装性。
- 可以以不同的遍历方式来遍历一个聚合。
- 使用迭代器模式使得聚合对象的内容和具体的迭代算法分开。这样就可以通过使用不同的迭代器的实例，不同的遍历方式来遍历一个聚合对象了。
- 迭代器简化了聚合的接口。
- 有了迭代器的接口，则聚合本身就不需要再定义这些接口了，从而简化了聚合的接口定义。
- 简化客户端调用。
- 迭代器为遍历不同的聚合对象提供了一个统一的接口，使得客户端遍历聚合对象的内容变得更简单。
- 同一个聚合上可以有多个遍历。
- 每个迭代器保持它自己的遍历状态。

思考迭代器模式

迭代器模式的本质：控制访问聚合对象中的元素。

何时选用迭代器模式

- 如果你希望提供访问一个聚合对象的内容，但是又不想暴露它的内部表示的时候，可以使用迭代器模式来提供迭代器接口，从而让客户端只是通过迭代器的接口来访问聚合对象，而无需关心聚合对象的内部实现。
- 如果你希望有多种遍历方式可以访问聚合对象，可以使用迭代器模式。
- 如果你希望为遍历不同的聚合对象提供一个统一的接口，可以使用迭代器模式。

第15章 组合模式

模式讲解

1.组合模式的目的

让客户端不再区分操作的是组合对象还是叶子对象，而是以一个统一的方式来操作。

实现这个目标的关键之处，是设计一个抽象的组件类，让它可以代表组合对象和叶子对象。这样一来，客户端就不用区分到底操作的是组合对象还是叶子对象了，只需要把它们全部当作组件对象进行统一的操作就可以了。

2.对象树

通常，组合模式会组合出树形结构来，组成这个树形结构所使用的多个组件对象，就自然地形成了对象树。

3.组合模式中的递归

4.Component中是否应该实现一个Component列表

不太好，因为在父类来存放子类的实例对象中，对于composite节点来说没有什么，它本来就需要存放子节点；但是对于叶子节点来说，就会导致空间的浪费，因为叶节点本身不需要子节点。

5.最大化Component定义

类设计有这样的原则：一个父类应该只定义那些对它的子类有意义的操作。常见的做法是对于子类的某些实现提供默认的实现，比如返回不支持该操作，如果子类对象需要这个功能，那就覆盖实现它，如果不需要，使用父类的默认实现就可以了。

安全性和透明性

一个很重要的问题：在组合模式的类层次结构中，到底在哪一些类里面定义这些管理子组件的操作，是应该在Component中声明这些操作呢，还是在Composite中声明这些操作?

这就需要仔细思考，在不同的实现中，进行安全性和透明性的权衡选择。

- 这里所说的安全性是指：从客户使用组合模式上看是否更安全。如果是安全的，那么就不会有发生误操作的可能，能访问的方法都是被支持的功能。
- 这里所说的透明性是指：从客户使用组合模式上，是否需要区分到底是组合对象还是叶子对象。如果是透明的，那就不再区分，对于客户而言，都是组件对象，具体的类型对于客户而言是透明的，是客户无须关心的。

1.透明性的实现

如果把管理子类组件的操作定义在Component中，那么客户端只需要面对Component，而无须关心具体的组件类型，这种实现方式就是透明性的实现。

但是透明性的实现是以安全为代价的，因为在Component中定义的一些方法，对于叶子对象来说是没有意义的，比如增加删除子组件对象。而客户端不知道这些区别，对客户是透明的，这样客户端会误操作，很不安全。

组合模式的透明性实现，通常的方式是，在Component中声明管理子组件的操作，并在Component中为这些方法提供默认的实现，如果子对象不支持的功能，默认的实现可以是抛出一个异常，来表示不支持这个功能。

2.安全性的实现

如果把管理子组件的操作定义在Composite中，那么客户在使用叶子对象的时候，就不会发生使用添加子组件或是删除子组件的操作了，因为压根就没有这样的功能，这种实现方式是安全的。

