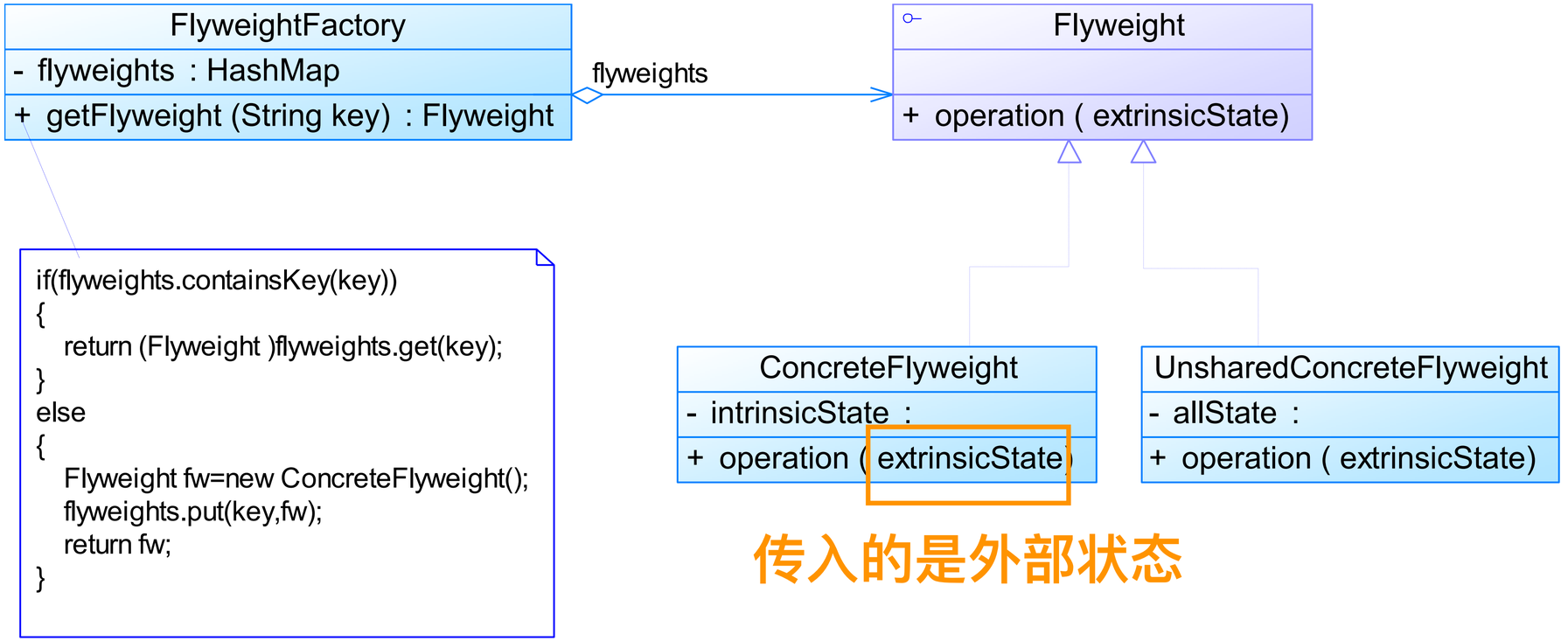
享元模式



https://mp.weixin.qq.com/s/P94B2PPS2nnYWRuWFfCDDQ

行为型模式

1. **职责链模式**：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止

2. **命令模式**：将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作

3. **解释器模式**：定义一个语言的文法，并且建立一个解释器来解释该语言中的句子，这里的“语言”是指使用规定格式和语法的代码

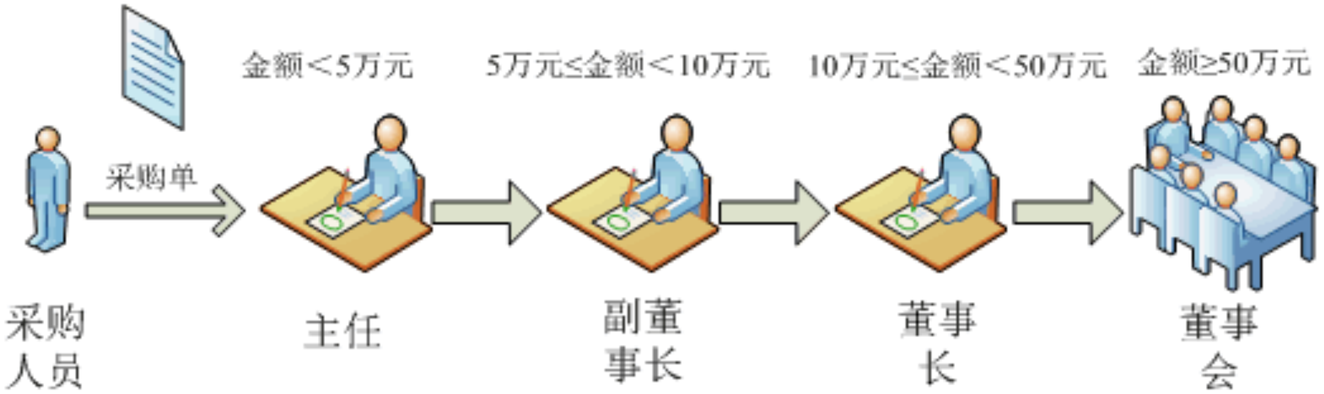
4. **迭代器模式**：提供一种方法来访问聚合对象，而不用暴露这个对象的内部表示，其别名为游标(Cursor)

5. **中介者模式**：用一个中介对象（中介者）来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互

6. **备忘录模式**：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态

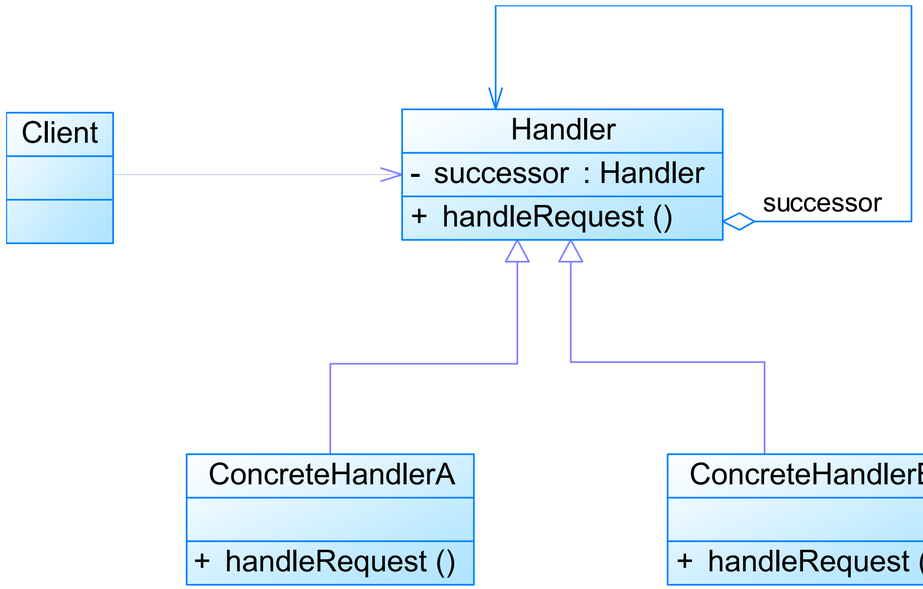
职责链模式

- if…else if….else if….
- 避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止



