设计模式报告

论坛模块

Forum Module

作者

3170103307　魏秋阳

3170103058　龚城庆

3170103291　方誉州

3170105797　王北辰

3160105335　彭自涵

浙江大学计算机科学与技术学院战疫软工开发组

2020 年 5 月

目录

1 设计模式调查

1.1 概述

设计模式代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。在软件设计中，合适的设计模式可以帮助软件设计者更好的进行软件设计，使得设计变得规范。

根据《设计模式：可复用面向对象软件的基础》在书中的描述，设计模式主要可以分为三大类：创建型、结构型与行为型。下面将对其进行具体介绍，并对三大类中的几小类进行举例分析。

1.2 设计模式调查与分析

1.2.1 创新型设计模式

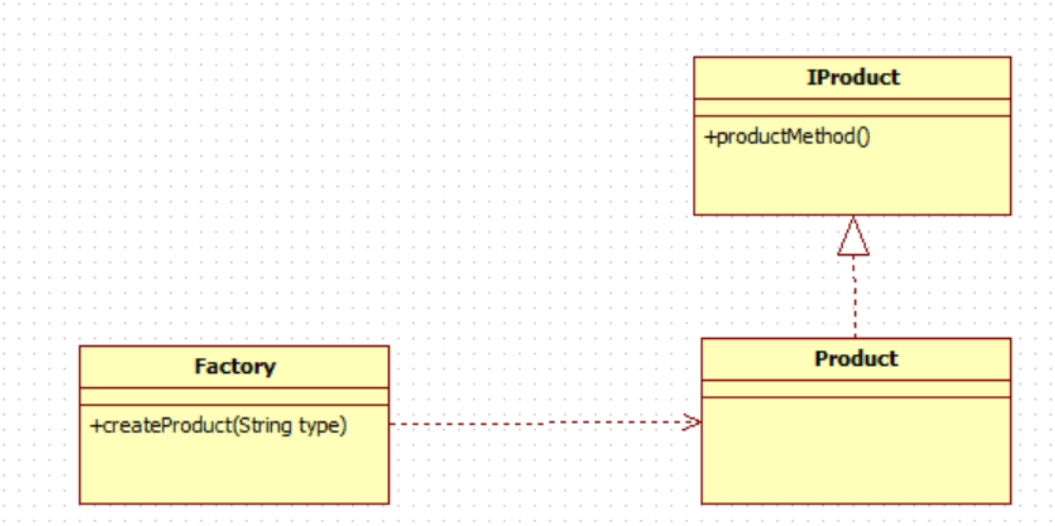
这类设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

我们主要调查了其中的简单工厂模式和工厂方法模式。

（1） 简单工厂模式

简单工厂模式属于创建型模式又叫做静态工厂方法模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

简单工厂模式结构图：



- Factory：工厂类，简单工厂模式的核心，它负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。
- IProduct：抽象产品类，简单工厂模式所创建的所有对象的父类，它负责描述所有实例所共有的公共接口。
- Product：具体产品类，是简单工厂模式的创建目标。

