设计模式的思考

注: 本文只描述设计模式,不实现设计模式.

结构型模式： Adapter 让我们改变一个类的接口。 Bridge 让我们改变一个类的实现。 Composite 让我们改变对象的结构和组成。 Decorator 让我们改变职责而无需派生子类。Facade 让我们改变一个子系统的接口。Flyweight 让我们改变对象的存储开销。 Proxy 让我们改变如何访问一个对象以及改变对象的位置。

创建型

创建型设计模式有好几种,没有全部写出(抽象工厂模式, 工厂方法,单例).

建造者模式

建造模式是对象的创建模式。建造模式可以将一个产品的内部表象（*internal representation*）与产品的生产过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。

产品的内部表象

一个产品常有不同的组成成分作为产品的零件，这些零件有可能是对象，也有可能不是对象，它们通常又叫做产品的内部表象（*internal representation*）。不同的产品可以有不同的内部表象，也就是不同的零件。使用建造模式可以使客户端不需要知道所生成的产品有哪些零件，每个产品的对应零件彼此有何不同，是怎么建造出来的，以及怎么组成产品。

建造模式利用一个导演者对象和具体建造者对象一个个地建造出所有的零件，从而建造出完整的产品对象。建造者模式将产品的结构和产品的零件的建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

使用建造模式构建复杂对象

建造模式也适用于构造一个复杂,约束性强的对象,将对象的构造过程''封装''起来.因为目的明确且流程固定,所以可以考虑将''导演''去除,客户端自己''引导''完成建造对象.

在什么情况下使用建造模式

1. 需要生成的产品对象有复杂的内部结构，每一个内部成分本身可以是对象，也可以仅仅是一个对象（即产品对象）的一个组成部分。

2. 需要生成的产品对象的属性相互依赖。建造模式可以强制实行一种分步骤进行的建造过程，因此，如果产品对象的一个属性必须在另一个属性被赋值之后才可以被赋值，使用建造模式是一个很好的设计思想。

3. 在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。(其实就是一些业务中的对象不能''对外开放''.像 HashMap 中的 entrySet).

原型模式

原型模式属于对象的创建模式。通过给出一个原型对象来指明所有创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。这就是选型模式的用意。

原型模式要求对象实现一个可以"克隆"自身的接口，这样就可以通过复制一个实例对象本身来创建一个新的实例。这样一来，通过原型实例创建新的对象，就不再需要关心这个实例本身的类型，只要实现了克隆自身的方法，就可以通过这个方法来获取新的对象，而无须再去通过 new 来创建。

原型模式有两种表现形式：（1）简单形式、（2）登记形式，这两种表现形式仅仅是原型模式的不同实现。

浅度克隆

只负责克隆按值传递的数据（比如基本数据类型、String 类型），而不复制它所引用的对象，换言之，所有的对其他对象的引用都仍然指向原来的对象。

深度克隆

除了浅度克隆要克隆的值外，还负责克隆引用类型的数据。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深度克隆把要复制的对象所引用的对象都复制了一遍，而这种对被引用到的对象的复制叫做间接复制。

深度克隆要深入到多少层，是一个不易确定的问题。在决定以深度克隆的方式复制一个对象的时候，必须决定对间接复制的对象时采取浅度克隆还是继续采用深度克隆。因此，在采取深度克隆时，需要决定多深才算深。此外，在深度克隆的过程中，很可能会出现循环引用的问题，必须小心处理。

代码案例:

```
ByteArrayOutputStream bos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(this);
//从流里读回来
ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
ObjectInputStream ois = new ObjectInputStream(bis);
return ois.readObject();
```

原型模式的缺点

原型模式最主要的缺点是每一个类都必须配备一个克隆方法。配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类来说不是很难，而对于已经有的类不一定很容易，特别是当一个类引用不支持序列化的间接对象，或者引用含有循环结构的时候。

结构型

适配器模式

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

适配器模式的结构

适配器模式有**类的适配器模式(基于继承)**和**对象的适配器模式(基于依赖对象)**两种不同的形式。

类适配器和对象适配器的权衡

类适配器使用对象继承的方式，是静态的定义方式；而**对象适配器**使用对象组合的方式，是动态组合的方式。

对于类适配器，由于适配器直接继承了 Adaptee，使得适配器不能和 Adaptee 的子类一起工作，因为继承是静态的关系，当适配器继承了 Adaptee 后，就不可能再去处理 Adaptee 的子类了。

