设计模式报告

基础信息模块

basic information Module

作者

3170104705 杨淯晴

3170106317 仇盛显

3170106089 马跃洋

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](#_Toc39582594)

[2基础模块体系结构分析 2](#_Toc39582595)

[2.1 Performance Tactics 2](#_Toc39582596)

[2.1.1增加资源 2](#_Toc39582597)

[2.1.2合理使用资源 2](#_Toc39582598)

[2.1.3合理分配资源 2](#_Toc39582599)

[2.2 Security Tactics 2](#_Toc39582600)

[2.2.1抵抗攻击 3](#_Toc39582601)

[2.2.2检测攻击 3](#_Toc39582602)

[2.2.3从攻击中恢复 3](#_Toc39582603)

[2.3 Testability Tactics 3](#_Toc39582604)

[2.3.1 I/O 数据控制 3](#_Toc39582605)

[2.3.2内部检测 4](#_Toc39582606)

[2.4 Availability Tactics 4](#_Toc39582607)

[2.4.1错误检测 4](#_Toc39582608)

[2.4.2错误恢复 4](#_Toc39582609)

[2.4.3错误预防 5](#_Toc39582610)

[3创建型模式 6](#_Toc39582611)

[3.1工厂方法(Factory Method) 6](#_Toc39582612)

[3.2原型(Prototype) 7](#_Toc39582613)

[3.3单件(Singleton) 7](#_Toc39582614)

[4结构型模式 8](#_Toc39582615)

[4.1适配器(Adaptor) 8](#_Toc39582616)

[4.2外观(Facade) 8](#_Toc39582617)

[4.3享元(Flyweigh) 8](#_Toc39582618)

[4.4代理(Proxy) 9](#_Toc39582619)

[5行为模式 10](#_Toc39582620)

[5.1 责任链 （Chain of Responsibility） 10](#_Toc39582621)

[5.2 命令（Command） 11](#_Toc39582622)

[5.3 循环体（Iterator） 11](#_Toc39582623)

[5.4 中介（Mediator） 12](#_Toc39582624)

[5.5 备忘录（Memento (Snapshot)） 13](#_Toc39582625)

[5.6 观察者（Observer） 13](#_Toc39582626)

[5.7 状态（State） 14](#_Toc39582627)

[5.8 策略（Strategy） 15](#_Toc39582628)

[5.9 模板（Template Method） 15](#_Toc39582629)

[5.10 浏览者（Visitor） 16](#_Toc39582630)

[6基础信息模块设计模式分析 17](#_Toc39582631)

[架构具体实现方法 18](#_Toc39582632)

[Model 18](#_Toc39582633)

[ViewModel 19](#_Toc39582634)

[View 20](#_Toc39582635)

[7参考资料 21](#_Toc39582636)

# 1设计模式调查

在对 OOP 程序设计之初，就遇到了不少共性的问题，为了解决这些问题，程序员在其中探索出了一些用于解决一般性问题的方法，这些方式就被称为设计模式。程序员可以在已有的设计模式基础上再做修改来达到解决具体问题的效果。

设计模式首先被 Christopher Alexander 在 1996 年提出，当时被称为 architectural concept。在之后，由于 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994) 的提出，一共有23条设计模式。设计模式从此变得十分热门。许多新的研究成果也在之后大量出现，设计模式也因此不断被扩充，远不止于先前的23条。

设计模式源于 SOLID 的设计原则，即：

* Single Responsibility Principle
* Open/Closed Principle
* Liskov Substitution Principle
* Interface Segregation Principle
* Dependency Inversion Principle

在这五条的原则之上，设计模式具体化了许多种不同的应用场景，并根据此总结出最优的做法。由于被许多人使用和验证，设计模式被证明是十分高效且利于将来代码的维护。

由于设计模式的繁杂，根据他们不同的目的和目标，这些模式被切分为三类：

* Creational patterns
* Structural patterns
* Behavioral patterns

创建型模式主要应用于创建机制当中，并且能够使得代码提高灵活性和可复用性；结构模式主要将不同的 objects 形成一个复杂的结构，并且在同时能够保证结构的灵活性和高效性；行为模式着眼于提高各个 objects 之间的协作效果，增加他们之间的沟通效率。

