设计模式汇总

一、 设计模式分类

	创建型	结构型	行为型	
类	factory method 工厂方法模式	adapter 适配器模式	template method 模板方法模式	interpreter 解释器模式
对象	abstract factory 抽象工厂模式 prototype 原型模式 singleton 单例模式 builder 构建器模式	bridge 桥接模式 composite 组合模式 decorator 装饰模式 facade 外观模式 flyweight 享元模式 proxy 代理模式	chain of responsibility 职责链模式 command 命令模式 iterator 迭代器模式 mediator 中介者模式	memento 备忘录模式 observer 观察者模式 state 状态模式 strategy 策略模式 visitor 访问者模式

二、设计模式介绍

（1）创建型模式

设计模式名称	简要说明	速记关键字
Abstract Factory 抽象工厂模式	提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类	生产成系列对象
Builder 构建器模式	将一个复杂类的表示与其构造相分离，使得相同的构建过程能够得出不同的表示	复杂对象构造
Factory Method 工厂方法模式	定义一个创建对象的接口，但由子类决定需要实例化哪一个类。工厂方法使得子类实例化的过程推迟	动态生产对象
Prototype 原型模式	用原型实例指定创建对象的类型，并且通过拷贝这个原型来创建新的对象	克隆对象

Singleton 单例模式	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	单实例

（2）结构型模式

设计模式名称	简要说明	速记关键字
Adapter 适配器模式	将一个类的接口转换成用户希望得到的另一种接口。它使原本不相容的接口得以协同工作	转换接口
Bridge 桥接模式	将类的抽象部分和它的实现部分分离开来，使它们可以独立地变化	继承树拆分
Composite 组合模式	将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性	树形目录结构
Decorator 装饰模式	动态地给一个对象添加一些额外的职责。它提供了用子类扩展功能的一个灵活的替代，比派生一个子类更加灵活	附加职责
Facade 外观模式	定义一个高层接口，为子系统中的一组接口提供一个一致的外观，从而简化了该子系统的使用	对外统一接口
Flyweight 享元模式	提供支持大量细粒度对象共享的有效方法	文章共享文字对象
Proxy 代理模式	为其他对象提供一种代理以控制这个对象的访问	快捷方式

（3）行为型模式

设计模式名称	简要说明	速记关键字
Chain of Responsibility 职责链模式	通过给多个对象处理请求的机会，减少请求的发送者与接收者之间的耦合。将接收对象链接起来，在链中传递请求，直到有一个对象处理这个请求	传递职责

Command 命令模式	将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作	日志记录，可撤销
Interpreter 解释器模式	给定一种语言，定义它的文法表示，并定义一个解释器，该解释器用来根据文法表示来解释语言中的句子	虚拟机的机制
Iterator 迭代器模式	提供一种方法来顺序访问一个聚合对象中的各个元素，而不需要暴露该对象的内部表示	数据库数据集
Mediator 中介者模式	用一个中介对象来封装一系列的对象交互。它使各对象不需要显式地相互调用，从而达到低耦合，还可以独立地改变对象间的交互	不直接引用
Memento 备忘录模式	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，从而可以在以后将该对象恢复到原先保存的状态	可恢复
Observer 观察者模式	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自动更新	联动
State 状态模式	允许一个对象在其内部状态改变时改变它的行为	状态变成类
Strategy 策略模式	定义一系列算法，把它们一个个封装起来，并且使它们之间可互相替换，从而让算法可以独立于使用它的用户而变化	多方案切换

Template Method 模板方法模式	定义一个操作中的算法骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤	文档模板填空
Visitor 访问者模式	表示一个作用于某对象结构中的各元素的操作，使得在不改变各元素的类的前提下定义作用于这些元素的新操作	数据与操作分离

三、设计模式类图

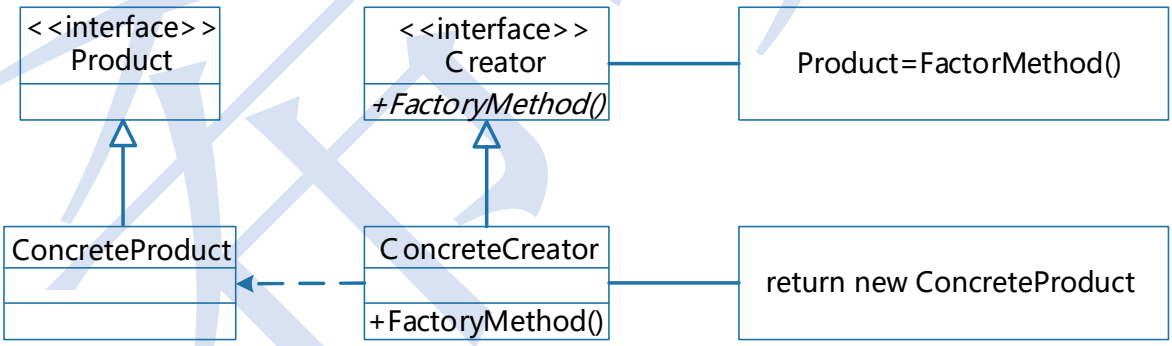
（一）创建型设计模式

【创建型模式与对象的创建有关，抽象了实例化过程，它们帮助一个系统独立于如何创建、组合和表示它的那些对象。一个类创建型模式使用继承改变被实例化的类，而一个对象创建型模式将实例化委托给另一个对象。】

1、工厂方法（Factory Method）模式

（1）工厂模式的意图是：定义一个创建对象的接口，但由子类决定需要实例化哪一个类。 工厂方法使得子类实例化的过程推迟。

（2）类图



- Product：产品角色定义产品的接口。
- ConcreteProduct：真实的产品,实现接口 Product 的类。
- Creator：工厂角色声明工厂方法（Factory Method），返回一个产品。
- ConcreteCreator：真实的工厂实现 Factory Method 工厂方法，由客户调用 ，返回一个产品的实例。

（3）适用场景（动态产生对象）

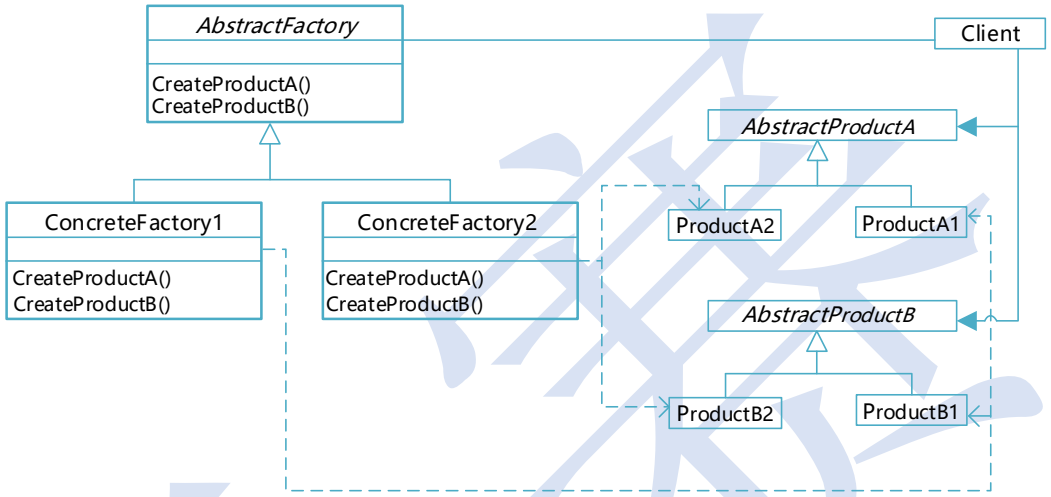
- 当一个类不知道它所必须创建的对象的类的时候。
- 当一个类希望由它的子类来指定它所创建的对象的时候。