职责链模式

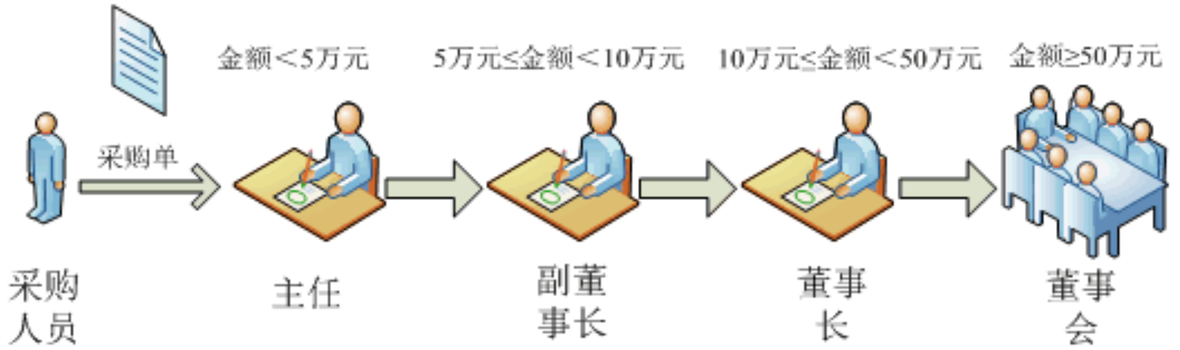
- 很多对象由每一个对象对其下家的引用而连接起来形成一条链
- 请求在这条链上传递，直到链上的某一个对象处理此请求为止
- 对客户端透明，系统可以在不影响客户端的情况下动态地重新组织链和分配责任




```
class ConcreteHandler extends Handler {
        public void handleRequest(String request) {
                if (请求满足条件) {
                        //处理请求
                }
                else {
                        this.successor.handleRequest(request);  //转发请求
                }
        }
}
```

Handler: 抽象处理者
ConcreteHandler: 具体处理者
Client: 客户类

职责链模式实例



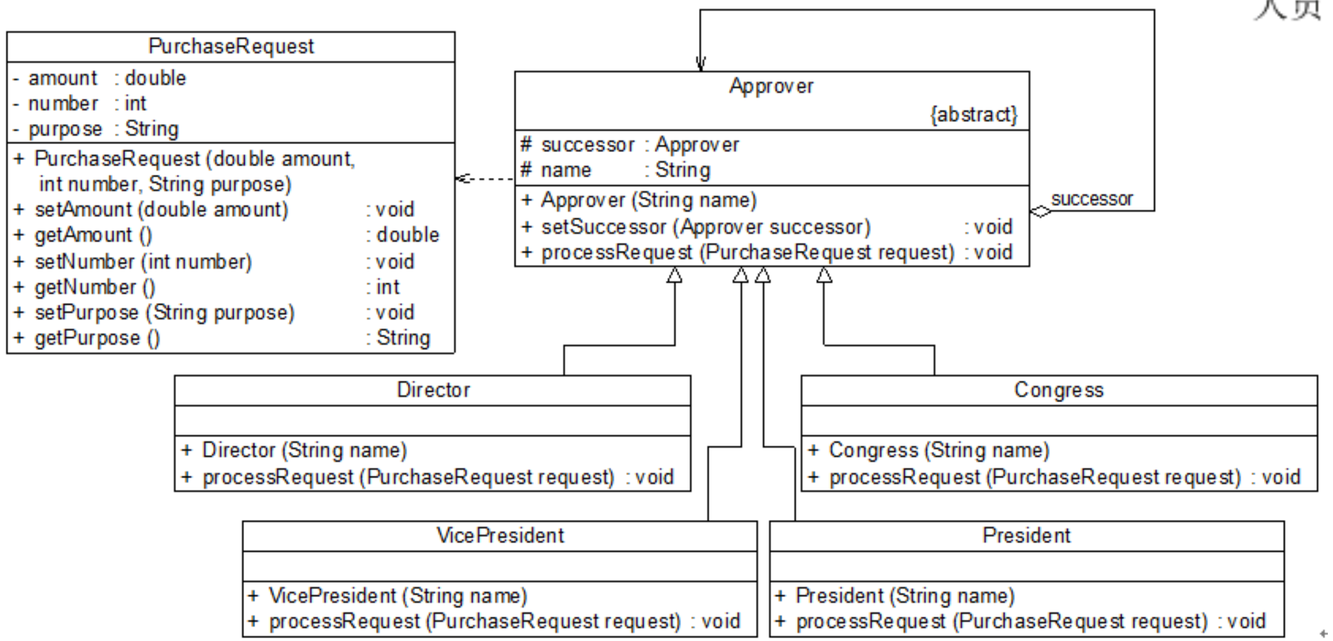


图 16-3 采购单分级审批结构图

```
//创建职责链
wjzhang.setSuccessor(rhuang); //将“黄蓉”作为“张无忌”的下家
rhuang.setSuccessor(gyang); //将“杨过”作为“黄蓉”的下家
gyang.setSuccessor(jguo);
jguo.setSuccessor(meeting);
```

职责链模式总结

优点

降低耦合度;

可简化对象的相互连接;

增强给对象指派职责的灵活性;

增加新的请求处理类很方便

缺点

不能保证请求一定被接收;

系统性能将受到一定影响，而且在进行代码调试时不太方便;

可能会造成循环调用

适用环境

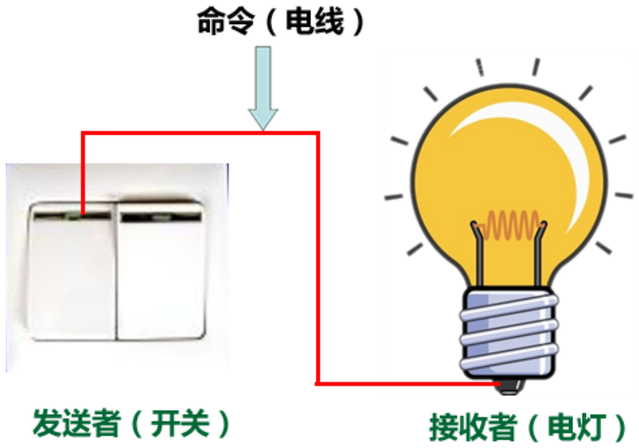
有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定;

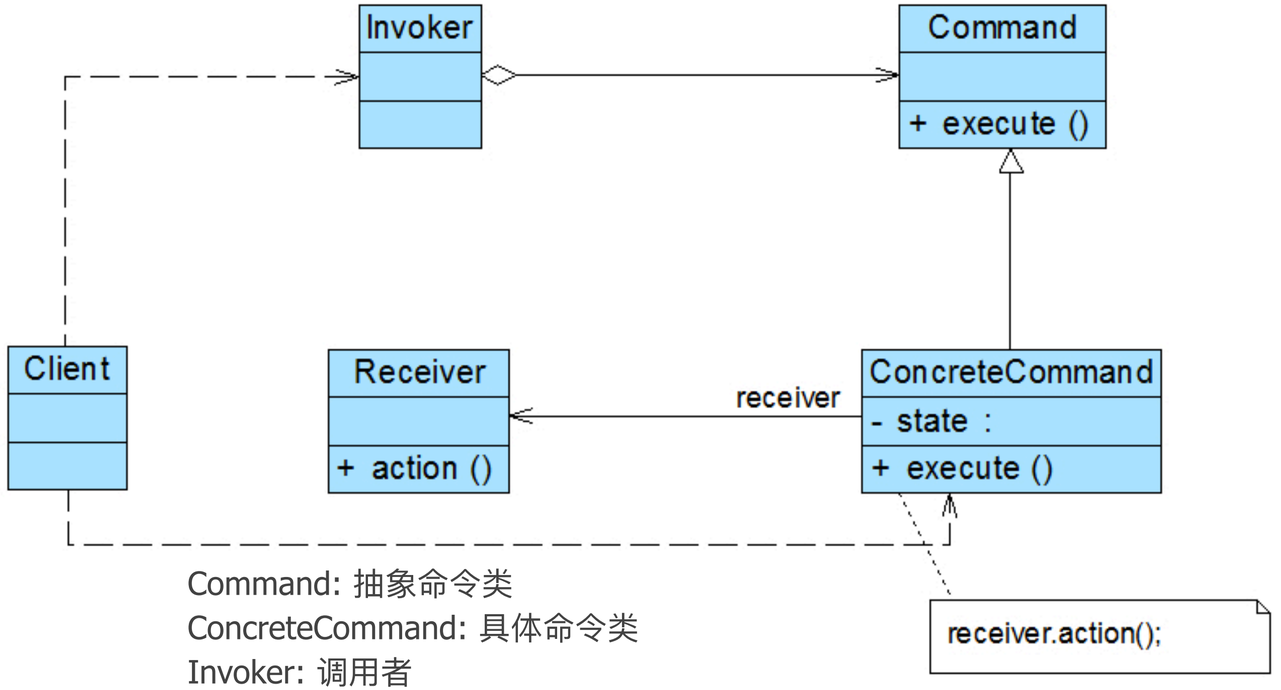
在不明确指定接收者的情况下，向多个对象中的一个提交一个请求;