分析：

工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的责任，而仅仅“消

费”产品；简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象。

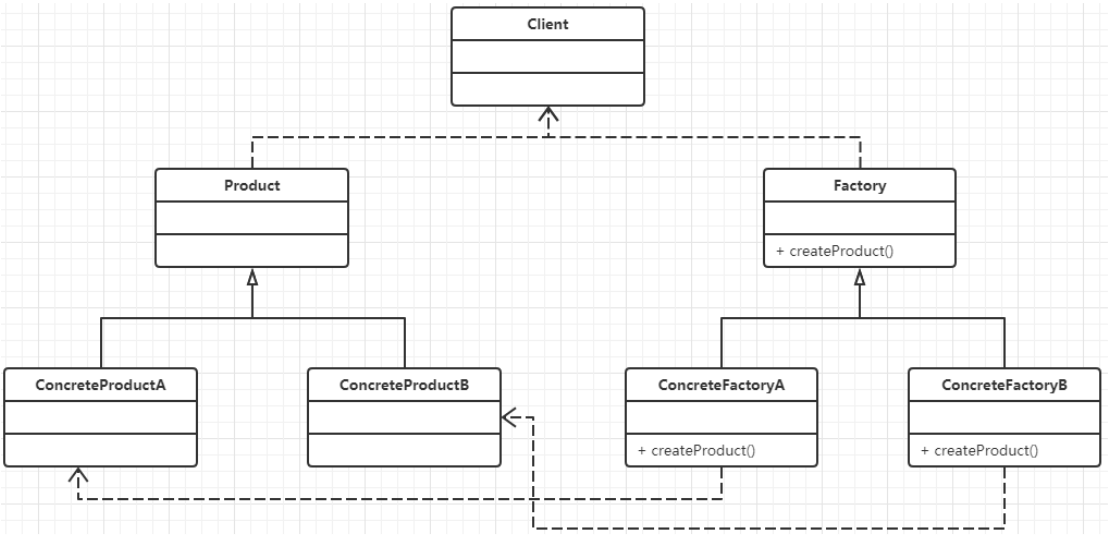
但也正是由于工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响。而且系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，同样破坏了“开闭原则”；在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。

（2） 工厂方法模式

工厂方法模式是简单工厂模式的延伸，它继承了简单工厂模式的优点，同时还弥补了简单工厂模式的缺陷，更好地符合开闭原则的要求，在增加新的具体产品对象时不需要对已有的系统做任何修改。

在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

工厂方法模式结构图：



- Factory（抽象工厂类）：在抽象工厂类中，声明了工厂方法(Factory Method)，用于返回一个产品。抽象工厂是工厂方法模式的核心，所有创建对象的工厂类都必须实现该接口。
- ConcreteFactory（具体工厂类）：它是抽象工厂类的子类，实现了抽象工厂中定义的工厂方法，并可由客户端调用，返回一个具体产品类的实例。
- Product（抽象产品类）：它是定义产品的接口，是工厂方法模式所创建对象的超类型，也就是产品对象的公共父类。
- （ConcreteProduct 具体产品类）：它实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建，具体工厂和具体产品之间一一对应。

分析：

与简单工厂模式相比，工厂方法模式最重要的区别是引入了抽象工厂角色，抽象工厂可以是接口，也可以是抽象类或者具体类。

在工厂方法模式中，客户端只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。但区别于区简单工厂模式，该模式中工厂不再负责产品的创建，由接口针对不同条件返回具体的类实例，而由具体类实例去实现。在该系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。

