私塾在线《设计模式综合项目实战》

——跟着CC学设计系列精品教程

本阶段课程概览

- 设计并实现核心框架中的分发调度模块

一：分发调度模块的详细功能

二：分发调度模块的功能边界

三：分发调度模块对外的接口

四：分发调度模块的内部实现

包括：外观模式、命令模式、职责链模式的综合应用

五：分发调度模块和其他模块的交互实现

主要是使用中介者模式

分发调度模块的详细功能

- 提供操作生成器的调用入口，接受客户端的请求来触发生成器运行
  （1）提供一个默认的，无参的接口，表明一切按照配置文件进行生成
  （2）考虑提供一个可以指定生成参数的接口

- 然后按照生成配置来组合需要生成的功能
  不同的生成配置，需要生成的东西是不一样的，比如一个模块需要生成表现层、逻辑层和数据层，而另外一个模块可能只需要逻辑层和数据层，因此需要能够灵活的组合需要生成的功能。

- 然后进行分发调度，发出执行生成的命令，让后续模块去真正实现功能
  为了让程序结构更为松散耦合，灵活度更高，在分发调度模块并不真正去执行生成，它只是发出执行生成的命令，具体的工作由其他模块去完成。

分发调度模块的功能边界

- 分发调度模块的功能边界

1：分发调度模块主要负责接收客户端的请求，然后根据配置来组合需要生成的功能，然后把需要生成具体功能的命令分发出去就可以了。

2：分发调度模块本身不负责配置数据的获取或者解析，它只是向配置管理模块去获取它需要的数据。具体的配置数据由使用人员进行配置，然后由配置管理模块去读取、解析并进行管理。

3：分发调度模块本身并不知道每个模块需要生成些什么，它只是从配置数据中获取相应的数据，然后按照这些数据来组合相应的功能。具体每个模块要生成什么，是由使用人员在配置里面指定的。

4：分发调度模块本身并不知道如何去实现用户要求的生成功能，因此它只是负责把命令分发出去，具体谁来生成、如何生成、生成成什么东西，它都管不着了。

分发调度模块的对外接口

- 对外程序接口设计

分发调度模块的内部实现

- 实现的起点

为了让大家更好的理解分发调度模块的内部实现架构，因此先以一个最简单的实现结构为起点，采用重构的方式，逐步把相关的设计模式应用进来，从简单到复杂，从而让大家更好的看到如何选择要使用的设计模式、如何实际应用设计模式以及如何让多种设计模式协同工作。