但是这样一来，客户端在使用的时候，就必须区分到底使用的是Composite对象，还是叶子对象，不同对象的功能是不一样的。也就是说，这种实现方式，对客户端而言就不是透明的了。

3.两种实现方式的选择

对于组合模式而言，在安全性和透明性上，会更看重透明性，毕竟组合模式的功能就是要让用户对叶子对象和组合对象的使用具有一致性。

而且对于安全性的实现，需要区分是组合对象还是叶子对象。有的时候，需要将对象进行类型转换，却发现类型信息丢失了，只好强行转换，这种类型转换必然是不够安全的。

对于这种情况的处理方法是在Component中定义一个getComposite方法，用来判断是组合对象还是叶子对象，如果是组合对象，就返回组合对象，如果是叶子对象，就返回null，这样就实现了先判断，然后再强制转换。

因此在使用组合模式的时候，建议多采用透明性的实现方式，而少用安全性的实现方式。

环状引用

A包含B，B包含C，C又包含A。

组合模式不应该出现环状引用。当然可以有环状引用，但是需要特殊构建环状引用，并提供相应的检测和处理。

如何检测是否有环状引用?

思路：记录下每个组件从根节点路径开始，只要出现环状引用在一条路径上，某个对象就必然出现两次。

组合模式的优缺点

组合模式有以下优点

- 定义了包含基本对象和组合对象的类层次结构
- 统一了组合对象和叶子对象
- 简化了客户端调用
- 更容易扩展 **组合模式缺点是很难限制组合中的组件类型**

思考组合模式

组合模式的本质：统一叶子对象和组合对象。

何时选用组合模式

- [ ] 如果你想表示对象的部分一整体层次结构，可以选用组合模式，把整体和部分的操作统一起来，使得层次结构实现更简单，从外部来使用这个层次结构也容易。
- [ ] 如果你希望统一地使用组合结构中的所有对象，可以选用组合模式，这正是组合模式提供的主要功能。

第16章 模版方法模式

1.模版方法模式的定义

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模版方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

模式讲解

1.模版方法模式的功能

该功能在于固定算法骨架，而让具体算法实现可扩展。

模版方法模式还额外提供了一个好处，就是可以控制子类的扩展。因为在父类中定义好了算法的步骤，只是在某几个固定的点才会调用到被子类实现的方法，因此也就只允许在这几个点来扩展功能。这些可以被子类覆盖以扩展功能的方法通常被称为“钩子”方法。

2.为何不是接口

抽象类不一定包含抽象方法；有抽象方法的类一定是抽象类。

使用抽象类的场景是：**既要约束子类的行为，又要为子类提供公共功能的时候使用抽象类。**

3.变与不变

程序设计的一个很重要的思考点就是“变与不变”，分析程序中哪些功能是可变的，哪些功能是不变的，然后把不变的部分抽象出来，进行公共的实现，把变化的分离出去，用接口来封装隔离，或者是用抽象类来约束子类行为。

4.好莱坞法则：HeadFirst设计模式里有[不要找我们，我们会联系你]

模版方法模式很好的体现了这一点，作为父类的模板会在需要的时候，调用子类相应的方法，也就是由父类来找子类，而不是让子类来找父类。

这其实也是一种反向的控制结构，按照通常的思路本来是应该子类来寻找父类才对，模版方法模式是反过来了所以说是反向控制结构。

Java中能实现这样功能的理论依据在哪里呢?

就在于Java提供了动态绑定采用的是“后期绑定”技术，对于出现子类覆盖父类方法的情况，在编译时是看数据类型，运行时则看实际的对象类型(new 操作符后跟的构造方法是哪个类的)。**一句话：new 谁就调用谁的方法。**