# 2基础模块体系结构分析

对于基础信息模块，为了保证模块的正常工作，我们需要关注并且保证以下几种质量：

* Performance
* Security
* Testability
* Availability

性能十分关键，他是用户能够直接观察到的性质。安全性是数据处理的极为重要的一个环节。可测试性能够显著降低问题代码，最大程度保证模块逻辑的正常运作。可达性也是用户能够直接体会到的性质，作为用户导向的开发理念，可达性也必须考虑在内。

## 2.1 Performance Tactics

为了解决性能问题，我们会从三个方面着手处理

### 2.1.1增加资源

我们可以选择增加运算效率，比如说使用有着良好网络和硬件状态的云端机器作为我们的服务器。我们也会减少运算时的不必要消耗，比如说压缩计算复杂度。还有一种做法就是限制 accept queue 的大小，控制在机器可以容忍的程度。

### 2.1.2合理使用资源

第一种方式就是使用并发的技术来实现并发处理请求。除此之外，我们还可以以空间换时间，适当增加一些冗余信息来加速数据库查询速度。

### 2.1.3合理分配资源

我们可以选择合适的资源调度工具来处理请求，比如说使用 Nginx 来应用高效的调度政策，从而加快服务器响应速度。

## 2.2 Security Tactics

为达成安全性，主要可以通过抵抗攻击，检测攻击，和从攻击中恢复的能力。

### 2.2.1抵抗攻击

* 身份确认 - 我们可以在服务提供前要求用户给出身份认证的凭证，比如说输入账号密码等，或者通过之前使用的 Cookie 来证明自己的身份。
* 身份授权 - 由于服务的数据敏感程度不同，普通用户和管理员能够接触到的数据也不相同，因为需要区分高低权限来阻止越权操作。
* 保密 - 所有数据都需要保证不被未授权者了解数据内容。我们可以使用 TLS 来保证。
* 保持完整性 - 数据在传输过程中不得被修改，即使是被修改，也要被发现并且向上汇报。我们也可以使用 TLS 来保证。
* 保证可达性 - 接收者能够成功接收到信息。我们会选择使用网速较快，并且离用户较近的机器充当服务器。

### 2.2.2检测攻击

* 在攻击发生时，我们能够检测到攻击。我们会将服务器 IP 隐藏在 CDN 之后，一切流量都要先经过 CDN 的清洗才能到达我们的服务器。一旦发生攻击，CDN 会向服务器拥有者汇报当前遭到了攻击。

### 2.2.3从攻击中恢复

* 恢复到上一次可用状态 - 如果攻击发生并且数据遭到损坏，我们可以把数据还原为上一次可用的状态。具体来讲，我们会采用备份的手段来保证一次数据损坏之后还能完整地恢复到上一次备份的节点上。
* 识别攻击者 - 我们会收集一切攻击者留下的痕迹，根据痕迹我们能够做出更加精确的数据恢复判断，并且能够作为证据来反追溯攻击者，并且能够作为教训来防范类似攻击再次发生。

## 2.3 Testability Tactics

为保证可测试性，我们主要关注两个问题：I/O 数据可控制性和内部可检测性。

### 2.3.1 I/O 数据控制

* 记录/回放 - 记录指的是记录经过 input 和 output 时的数据。回放指的是在记录可以在之后的测试中重新使用，用作测试用例。
* 分离接口和 implementation - 拆解后的部分可以更加方便拿具体的接口来在 I/O 出入口做测试。
* 开放特殊接口 - 有一些数据本来是不应该从封装中泄露的，但是出于测试的角度来看，我们就需要开放一些接口来做测试，这些接口可能是针对 metadata 来做的，因此就需要特殊开辟接口。

### 2.3.2内部检测

* 提供内部检测 - 程序内部需要提供一些内置的检测模块，这些模块需要考虑在整体设计当中。这些检测模块同时需要像正常模块一样防止安全问题。有了内部检测模块，我们就可以更好地发现问题。具体上我们可以使用 assert 等语句来设定一些测试值和预期值来最大程度上保证软件的逻辑正确性。

## 2.4 Availability Tactics

可用性主要分为错误检测，恢复，预防等方面。

### 2.4.1错误检测

* Ping - 我们可以使用已有的 ICMP 协议来检测网络层是否出现了故障。
* Heartbeat - 由于 TCP 协议可能出现半连接的窘况，我们会引入心跳来定时维持 TCP 的链接，如果 Heartbeat 超时了，我们就认为这个链接断开了。
* Exception - 我们会在程序里面安插尽量全面的 exceptions，如果错误发生，就会主动触发 exception。