1：针对前面定义的API，提供一个最基本的实现。

加入外观模式

- 面临的问题

  假定系统内部的各个模块功能都已经实现好了，而客户是不需要和系统内部的各个模块交互的。

  也就是说，客户端希望以一种简单的方式来和x-gen系统进行交互，而且客户端并不希望了解系统内部的细节。

  该怎么办呢？

- 用外观模式来解决

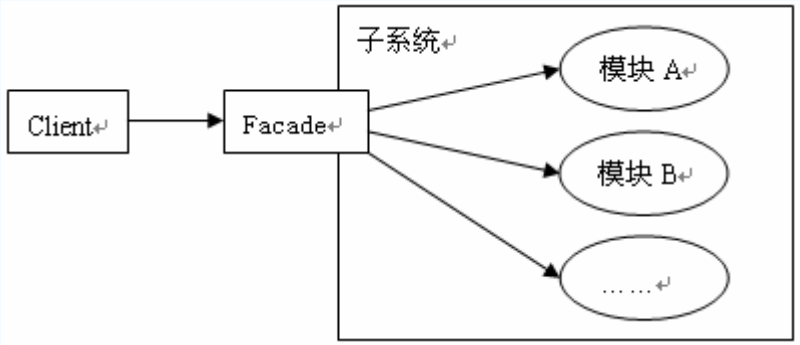
- 外观模式基础回顾

初识外观模式

- 定义

  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。

- 结构和说明



Facade：定义子系统的多个模块对外的高层接口，通常需要调用内部多个模块，从而把客户的请求代理给适当的子系统对象。

模块：接受Facade对象的委派，真正实现功能，各个模块之间可能有交互。
注意，Facade对象知道各个模块，但是各个模块不应该知道Facade对象。

外观模式的知识要点

- 外观模式的知识要点

1：首先要正确理解界面和接口在这里的含义。这里的界面主要指的是从一个组件外部来看这个组件，能够看到什么，这就是这个组件的界面。而这里的接口并不是指Java的interface，而是指的外部和内部交互的这么一个通道，通常是指的一些方法，可以是类的方法，也可以是interface的方法。也就是说，这里说的接口，并不等价于interface，也有可能是一个类