模板里边应该包含的操作：

- 模板方法：定义算法骨架的方法。
- 具体的操作：在模板中直接实现某些步骤的方法。
- 具体的AbstractClass操作：在模板中实现某些公共功能，可以提供给子类使用，一般不是具体的算法步骤的实现，而是一些辅助的公共功能。
- 原语操作：就是在模板中定义的抽象操作，通常是模板方法需要调用的操作，是必须的操作，而且在父类中还没有办法确定下来如何实现，需要子类来真正实现的方法。
- 钩子操作：在模板中定义，并提供默认实现的操作。钩子操作是可以被扩展的点，但不是必须的。
- FactoryMethod：在模板方法中，如果需要得到某些对象实例的话，可以考虑通过工厂方法模式来获取，把具体的构建对象的实现延迟到子类中去。

Java回调与模板方法模式

模板方法模式的一个目的，就在于让其他类来扩展或具体实现在模板中固定的算法骨架中的某些算法步骤。

在标准的模板方法模式实现中，主要是使用继承的方式，来让父类在运行期间可以调用到子类的方法。

模板方法模式的优缺点

- 模板方法模式的优点是实现代码复用
- 模板方法模式的缺点是算法骨架不容易升级

思考模版方法模式

1.模板方法模式的本质：固定算法骨架。

2.对设计原则的体现

很好地体现了开闭原则和里氏替换原则。

何时选用模板方法模式

- 需要固定定义算法骨架，实现一个算法的不变的部分，并把可变的行为留给子类来实现的情况。
- 各个子类中具有公共行为，应该抽取出来，集中在一个公共类中去实现，从而避免带代码重复。
- 需要控制子类扩展的情况。模板方法模式会在特定的点来调用子类的方法，这样只允许在这些点进行扩展。

第17章 策略模式

使用策略模式来解决问题

1.策略模式的定义：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

模式讲解

1.策略模式的功能

把具体的算法实现从具体的业务处理中独立出来，把它们实现成单独的算法类，从而形成一系列的算法，并让这些算法可以相互替换。

2.策略模式和if else语句

策略模式就是把各个平等的具体实现封装到单独的策略实现类了，然后通过上下文来与具体的策略类进行交互。

3.算法的平等性

对于一系列具体的策略算法，大家的地位是完全一样的，这些策略算法在实现上也是相互独立的，相互之间是没有依赖的。

策略算法是相同行为的不同实现。

4.谁来选择具体的策略算法

可以在两个地方来进行具体的策略选择。

一是客户端，当使用上下文的时候，由客户端来选择具体的策略算法，然后把这个策略算法设置给上下文。

一是由上下文来选择具体的策略算法。

5.strategy的实现方式

如果多个算法具有公共功能的话，可以把strategy实现成为抽象类，然后把多个算法的公共功能实现到strategy中。

6.运行时策略的唯一性

运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象。

7.增加新的策略

写一个新的类实现strategy接口，然后客户端直接使用。

Context和Strategy的关系

在策略模式中，通常是上下文使用具体的策略实现对象。反过来，策略实现对象也可以从上下文中获取所需要的数据。 因此可以将上下文当作参数传递给策略实现对象，这种情况下上下文和策略实现对象是紧密耦合的。 在这种情况下，上下文封装着具体策略对象进行算法运算所需要的数据，具体策略对象通过回调上下文的方法来获取这些数据。

策略模式的优缺点

优点

- 定义一系列算法：策略模式的功能就是定义一系列算法，实现让这些算法可以相互替换。
- 避免多重条件语句
- 更好的扩展性

缺点

- 客户必须了解每种策略的不同
- 增加了对象数目
- 只适合扁平的算法结构

策略模式本质：分离算法，选择实现。

何时选用策略模式

- 出现许多相关的类，仅仅是行为有差别的情况下，可以使用策略模式来使用多个行为中的一个来配置一个类的方法，实现算法动态切换。
- 出现同一个算法，有很多不同实现的情况下，可以使用策略模式来把这些“不同的实现”实现成为一个算法的类层次。
- 需要封装算法中，有与算法相关数据的情况下，可以使用策略模式来避免暴露这些跟算法相关的数据结构。
- 出现抽象一个定义了很多行为的类，并且是通过多个ifelse语句来选择这些行为的情况下，可以使用该模式来替换这些条件语句。