但是在添加新产品时，需要编写新的具体产品类，而且还要提供

与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销，也增加了实现的难度。

1.2.2 结构型设计模式

结构型模式关注如何将现有类或对象组织在一起形成更加强大的结构。可分为两种：

（1）类结构型模式：关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系；

（2）对象结构型模式：关心类与对象的组合，通过关联关系使得在一个类中定义另一个类的实例对象，然后通过该对象调用其方法。更符合“合成复用原则”。

由于发现其并不是特别适用于我们的系统，所以我们并没有更深入地研究。

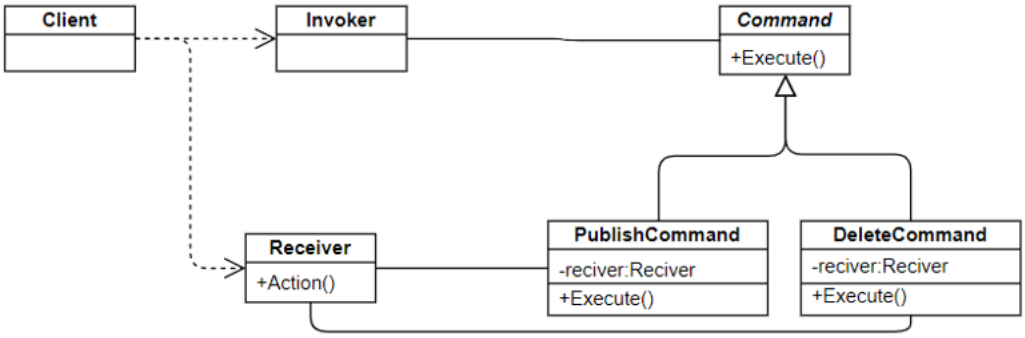
1.2.3 行为型设计模式

行为型设计模式，通常用来识别对象之间的常用交流模式并加以实现。我们主要调查了其中的命令模式、备忘录模式和责任链模式。

（1） 命令模式

该模式将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，将请求排队或记录请求日志，支持可撤销的操作。

命令模式结构图：



- Client：即用户，将通过操作创建一个具体命令对象。
- Command：抽象命令，包含命令执行的抽象方法
- Receiver:命令接收者角色,它包含所有命令的具体行为实 现方法。
- PublishCommand、DeleteCommand: 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。在我们的子系统中，为发布帖子和删除帖子命令。
- Invoker：提供给客户端调用，接收客户端所传递的具体命令对象。

分析：

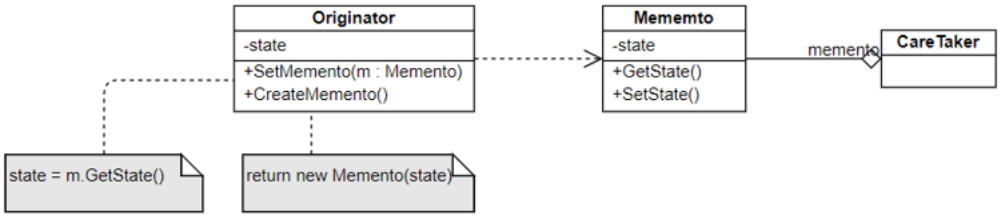
这是一种对象行为性模式，类似于传统设计方法的回调机制，将

一个请求封装为一个对象，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，这样做的目的就可以支持可撤销的操作。对这个命令的封装，将发出命令的责任和执行命令的责任分开，委派给不同的对象，这样就能实现发送者和接收者的完全的解耦，以达到松耦合的目的，提高系统的可扩展性和灵活性。

（2） 备忘录模式

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。

备忘录模式结构图：



- Originator: 发起人角色，创建一个含有当前的内部状态的备忘录对象。使用备忘录对象存储其内部状态。
- Memento: 备忘录角色，负责存储发起人对象的内部状态，在需要的时候提供 发起人需要的内部状态。
- Caretaker: 负责人角色，负责保存备忘录对象，不检查备忘录对象的内容。

分析：

由于在备忘录中存储的是原发器的中间状态，因此需要防止原发

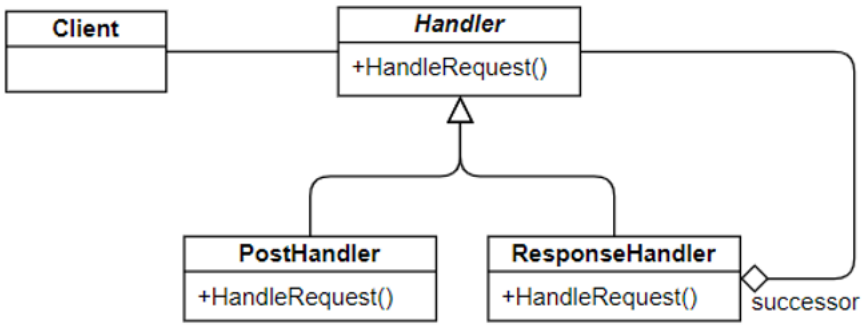
器以外的其他对象访问备忘录。

备忘录对象通常封装了原发器的部分或所有的状态信息，而且这些状态不能被其他对象访问，也就是说不能在备忘录对象之外保存原发器状态，因为暴露其内部状态将违反封装的原则，可能有损系统的可靠性和可扩展性。