2：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的使用子系统

3：外观是位于提供外观的模块内部的，也就是模块对外提供的外观

4：外部可以绕开Facade，直接调用某个具体模块的接口，这样就能实现兼顾组合功能和细节功能

5：外观通常实现成为类，提供缺省的功能实现，当然外观也可以实现成为interface

6：外观里面一般也不去真正实现功能，而是组合调用模块内部已有的实现，外观只是为了给客户端一个简洁的接口，并不是要外观自己来实现相应的功能

思考外观模式

- 外观模式的本质

  外观模式的本质是：**封装交互，简化调用**

- 何时选用外观模式

1：如果你希望为一个复杂的子系统提供一个简单接口的时候，可以考虑使用外观模式，使用外观对象来实现大部分客户需要的功能，从而简化客户的使用

2：如果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，使用外观对象来将这个子系统与它的客户分离开来，从而提高子系统的独立性和可移植性

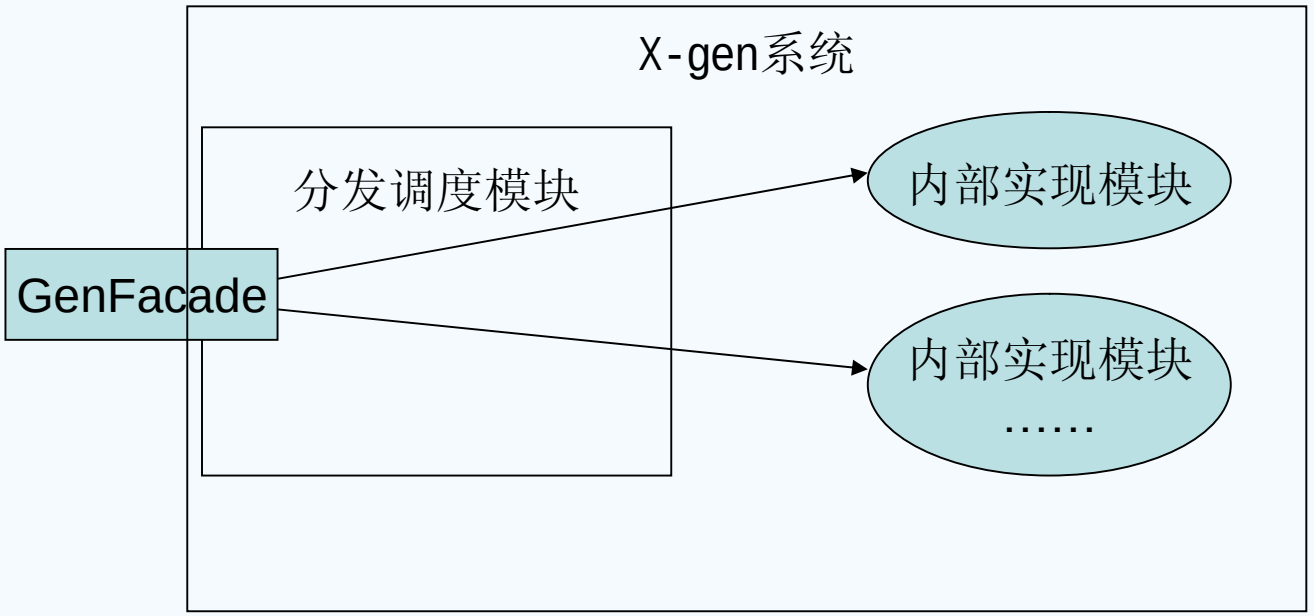
3：如果构建多层结构的系统，可以考虑使用外观模式，使用外观对象作为每层的入口，这样可以简化层间调用，也可以松散层次之间的依赖关系