第18章 状态模式

状态模式的定义

允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

模式讲解

1.状态和行为

所谓对象的状态，通常指的是对象实例的属性的值；而行为指的就是对象的功能，具体点就是行为大多可以对应到方法上。

状态模式的功能就是分离状态的行为，通过维护状态的变化，来调用不同状态对应的不同功能。状态相互之间是不可替换的。

状态决定行为

2.行为的平行性

平行性是指各个状态的行为所处的层次是一样的，相互是独立的，没有关联的。

3.上下文和状态处理对象

该模式中，上下文是持有状态的对象，但是上下文自身并不处理跟状态相关的行为，而是把处理状态的功能委托给了状态对应的状态处理类来处理。

4.创建和销毁状态对象

何时创建和销毁状态对象：

- 当需要使用状态对象的时候创建，使用完后就销毁它们。
- 提前创建它们并且始终不销毁。
- 采用延迟加载和缓存合用的方式。

状态的维护和转换控制

通常有两个地方可以进行状态的维护和转换控制。

一个是在上下文中

另外一个就是在状态的处理类中

如何选择这两种方式呢?

- 如果状态转换的规则是一定的，一般不需要进行什么扩展规则，那么就适合在上下文中统一进行状态的维护。
- 如果状态的转换取决于前一个状态动态处理的结果，或者是依赖于外部数据，为了增强灵活性，这种情况下，一般是在状态处理类中进行状态的维护。

使用数据库来维护状态

状态模式的优缺点

优点

- 简化应用逻辑控制
- 更好地分离状态和行为
- 更好的扩展性
- 显式化进行状态转换

缺点

会使得程序引入过多的状态类，程序会变得类比较多。

思考状态模式

状态模式的本质：根据状态来分离和选择行为。

何时选用状态模式

- 如果一个对象的行为取决于它的状态，而且它必须在运行时刻根据状态来改变它的行为，可以使用状态模式，来把状态和行为分离开。
- 如过一个操作中含有庞大的多分支语句，而且这些分支依赖于该对象的状态，可以使用状态模式，把各分支的处理分散包装到单独的对象处理类中。

第19章 备忘录模式

备忘录模式定义：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

一个备忘录是一个对象，它存储另一个对象在某个瞬间的内部状态，后者被称为备忘录的原发器。

如何能够在不破坏对象的封装性的前提下，来保存和恢复对象的状态。

备忘录模式引入一个存储状态的备忘录对象，为了让外部无法访问这个对象的值，一般把这个对象实现称为需要保存数据的对象的内部类，通常还是私有的。

但是这个备忘录对象需要存储在外部。为了避免让外部访问到这个对象内部的数据，备忘录模式引入一个备忘录对象的窄接口，这个接口一般是空的，什么方法都没有，这样外部存储的地方，只是知道存储了一些备忘录接口的对象，但是由于接口是空的，它们无法通过接口去访问备忘录对象内部的数据。

模式讲解

1.备忘录模式的功能

首先在不破坏封装性的前提下，捕获一个对象的内部状态。

捕获这个内部状态的作用：**是为了在以后的某个时候，将该对象的状态恢复到备忘录所保存的状态。**

捕获的状态放在哪里?

存储在备忘录对象中；而备忘录对象通常会被存储在原发器对象之外，也就是被保存状态的对象的外部，通常是存放在管理者对象那里。

2.备忘录对象

该模式中，备忘录对象通常就是用来记录原发器需要保存的状态的对象，简单点的实现就是封装数据的对象。

备忘录对象一般只让原发器对象来操作，而不是普通的封装数据的对象那样，谁都可以操作。为了保证这一点，通常会把备忘录对象作为原发器对象的内部类来实现，而且实现成私有的，这就断绝了外部来访问这个备忘录对象的途径。

备忘录对象需要保存在原发器对象之外，为了与外部交互，通常备忘录对象都会实现一个窄接口，来标识别对象的类型。

3.原发器对象

就是需要被保存状态的对象，也是有可能需要恢复状态的对象。原发器一般会包含备忘录对象的实现。

通常原发器对象应该提供捕获某个时刻对象内部状态的方法，在这个方法中，原发器对象会创建备忘录对象，把需要保存的状态数据设置到备忘录对象中，然后把备忘录对象提供给管理者对象来保存。