- 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

2、抽象工厂方法（Abstract Factory）模式

（1）抽象工厂模式的意图是：提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类。

（2）类图



- AbstractFactory：抽象工厂，声明生成抽象产品对象的方法接口。
- ConcreteFactory：具体工厂，执行生成抽象产品对象的方法，生成一个具体的产品对象。
- AbstractProduct：抽象产品，为一种产品声明接口。
- Product：具体产品，定义具体工厂生成的具体产品的对象，实现产品接口。
- Client：客户，仅使用由抽象产品 AbstractProduct 和抽象工厂 AbstractFactory 声明的接口。

（3）适用场景（生成系列对象）

一个系统要独立于它的产品的创建、组合和表示时。

一个系统要由多个产品系列中的一个来配置时。

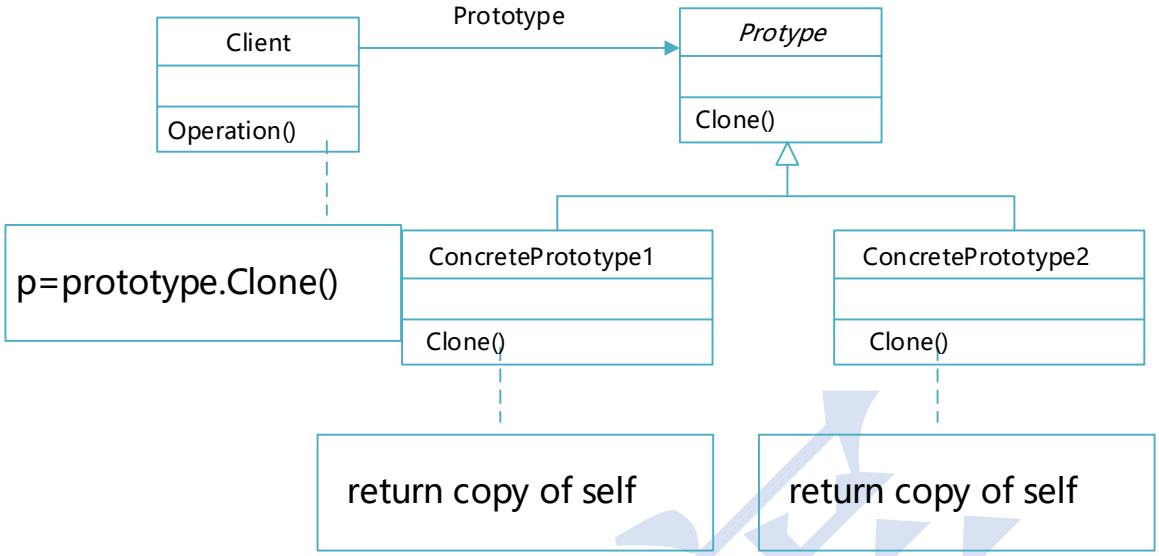
当要强调一系列相关的产品对象的设计以便进行联合使用时。

当提供一个产品类库，只想显示它们的接口而不是实现时。

3、原型（Prototype）模式

（1）原型模式的意图是：用原型实例指定创建对象的类型，并且通过拷贝这个原型来创建新的对象。

（2）类图



- Prototype：抽象原型类,定义具有克隆自己的方法的接口。
- ConcretePrototype：具体原型类,实现具体的克隆方法。
- Client：客户，让一个原型赋值自身从而创建一个新的对象。

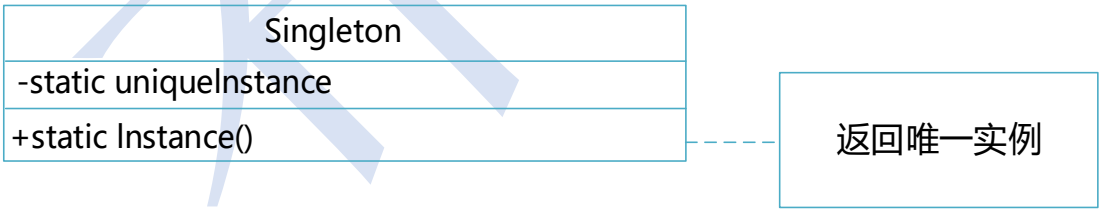
（3）适用场景（克隆对象）

- 当一个系统应该独立于它的产品创建、构成和表示时。
- 当要实例化的类是在运行时刻指定时，例如，通过动态装载。
- 为了避免创建一个与产品类层次平行的工厂类层次时。
- 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们，可能比每次用合适的状态手工实例化该类更方便一些。

4、单例（Singleto）模式

（1）单例模式（单件模式）的意图是：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。

（2）类图



- Singleton：单例,提供一个 instance 的方法，让客户可以使用它的唯一实例。内部实现只生成一个实例。

（3）适用场景（单实例）

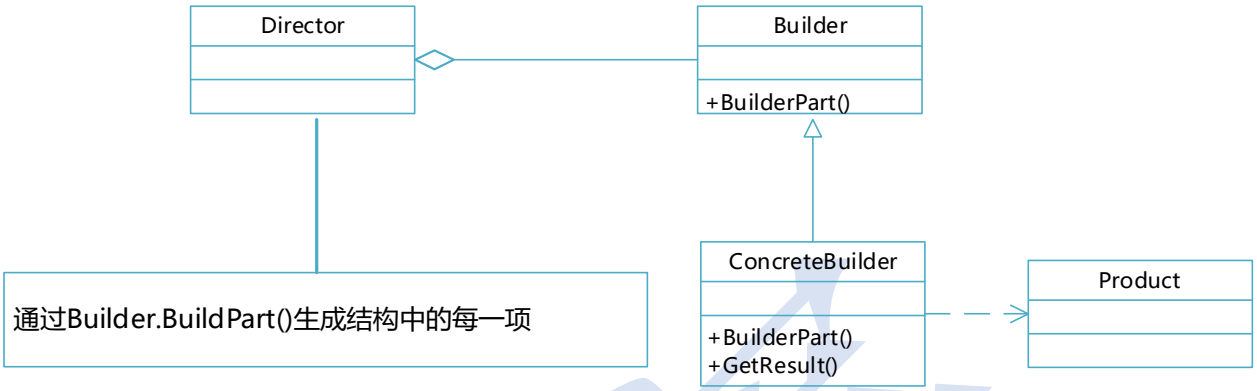
当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

当这个唯一实例应该是通过子类化可扩展的，并且客户无须更改代码就能使用一个扩展的实例时。

5、构建器（Builder）模式

（1）生成器（构建器）模式的意图是：将一个复杂类的表示与其构造相分离，使得相同的构建过程能够得出不同的表示。

（2）类图



- Builder：抽象建造者，为创建一个 Product 对象各个部件指定抽象接口，把产品的生产过程分解为不同的步骤，从而使具体建造者在具体的建造步骤上具有更多弹性，从而创造出不同表示的产品。
- ConcreteBuilder：具体建造者，实现 Builder 接口，构造和装配产品的各个部件定义并明确它所创建的表示，提供一个返回这个产品的接口。
- Director：指挥者，构建一个使用 Builder 接口的对象。
- Product：产品角色，被构建的复杂对象，具体产品建造者，创建该产品的内部表示并定义它的装配过程。包含定义组成组件的类，包括将这些组件装配成最终产品的接口。

（3）适用场景（复杂对象构造）

- 当创建复杂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。
- 当构造过程必须允许被构造的对象有不同的表示时。

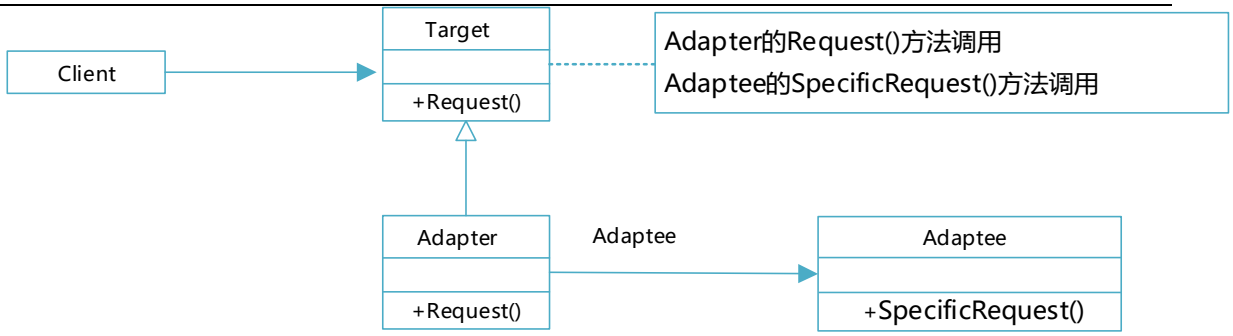
（二）结构型设计模式

【结构型模式处理类或对象的组合，结构型设计模式涉及如何组合类和对象以获得更大的结构。结构型类模式采用继承机制来组合接口或实现。结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能的一些方法。】