可动态指定一组对象处理请求

命令模式

- 向某些对象发送请求（调用其中的某个或某些方法），但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个
- 将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作

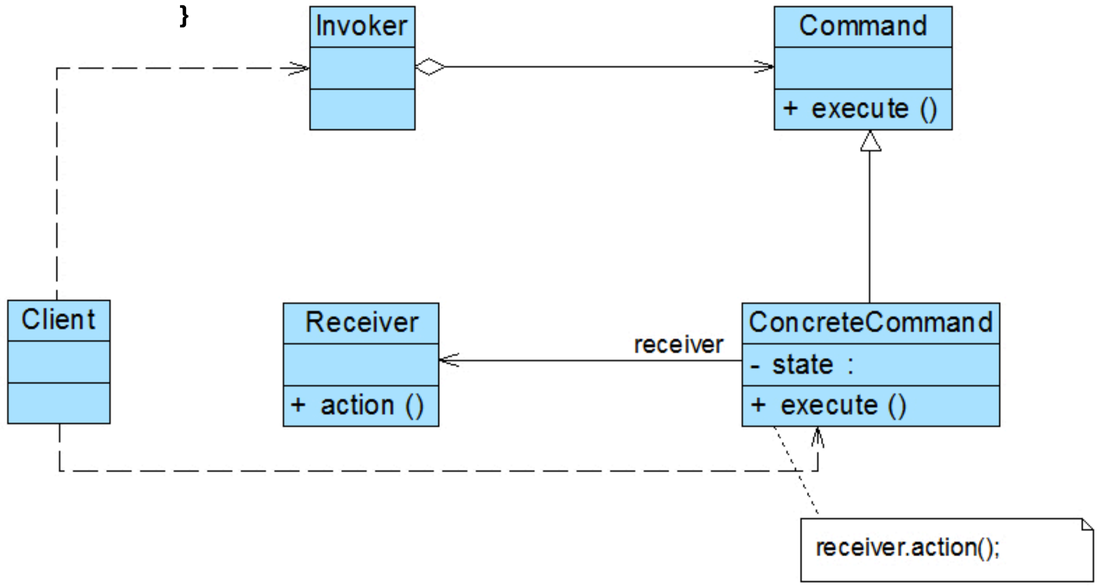


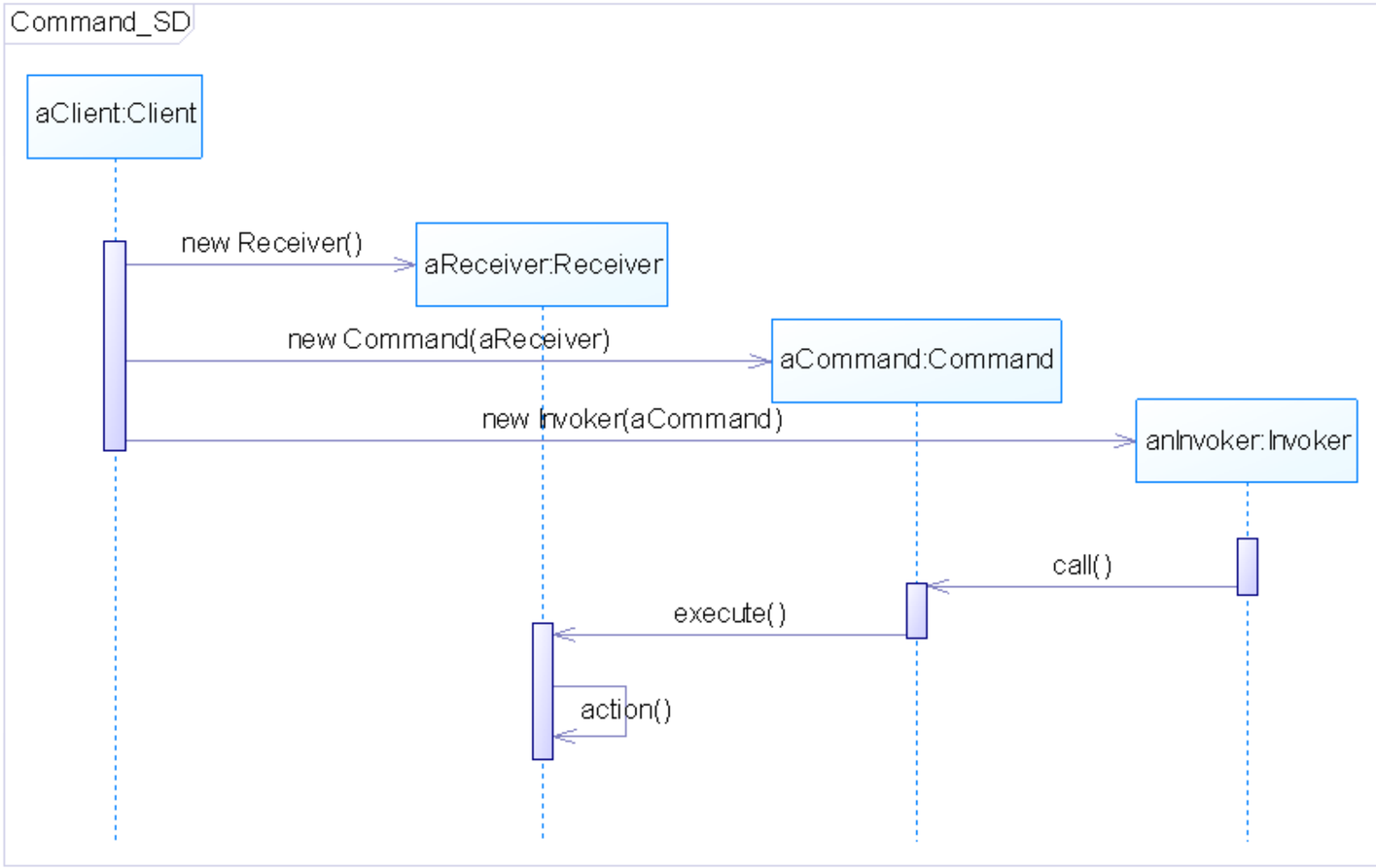


Command: 抽象命令类
ConcreteCommand: 具体命令类
Invoker: 调用者
Receiver: 接收者
Client:客户类

命令模式

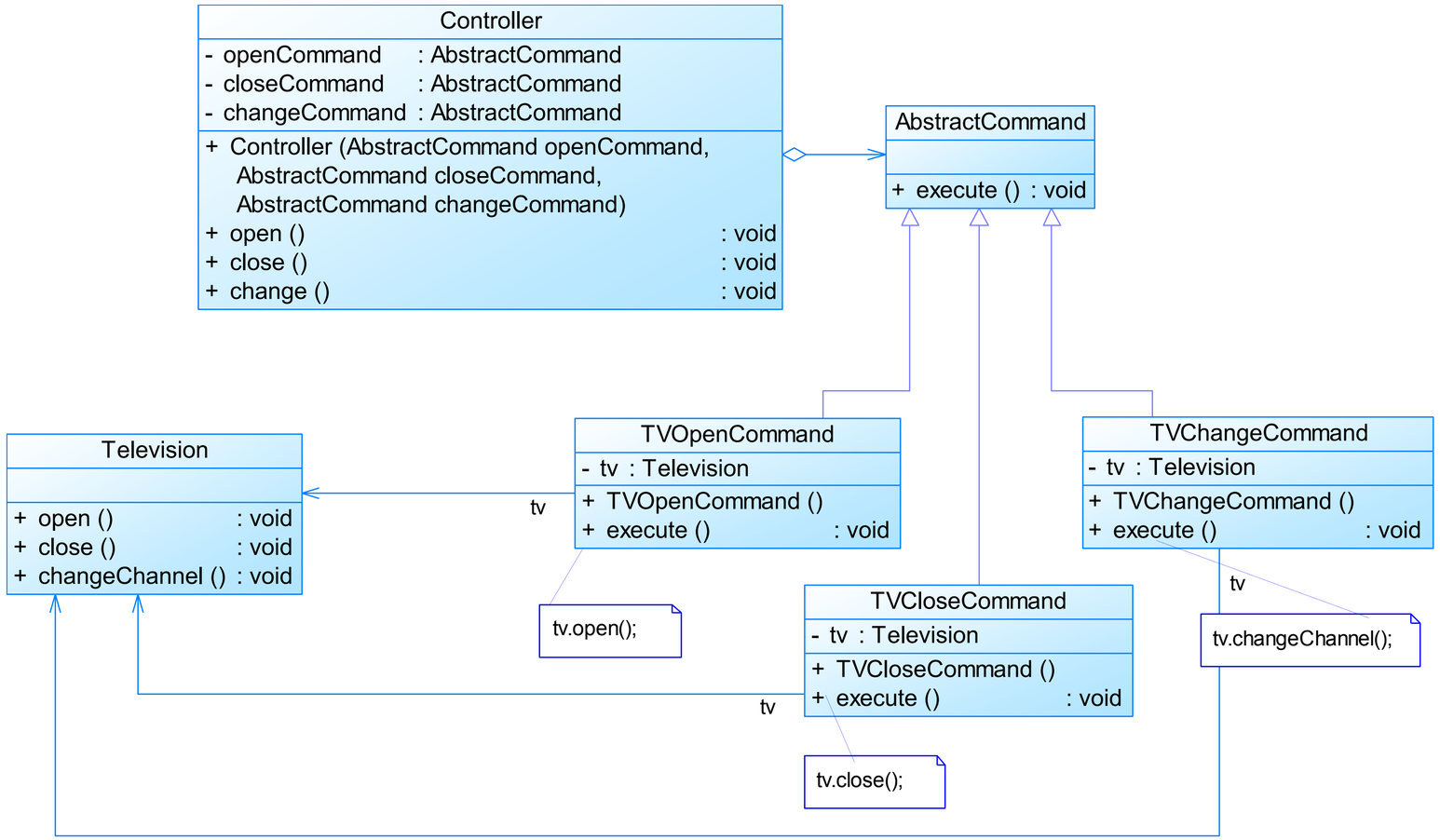
```
public class Invoker{
        Command command;
        //业务方法，用于调用命令类的方法
        public void call(){
                command.execute();
        }
}
```





命令模式实例

- 电视机遥控器：打开电视机、关闭电视机和切换频道



命令模式总结

优点

- 降低系统的耦合度
- 新的命令可以很容易地加入到系统中
- 可以比较容易地设计一个命令队列或宏命令（组合命令）
- 为请求的撤销(Undo)和恢复(Redo)操作提供了一种设计和实现方案