### 2.4.2错误恢复

* 投票 - 计算机处理一个输入，并且对同一输入重复多次输出，如果输出内容不完全一致，那么将会采用多数服从少数的方式来决定到底使用哪个输出。由于现在的处理器不太可能出现这种问题，因此不太使用。这种技术主要使用在太空等高辐射的地区，以防处理器由于辐射而导致运算错误。
* 主动冗余 - 通过增加冗余，也就是说引入多个冗余组件，这些组件都是在相同的状态。但是只有其中一个组件的输出是被使用的，其他都要被丢弃。如果发生了错误，数据会马上从其他并行的组件中恢复，因此宕机时间十分短。
* 被动冗余 - 与上面类似，但是这种冗余是需要有一个带头组件来发号施令，当错误发生之后，才会去运行其他冗余部分，从而减少运算开支，但是恢复时间会有所延长。
* 空闲机器 - 随时准备一个空闲的机器等待，如果发生问题，空闲的机器会马上从一个快照中恢复，进而迅速承接服务，直到原机器从错误中恢复，比如说重启结束。
* 影子操作 - 挂掉的组件如果需要重新投入生产，就需要首先在影子环境中启动，确认不会发生错误以后再投入生产。
* 状态同步- 上面提到的两种冗余方式都需要一个状态来记录当前运行到什么阶段了。这个状态就需要在不同的组件中保证其一致性。
* Checkpoint/rollback - 检查点其实就是快照，当系统挂掉以后，就可以直接从某个快照中恢复。

### 2.4.3错误预防

* 组件下线 - 如果我们发现有组件发生了问题，我们会主动将该组件下线来防止错误被利用。
* Transactions - 一次 Transaction 被认为是一次原子性操作。如果出现错误，我们将不会采用这个过程中的所有操作。
* 进程守护 - 如果进程出现错误，守护进程将会把挂掉的程序撤换成新的程序继续保持服务。