6、适配器（Adapter）模式

（1）适配器模式的意图是：将一个类的接口转换成用户希望得到的另一种接口。它使原本不相容的接口得以协同工作。

（2）类图



- 目标抽象（Target）类：定义客户 Client 要用的特定领域的接口。
- 适配器公接口（Adapter）：调用另一个接口，作为一个转换器，对已有母接口 Adaptee 和目标接口 Target 进行适配。
- 适配器母接口（Adaptee）：定义一个已存在的需要被适配的接口，Adapter 需要接入。
- 客户调用（Client）类：与符合 Target 接口的对象协同。

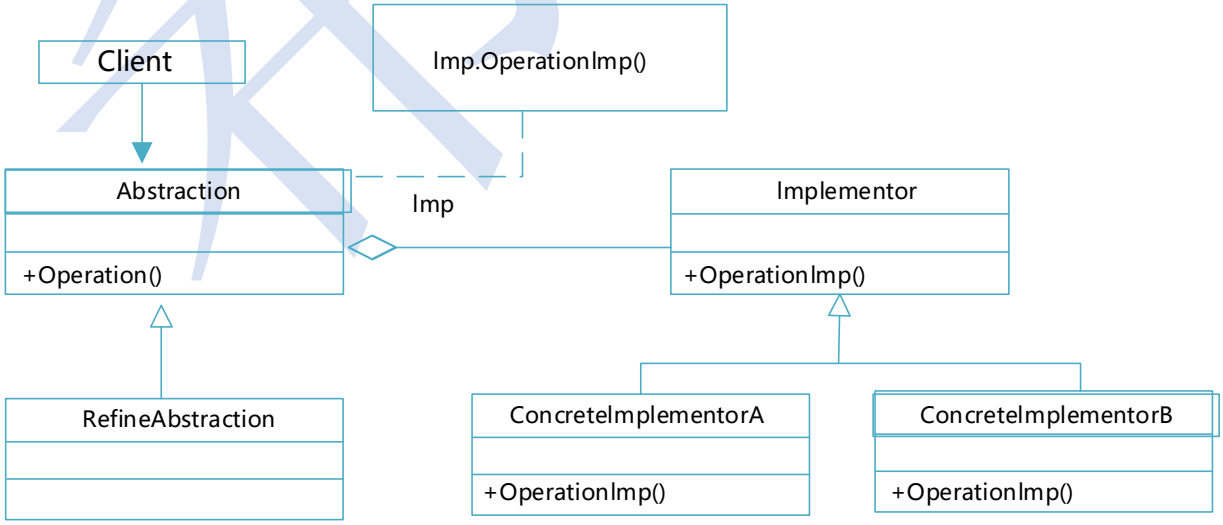
（3）适用场景（转换接口）

- 想使用一个已经存在的类，而它的接口不符合要求。
- 像创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
- （仅适用于对象 Adapter）想使用一个已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

7、桥接（Bridge）模式

（1）桥接模式的意图是：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。

（2）类图



- Abstraction：抽象类定义抽象类的接口。维护一个指向 Implementor（实现抽象类）

对象的一个指针

- RefinedAbstraction：扩充的抽象类,扩充由 Abstraction 定义的接口。
- Implementor：实现类接口,定义实现类的接口，这个接口不一定要与 Abstraction 的接口完全一致，事实上这两个接口可以完全不同，一般的讲 Implementor 接口仅仅给出基本操作，而 Abstraction 接口则会给出很多更复杂的操作。
- ConcreteImplementor：具体实现类,实现 Implementor 定义的接口并且具体实现它。

（3）适用场景（继承树拆分）

不希望在抽象和它的实现部分之间有一个固定的绑定关系。

类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时 Bridge 模式使得开发者可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

对一个抽象的实现部分的修改应对客户不产生影响，即客户代码不必重新编译。

（C++）想对客户完全隐藏抽象的实现部分。

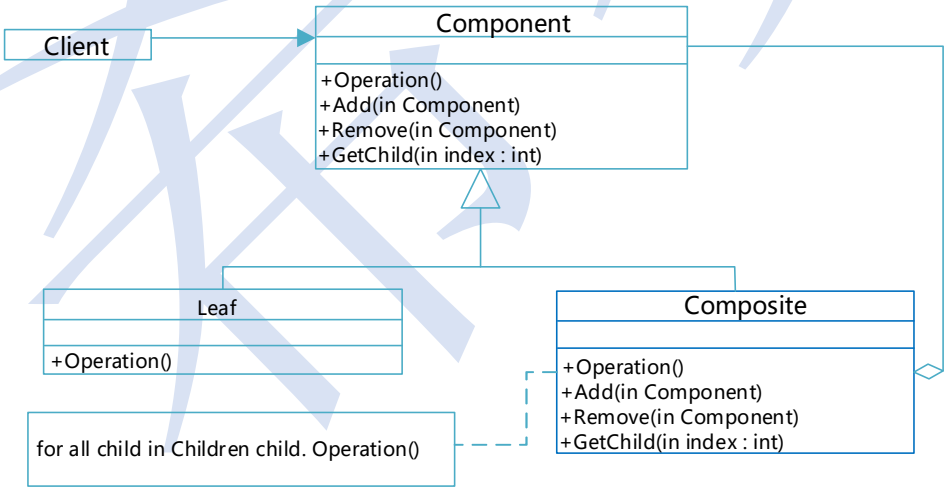
有许多类要生成的类层次结构。

想在多个对象之间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

8、组合（Composite）模式

（1）组合模式的意图是：将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

（2）类图



- Leaf：叶部件,在组合中表示叶节点对象，叶节点没有子节点。定义组合中图元对象的行为。
- Composite：组合类，定义有子节点（子部件）的部件的行为；存储子部件；在 Component 接口中实现与子部件相关的操作。
- Component：为组合中的对象声明接口；声明一个接口用于访问和管理 Compoment

的子部件；在适当情况下实现所有类共有接口的默认行为；（可选）在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况实现它。

