# 设计模式复习

从全C编程转到C++编程已经快两年了，回顾曾经看设计模式的经历，几多感概。现在，大家讲设计模式，一般都是针对面向对象的语言来说，就面向对象而言，它其实不分语言，c++,java,python等都能提炼这些经典模式。花了几天时间复习了23种经典设计模式：

## 创建型

### 三种工厂模式



简单工厂模式：算法和界面接口分离，降低耦合度。



工厂模式：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。



抽象工厂模式：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。

上面对三种工厂模式分别有UML图示和最精炼的介绍，应该说看图示更好理解，因为介绍的内容是站在一个高度的提炼，太抽象。三种工厂模式在实际项目开发中，使用频率是递减的，因为抽象程度不断加深，而在项目中结合了具体业务后，往往在有些基本场景中并不需要那么高度的抽象，尤其是在C++中，繁冗的抽象甚至会影响开发和运行效率，尤其是抽象工厂模式，基本上在进程或库中的使用往往是全局框架下的大结构。

### 单例模式



单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

单例模式有几个需要注意的地方：

* 单例类不能有实例，所以构造函数要处理成私有；
* 单例类实现要注意并行操作，所以要注意加互斥锁；
* 单例类的析构，一般另外起一个carbo类来做，carbo类就写在单例类里面，然后在carbo类的析构里面做单例类的析构，单例类添加一个静态的carbo实例即可。这是为了避免死循环。

另外，说到析构，就想到一个话题：进程退出时不主动释放资源，会有什么影响，有内存泄露吗？个人认为，进程退出，系统会自动回收进程申请的所有资源，包括内存、文件句柄等等，因此不会有内存泄露。虽然单例类的析构往往是伴随进程的整个生命期，但我们还是认真对待其析构的意义在于两方面：

* 类对象有业务场景，在对象销毁前需要做业务的回滚等清理工作，比如文件、数据库操作等，这时需要在析构中写完善，在进程退出时会调用。
* 代码素养，这个无需多说，哪说哪了的事。

### 建造者模式



建造者模式：将一个复杂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。

同为创建型的设计模式，建造者模式和工厂模式有很强的参照性：

* 前者的指挥类相当于后者的工厂类，都用来生成合适的计算类；
* 但工厂模式的计算类自身包含所有的计算需求，但建造者模式的计算类是由建造者和产品类一起组合出计算需求；
* 工厂模式意在获取具有某种计算能力的类，而建造者模式意在通过得到具有种计算能力的类之后再通过这个类组装得到目标产品类。建造者模式的使用场景是目标产品类有多个组成部分且各自构造复杂，需要制定产品的创建顺序且创建过程独立于产品类本身。

### 原型模式



原型模式：用原型实例指定创建对象的种类，用该原型创建新对象。

C++的拷贝构造函数就是原型模式的样板，需要注意的是，C++的默认拷贝构造函数是浅拷贝，如果要深拷贝，需要手动处理。

## 结构型

### 桥接模式



桥接模式：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。

首先，桥接模式最好的诠释了一个道理：多用组合来替代继承实现算法。抽象部分是对外的接口，实现部分是具体的算法内容，二者通过组合发生关系，灵活易扩展。

其次，桥接模式常见的一个场景就是很多C++应用库的实现，库向外提供头文件，即抽象部分，其实现文件中用内部组合的计算类来实现功能，这样就很好的实现了封装。

### 适配器模式



适配器模式：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。

适配器模式有类适配器和对象适配器两种，上图中左图为类适配器、右图为对象适配器。类适配器通过多重继承来做，adapter直接使用adaptee的方法；对象适配器通过组合来实现request，adapter通过组合调用组合对象的方法来request。

### 装饰者模式



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

首先，装饰者模式又是一个有力支撑“多用组合代替继承”的例子，给对象增添职责，如果一味采用继承，会增加代码层次复杂性。这里需要强调的是，继承的缺点中，并不一定是增加了对象大小，除非子类继承了父类很多成员变量，不然成员函数并不占对象空间，继承的缺点大多是增加了代码逻辑复杂性以及产生不必要的代码风险。