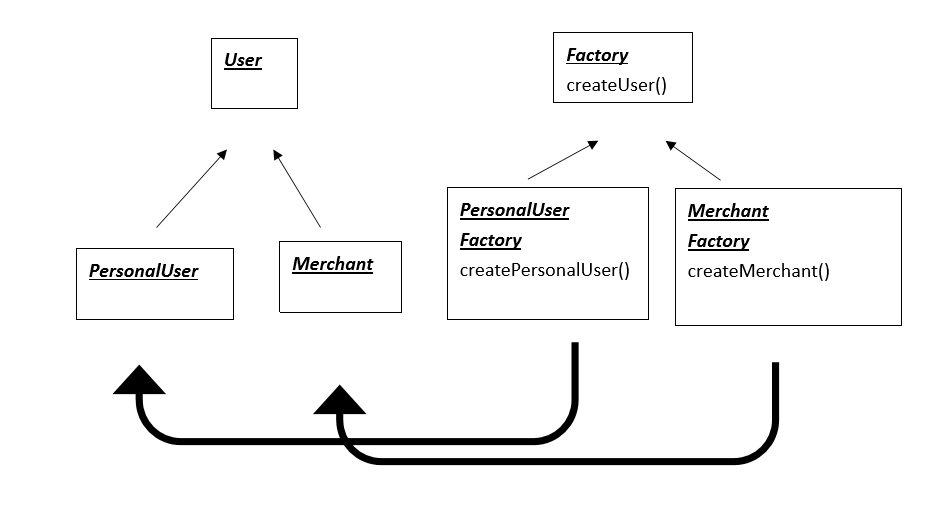
# 3创建型模式

创建型模式主要讨论以下几种模式：

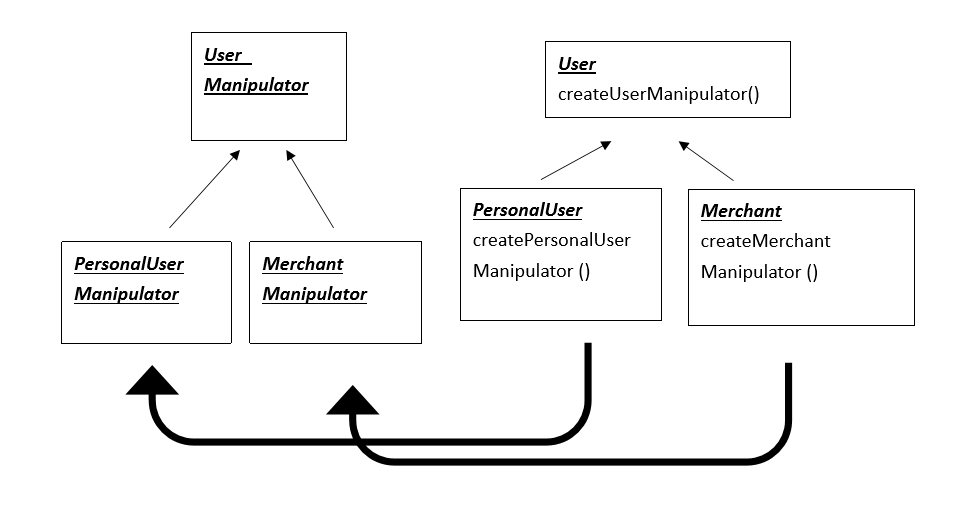
* Factory Method
* Prototype
* Singleton

## 3.1工厂方法(Factory Method)

工厂方法的意图是将类的实例化延迟至子类，让子类决定实例化哪一个类。工厂方法使得创建对象的操作更加灵活而不必局限于类的直接创建。 我们的用戶信息存放在数据库中，我们可以单独设立一个类来实现这些类的实例化，而不是通过直接调用这些类自己的构造函数来实例化。对于用戶类来 说，其子类的实例化通过调用工厂类的子类来进行实例化。在这里，User类及其子类的构造函数被忽略，代替它们的是工厂中的工厂函数。如下图所示，通过调用 Factory 类来生成对应的User 类，从而达到创建对象的目的。



对于各种数据的操纵类也可以使用工厂方法来生成。如下图所示，通过 User 类来生成对应的 manipulator，这种对应关系即为一种平行关系。 在这种情况下，工厂方法用于连接平行的类的层次。但是考虑到各个数据总是一直存在的，我们在软件运行过程中并不需要产生新的操纵类，因此我们通过工厂类来生成并不合算，所以这里不会采用此方法。



## 3.2原型(Prototype)

原型这一设计模式的意图在于用原型的实例来创建对象，通过拷贝原型来完成这个目的。我们能够将原型模式认作是基于原型的继承中,我们创建作为其它对象原型的对象，原型对象自身被当做构造器创建的每一个对象的蓝本高效的使用着，如果构造器函数使用的原型包含例如叫做name的属性,那么每一个通过同一个构造器创建的对象都将拥有这个相同的属性。当我们实例化多个几乎相同的User类时，我们可以用原型设计模式。使用这个模式，可以通过拷贝已有实例来进行实例化。

## 3.3单件(Singleton)

当我们需要保证一个类最多只有一个实例时，并且让它能够被全局访问时，我们可以使用单件模式。对于各个数据进行的操作被包裹在单独的类中，这个类可以应用单件模式。因此，对于前面所阐述的工厂类我们也可以应用单件模式，因为我们只需要一个工厂，而不需要多个。我们项目的数据操作类也可以使用单件模式。因为这些类只包含操作，没有必要有多个实例。而且，利用单件模式，可以避免数据库操作的冲突。而且，单件具有较好的扩展性，可以通过子类继承的方式对操作进行一些列拓展。

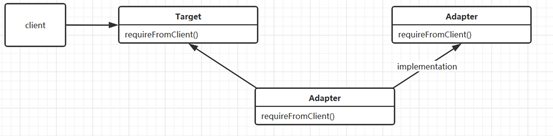
# 4结构型模式

结构型模式主要讨论以下几种模式：

* Adaptor
* Facade
* Flyweigh
* Proxy

## 4.1适配器(Adaptor)

适配器模式的重要作用在于使一些原本因为接口不兼容而无法一起工作的类能够在一起工作，作为两个不兼容的接口之间的桥梁。 当代码有多人协作时，我们最初定义的类之间的接口调用可能无法实现，从而导致接口相互不兼容，因此我们可以利用适配器模式来实现接口的兼容。例如，我们可能已经开发了一个类Adapter，但是我们所开发的另一个却是以Target类的形式想要调用Adapter这个类的，那么我们需要开发一个新的类Adapter去适配这一接口的不兼容情况。