- Client：客户应用程序,通过 Component 接口控制组合部件的对象。

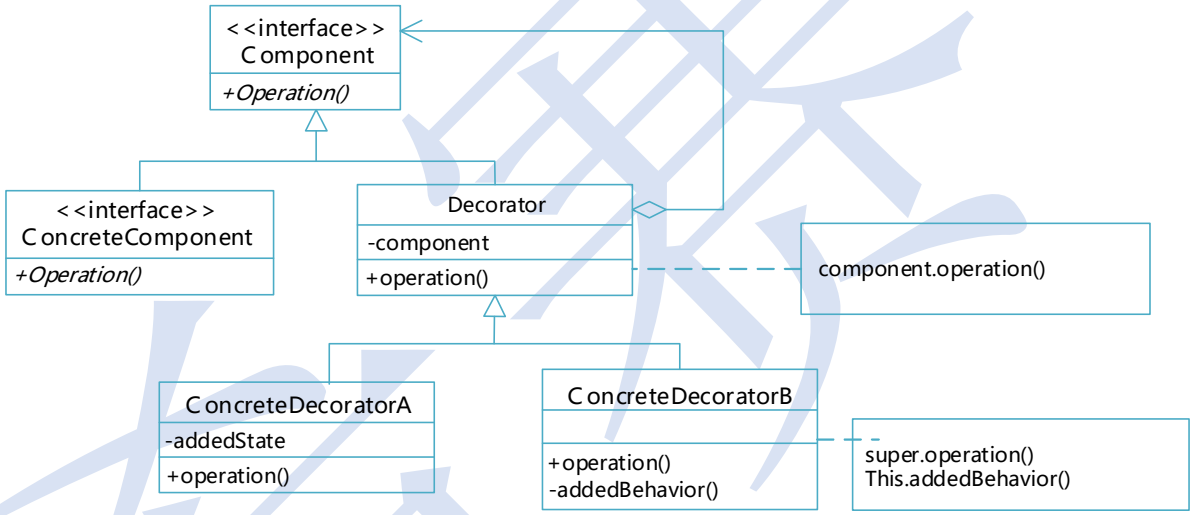
（3）适用场景（树形目录结构）

- 想表示对象的部分-整体层次结构。
- 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

9、装饰（Decorator）模式

（1）装饰模式的意图是：动态地给一个对象添加一些额外的职责。它提供了用子类扩展功能的一个灵活的替代，比派生一个子类更加灵活。

（2）类图



- Component：部件,定义对象的接口，可以给这些对象动态的增加职责（方法）。
- Concrete Component：具体部件,定义具体的对象，Decorator 可以给这个对象增加额外的职责（方法）。
- Decorator：装饰抽象类,维护一个内有的 Component 对象的指针，并且定义一个与 Component 接口一致的接口。
- Concrete Decorator：具体装饰对象，给内在的具体部件对象增加具体的职责（方法）。

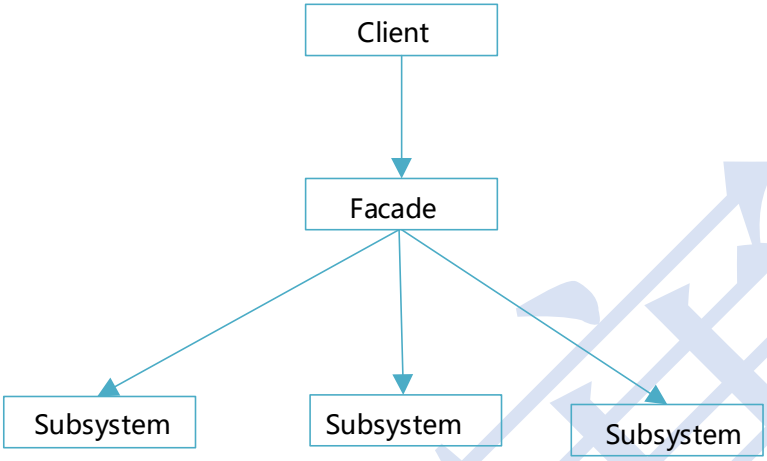
（3）适用场景（附加职责）

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤销的职责。
- 当不能采用生成子类的方式进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是，由于类定义被隐藏，或类定义不能用于生成子类。

10、外观（Facade）模式

（1）外观（门面）模式的意图是：定义一个高层接口，为子系统中的一组接口提供一个一致的外观（界面），从而简化了该子系统的使用。

（2）类图



- Facade：外形类，知道哪些子系统负责处理那些请求，将客户的请求传递给相应的子系统对象处理。
- Subsystem：子系统类，实现子系统的功能，处理由 Façade 传过来的任务。子系统不用知道 Façade，在任何地方也没有引用 Façade，没有指向 Façade 的指针。

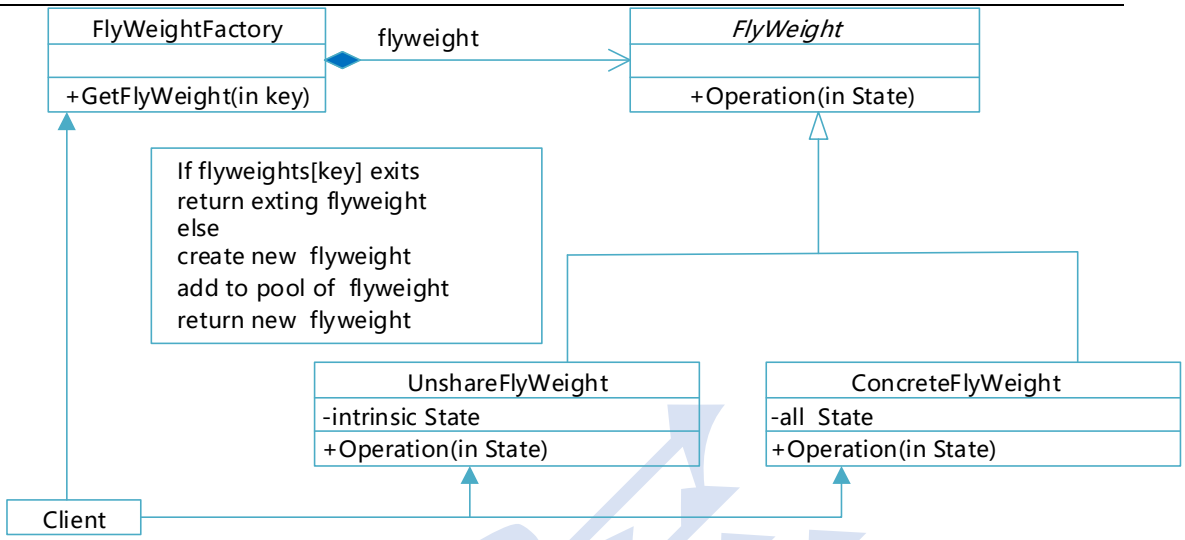
（3）适用场景（对外统一接口）

- 要为一个复杂子系统提供一个简单接口时，子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类，这使得子系统更具有可重用性，也更容易对子系统进行定制，但也给那些不需要定制子系统的用户带来一些使用上的困难。Façade 可以提供一个简单的默认视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过 Façade 层。
- 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 Façade 将这个子系统与客户及其他的子系统分离，可以提供子系统的独立性和可移植性。
- 当需要构建一个层次结构的子系统时，使用 Façade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，则可以让它们仅通过 Façade 进行通信，从而简化了它们之间的依赖关系。

11、享元（Flyweight）模式

（1）享元（轻量级）模式的意图是：提供支持大量细粒度对象共享的有效方法。

（2）类图



- Flyweight：抽象轻量级类,声明一个接口，通过它可以接受外来的状态并作出处理。
- ConcreteFlyweight：具体轻量级类,实现 Flyweight 接口，如果有内部状态，则为其增加存储空间。ConcreteFlyweight 对象必须是可共享的。它所存储的状态必须是内部的，即它必须独立于 ConcreteFlyweight 对象的场景。
- UnsharedConcreteFlyweight：不共享的具体轻量级类，UnsharedFlyweight 对象常常将 ConcreteFlyweight 对象作为子节点。
- FlyweightFactory：轻量级类工厂,创建并且管理 flyweight 对象确保享用 flyweight。
- Client：客户应用程序，维护一个对 Flyweight 的引用；计算或存储一个或多个 Flyweight 的外部状态。

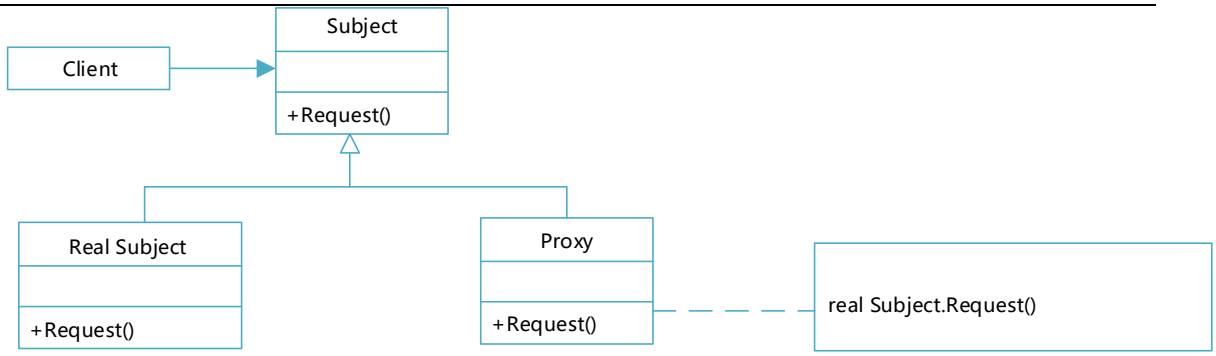
（3）适用场景（文章共享文字对象）

- 一个应用程序使用了大量的对象。
- 完全由于使用大量的对象，造成了很大的存储开销。
- 对象的大多数状态都可变为外部状态。
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
- 应用程序不依赖于对象标识。由于 Flyweight 对象可以被共享，所以对于概念上明显有别的对象，标识测试将返回真值。