应用外观模式

- 使用外观模式来解决问题的思路

  通过给客户端提供一个外观对象，客户端就只需要和这个外观对象交互，而这个外观对象负责与系统内部的各个模块交互。

- 此时分发调度模块的结构示意如图



加入中介者模式

- 面临的问题

  分发调度模块，需要与其它模块交互，有可能会有较为复杂的交互。如果直接让分发模块去调用这些模块，可能会导致较强的耦合性，该如何处理呢？

- 用中介者模式来解决

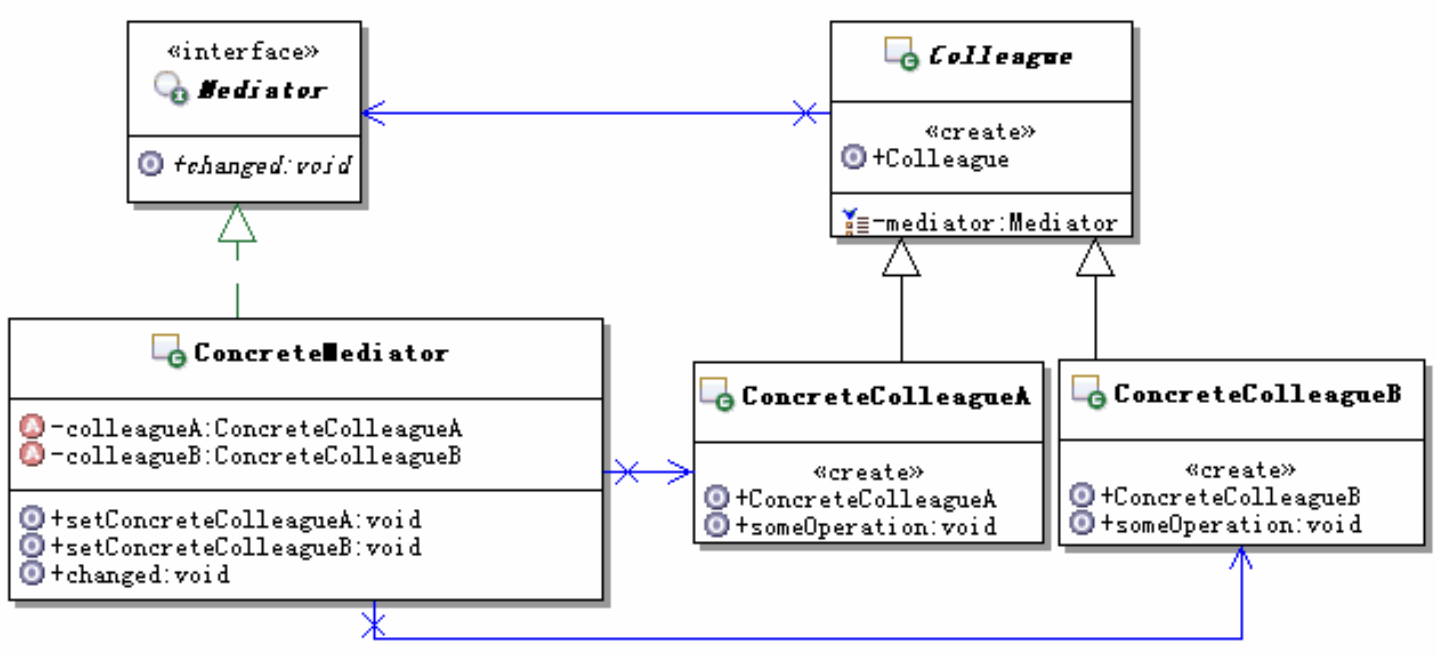
- 中介者模式基础回顾

初识中介者模式

- 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式地相互引用，从而使其耦合松散，而且可以独立的改变它们之间的交互。