缺点

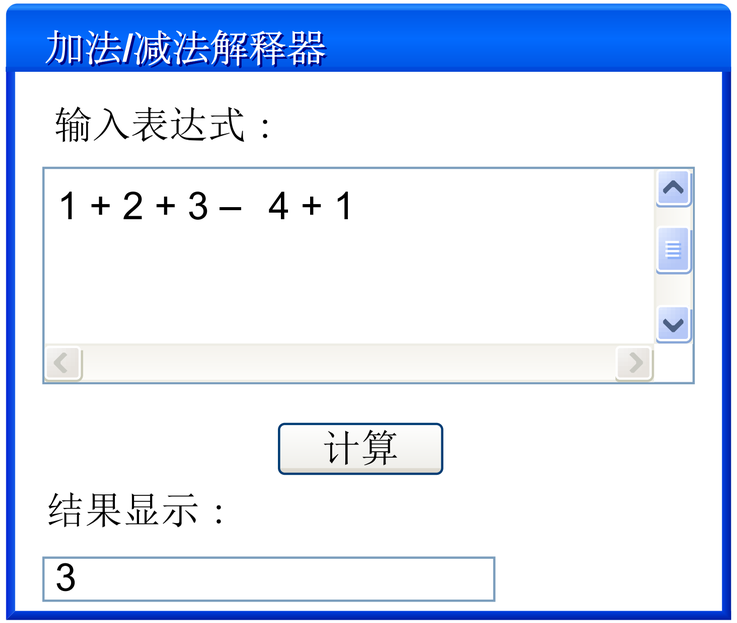
- 使用命令模式可能会导致某些系统有过多的具体命令类

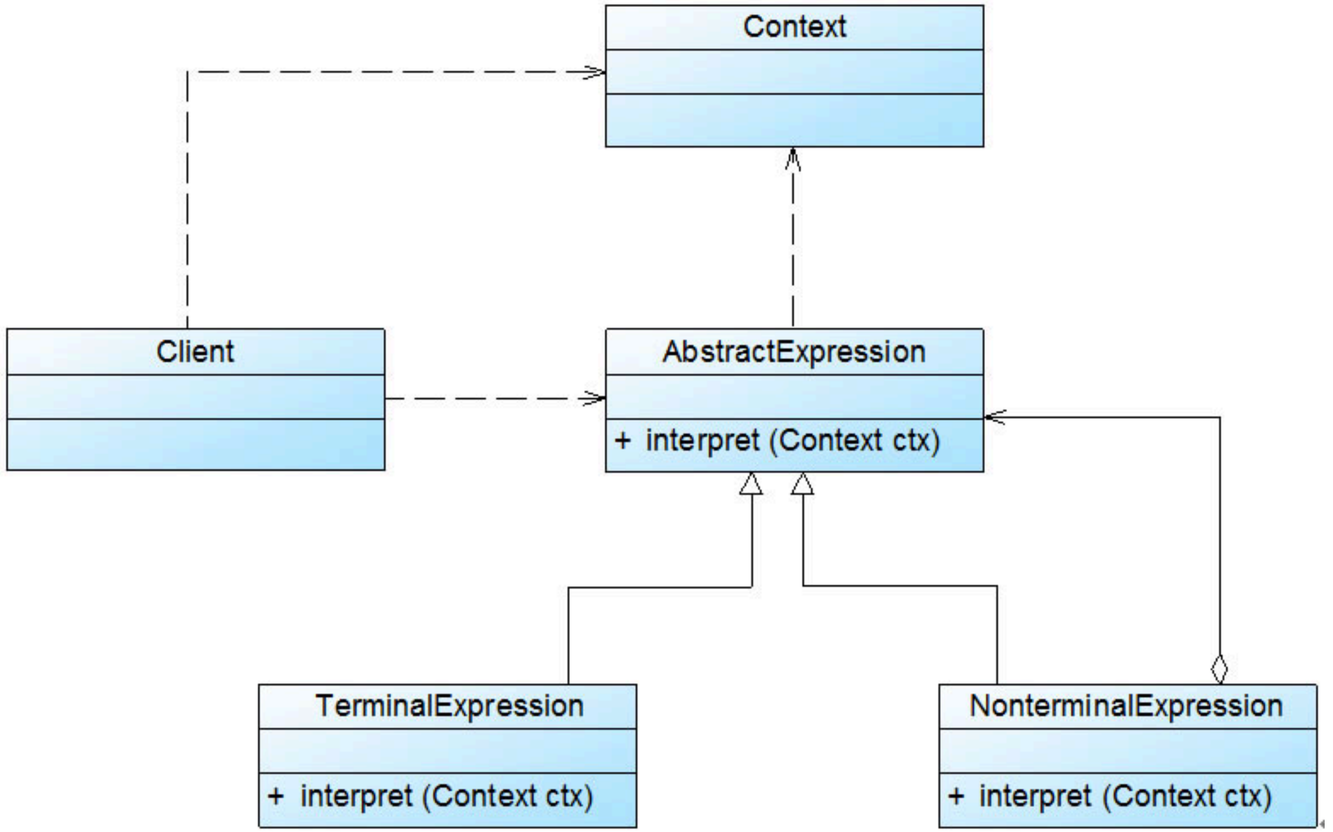
适用场景

- 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互
- 系统需要在不同的时间指定请求、将请求排队和执行请求
- 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。
- 系统需要将一组操作组合在一起形成宏命令