12、代理（Proxy）模式

（1）代理模式的意图是：为其他对象提供一种代理以控制对这个对象的访问。

（2） 类图



- Proxy：代理维护一个引用使得代理可以访问实体；提供一个与 Subject 的接口相同的接口，使代理可以用来代替实体；控制对实体的存取，并可能负责创建和删除它；其他功能取决于 Proxy 的类型。【远程代理（Remote Proxy）负责对请求及其参数编码，向不同地址空间中的实体发送已编码的请求；虚代理（Virtual Proxy）可以缓存实体的其他信息，以便延迟对它的访问；保护代理（Protection Proxy）检查调用者的请求是不是有所需的权限。】
- Subject：抽象实体接口,为 RealSubject 实体和 Proxy 代理定义相同的接口，使得 RealSubject 在任何地方都可以使用 Proxy 来访问。
- RealSubject：真实对象，定义 Proxy 代理的实体。

（3）适用场景（快捷方式、远程调用）

Proxy 模式适用于在需要比较通用和复杂的对象指针代替简单的指针的时候，常见情况有：

- 远程代理：为一个对象在不同地址空间提供局部代表。
- 虚代理：根据需要创建开销很大的对象。
- 保护代理：控制对原始对象的访问，用于对象应该有不同的访问权限的时候。
- 智能引用：取代了简单的指针，它在访问对象时执行一些附加的操作，如对指向实际对象的引用计数等。

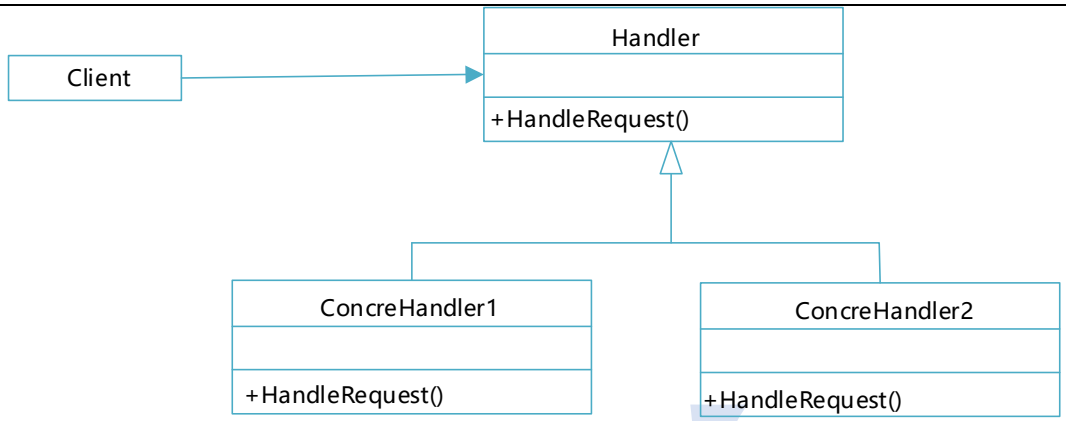
（三）行为型设计模式

【行为模式涉及算法和对象间职责的分配。行为模式不仅描述对象或类的模式，还描述它们之间的通信模式。行为型类模式使用继承机制在类间分配行为，这里包括模板类模式和解释器类模式。行为对象模式使用对象复合而不是继承。一些行为对象模式描述了一组对等的对象怎样相互协作以完成其中任一对象都无法单独完成的任务。】

13、职责链（Chain of Responsibility）模式

（1）职责链模式的意图是：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。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。

（2）类图



- Handler：传递者接口，定义一个处理请求的接口。（可选）实现后继链。
- ConcreteHandler：具体传递者，处理它所负责的请求。可以访问链中下一个对象，如果可以处理请求，就处理它，否则将请求转发给后继者。
- Client：客户应用程序，向链中的具体传递者对象（ConcreteHandler）提出最初的请求。

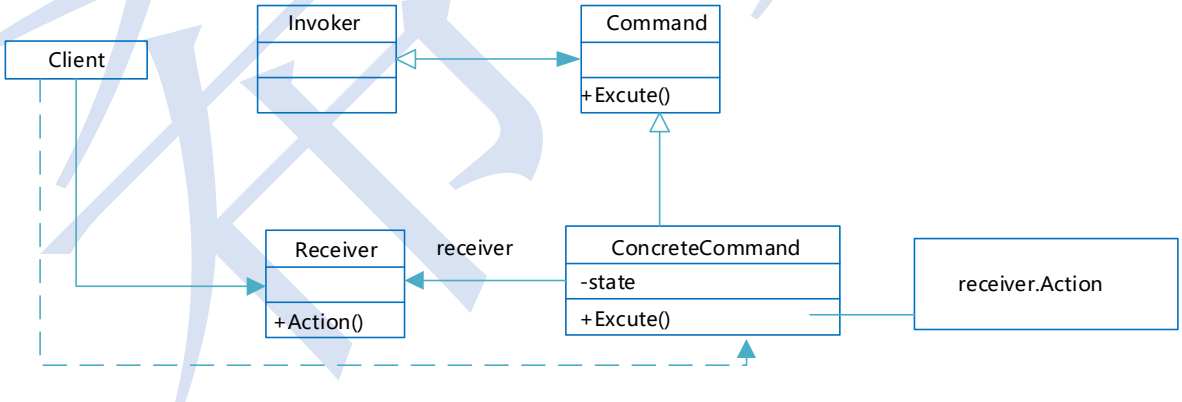
（3）适用场景（传递职责）

- 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 想在不明确指定接收者的情况下向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

14、命令（Command）模式

（1）命令模式的意图是：将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作。

（2）类图



- Command：抽象命令类，声明执行操作的一个接口。
- ConcreteCommand：具体命令类将一个接收者对象绑定于一个动作。实现 execute 方法，以调用接收者的相关操作（Action）。
- Invoker：调用者，要求一个命令对象执行一个请求。
- Receiver：接收者，知道如何执行关联请求的相关操作。
- Client：客户应用程序，创建一个具体命令类对象，并且设定它的接收者。

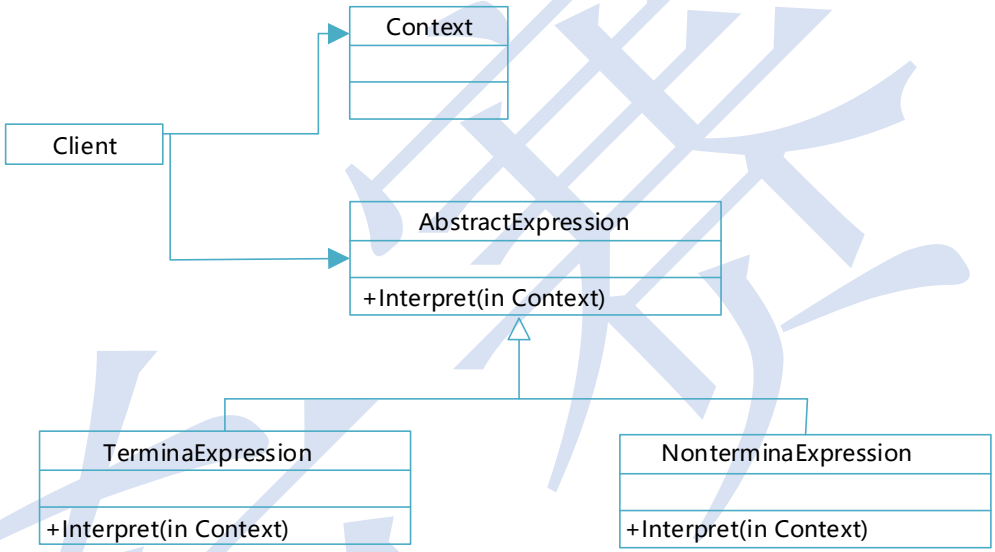
（3）适用场景（日志记录，可撤销）

- 抽象出待执行的动作以参数化某对象。
- 在不同的时刻指定、排列和执行请求。
- 支持取消操作。
- 支持修改日志。
- 用构建在原语操作上的高层操作构造一个系统。