（3） 责任链模式

责任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。

责任链模式结构：



Handler：抽象处理者。定义一个请求的接口。如果需要可以定义个一个方法用来设定和返回下家对象的引用。

ConcreteHandler：具体处理者。如果可以处理就处理请求，如果不能处理，就把请求传给下家，让下家处理。

分析：

责任链模式最大的缺点在于不能保证每个请求都被接受，因为一个请求没有一个明确的接收者的时候，不能保证一定会被处理，就会

一直被传递下去。

1.3 总结

下面我们简单总结了调查的几种设计模式的优缺点以及其应用：

模式	优点	缺点	应用
简单工厂	客户端无须创建产品对象，只需要提供产品的创建参数即可。	违反了开闭原则，系统的拓展和修改比较麻烦。	Product数量不再或者极少改变的系统
工厂方法	与简单工厂相比，遵守了开闭原则（增加新的产品类，只需要增加新的工厂类）；客户端更换产品对象时，改动较少（只需要改动具体工厂类即可）	每增加一个产品类，就会增加一个具体工厂类，增加额外的开发量，类的数目也较多。	客户端不需要知道具体的产品类的创建逻辑甚至具体产品的名称，但是需要知道创建具体产品的工厂类
命令	命令分步执行，	会有过多的具	请求一个操作

	低耦合度，高灵活性	体命令类	的对象与知道怎么执行一个操作的对象不直接交互
备忘录	为回滚操作提供了可能	备份对于资源消耗比较严重	需要执行回滚操作的系统
责任链	降低耦合度；增强了给对象指派职责的灵活性	一个请求极有可能到了链的末端都得不到处理，或者因为没有正确配置而得不到处理	多个对象都有机会处理请求，哪一个对象最终处理并不不确定

2 疫情管理系统体系结构分析

2.1 系统关键质量

依据本子系统的软件需求分析说明书，本子系统的关键质量属性为性能，安全性，使用性和服务获得性。其中在性能方面，负载容量和操作响应时间有明确的下界要求：

负载容量方面必须保证可允许 10000 名用户同时在线。

响应时间方面，单个用户在线 web 响应时间不超过 0.5s，信息查询时间不超过 2s。多名用户同时在线时，web 响应时间不超过 1s，信息查询时间不超过 3s.

2.2 系统体系结构

本系统分为客户端和服务器端：

服务器端：采用 Nodejs 语言在 express 框架下编写，数据库使用 Mysql。

客户端：浏览网页主要采用 Edge 浏览器或 Chrome，在移动端方面，同时支持 IOS 与 Android。

2.3 在关键质量属性中选择的策略

2.3.1 性能

（1）支持并发。引入并发能够极大的提高工作效率，满足多人同时使用系统的需求。

（2）设置优先级。引入优先级能够在系统负载过大时优先处理紧急事务。如在负载过大时，管理员的删除或修改帖子的优先级应高于普通用户。

（3）减少开销。减少计算开销与通信开销。如用户在进行注册时，一些格式匹配的工作可以直接在本地进行，不必发送至服务器，以减少网络的通信开销；在进行帖子的更新与排序时，尽可能的少修改数据库，减少计算开销。

2.3.2 安全性

（1）用户认证。确保进入系统的用户是合法用户。

（2）用户授权。分别管理管理员权限与普通用户权限。管理员权限能够对所有人的帖子进行编辑，而普通用户只能修改自己发布的内容。

（3）限制访问。通过防火墙限制一些危险用户的访问，确保校内用户能够稳定访问系统。

（4）数据加密。重要内容必须进行加密。储存在数据库中的内容

也必须加密。

（5）攻击后恢复。进行数据备份，结合系统日志和检查点，在被攻击后及时进行恢复。

2.3.3 服务获得性

2.3.3.1 概述

衡量指标 α = $\frac{MTBF}{(MTBF+MTTR)}$ ，其中：

MTBF 为平均故障时间，MTTR 为平均恢复时间。

2.3.3.2 策略

（1）异常检测。异常检测包括操作系统异常，常数异常等。这里主要讨论的是超时异常。用户进行操作时，应设置一个超时时间。若用户网络中断，则能够及时发现超时异常，进行回退。