对于对象适配器，一个适配器可以把多种不同的源适配到同一个目标。换言之，同一个适配器可以把源类和它的子类都适配到目标接口。因为对象适配器采用的是对象组合的关系，只要对象类型正确，是不是子类都无所谓。

对于类适配器，适配器可以重定义 Adaptee 的部分行为，相当于子类覆盖父类的部分实现方法。

对于对象适配器，要重定义 Adaptee 的行为比较困难，这种情况下，需要定义 Adaptee 的子类来实现重定义，然后让适配器组合子类。虽然重定义 Adaptee 的行为比较困难，但是想要增加一些新的行为则方便的很，而且新增加的行为可同时适用于所有的源。

对于类适配器，仅仅引入了一个对象，并不需要额外的引用来间接得到 Adaptee。

对于对象适配器，需要额外的引用来间接得到 Adaptee。

建议尽量使用**对象适配器**的实现方式，多用合成/聚合、少用继承。当然，具体问题具体分析，根据需要来选用实现方式，最适合的才是最好的。

适配器模式的优点

更好的复用性

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

更好的扩展性

在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的功能。

适配器模式的缺点

过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

缺省适配模式

缺省适配(Default Adapter)模式为一个接口提供缺省实现，这样子类型可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。作为适配器模式的一个特例，缺省是适配模式在 JAVA 语言中有着特殊的应用。

适配器模式的用意是要改变源的接口，以便于目标接口相容。

缺省适配的用意稍有不同，它是为了方便建立一个不平庸的适配器类而提供的一种平庸实现。

组合模式(平时没碰到过,不太好理解这跟普通的''树''有什么区别.)

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

解刨这句话: 树形结构,使用一致性(旁外话:并不代表内部真的一致)

什么情况下使用组合模式

引用大话设计模式的片段:"当发现需求中是体现部分与整体层次结构时,以及你希望用户可以忽略组合对象与单个对象的不同,统一地使用组合结构中的所有对象时,就应该考虑组合模式了。"

外观模式(门面模式)

org.apache.catalina.connector.RequestFacade

它隐藏了系统的复杂性,并向客户端提供了一个可访问系统的接口. 这种类型的设计模式属于结构性模式.为子系统中的一组接口提供了统一的访问接口,这个接口使得子系统更容易被访问或者使用.

门面模式的优点

1. **减少系统的相互依赖:** 想想看,如果我们不使用门面模式,外界访问直接深入到子系统内部,相互之间是一种强耦合关系,你死我就死,你活我才能活,这样的强依赖是系统设计所不能接受的,门面模式的出现就很好地解决了该问题,所有的依赖都是对门面对象的依赖,与子系统无关。
2. **强约束性:** 想让你访问子系统的哪些业务就开通哪些逻辑,不在门面上开通的方法,不允许访问.

门面模式的缺点

门面模式最大的缺点就是不符合开闭原则,对修改关闭,对扩展开放.

需要注意的几点

1.为子系统增加新行为

初学者往往以为通过继承一个门面类便可在子系统中加入新的行为，这是错误的。门面模式的用意是为子系统提供一个集中化和简化的沟通管道，而不能向子系统加入新的行为。比如医院中的接待员并不是医护人员，接待员并不能为病人提供医疗服务。

2.外观模式与迪米特法则

外观模式创造出一个外观对象，将客户端所涉及的属于一个子系统的协作伙伴的数量减到最少，使得客户端与子系统内部的对象的相互作用被外观对象所取.

3.抽象外观类的引入

外观模式最大的缺点在于违背了“开闭原则”，当增加新的子系统或者移除子系统时需要修改外观类，可以通过引入抽象外观类在一定程度上解决该问题，客户端针对抽象外观类进行编程。对于新的业务需求，不修改原有外观类，而对应增加一个新的具体外观类，由新的具体外观类来关联新的子系统对象，同时通过修改配置文件来达到不修改源代码并更换外观类的目的。

模式适用场景

当一个强耦合性的程序需要解耦时,”门面模式”是个不错的选择.