原发器对象也应该提供这样的方法：按照外部要求来恢复内部状态到某个备忘录对象记录的状态。

4.管理者对象

主要是负责保存备忘录对象，注意几点：

- 并不一定要特别的做出一个管理者对象来。
- 管理者对象并不是只能管理一个备忘录对象，一个管理者对象可以管理很多的备忘录对象。
- 狭义的管理者对象是只管理同一类的备忘录对象，但广义的管理者对象是可以管理不同类型的备忘录对象的。
- 管理者对象需要实现的基本功能主要是：存入备忘录对象，保存备忘录对象和获取备忘录对象。
- 管理者虽然能存取备忘录对象，但是不能访问备忘录对象内部的数据。

5.窄接口和宽接口

- 窄接口：管理者只能看到备忘录的窄接口，窄接口的实现中通常没有任何的方法，只是一个类型标识。窄接口使得管理者只能将备忘录传递给其他对象。
- 宽接口：原发器能够看到一个宽接口，允许它访问所需要的所有数据，来返回到先前的状态。理想的状况是：只允许生成备忘录的原发器来访问该备忘录的内部状态，通常实现成为原发器内的一个私有内部类。

备忘录模式的优缺点

优点

- 更好的封装行
- 简化了原发器
- 窄接口和宽接口

缺点是可能导致高开销。

备忘录模式的本质：保存和恢复内部状态

何时选用备忘录模式

- 如果必须保存一个对象在某一时刻的全部或者部分状态，方便在以后需要的时候，可以把该对象恢复到先前的状态，可以使用备忘录模式。
- 如果需要保存一个对象的内部状态，但是如果用接口来让其他对象直接得到这些需要保存的状态，将会暴露对象的实现细节并破坏对象的封装行，这时可以使用备忘录模式，把备忘录对象实现成为原发器对象的内部类，而且还是私有的，从而保证只有原发器对象才能访问该备忘录对象。

第20章 享元模式

在系统当中，存在大量的细粒度对象，而且存在大量的重复数据，严重耗费内存，如何解决呢?

使用享元模式来解决问题

运用共享技术有效地支持大量细粒度的对象。

模式讲解

1.变与不变

把一个对象的状态分成内部状态和外部状态，内部状态是不变的，外部状态是可变的。

然后通过共享不变的部分，达到减少对象数量并节约内存的目的。

2.共享与不共享

享元对象又有共享与不共享之分，这种情况通常出现在和组合模式合用的情况，通常共享的是叶子对象，一般不共享的部分是由共享部分组合而成的，由于所有细粒度的叶子对象都已经缓存了，那么缓存组合对象就没有什么意义了。

3.内部状态和外部状态

享元模式的内部状态通常指的是包含在享元对象内部的，对象本身的状态，是独立于使用享元的场景的信息，一般创建后就不再变化的状态，因此可以共享。

外部状态指的是享元对象之外的状态，取决于使用享元的场景，会根据使用场景而变化，因此不可共享。

享元模式真正缓存和共享的数据是享元的内部状态，而外部状态是不应该被缓存共享的。

内部状态和外部状态是独立的，外部状态的变化不应该影响到内部状态。

4.实例池

为了创建和管理共享的享元部分，引入了享元工厂。

所谓实例池，指的是缓存和管理对象实例的程序，通常实例池会提供对象实例的运行环境，并控制对象实例的生命周期。

何时选用享元模式

- 如果一个应用程序使用了大量的细粒度对象，可以使用享元模式来减少对象数量。
- 如果由于使用大量的对象，造成很大的存储开销，可以使用享元模式来减少对象数量并节约内存。
- 如果对象的大多数状态都可以转变为外部状态，可以使用享元模式来实现内部状态和外部状态的分离。
- 如果不考虑对象的外部状态，可以用相对较少的共享对象取代很多组合对象，可以使用享元模式来共享对象，然后组合对象来使用这些共享对象。

第21章 解释器模式

如何能够灵活的读取配置文件的内容?