- 结构和说明



中介者模式的知识要点

- 中介者模式的知识要点

1：中介者主要用来封装对象之间的交互，把对象之间的交互全部集中到中介者对象里面，所有的对象就只是跟中介者对象进行通信，相互之间不再有联系，从而松散对象间的耦合，并对交互关系进行统一的管理

2：在实现中介者的时候，如果中介者的实现只有一种，而且今后也没有扩展的需要，那么可以不要Mediator

3：同事和中介者必须有关系，首先是同事对象需要知道中介者对象是谁；反过来，中介者对象也需要知道相关的同事对象，这样它才能与同事对象进行交互。也就是说中介者对象和同事对象之间是相互依赖的

4：同事和中介者通信的方式，通常的实现方式，一种是在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；另一种实现方式是可以采用观察者模式，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

5：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，我们称其为广义中介者模式

思考中介者模式

- 中介者模式的本质

  中介者模式的本质是：**封装交互**

- 何时选用中介者模式

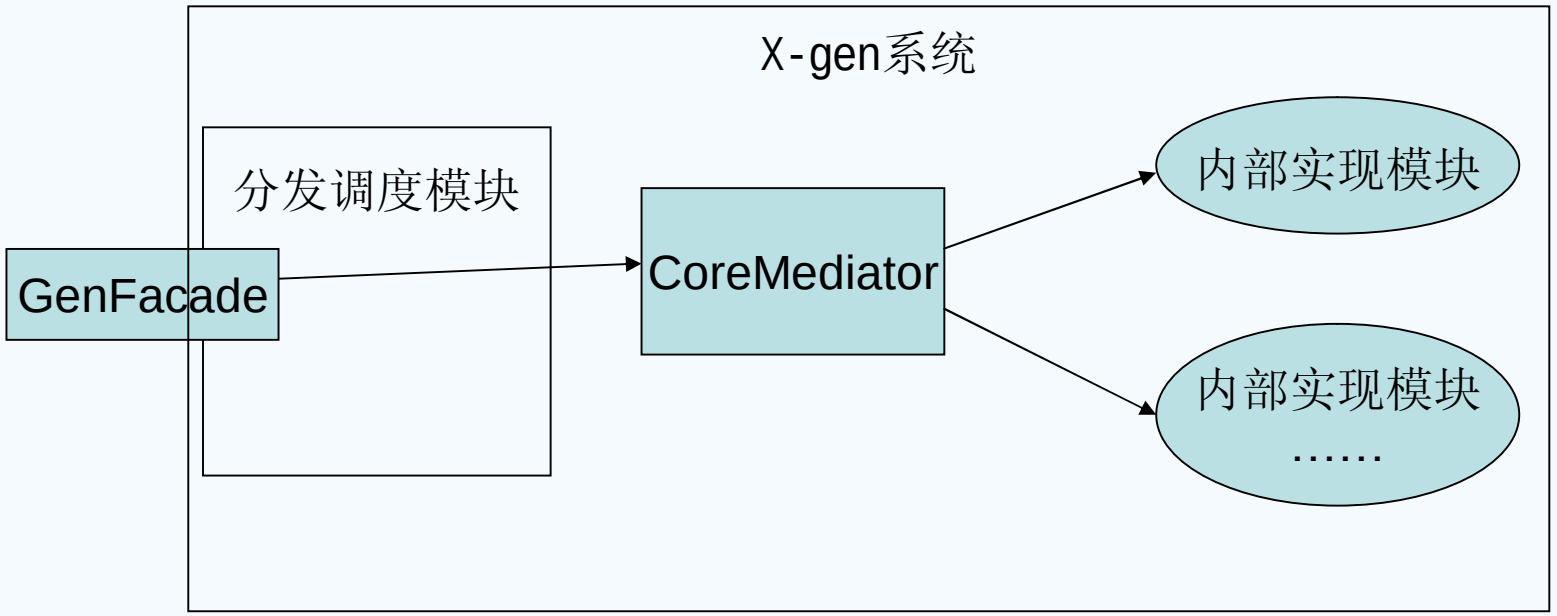
1：如果一组对象之间的通信方式比较复杂，导致相互依赖、结构混乱，可以采用中介者模式，把这些对象相互的交互管理起来，各个对象都只需要和中介者交互，从而使得各个对象松散耦合，结构也更清晰易懂