另外，组合就有谁包含谁的问题，装饰者模式中，是装饰者包含被装饰者，在装饰者类中丰富被装饰者的方法。

### 组合模式 + 享元模式



组合模式：将对象组合成树形结构以表示“整体-部分”的层次结构，使用户对单个对象和组合对象具有一致性。



享元模式：当某个类生成大量细粒度的实例，并且这些实例除了几个参数外大部分是相同的，而这些参数可以在方法调用时传递进来，就可以通过共享实例的方式。

把这两个模式放在一起，一来它们都是结构型模式、另一来它们都有内部维护一个对象数组。组合模式通过内部数组关联父子节点关系、享元模式通过内部数组保存用于共享的实例。

### 门面模式



门面模式：一个子系统的外部和内部通信必须通过一个统一的接口类完成。

门面模式的使用很普遍，因为面向对象程序设计的一大原则就是封装，对于高度内聚的子系统，当外部需要调用其中的数据时，需要由统一的入口，门面类即充当这个角色。对外部而言，只需要与门面类的接口交互就行了。

### 代理模式



代理模式：为其他对象提供一种代理以控制对这个对象的访问。

从UML图中可以看出，代理类和实际类是同层次子类，外部访问的是代理类对象，然后代理类对象调用实际处理类对象的算法，这样有几种可能的好处：访问控制；智能指引等。

代理模式和装饰者模式很像，但侧重完全不同，前者用于丰富目标对象的接口，后者只是控制对象接口的访问。

## 行为型

### 模板模式



模板模式：算法/逻辑框架放在抽象基类中，并定义好实现接口，在子类中实现细节接口。

C++中通过继承、抽象基类这些语法来天然支持模板模式。

### 观察者模式 + 中介者模式



观察者模式：定义对象间一对多的关系，当一个对象状态发生改变，所有依赖它的对象都得到通知并且状态自动改变。



中介模式：用一个中介对象来封装一系列对象交互。中介者可以使各对象不需要显示的相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

这里把观察者模式和中介者模式放在一起，是因为二者在结构上几乎一样。观察者模式行为上更固定，即主体对象接受观察者注册上来，通过数组保存这些观察者对象，之后动态更新数组中观察者的状态。中介者模式也类似于接受一系列对象注册，但行为更灵活，即在如何处理不同对象间的交互上，中介类可以有很大的处理空间。

### 备忘录模式



备忘录模式：在不破话封装性的情况下，在某个时刻捕获一个类对象的内部状态，并在外部保存这个状态，在适当时候可以恢复之前的状态。

备忘录模式很简单，需要保存的快照信息包括对象的属性，保存时刻的一些即时数据等等。

### 命令模式



命令模式：将请求封装为一个对象，从而可以根据不同的请求对客户进行参数化、对请求进行排队或记录日志，以及支持可撤销的操作。

创建个具体的命令对象，为它确定命令接受者，当触发条件产生使得命令对象的方法执行后，最后实际调用接收者的方法。接收者可以由客户端选择。命令模式在结构上使用聚合和关联使触发事件得以最后施行，在行为上表现为**对象的回调**。

### 职责链模式



职责链模式：一系列有层次关系的对象，为了使所有对象都有机会处理请求，让请求从层次链逐个传递，直到有一个对象处理该请求为止。

职责链在很多库框架中有应用，比如MFC的消息机制，QT的消息机制等等，需要注意的是，职责链中层次关系是通过聚合而来，但可以通过让所有对象共同继承一个基类使层次更容易控制且更清晰。

### 策略模式 + 状态机模式



策略模式：客户使用某类算法，算法在各个子类单独实现，算法的变化不会影响使用算法的客户。



状态模式：允许对象在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。

这里把策略模式和状态机模式放在一起，是因为二者在结构上几乎一样，但既然被放到行为型模式的范畴，那么从行为上看它们肯定区别很大了。策略模式中算法的变化是客户自己选择的、关注的是方法本身，而状态机模式注重的是整个对象（状态）的变化且变化可以因某些触发条件自行发生、关注的是对象（状态）。

### 访问者模式 + 迭代器模式



访问者模式：当一个对象结构有很多类对象，而外部需要对这些类对象实施依赖其具体类的操作，为了避免这些操作污染这些对象的类，使用visitor类来统一调度对这些类对象的接口访问。



迭代器模式：提供一种顺序访问聚合对象中元素，而又不暴露对象的内部表示。

这里把访问者模式和迭代器模式放在一起，是因为个人觉得迭代器模式其实就是一种特化的访问者模式。二者都维护了一些对象，都希望通过另外的类来调度对聚合对象的访问，只不过访问者模式更灵活，因为visitor对象本身可以在接口调度时添加一些更丰富的内容。

### 解释器模式



解释器模式：给定一个上下文对象，根据对象的情况选择一种解释/处理方式，即创造出解释器类，解释器的终止类型表示一种解释语法，非终止类型表示组合得到的解释语法。

解释语法的应用场景也有很多，比如正则表达式内部实现应该就用到了，还有像命令行操作的实现，应该也是对输入进行命令解释，根据基本语法和组合语法对输入进行解释并响应。