15、解释器（Interpret）模式

（1）解释器模式的意图是：给定一种语言，定义它的文法表示，并定义一个解释器，该解释器用来根据文法表示来解释语言中的句子。

（2）类图



- NonTerminalExpression：非终结符表达式。对文法中的每一条规则都需要一个 NonTerminalExpression 类；为每个符号都维护一个 AbstractExpression 类型的实例变量；为文法中的非终结符实现解释（Interpret）操作。
- Context：场景，包含解释器之外的一些全局信息。
- Client：客户程序，构建（或被给定）由这种语言表示的一个特定句子的抽象文法树，文法树由终结符表达式或非终结表达式的实例组成，调用解释操作。
- AbstractExpression：抽象表达式类，定义一个接口来执行解释操作，实现与文法中的元素相关联的解释操作。这个接口为抽象语法树中所有的结点所共享。
- TerminalExpression：终结符表达式，实现文法中关联终结符的解释操作。文句中的每个终结符都需要该类的一个实例。

（3）适用场景（虚拟机的机制）

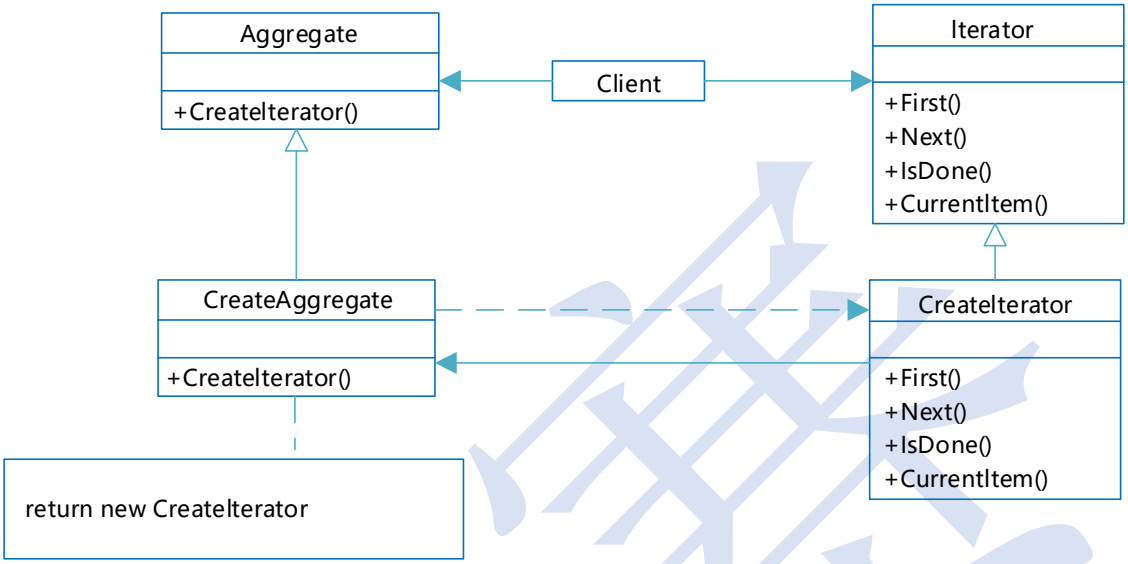
【解释器（Interpret）模式适用于当有一个语言需要解释执行，并且可将该语言中的句子表示为一个抽象语法树时，以下情况效果最好：】

- 该文法简单。
- 效率不是一个关键问题。

16、迭代器（Iterator）模式

（1）迭代器模式的意图是：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

（2）类图



- Iterator：迭代器，迭代器定义访问和遍历元素的接口。
- ConcreteIterator：具体迭代器，实现迭代器的接口，在遍历时跟踪当前聚合对象中的位置。
- Aggregate：聚合，定义一个创建迭代器对象的接口。
- ConcreteAggregate：具体聚合，实现创建迭代器对象，返回一个具体迭代器的实例。

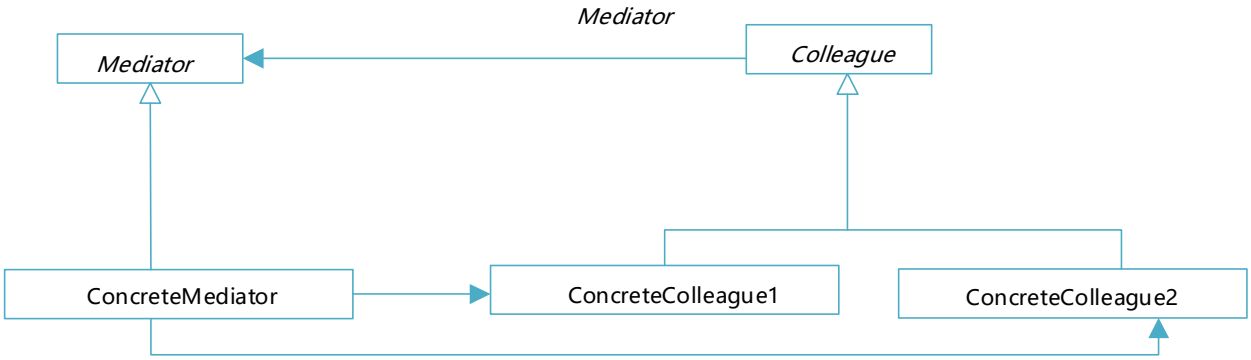
（3）适用场景（数据库数据集）

- 访问一个聚合对象的内容而无须暴露它的内部表示。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口。

17、中介者（Mediator）模式

（1）中介者模式的意图是：用一个中介对象来封装一系列的对象交互。它使各对象不需要显式地相互调用，从而达到低耦合，还可以独立地改变对象间的交互。

（2）类图



- Mediator：抽象中介者，定义统一的接口用于各同事（Colleague）对象间的通信。
- ConcreteMediator：具体中介者，协调各个同事对象实现协作的行为，掌握并且维护它的各个同事对象引用。
- Colleague：同事类，每一个同事角色都知道对应的具体中介者角色，而且与其他的同事角色通信的时候，一定要通过中介者角色协作。

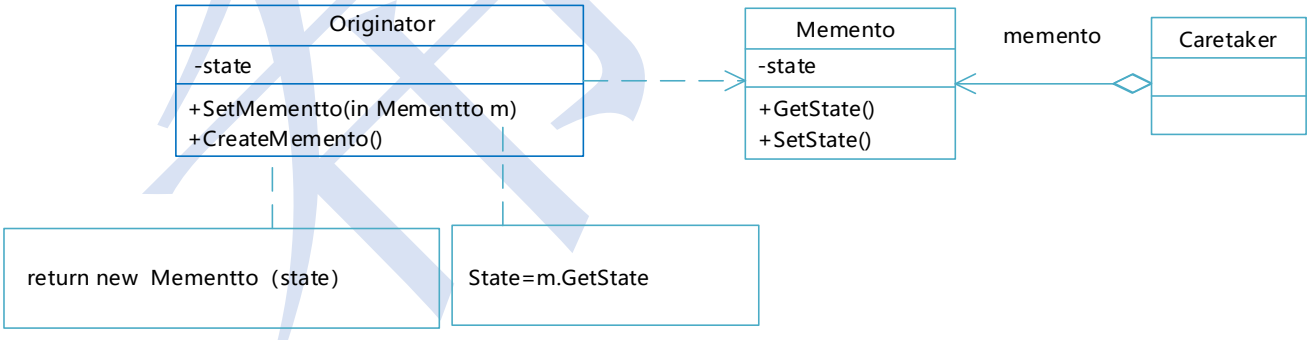
（3）适用场景（不直接引用）

- 一组对象以定义良好但是复杂的方式进行通信，产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象。
- 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