2：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。可以采用中介者模式，把这个对象跟其它对象的交互封装到中介者对象里面，这个对象就只需要和中介者对象交互就可以了

应用中介者模式

- 使用中介者模式来解决问题的思路

  引入一个中介者对象，所有的模块都和他进行交互，模块之间不再相互调用，所有模块间的关系都是松散耦合的。
- 此时分发调度模块的结构示意如图



加入命令模式

- 面临的问题

  接着来看，按照上面的实现，现在在外观类里面，需要去调用真正的实现，因为分发调度模块本身是不负责具体实现的，那么现在的问题是：到底外观类要如何去调用具体的实现呢？

  有朋友会说，直接调用具体实现的对象不就可以了，好像听起来是这么一回事情。但仔细想想，如果调用的顺序比较复杂，或者是调用的过程本身并不是固定的呢？那么直接调用就会有问题了，另外，如果调用的程序不仅仅在本地，也有可能在远程呢？

- 用命令模式来解决

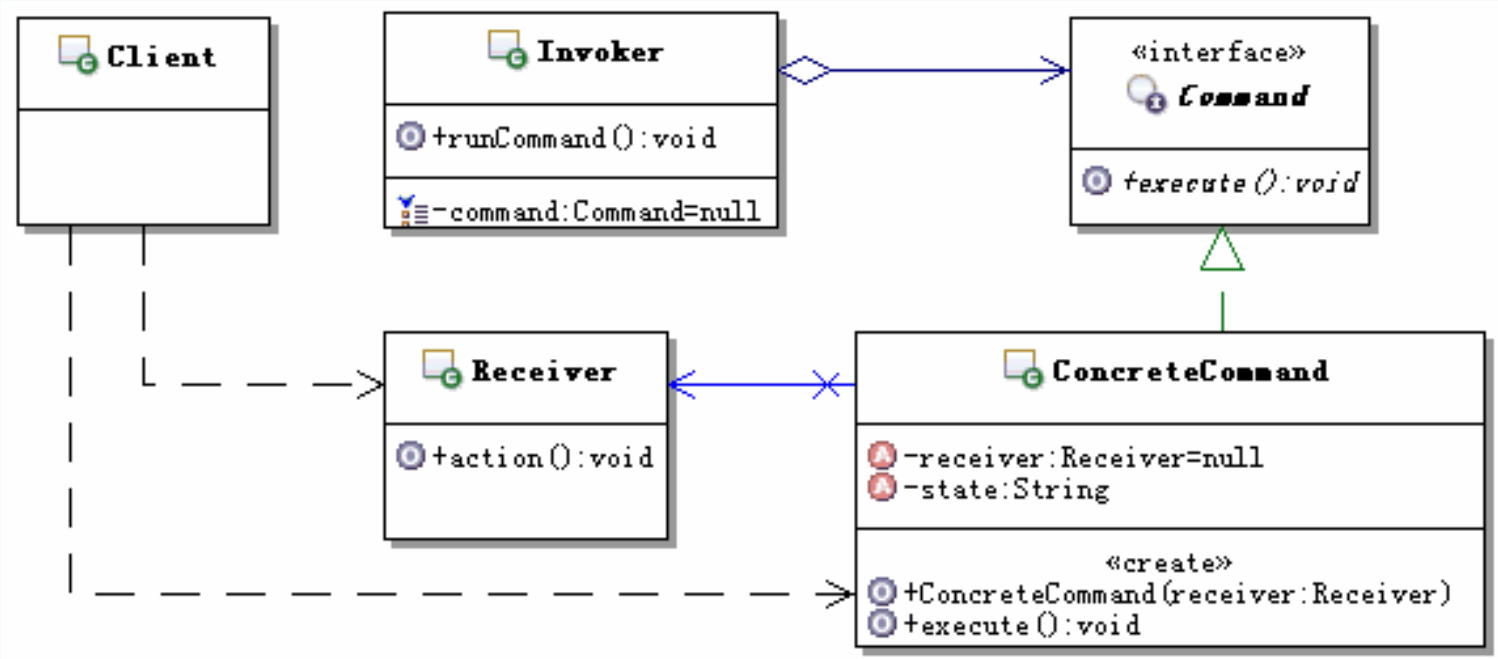
- 命令模式基础回顾

初识命令模式

- 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。

- 结构和说明



命令模式的知识要点

- 命令模式的知识要点

1：命令模式的关键之处就是把请求封装成为对象，也就是命令对象，并定义了统一的执行操作的接口，这个命令对象可以被存储、转发、记录、处理、撤销等，整个命令模式都是围绕这个对象在进行

2：在命令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

3：命令模式的接收者可以是任意的类，只要完成命令要求的功能即可

4：可以有智能的命令，也就是不需要接受者，命令的实现类就真正实现了命令要求的功能

5：命令模式实现了命令的发起方和真正实现对象之间的解耦

6：命令的参数化配置的意思是：可以用不同的命令对象，去参数化配置客户的请求

7：命令模式实现可撤销操作的常见方式：一种是补偿式；另一种是存储恢复式

8：多个命令还可以组合在一起，形成宏命令，宏命令从本质上来说仍然是命令，运行宏命令，其实就是循环运行一个宏命令内的每一个命令

9：命令模式可以退化成为类似于Java回调的实现形式

思考命令模式

- 命令模式的本质

  命令模式的本质是：**封装请求**

- 何时选用命令模式

1：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，把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

2：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，把这些请求封装成为命令对象，然后实现把请求队列化

3：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，能很容易的实现命令的恢复和重做的功能

4：如果需要支持当系统崩溃时，能把对系统的操作功能重新执行一遍，可以选用命令模式，把这些操作功能的请求封装成命令对象，然后实现日志命令，就可以在系统恢复回来后，通过日志获取命令列表，从而重新执行一遍功能