解释器模式的定义：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。

模式讲解

语法规则和解释器

一般一个解释器处理一条语法规则，但是反过来并不成立，一条语法规则是恶意有多种解释和处理的，也就是一条语法规则可以对应多个解释器对象。

上下文的公用性

上下文在解释器中起着非常重要的作用。上下文会被传递到所有的解释器中，因此可以在上下文中存储和访问解释器的状态，还可以通过上下文传递一些在解释器外部但是解释器需要的数据，也可以是一些全局的，公共的数据。

上下文还有一个功能就是可以提供所有解释器对象的公共功能，类似于对象组合，而不是使用继承来获取公共功能，在每个解释器对象中都可以调用。

解释器模式的优缺点

优点

- 易于实现语法
- 易于扩展新的语法

缺点

不适合复杂的语法。如果语法特别复杂，构建解释器模式需要的抽象语法树的工作是非常艰巨的，再加上有可能需要构建多个抽象语法树。对于复杂的语法，使用语法分析程序或编译生成器可能会更好一些。

思考解释器模式

解释器模式的本质：分离实现，解释执行。

何时使用解释器模式

当有一个语言需要解释执行，并且可以将该语言中的句子表示为一个抽象语法树的时候，就可以考虑解释器模式。

使用解释器模式的时候有两个特点需要考虑，一是语法相对应该比较简单，另一个是效率要求不是很高，效率要求很高的情况下不适合使用解释器模式。

第22章 装饰模式

此处的例子是奖金的计算，要求灵活，因为奖金计算的方式随着时间和人员的职级的不同的不同而变化。

装饰器模式定义：

动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式比生成子类更为灵活。

所谓透明地给一个对象增加功能，换句话说就是要给一个对象增加功能，但是不能让这个对象知道，也就是不能去改动这个对象。

在装饰器模式中，为了能够实现和原来使用被装饰对象的代码无缝结合，是通过定义一个抽象类，让这个类实现与被装饰对象相同的接口，然后再具体的实现类中，转调被装饰的对象，在转调的前后添加新的功能，这就实现了给被装饰对象增加功能

模式讲解

1.装饰器模式的功能

该模式能够实现动态地为对象添加功能，是从一个对象外部来给对象增加功能，相当于改变了对象的外观。

2.对象组合

在面向对象的设计中，有一条基本的原则是尽量使用对象组合，而不是对象继承来扩展和复用功能。

装饰器模式和AOP

类似于AOP，AOP也是通过底层的要么是继承要么是实现接口或者组合。

装饰器模式的优缺点

优点

- 比继承更灵活
- 更容易复用功能
- 简化高层定义

缺点：会产生很多细粒度的对象

何时选用装饰器模式

- 如果需要在不影响其他对象的情况下，以动态，透明的方式给对象添加职责，可以选用装饰器模式。
- 如果不适合使用子类来进行扩展的时候，就可以考虑使用装饰器模式。

第23章 责任链模式

此处举例是使用的申请聚餐费用的管理。

问题抽象一下就是：客户端发出一个请求，会有很多对象都可以来处理这个请求，而且不同对象的处理逻辑是不一样的。对于客户端而言，无所谓谁来处理，反正有对象处理就可以了。而且在上述处理中，还希望流程处理是可以灵活变动的，而处理请求的对象需要能方便地修改或者是被替换掉，以适应新的业务功能的需要。

解决上述问题：**责任链模式**

使多个对象有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

模式讲解

如何构建链

归结起来大致有以下一些方式。

首先是按照实现的地方来说：

- 可以是现在客户端提交请求前组合链。也就是在使用的时候动态组合链，称为外部链。
- 也可以在Handler里面实现链的组合，算是内部链的一种。
- 还有一种就是在各个职责对象中，由于各个职责对象自行决定后续的处理对象。