18、备忘录（Memento）模式

（1）备忘录模式的意图是：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，从而可以在以后将该对象恢复到原先保存的状态。

（2）类图



- Memento：备忘录对象，保持 Originator（原发器）的内部状态，原发器根据需要来决定保存哪些内部的状态。防止原发器之外的对象访问备忘录。
- Originator：原发器，通常是需要备忘的对象自己，创建一个备忘录，记录它的当前内部状态。可以利用一个备忘录来恢复它的内部状态。
- CareTaker：备忘录管理者，只负责看管备忘录，不可以对备忘录的内容操作或者检查。
- 备忘录可以有效地利用两个接口。看管者只能调用狭窄（功能有限）的接口——它只能传递备忘录对象给其他对象。而原发器能调用一个宽接口（功能强大的接口），通过这个接口可以访问所有需要的

数据，使原发器可以返回原先的状态。理想的情况是，只允许生成本备忘录那个原发器访问本备忘录的状态。

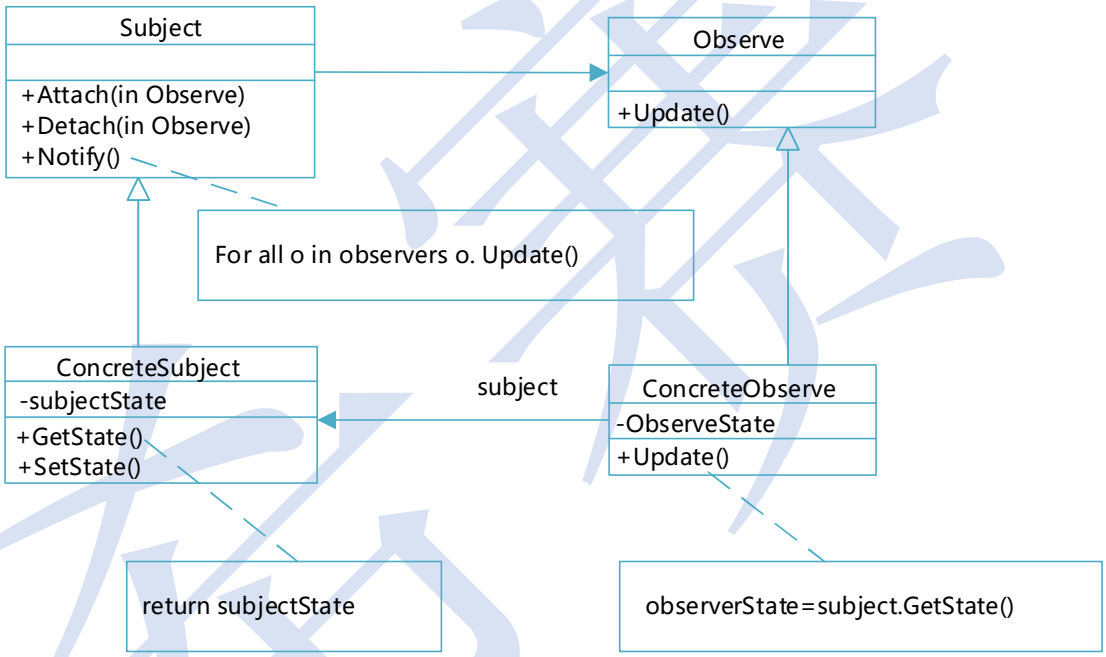
（3）适用场景（可恢复-数据库备份）

- 必须保存一个对象在某一时刻的（部分）状态，这样以后需要时它才能恢复到先前的状态。
- 如果一个用接口来让其他对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

19、观察者（Observer）模式

（1）观察者模式的意图是：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。

（2）类图



- Observer：观察者，定义一个更新接口，在一个被观察对象改变时应被通知。
- Subject：被观察对象（目标），了解其多个观察者，任意数量的观察者可以观察一个对象，提供一个接口用来绑定（注册）以及分离（删除）观察者对象。
- Concrete Subject：具体被观察对象，存储具体观察者 Concrete Observer 有兴趣的状态。当其状态改变时，发送一个通知给其所有的观察者对象。
- Concrete Observer：具体观察者，维护一个指向 Concrete Subject 对象的引用。存储有关状态，这些状态应与目标的状态保持一致；实现 Observer 的更新接口，以使自身状态与目标的状态保持一致。

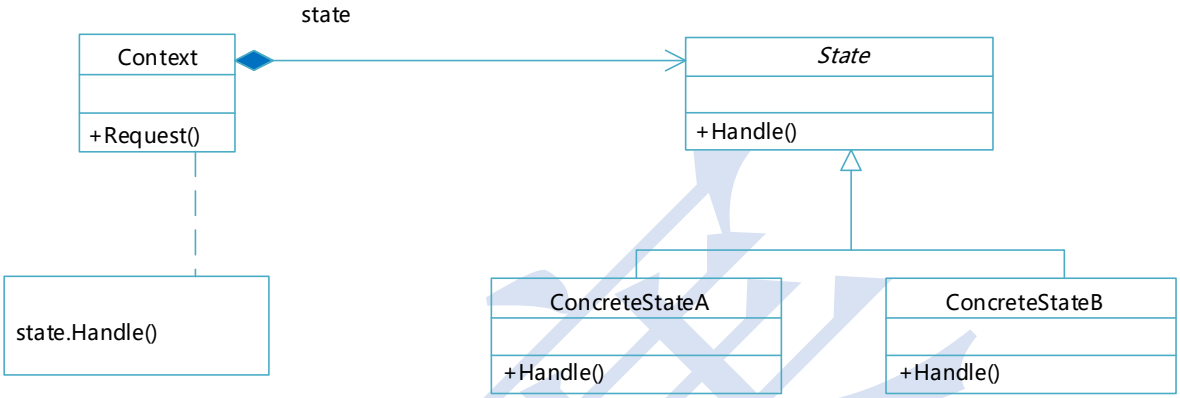
（3）适用场景（联动）

- 当一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两者封装在独立的对象中以使它们可以各自独立地改变和复用。
- 当对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象有待改变时。
- 当一个对象必须通知其他对象，而它又不能假定其他对象是谁，即不希望这些对象是紧耦合的。

20、状态（State）模式

（1）状态模式的意图是：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。

（2）类图



- Context：（上下文）情景类,定义客户应用程序有兴趣的接口,维护一个 Concrete State（具体状态）子类的实例对象，这个实例定义当前状态。
- State：抽象状态类,定义一个接口用来封装与 Context 的一个特定状态相关的行为。
- Concrete State：具体状态类,每一个具体状态类（Concrete State）实现了一个 Context 的一个状态相关的行为。

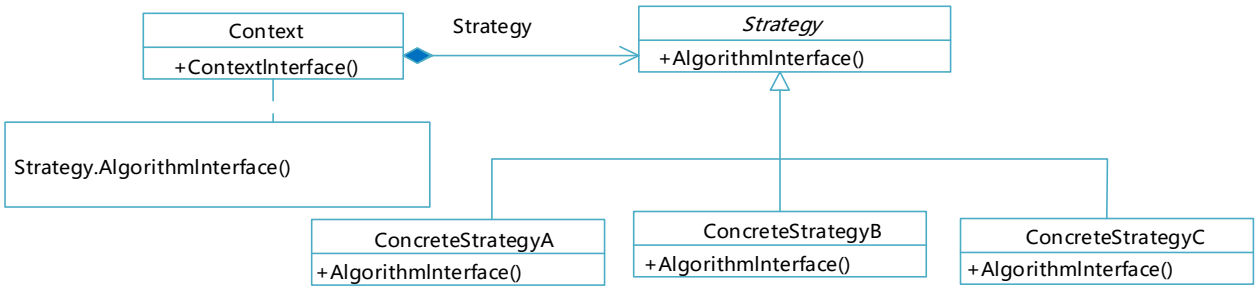
（3）适用场景（状态变成类）

- 一个对象的行为决定于它的状态，并且它必须在运行时刻根据状态改变它的行为。
- 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态常用一个或多个枚举常量表示。通常，有多个操作包含这一相同的条件结构。State 模式将每一个条件分支放入一个独立的类中。这使得开发者可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象独立变化。