5：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。

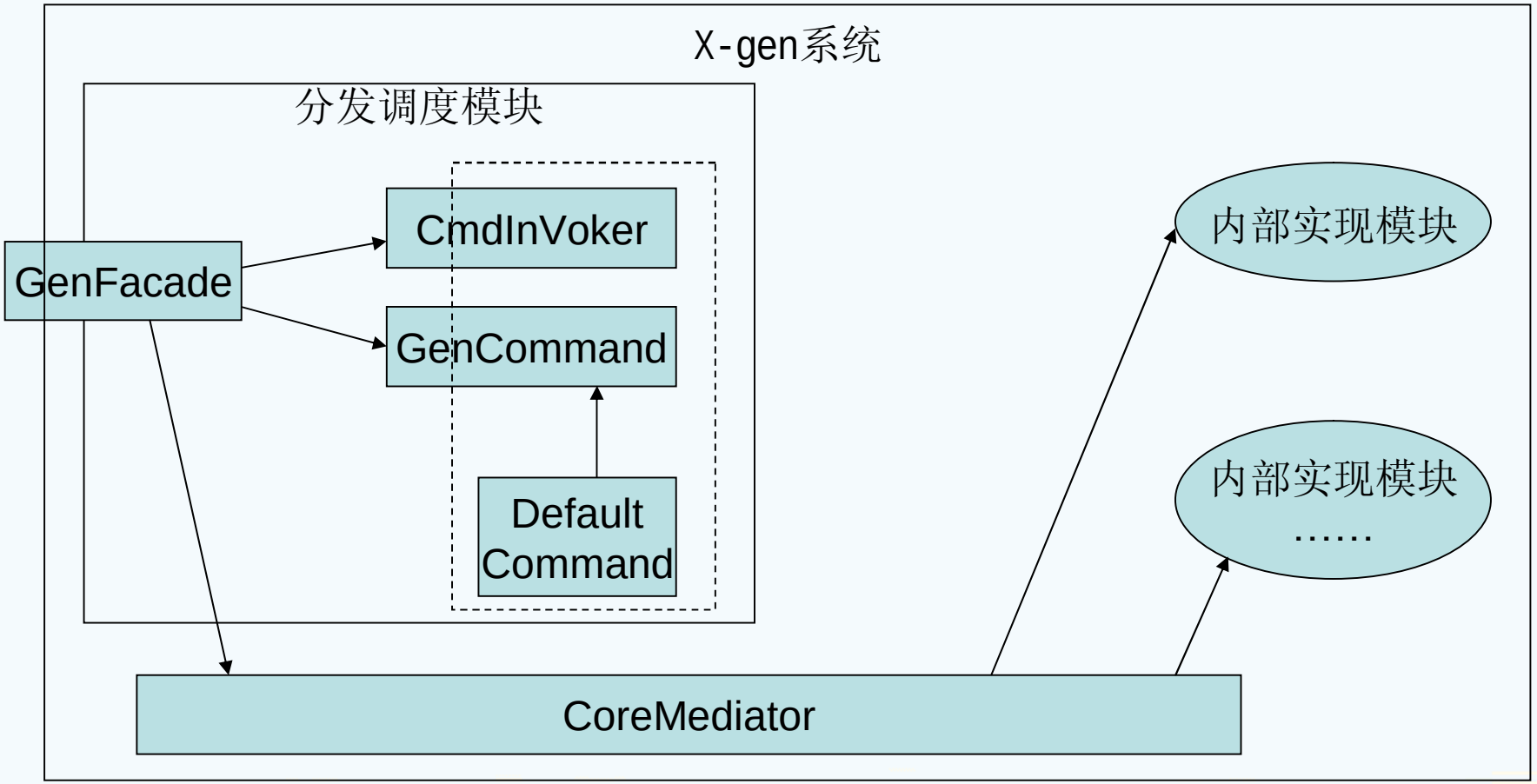
应用命令模式

- 使用命令模式来解决问题的思路

  要解决前面提到的问题，就应该让外观类和真正的实现分离开来，实现松散的耦合，从而可以灵活的组合。外观类里面只需要发出命令，其它的就不管了，当然在分发模块里面需要实现这个命令，但是这个实现只是命令的一个虚实现，而真正的实现当作命令的接收者对象。

应用命令模式

- 此时分发调度模块的结构示意如图



加入职责链模式

- 面临的问题

  接着来看实现，当外观类在循环多个模块的generate要求，然后发出命令，要求按照配置来生成一个模块的时候，出现了一个新的问题，那就是每个模块要求生成的具体功能是不一样的，比如：有些模块要求生成Ebi、Ebo，而有些模块要求生成DAO等等。