解释器模式

- 定义一个语言的文法，并且建立一个解释器来解释该语言中的句子，这里的“语言”是指使用规定格式和语法的代码





解释器模式实例

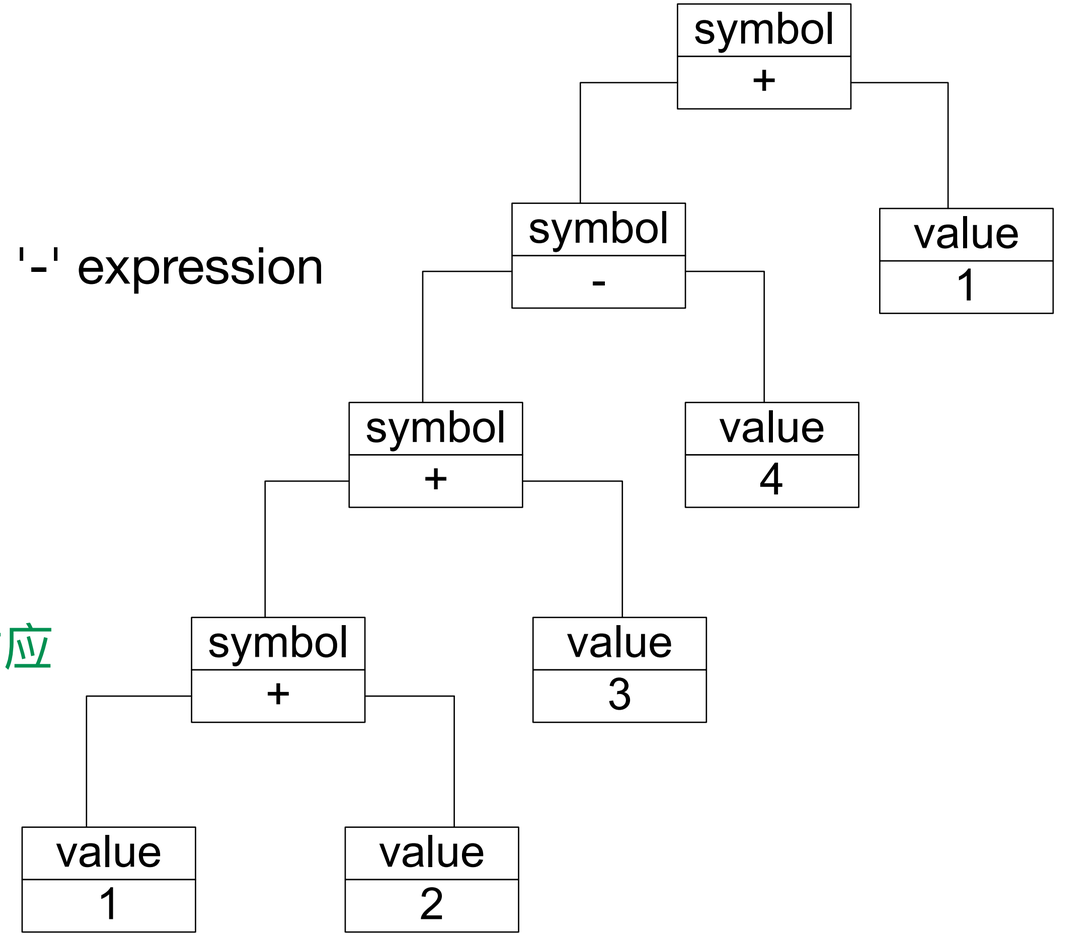
- 文法规则实例：

expression ::= value | symbol

symbol ::= expression '+' expression | expression '-' expression

value ::= an integer //一个整数值

- 每一种终结符和非终结符都有一个具体类与之对应



解释器模式总结

优点

- 易于改变和扩展文法
- 易于实现文法
- 增加了新的解释表达式的方式

缺点

- 对于复杂文法难以维护
- 执行效率较低
- 应用场景很有限

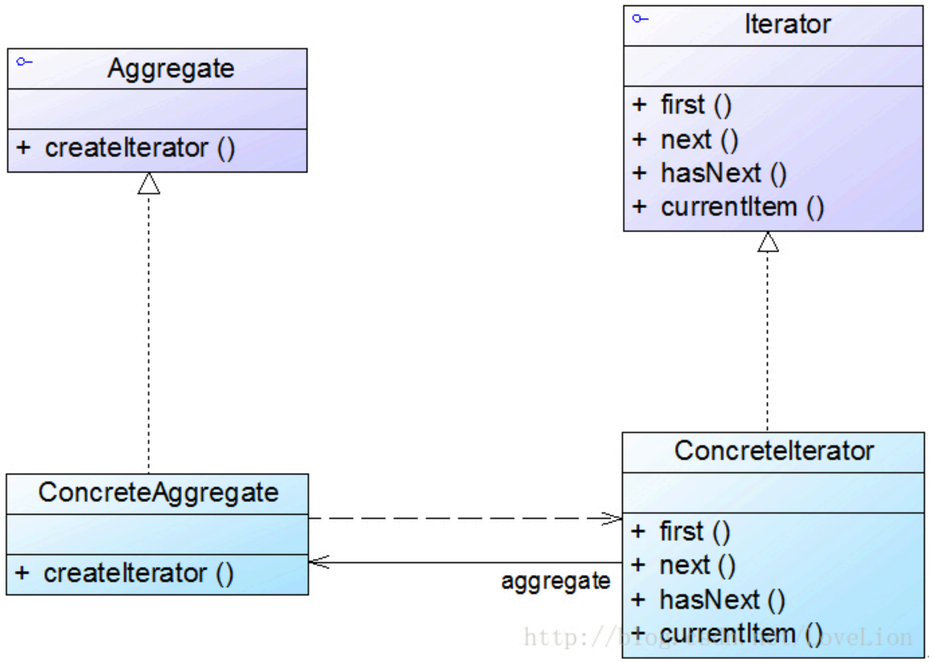
适用环境

- 可以将一个需要解释执行的语言中的句子表示为一个抽象语法树
- 一些重复出现的问题可以用一种简单的语言来进行表达
- 文法较为简单
- 效率不是关键问题

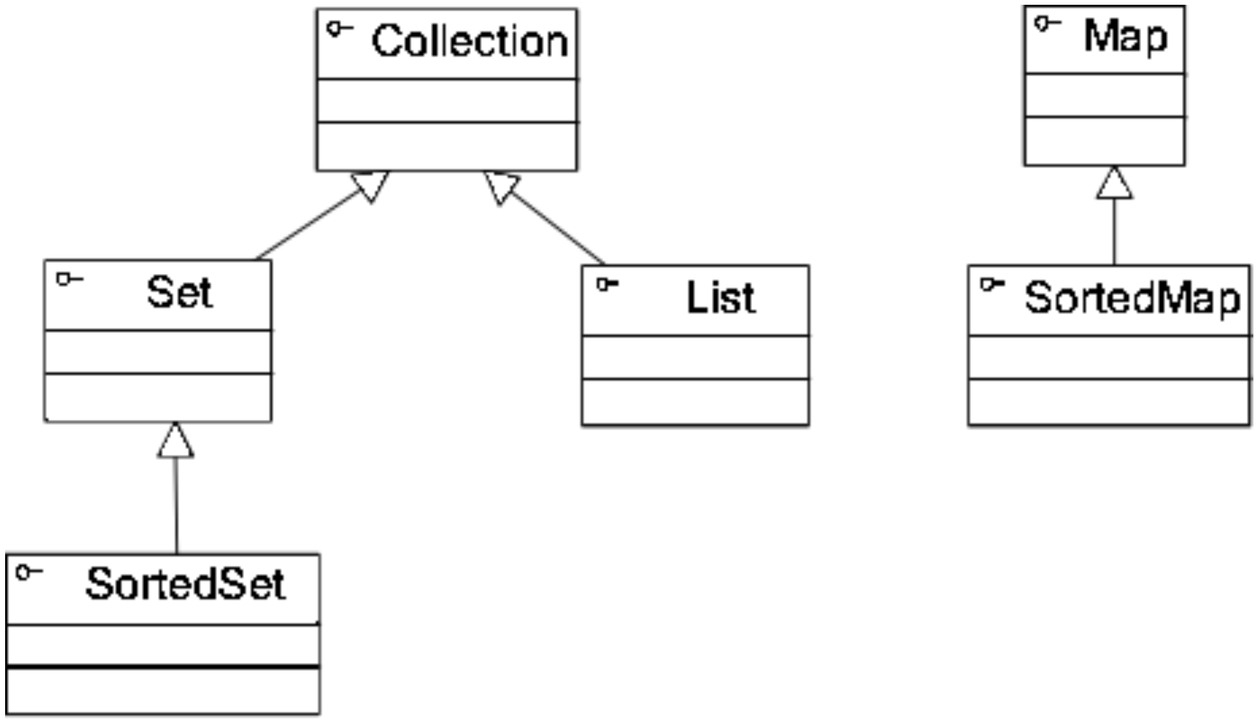
迭代器模式

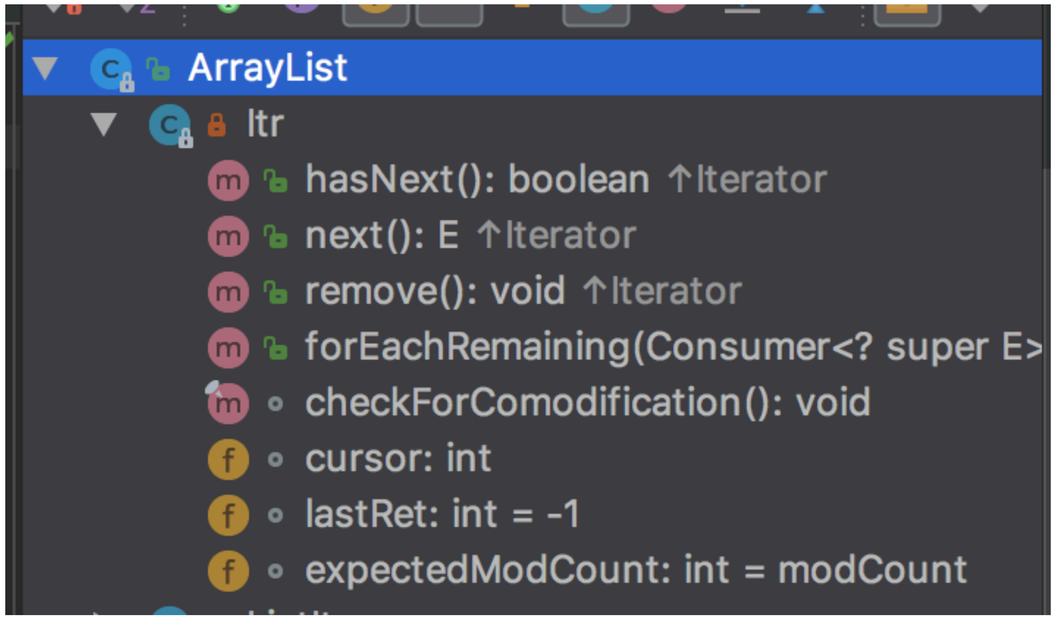