按照构建链的数据来源，也就是决定了按照什么顺序来组合链的数据，又分为以下几种。

- 一种就是在程序中动态组合。
- 也可以通过外部，比如数据库来获取组合数据，这种属于数据库驱动的方式。
- 通过配置文件传递进来，也可以是流程的配置文件。

职责链模式的优缺点

优点

- 请求者和接收者松散耦合
- 动态组合职责

缺点

- 产生很多细粒度对象
- 不一定能被处理

职责链模式的本质：分离职责，动态组合

何时选用职责链模式

- 如果有多个对象可以处理同一个请求，但是具体由那个对象处理该请求，是运行时动态决定的。
- 如果你想在不明确指定接收者的情况下，向多个对象中的其中一个提交请求的话，可以使用责任链模式。
- 如果想要动态指定处理一个请求的对象的集合，可以使用责任链模式。

第24章 桥接模式

举的例子是发送消息的例子。

桥接模式的定义：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。

模式讲解

什么是桥接

所谓桥接就是在不同的事物之间搭一个桥，让它们能够连接起来，可以相互通讯和使用。

给什么东西来搭桥呢？为被分离的抽象部分和实现部分来搭桥。

为何需要桥接

为了达到让抽象部分和实现部分都可以独立变化的目的。

如何桥接

只要让抽象部分拥有实现部分的接口对象，就桥接上了。桥接在程序上体现了在抽象部分拥有实现部分的接口对象，维护桥接就是维护这个关系。

独立变化

桥接模式的意图是使得抽象和实现可以独立变化，都可以分别扩充。

动态变换功能

由于桥接模式中的抽象部分和实现部分是完全分离的，因此可以在运行时动态组合具体的真实实现，从而达到动态变换功能的目的。

从另一个角度看，抽象部分和实现部分没有固定的绑定关系，因此同一个真实实现可以被不同的抽象对象使用；反过来，同一个抽象也可以有多个不同的实现。

退化的桥接模式

如果Implementor仅有一个实现，那么就没有必要创建Implementor接口了，这是一种桥接模式退化的情况。这个时候Abstraction和Implementor是一对一的关系，虽然如此，也还是要保持它们的分离状态，这样的话，才不会相互影响，才可以分别扩展。

也就是说，就算不要Implementor接口了，也要保持Abstraction和Implementor是分离的，模式的分离机制仍然是非常有用的。

桥接模式和继承

继承是扩展对象功能的一种常见手段，通常情况下，继承扩展的功能变化纬度都是一维的，也就是变化的因素只有一类。

对于出现变化因素有两类的，也就是两个变化纬度的情况，继承实现就会比较痛苦。

桥接模式就是用来解决这种有两个变化纬度的情况下，如何灵活地扩展功能的一个很好的方案。

谁来桥接

就是谁来负责创建抽象部分和实现部分的关系，说的更直白点，就是谁来负责创建Implementor对象，并把它设置到抽象部分的对象中去，这点对于使用桥接模式来说，是十分重要的一点。

大致有以下几种实现方式：

- 由客户负责创建Implementor对象，并在创建抽象部分对象的时候，把它设置到抽象部分的对象中去。
- 可以在抽象部分对象构建的时候，由抽象部分的对象自己来创建相应的Implementor对象，当然可以给它传递一些参数，它可以根据参数来选择并创建具体的Implementor对象。
- 可以在Abstraction中选择并创建一个默认的Implementor对象，然后子类可以根据需要改变这个实现。
- 也可以使用抽象工厂或者简单工厂来选择并创建具体的Implementor对象，抽象部分的类可以通过调用工厂的方法来获取Implementor对象。
- 如果使用IOC/DI容器的话，还可以通过IOC/DI容器来创建具体的Implementor对象，并注入回Abstraction中。