（2）回滚。在发生错误的情况下，通过系统安全日志和检查点回滚到系统的前一个状态。本系统中，各种重要操作均应该支持回滚操作。

2.3.4 使用性

（1）取消与撤销。取消与撤销策略几乎用在了所有设计中，用户可以撤回其操作，减小了用户误操作的损失。

（2）维护用户模型。记录用户设置，便于用户在多环境下使用本系统。

2.3.5 可变更性

2.3.5.1 概述

在本系统的软件需求与分析说明书中，说明了本子系统需要实现易于修改和维护，以达到增量迭代开发的要求。

2.3.5.2 策略

（1）耦合。耦合性描述了两个模块之间的互相影响性。即修改一个模块后另一个模块需要修改的程度。本子系统要求达到低耦合性的要求。

（2）内聚。内聚性描述了一个模块的功能紧密结合的程度。本系统需要达到高内聚性，在模块划分时需要进行精心设计。

（3）模块分解。当一个模块被修改时，如果该模块十分复杂，那么修改所带来的开销将会大大增加。本系统需要适当地进行模块分解，减少错误带来的损失。

2.3.5.3 总结

良好的可变更性要求高内聚，低耦合，易分解，这需要开发人员有良好的设计经验，并对业务逻辑有着深入的掌握。

3 论坛模块设计模式分析

3.1 工厂模式（factory pattern）

——用户交互模块

3.1.1 概述

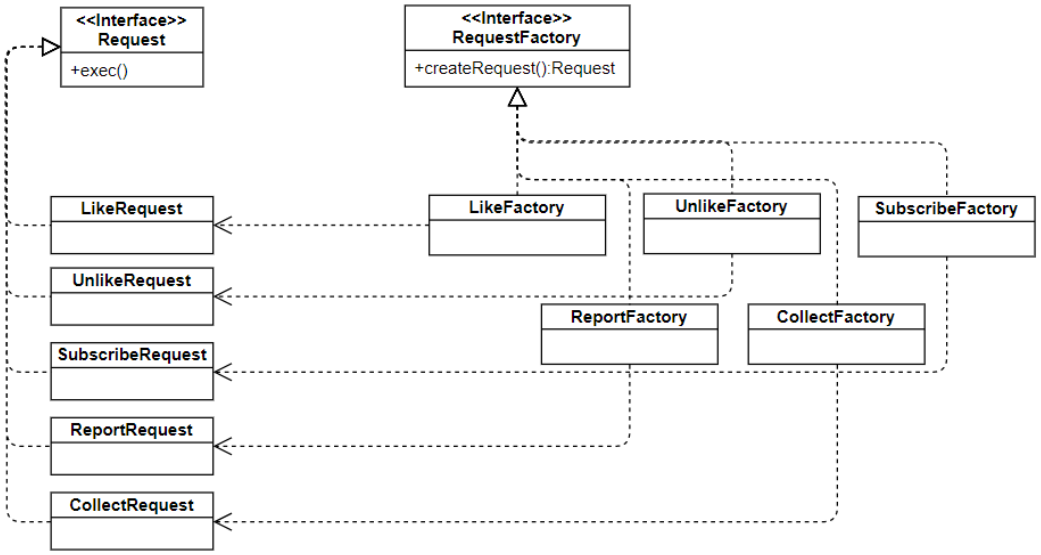
工厂方法模式将生成具体产品的任务分发给具体的产品工厂。工厂模式中包括以下主要的功能部门：