- 聚合对象应该提供一种方法来让别人可以访问它的元素，而又不需要暴露它的内部结构
- 可能还要以不同的方式遍历整个聚合对象，但是我们并不希望在聚合对象的抽象层接口中充斥着各种不同遍历的操作

迭代器模式提供一种方法来访问聚合对象，而不用暴露这个对象的内部表示，其别名为游标(Cursor)



迭代器实例


Collection
Map
Set
List
SortedMap
SortedSet


ArrayList
Itr
hasNext(): boolean ↑Iterator
next(): E ↑Iterator
remove(): void ↑Iterator
forEachRemaining(Consumer<? super E>
checkForComodification(): void
cursor: int
lastRet: int = -1
expectedModCount: int = modCount

迭代器总结

优点

- 支持以不同的方式遍历一个聚合对象
- 迭代器简化了聚合类
- 在同一个聚合上可以有多个遍历
- 在迭代器模式中，增加新的聚合类和迭代器类都很方便

缺点

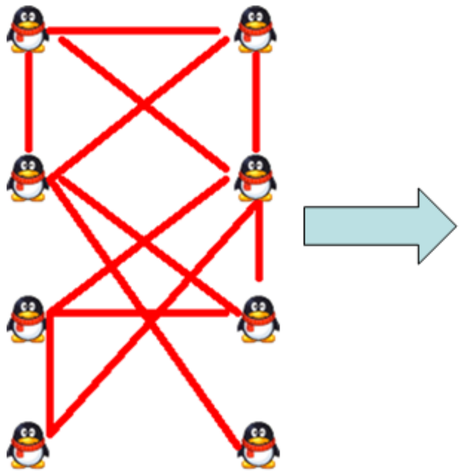
- 增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性