## 4.2外观(Facade)

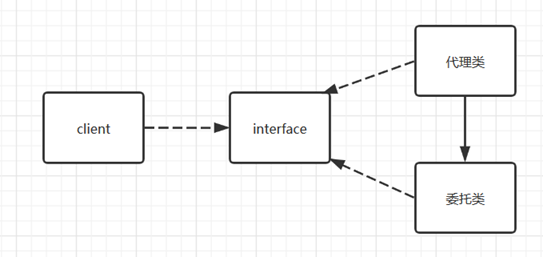
外观意在提供为子系统的一组接口提供一个高层的一致的接口，这一接口有利于降低系统的复杂性。 外观模式与适配器模式在本项目中的作用是类似的，都可以被用来解决由于合作编程中的不协调带来的一系列问题。在合作编程中，由于初始时对类的功能不明晰，各个成员写出来的相似的类的接口可能不一致，这时可以引⼊一个Facade对象来为其他调用者提供一致的API，⽽忽略每个类的接口可能不一致这样的事实。

## 4.3享元(Flyweigh)

享元模式主要用于减少创建对象的数量，以减少内存占用和提高性能。面向对象技术可以很好地解决一些灵活性或可扩展性问题，但在很多情况下需要在系统中增加类和对象的个数。当对象数量太多时，将导致运行代价过高，带来性能下降等问题。享元模式正是为解决这一类问题而诞生的。享元模式的宗旨是共享细粒度对象，将多个对同一对象的访问集中起来，不必为每个访问者创建一个单独的对象，以此来降低内存的消耗。享元模式把一个对象的状态分成内部状态和外部状态，内部状态即是不变的，外部状态是变化的；然后通过共享不变的部分，达到减少对象数量并节约内存的目的。享元模式其实就是工厂模式的一个改进机制，享元模式同样要求创建一个或一组对象，并且就是通过工厂方法生成对象的，只不过享元模式中为工厂方法增加了缓存这一功能。不过由于我们的用户数据是在数据库中一直存在的，因此使用享元模式的意义并不大。

## 4.4代理(Proxy)

代理模式为其他对象提供了一种代理以控制对这个对象的访问。对一个对象进行访问控制的一个原因是为了只有在我们确实需要这个对象时才对它进行创建和初始化，另一个原因则是为原有对象扩展其他自定义的功能。如下图所示，interface是一个普通的业务类型，无特殊要求；委托类是具体真实的角色，这个是处理真实的业务逻辑；代理类负责调用真实角色，可以在调用真实角色的方法前后添加一些自己的逻辑，即为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理。代理模式使得职责清楚，真实的角色实现自己的业务，代理角色可以额外处理一些事务，比如记录日志，访问控制等；扩展性好，如果业务逻辑变了，只修改真实具体角色就行。



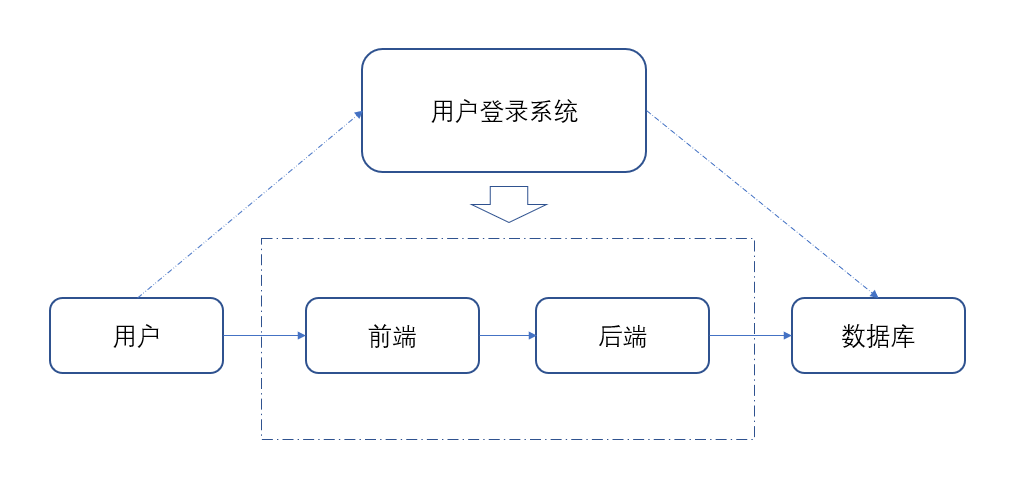
# 5行为模式

行为模式主要讨论以下几种模式：

* Chain of Responsibility
* Command
* Iterator
* Mediator
* Memento
* Observer
* State
* Strategy
* Template Method
* Visitor