- 抽象工厂（Abstract Factory）：抽象工厂是工厂模式的核心。其定义了产品的生产接口，但不负责具体的产品，将生产任务交给不同的派生类工厂。这样不用通过指定类型来创建对象了。
- 具体工厂（Concrete Factory）：具体工厂负责各项产品的生产。每一个工厂都只生产相应的产品。具体的工厂继承自抽象的工厂，实现所有产品的必要的一些功能。
- 产品（Product）：由具体的工厂创建的实例，这些产品都继承自抽象产品（Abstract Product）。

3.1.2 应用

用户交互模块包括了一下一些行为：点赞、点踩、收藏、举报、关

注等。每一个项交互功能的实现都需要用户提交相应的字段来实现相应的功能。在这里我们可以为这些交互的功能提供一个统一的接口，并创建一个工厂来处理这些交互的请求。



3.1.3 总结

工厂模式相比于简单工厂模式而言，其优势在于更加方便的扩展功能，只需要提供对应的工厂和产品即可而不用修改其他的代码。在我们论坛子系统中可以根据需要创建相应的工厂来生产相应的产品来提供相应的所需要的功能。使我们在用户交互模块具有更好的可扩展性。

3.2 命令模式（Command Pattern）

——帖子管理模块

3.2.1 概述

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。是一种数据驱动的设计模式，它属于行为型模式，请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

3.2.2 应用

帖子管理模块的主要功能是发布帖子和删除帖子。

Client：即用户，将通过操作创建一个具体命令对象。

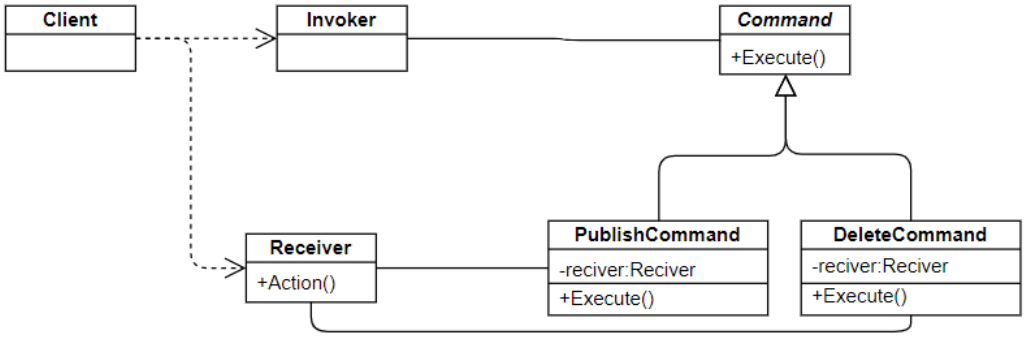
Invoker：负责调用命令对象执行请求，相关的方法叫做行动方法。

Command：声明了一个给有关帖子操作的抽象基类。

PublishCommand、DeleteCommand：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。在我们的子系统中，为发布帖子和删除帖子

命令。

Receiver：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。



3.2.3 总结

命令允许请求的一方和接收请求的一方能够独立演化，从而具有以下的优点：

- 命令模式使新的命令很容易地被加入到系统里。
- 允许接收请求的一方决定是否要否决请求。
- 能较容易地设计一个命令队列。
- 可以容易地实现对请求的撤销和恢复。
- 在需要的情况下，可以较容易地将命令记入日志。

帖子管理模块可拓展性较高，对于帖子来说，存在多种操作需求。所以使用命令模式，我们更容易为以后课程模块管理可能新增的命令留下拓展性，增加系统的灵活性和复用性。

3.3 备忘录模式（Memento Pattern）

——管理员/用户操作记录模块

3.3.1 概述

备忘录模式又叫做快照模式(Snapshot Pattern)或 Token 模式，是对象的行为模式。备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉(Capture)住，并外部化存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

备忘录模式给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。同时实现了信息的封装，使得用户不需要关心状态的保存细节。适用于需要回滚的场景。