桥梁模式

桥梁模式是对象的结构模式。又称为柄体(Handle and Body)模式或接口(Interface)模式。桥梁模式的用意是“将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化”。

桥梁模式所涉及的角色有：

抽象化(Abstraction)角色：抽象化给出的定义，并保存一个对实现化对象的引用。

修正抽象化(RefinedAbstraction)角色：扩展抽象化角色，改变和修正父类对抽象化的定义。 个人认为这个可要可不要,如果能抽象出来的话就抽象.

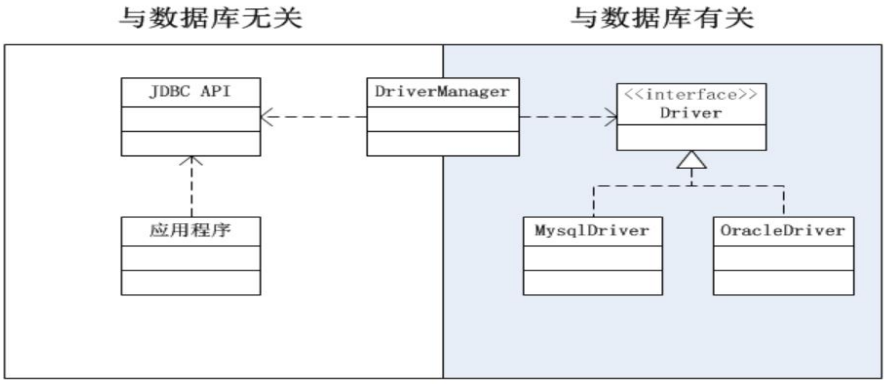
实现化(Implementor)角色：这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非

常不一样。实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。

具体实现化(*ConcreteImplementor*)角色：这个角色给出实现化角色接口的具体实现。

优点： 解耦.

桥梁模式在 Java 中的使用-- JDBC 驱动器(DriverManager)



JDBC 的这种架构，把抽象部分和具体部分分离开来，从而使得抽象部分和具体部分都可以独立地扩展。对于应用程序而言，只要选用不同的驱动，就可以让程序操作不同的数据库，而无需更改应用程序，从而实现在不同的数据库上移植；对于驱动程序而言，为数据库实现不同的驱动程序，并不会影响应用程序。

责任链模式

责任链模式是一种对象的行为模式。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

责任链模式在 Tomcat 中的应用: ApplicationFilterChain 中的 filters

命令模式

命令模式属于对象的行为模式。命令模式又称为行动(Action)模式或交易(Transaction)模式。

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

涉及角色

客户端(Client)角色：创建一个具体命令(ConcreteCommand)对象并确定其接收者。

命令(Command)角色：声明了一个给所有具体命令类的抽象接口。

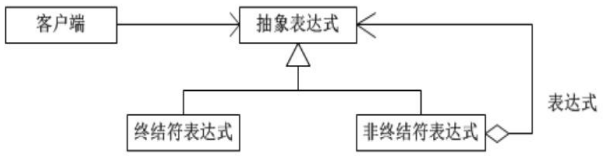
具体命令(ConcreteCommand)角色：定义一个接收者和行为之间的弱耦合；实现execute()方法，负责调用接收者的相应操作。execute()方法通常叫做执行方法。

请求者(Invoker)角色：负责调用命令对象执行请求，相关的方法叫做行动方法。

接收者(Receiver)角色：负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

解释器模式

解释器模式是类的行为模式。给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。



解释器涉及到 4 个角色:

抽象表达式: 声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个 interpret()方法，称做解释操作。

终结符表达式: 实现了抽象表达式角色所要求的接口，主要是一个 interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应,用来被非终结符表达式使用。比如有一个简单的公式 R=R1+R2，在里面 R1 和 R2 就是终结符，对应的解析 R1 和 R2 的解释器就是终结符表达式。

非终结符表达式: 文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式 R=R1+R2 中，"+"就是非终结符，解析"+"的解释器就是一个非终结符表达式。

环境角色: 这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如 R=R1+R2，我们给 R1 赋值 100，给 R2 赋值 200。这些信息需要存放到环境角色中，很多情况下我们使用 Map 来充当环境角色就足够了。