桥接模式的优点

- 分离抽象和实现部分
- 更好的扩展性
- 可动态地切换实现
- 可减少子类的个数

桥接模式的本质：分离抽象和实现。

对设计原则的体现：

1. 很好地实现了开闭原则
2. 很好地体现了多用对象组合，少用对象继承

何时选用桥接模式

- 如果你不希望在抽象部分和实现部分采用固定的绑定关系，可以采用桥接模式，来把抽象部分和实现部分分开，然后在程序运行期间来动态地设置抽象部分需要用到的具体的实现，还可以动态地切换具体的实现。
- 如果出现抽象部分和实现部分都能够扩展的情况，可以采用桥接模式，让抽象部分和实现部分独立地变化，从而灵活地进行单独扩展，而不是搅在一起，扩展一边就会影响另一边。
- 如果希望实现部分的修改不会对客户产生影响，可以采用桥接模式。
- 如果采用继承的实现方案，会导致产生很多子类，对于这种情况，可以考虑采用桥接模式，分析功能变化的原因，看看是否能够分离成不同的纬度，然后通过桥接模式来分离它们，从而减少子类的数目。

第25章 访问者模式

此处举例的是客户管理的功能。

访问者模式的定义：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各个元素的类的前提下定义作用于这些元素的新操作。

模式讲解

访问者的功能：能给一系列对象透明地添加新功能从而避免在维护期间对这一系列对象进行修改，而且还能变相实现复用访问者所具有的功能。

调用通路

访问者之所以能实现“为一系列对象透明地添加新功能”，也就是这一系列对象是不知道被添加功能的。

重要的就是依靠通用方法，访问者这边说要去访问，就提供一个访问的方法，如visit方法：而对象那边说，好的，我接受你的访问，提供一个接受访问的方法，如accept方法。这两个方法并不代表任何具体的功能，只是**构成一个调用的通路**，真正的功能实现在accept方法里边，回调visit方法，从而回调到访问者的具体的实现上，而这个访问者的具体实现的方法才是要添加的新的功能。

两次分发技术

访问者模式能够实现在不改变对象结构的情况下，就可以给对象结构中的类增加功能，实现这个效果所使用的核心技术就是两次分发技术。

空的访问方法

并不是所有的访问方法都需要实现，由于访问者模式默认的是访问对象结构中的所有元素，因此在实现某些功能的时候，如果不需要涉及到某些元素的访问方法，就可以实现成为空的。

操作组合对象结构

访问者模式一个很常见的应用，就是和组合模式结合使用，通过访问者模式给由组合模式构建的对象结构增加功能。

对于使用组合模式构建的组合对象结构，对外有一个统一的外观，要想添加新的功能也不是很困难，只要在组件的接口上定义新的功能就可以了，糟糕的是这样以来，需要修改所有的子类。而且每次添加一个新功能，都需要修改组件接口，然后修改所有的子类。

为了让组合对象结构更灵活，更容易维护和有更好的扩展性，可以把它改造成访问者模式和组合模式组合来实现。

访问者模式的优缺点

优点：

- 好的扩展性
- 好的复用性
- 分离无关行为

缺点：

- 对象结构变化很苦难：不适用于对象结构中的类经常变化的情况，因为对象结构发生了改变，访问者的接口和访问者的实现都要发生相应的改变，代价太高。
- 破坏封装性：访问者模式通常需要对象结构开放内部数据给访问者和ObjectStructure，这破坏了对象的封装性。

访问者模式的本质：预留通路，回调实现

何时选用访问者模式

- 如果想对一个对象结构实施一些依赖于对象结构中具体类的操作，可以使用访问者模式。
- 如果想对一个对象结构中的各个元素进行很多不同的而且不相关的操作，为了避免这些操作使类变得杂乱，可以使用访问者模式。把这些操作分散到不同的访问者对象中去，每个访问者对象实现同一类功能。
- 如果对象结构很少变动，但是需要经常给对象结构中的元素对象定义新的操作，可以使用访问者模式。

设计模式和设计原则

单一职责原则SRP(Single Responsibility Principle)

开放-关闭原则OCP(Open-Closed Principle)

里氏替换原则LSP(Liskov Substitution Principle)

依赖倒置原则DIP(Dependency Inversion Principle)

接口隔离原则ISP(Interface Segregation Principle)

最少知识原则LKP(Least Knowledge Principle)

其他原则