适用环境

- 访问一个聚合对象的内容而无须暴露它的内部表示
- 需要为聚合对象提供多种遍历方式
- 为遍历不同的聚合结构提供一个统一的接口

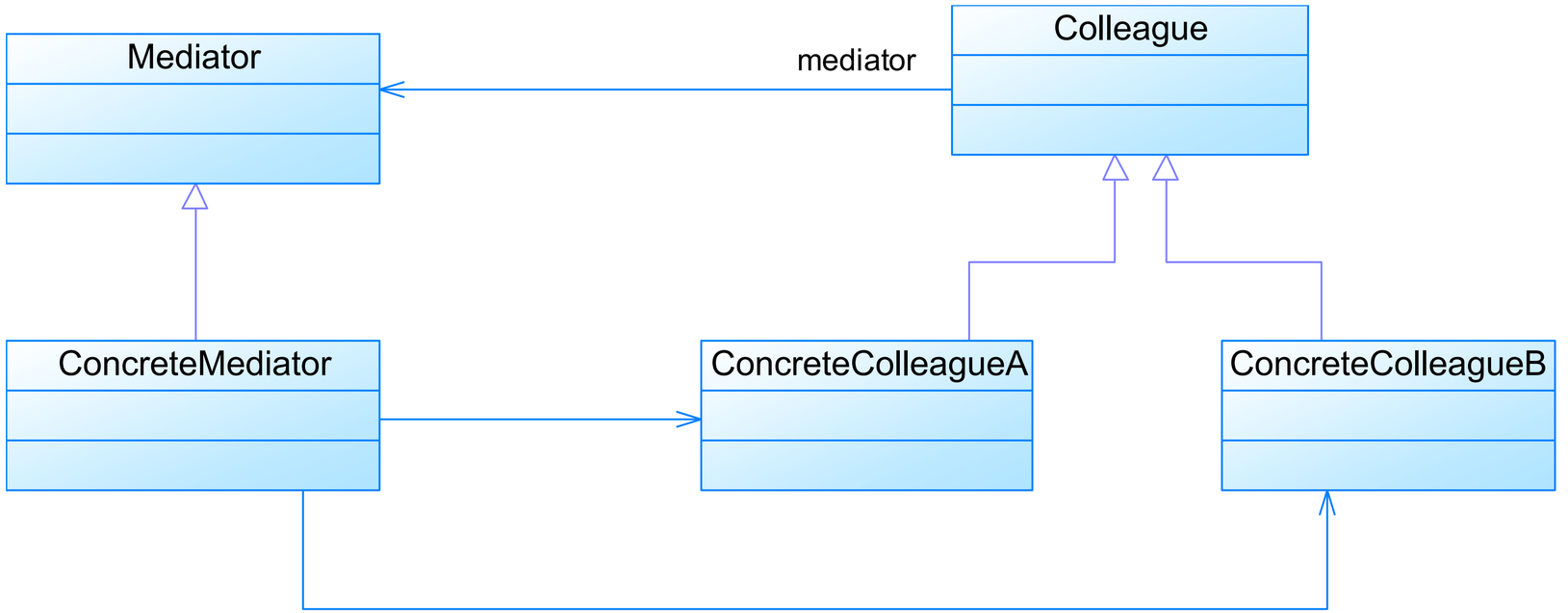
中介者模式

系统结构复杂, 对象可重用性差,系统扩展性低



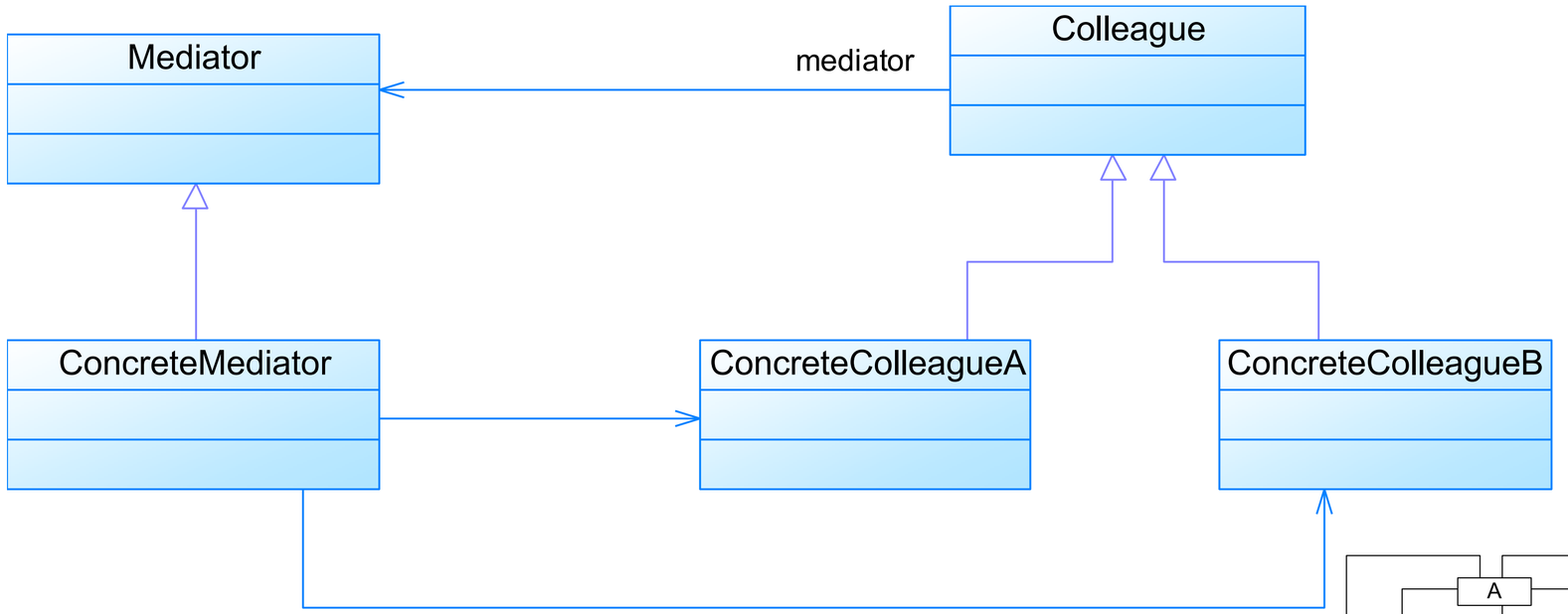
中介者模式

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

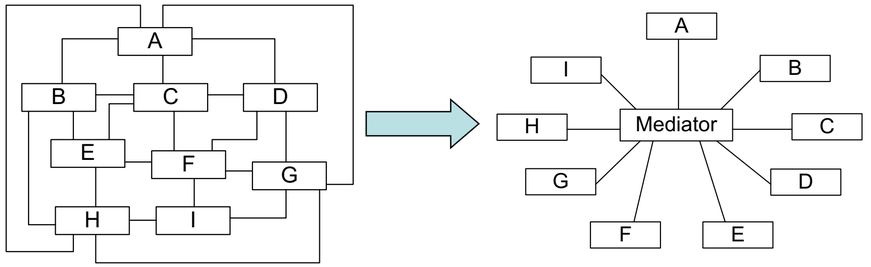


中介者模式

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

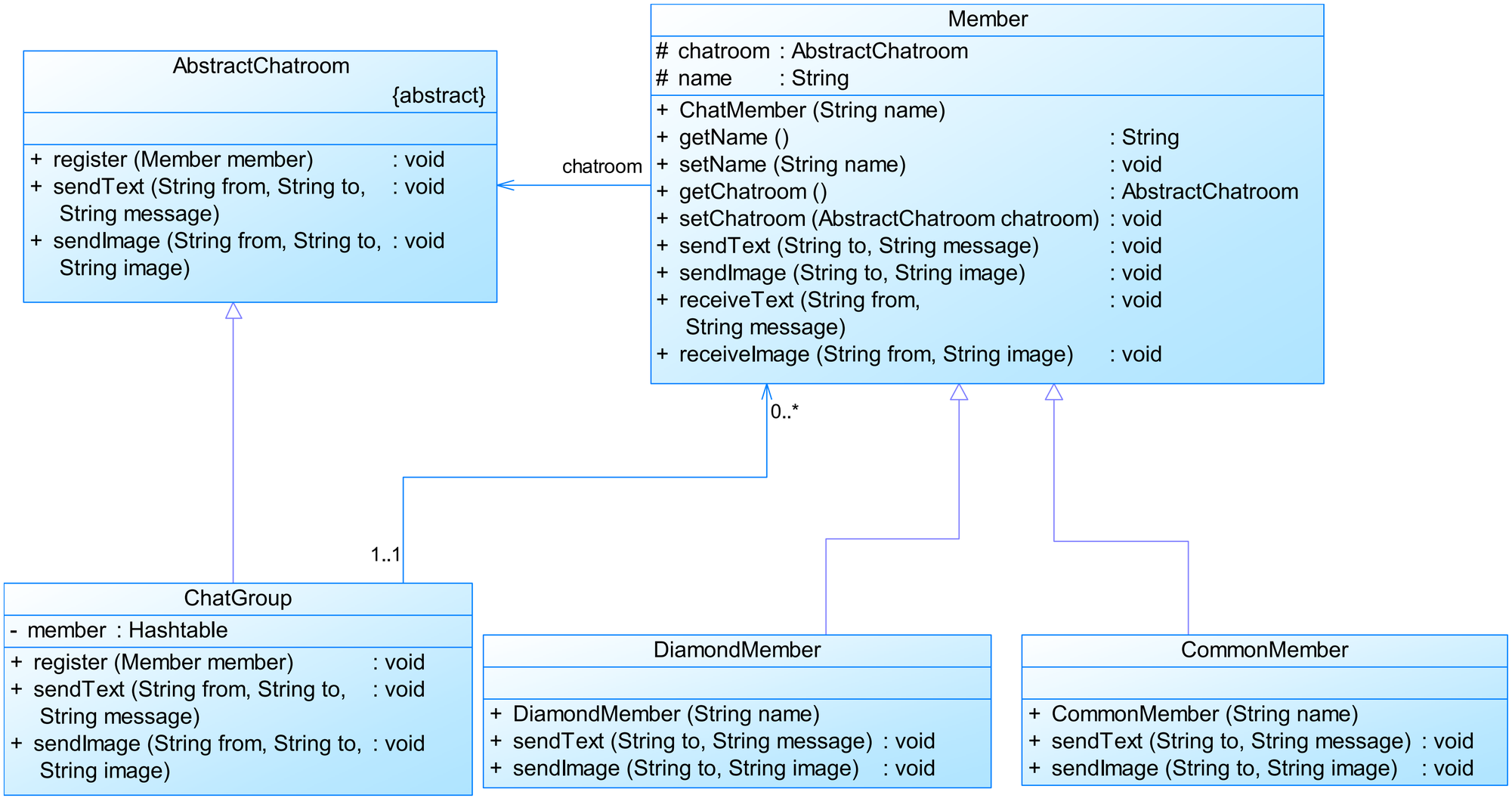


Mediator: 抽象中介者
ConcreteMediator: 具体中介者
Colleague: 抽象同事类
ConcreteColleague: 具体同事类



中介者模式实例

- 聊天室


AbstractChatroom
{abstract}
+ register (Member member) : void
+ sendText (String from, String to, String message) : void
+ sendImage (String from, String to, String image) : void
chatroom
Member
chatroom : AbstractChatroom
name : String
+ ChatMember (String name)
+ getName () : String
+ setName (String name) : void
+ getChatroom () : AbstractChatroom
+ setChatroom (AbstractChatroom chatroom) : void
+ sendText (String to, String message) : void
+ sendImage (String to, String image) : void
+ receiveText (String from, String message) : void
+ receiveImage (String from, String image) : void
0..*
1..1
ChatGroup
- member : Hashtable
+ register (Member member) : void
+ sendText (String from, String to, String message) : void