  那么该怎么解决这个问题呢？

- 用职责链模式来解决

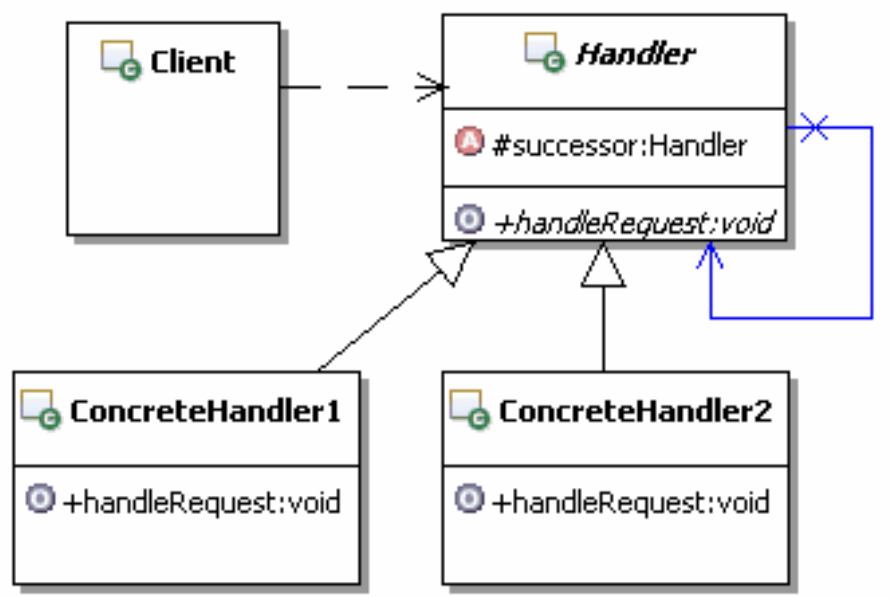
- 职责链模式基础回顾

初识职责链模式

- 定义

  使多个对象都有机会处理请求，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 。

- 结构和说明



职责链模式的知识要点

- 职责链模式的知识要点

1：职责链模式实现了请求者和具体的接收者解耦，这样就可以动态的切换和组合接收者了

2：客户端发出请求的时候，客户端并不知道谁会真正处理他的请求，客户端只知道他提交请求的第一个对象，然后这个请求会在整条链上传递，直到有人处理了这个请求为止

3：使用职责链模式，一个要注意的点就是谁来构建链，常见分为内部链和外部链，内部链又分在Handler里面构建链，还有一种是在各个职责里面指定后续的职责

4：在职责链模式中，请求不一定会被处理，因为可能没有合适的处理者

5：在实际开发中，经常会把标准的职责链变化成一个请求在职责链中传递，每个职责对象负责处理请求的某一方面的功能，处理完成后，不是停止，而是继续向下传递请求，当请求通过很多职责对象处理过后，功能也就处理完了，把这样的职责链称为功能链

6：动态组合才是职责链模式的精华所在，因为要实现请求对象和处理对象的解耦，请求对象不知道谁才是真正的处理对象，因此要动态的把可能的处理对象组合起来，由于组合的方式是动态的，这就意味着可以很方便的修改和添加新的处理对象，从而让系统更加灵活和具有更好的扩展性

思考职责链模式

- 职责链模式的本质

  职责链模式的本质是：**分离职责，动态组合**

- 何时选用职责链模式

1：如果有多个对象可以处理同一个请求，但是具体由哪个对象来处理该请求，是运行时刻动态确定的。这种情况可以使用职责链模式，把处理请求的对象实现成为职责对象，然后把它们构成一个职责链，当请求在这个链中传递的时候，具体由哪个职责对象来处理，会在运行时动态判断

2：如果你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的话，可以使用职责链模式，职责链模式实现了请求者和接收者之间的解耦，请求者不需要知道究竟是哪一个接收者对象来处理了请求 。

3：如果想要动态指定处理一个请求的对象集合，可以使用职责链模式，职责链模式能动态的构建职责链，也就是动态的来决定到底哪些职责对象来参与到处理请求中来，相当于是动态指定了处理一个请求的职责对象集合

应用职责链模式

- 使用职责链模式来解决问题的思路

  可以选用职责链来解决这个问题，当然不是用标准的职责链模式，而是使用变形的职责链，也就是功能链来解决这个问题。

  可以通过动态的组合功能来形成一个模块自己需要generate的功能链，然后要求执行这个功能链即可。

应用职责链模式

- 此时分发调度模块的结构示意如图