3.3.2 应用

无论是用户还是管理员都有对帖子的管理权限。尤其是管理员，我们需要储存管理员的操作以提供出现问题时的补救措施。

备忘录模式所涉及的角色有三个：备忘录(Memento)角色、发起人(Originator)角色、负责人(Caretaker)角色。

（1）备忘录(Memento)角色

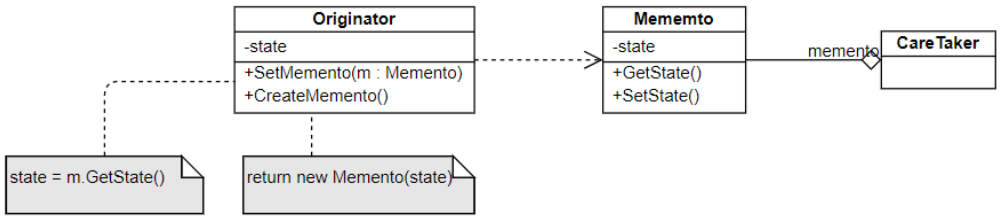
- 将发起人（Originator）对象的内战状态存储起来。备忘录可以根据发起人对象的判断来决定存储多少发起人（Originator）对象的内部状态。
- 备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。

（2）发起人（Originator）角色

- 创建一个含有当前的内部状态的备忘录对象。
- 使用备忘录对象存储其内部状态。

（3）负责人（Caretaker）角色

- 负责保存备忘录对象。
- 不检查备忘录对象的内容。



3.3.3 总结

通过使用备忘录模式记录管理员的操作，防止管理员出错或系统崩溃等意外状况发生，来提高我们系统的可靠性。使用备忘录模式，把复杂的发起人内部信息对其他的对象屏蔽起来，从而可以恰当地保持封装的边界。同时简化了管理，可自行进行存储和恢复状态。

3.4 责任链模式 (Chain of Responsibility)——帖子详情模块

3.4.1 概述

责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

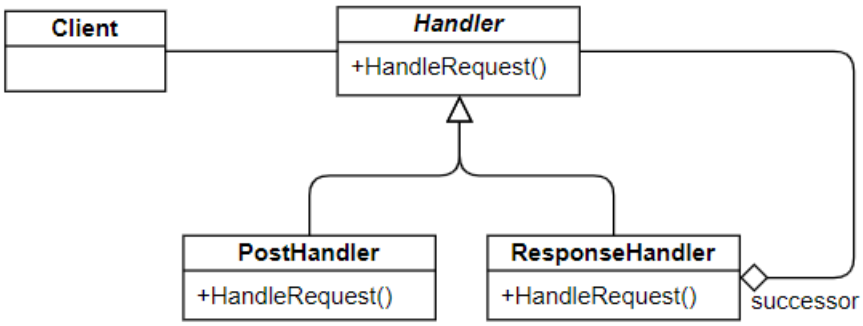
具体的请求处理者可以有多个，并且所有的请求处理者均具有相同的接口(继承于同一抽象类)。责任链模式主要包含如下两个角色：

- Handler（抽象处理者）：处理请求的接口，一般设计为具有抽象请求处理方法的抽象类，以便于不同的具体处理者进行继承，从而实现具体的请求处理方法。此外，由于每一个请求处理者的下家还是一个处理者，因此抽象处理者本身还包含了一个本身的引用（ successor）作为其对下家的引用，以便将处理者链成一条链；
- ConcreteHandler（具体处理者）：抽象处理者的子类，可以处理用户请求，其实现了抽象处理者中定义的请求处理方法。在具体处理请求时需要进行判断，若其具有相应的处理权限，那么就处

理它；否则，其将请求转发给后继者，以便让后面的处理者进行处理。

3.4.2 应用

在这个模块中我们会需要请求两部分的内容：帖子的内容和帖子的回复。在这个过程中我们根据客户端上传的请求的关键字段来请求相应的对象。我们为这些请求提供一个接口，通过这个接口将请求传入，经过责任链选择合适的处理者处理请求。