+ sendImage (String from, String to, String image) : void
DiamondMember
+ DiamondMember (String name)
+ sendText (String to, String message) : void
+ sendImage (String to, String image) : void
CommonMember
+ CommonMember (String name)
+ sendText (String to, String message) : void
+ sendImage (String to, String image) : void

中介者模式总结

优点

- 简化了对象之间的交互
- 将各同事解耦
- 减少子类生成
- 可以简化各同事类的设计和实现

缺点

- 在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护

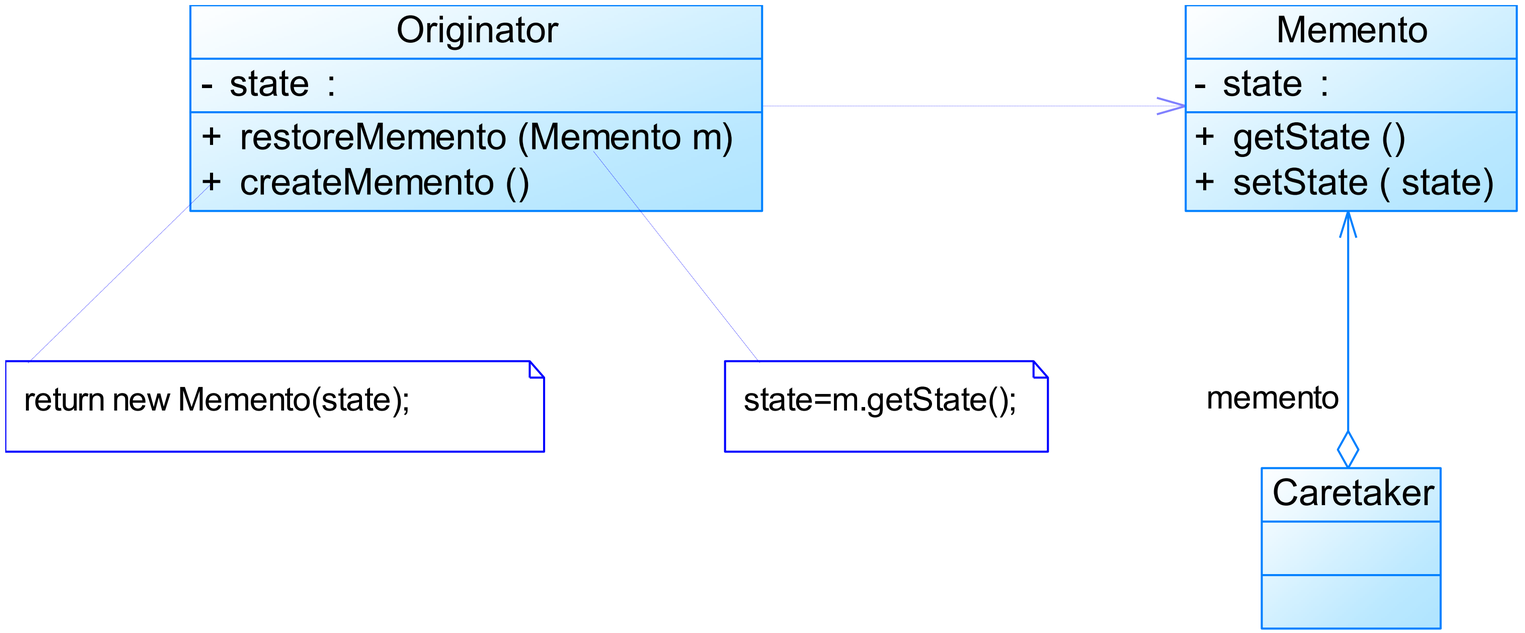
适用环境

- 系统中对象之间存在复杂的引用关系，产生的相互依赖关系结构混乱且难以理解
- 一个对象由于引用了其他很多对象并且直接和这些对象通信，导致难以复用该对象
- 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。可以通过引入中介者类来实现，在中介者中定义对象交互的公共行为，如果需要改变行为则可以增加新的中介者类

备忘录模式

撤销(Undo)功能

在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态



Originator: 原发器
Memento: 备忘录
Caretaker: 负责人

备忘录模式总结

优点

- 提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤，当新的状态无效或者存在问题时，可以使用先前存储起来的备忘录将状态复原
- 实现了信息的封装，一个备忘录对象是一种原发器对象的表示，不会被其他代码改动，这种模式简化了原发器对象，备忘录只保存原发器的状态，采用堆栈来存储备忘录对象可以实现多次撤销操作，可以通过在负责人中定义集合对象来存储多个备忘录

缺点

- 资源消耗过大
- 适用环境

在以下情况下可以使用备忘录模式：

- 保存一个对象在某一个时刻的状态或部分状态，这样以后需要时它能够恢复到先前的状态
- 如果用一个接口来让其他对象得到这些状态，将会暴露对象的实现细节并破坏对象的封装性，一个对象不希望外界直接访问其内部状态，通过负责人可以间接访问其内部状态