21、策略（Strategy）模式

（1）策略模式的意图是：定义一系列算法，把它们一个个封装起来，并且使它们之间可互相替换，从而让算法可以独立于使用它的用户而变化。

（2）类图



- Strategy：抽象策略类,定义一个公共接口给所有支持的算法，Context 使用这个接口调用 ConcreteStrategy 定义的算法。
- ConcreteStrategy：具体策略类,以 Strategy 接口实现某具体算法。
- Context（上下文）：用一个 ConcreteStrategy 对象配置其执行环境；维护一个对 Strategy 对象的引用；可定义一个接口来让 Strategy 访问它的数据。

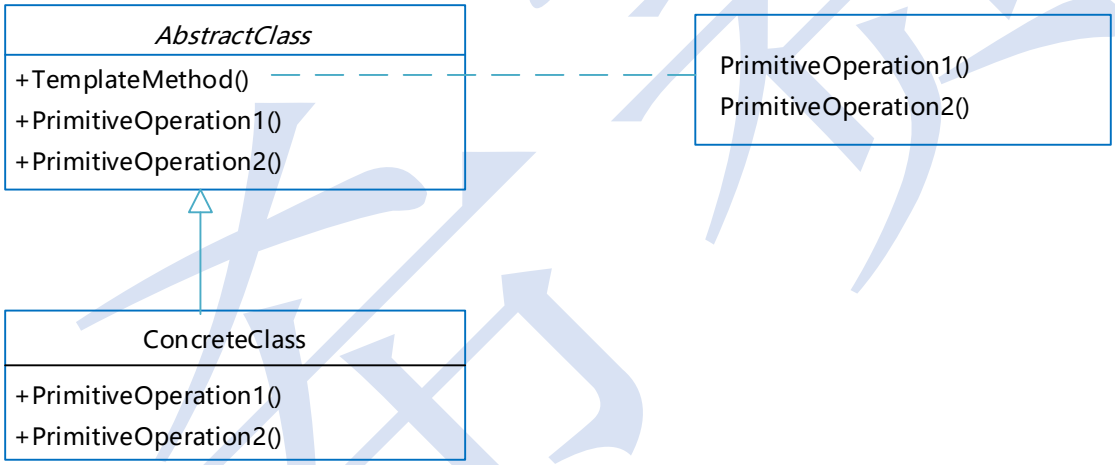
（3）适用场景（多方案切换）

- 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。
- 需要使用一个算法的不同变体。
- 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。
- 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，将相关的条件分支移入它们各自的 Strategy 类中，以替代这些条件语句。

22、模板方法（Template Method）模式

（1）模板方法模式的意图是：定义一个操作中的算法骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤。

（2）类图



- AbstractClass：抽象类,定义一个抽象的原语操作，其具体的子类可以重新定义它以实现一个算法的各个步骤。实现一个模板方法，定义一个算法的骨架，此模板方法不仅可以调用原语操作，还可以调用定义于 AbstractClass 或者其他对象中的操作。
- ConcreteClass：具体子类,实现原语操作以完成算法中与特定子类相关的算法步骤。

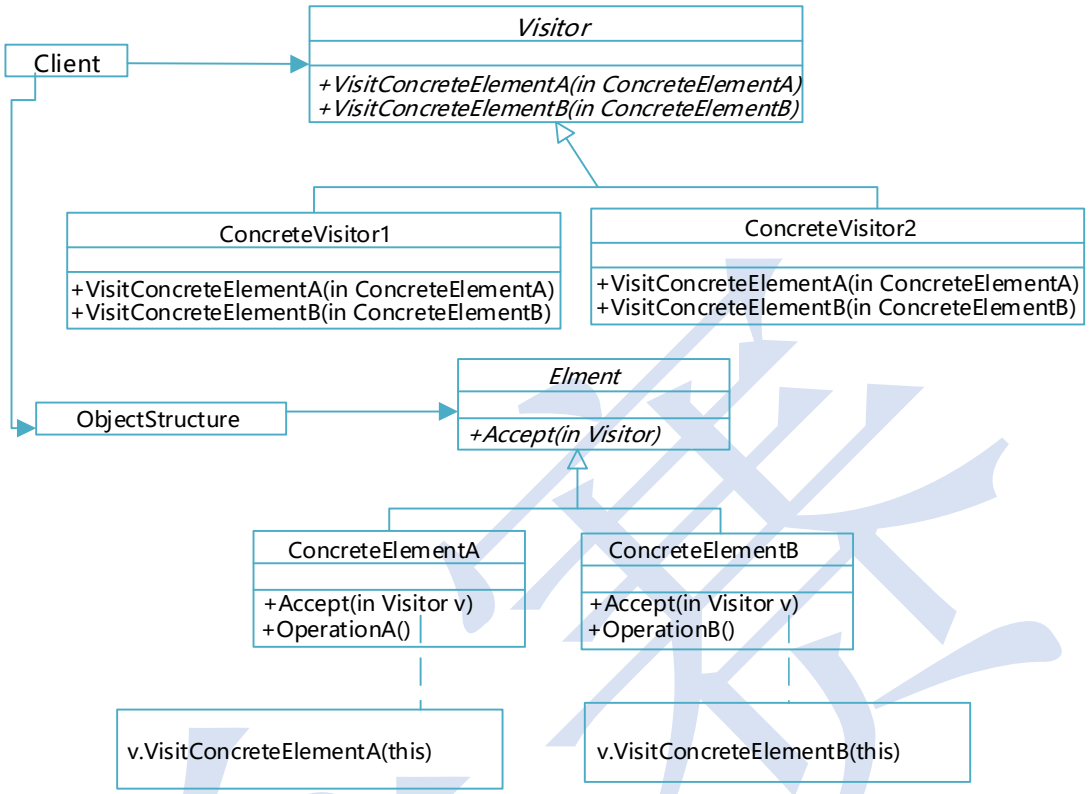
（3）适用场景（文档模板填空）

- 一次性实现一个算法的不变的部分，并将可变的行为留给子类实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中，以避免代码重复。
- 控制子类扩展。模板方法旨在特定点调用“hook”操作（默认行为，子类可以在必要时进行重定义扩展），这就只允许在这些点进行扩展。

23、访问者（Visitor）模式

（1）访问者模式的意图是：表示一个作用于某对象结构中的各元素的操作，使得在不改变各元素的类的前提下定义作用于这些元素的新操作。

（2）类图



- Visitor：抽象访问者，为对象结构类中每一个 ConcreteElement 的每一个类声明一个 Visit 操作。
- ConcreteVisitor：具体访问者，实现每个由 Visitor 声明的操作，每个操作实现本算法的一部分，而该算法片段乃是对应结构中对象的类。
- Element：元素，定义一个 Accept 操作，它以一个访问者为参数。
- ConcreteElement：具体元素，实现 Accept 操作，该操作以一个访问者为参数。
- ObjectStructure：对象结构。能枚举它的元素；可以提供一个高层的接口，以允许该访问者访问它的元素；可以是一个组合或者一个集合，如列表或一个无序集合。

（3）适用场景（数据与操作分离）

- 一个对象结构包含很多类对象，它们有不同的接口，而用户想对这些对象实施一些依赖于其具体类的操作。
- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而又想要避免这些操作“污染”这些对象的类。
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。