3.4.3 总结

在责任链模式里，由每一个请求处理者对象对其下家的引用而连接起来形成一条请求处理链。请求将在这条链上一直传递，直到链上的某一个请求处理者能够处理此请求。事实上，发出这个请求的客户端并不知道链上的哪一个请求处理者将处理这个请求，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链和分配责任。

4.其他可用设计模式分析

4.1 模板方法模式

4.1.1 概述

模板方法模式是行为模式的一种。在此模式中，类只定义一个操作中算法的骨架，更加具体的错将留在子类中延迟实现，使得子类可以在不改变算法结构的前提下重定义算法的某些特定步骤。

模板方法模式包含的角色及职责如下：

抽象类:定义一个操作中算法的股价,而将步骤延迟到子类中。其中模板方法可以实现代码共用。

具体类：重定义抽象类中的算法的某些特定步骤。

4.1.2 应用

对于所有的帖子与所有的用户我们都可以采用此方法。对于公共行为,如所有帖子的展示,用户登录等行为,我们可以集中在父类中，而对于特有的行为，如管理员和普通用户的权限区别，帖子和回帖的区别等，我们均可以在子类中完成。

4.1.3 总结

模板方法模式中，子类通过继承抽象基类以实现各个步骤的抽象方法，所以整体代码看上去简洁统一。并且，子类的流程会永远受到基类的控制，避免了一个流程被多个子类所修改。同时，代码的可读性也得到了提升，并且减少了代码重复量。

4.2 外观模式

4.2.1 概述

外观模式的动机时简化客户程序与系统之间的交互接口，降低复杂系统的内部子系统与客户程序之间的依赖。为了实现此目标，外观模式为子系统中的一系列接口提供了一个一致的界面，使得这一子系统更加便于使用。

4.2.2 应用

针对不同权限的用户，可以提供统一的主界面。然后根据用户种类的不同，使用统一的接口层将其导向不同的界面。然后不同的界面单独完成。例如针对管理员和不同用户的用户中心，分别定义一个单独的子类。

4.2.3 总结

使用外观模式可以有效提高系统的独立性，简化组件之间的接口，

将用户程序与子系统之间的依赖降低，避免子程序的变化直接影响到客户程序。在层次化接口中，可以使用外观模式定义每一层的入口。

4.3 装饰器模式

4.3.1 概述

装饰器模式允许动态地给一个对象添加额外的职责。与继承相比，装饰器模式允许动态地添加对象的功能，避免代码的重复，也能避免类数目过度增加。

装饰器模式中包含的角色与职责如下：

抽象组件角色：准备接受附加责任的对象，需要定义对象接口。

具体组件角色：被装饰者，定义一个需要被装饰的类，达到给类附加职责的目的。

抽象装饰器：维持一个指向构件对象的实例，并且提供一个与抽象组件角色一致的接口。

具体装饰器：向组件添加职责。

4.4.2 应用

例如，在浏览帖子的子系统中，可以首先定义最基本的查看帖子的抽象组件，然后查看帖子的子功能就可以通过具体组件来实现。由于查看帖子对于管理员用户来说应该具有许多附加功能，所以通过一个指向抽象组件的装饰器进行扩展功能的实现。针对管理员的具体装

饰器则可以在查看帖子的基础上扩展指定，修改，删除等功能。

4.4.3 总结

在编程中，应谨慎地运用子类修改类的扩展功能。若预计某类在将来会有较大的可能性多次修改，则尽可能的使用装饰器模式进行扩展，增加软件的可修改性。然而不应过度使用此模式，否则容易使得小类变得过于复杂，难以理解。

5 参考

[1]Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,《Design Patterns:Elements of Reusable Object-Oriented software》

[2]Gamma E, Helm R, Johnson R, John V. 设计模式可复用面向对象的软件基础[M]. 机械工 业出版社, 2007.

[3] Freeman E. HeadFirst 设计模式[M]. 中国电力出版社, 2007.