## 5.1 责任链 （Chain of Responsibility）

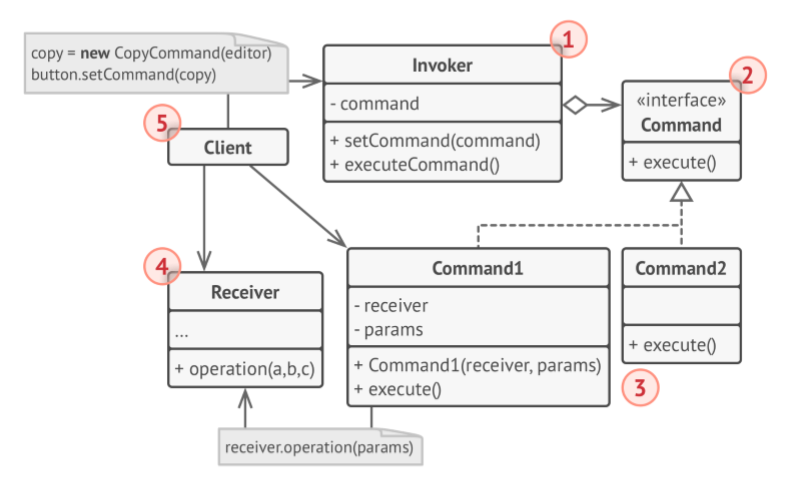
Chain of Responsibility 是通过将一个整体拆分成若干个线性的的逻辑分模块，确保每个逻辑链条上的分模块能够安全有序地处理信息，并且能够做到更加明确的责任分摊。这样有利于单个模块的测试，并且在修改单个模块的到时候，不会对上下游分模块造成影响。



在基础信息模块，我们可以用在项目分配中。具体来讲，我们会将“用户登录系统”分解为“前端”和“后端”，将责任合理分配到不同的模块当中，当用户对前端发生交互后，前端将信息初步整理成 JSON 格式，接着将结果传送到后端，由后端对 JSON 进行理解并且对数据库进行交互，最终会将结果格式化为 JSON 传送回前端，由前端自己负责以一个合理友好的方式来向用户汇报结果。此时，前端发生任何改变都不应该影响到后端的业务逻辑；同样的，后端的业务逻辑也不会改变前端的 UI 布局，这就是一种十分合理的 Chain of Responsibility.

## 5.2 命令（Command）

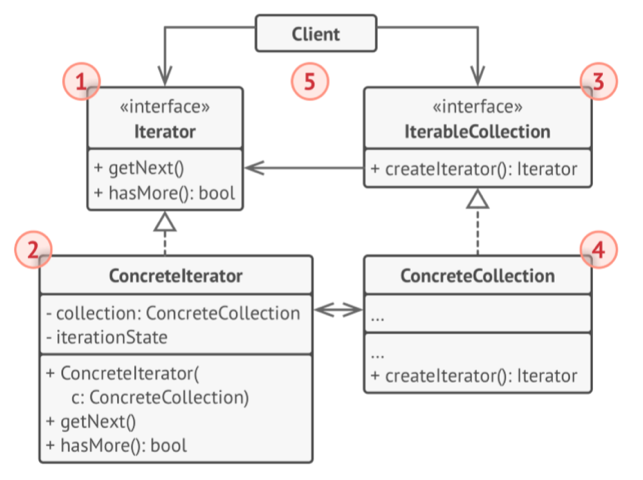
Command 的初衷是为了解决同一个业务逻辑在不同控件上重复定义导致的代码冗余问题。该设计模式本质是将调用者和具体 command 解耦分离，将 command 独立于调用者，因此能够既减少戴莫的冗余，又能使 command 免于受到调用者本身的改变而改变。



在基础模块中，我们也同样使用到了。在前端，按钮控件被划分为 View，而按键命令被划分为了 ViewModel。按键命令并不会知晓控件的存在，相反，是按钮作为调用者主动向命令发送调用请求。因此，如果接下来需求再次改变，将该命令同时实现在其他控件上面，比如说增加从菜单调用的方式，我们由于已经把该命令与控件分离，因此可以对菜单项增加相同的命令调用命令即可。再比如，如果新需求要求我们把业务逻辑做修改，即对命令做修改，那么修改的过程中不需要改动控件一个字。

## 5.3 循环体（Iterator）

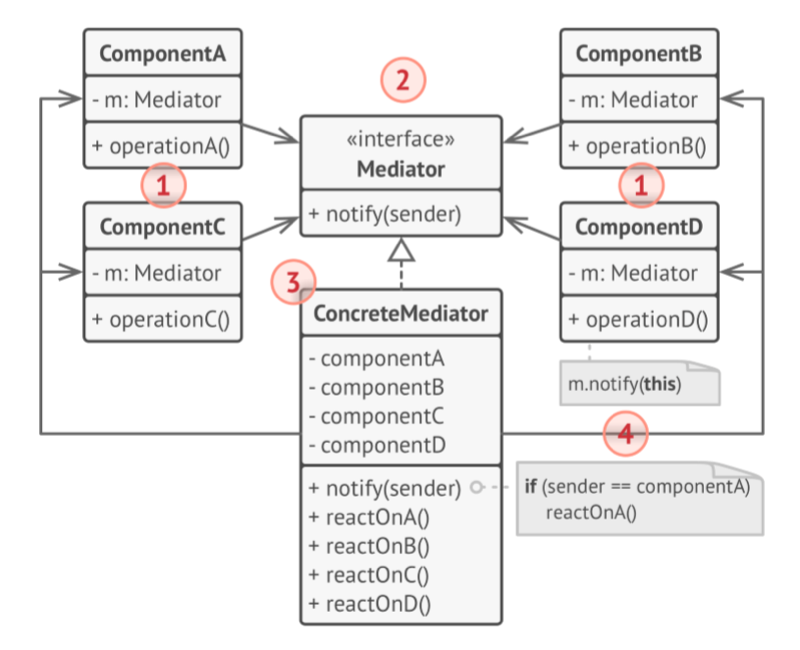
Iterator 目的是将数据结构隐藏的情况下仍然允许使用者能够对数据结构进行遍历的包装形式。在代码编写中，免不了会出现大量复杂的数据结构，那么如果需要代码写手能够专心于具体的业务逻辑实现，就需要将复杂的东西简单化，具体是将复杂包装在一个简单的外壳当中，允许程序员对封装类通过最简单的函数调用来执行十分复杂的过程。而对于 Iterator，主要是关注于数据结构的遍历过程。



在基础信息模块中，我们大量使用到了基础库的 Iterators，其中最具有代表性的就是 List。我们可以通过 for 循环体来将 List 内的所有元素都通过顺序遍历一遍，十分高效。

## 5.4 中介（Mediator）

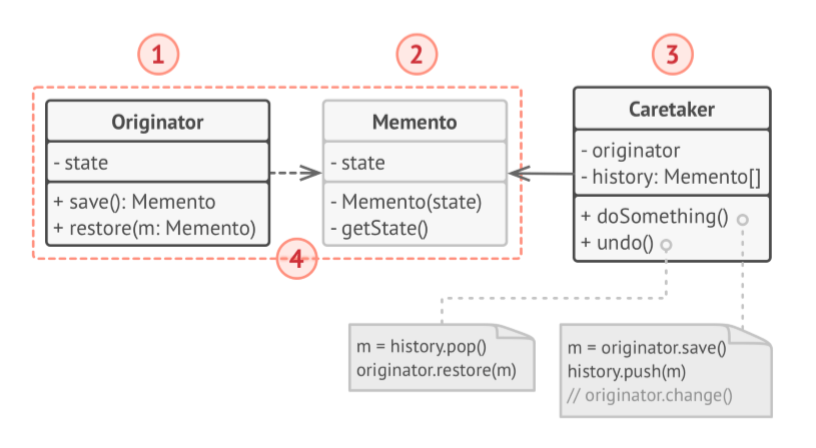
在开发前期，我们很容易陷入到错综复杂的代码逻辑当中，比如说，当一个按键被按下后，按钮会马上向后端发送请求，而按下菜单栏的一个项的时候，他也马上往后端发请求，直接接受后端的消息。这种方式是十分低效且容易发生依赖问题的，比如说在后端的接口发生修改后，刚才提到的两个控件都要修改其对应的代码。为了解决这种问题，我们会创建一个 Mediator 类，专门负责与后端的沟通交流，也是他集中收取后端发来的响应，由 Mediator 来做下一步信息分配。这样就能够解除大量错综复杂的依赖关系，大大减少耦合程度。



在基础信息模块，我们就完全通过这种方式来对后端进行交互，减少其他分模块的需求变更带来的修改压力。

## 5.5 备忘录（Memento (Snapshot)）

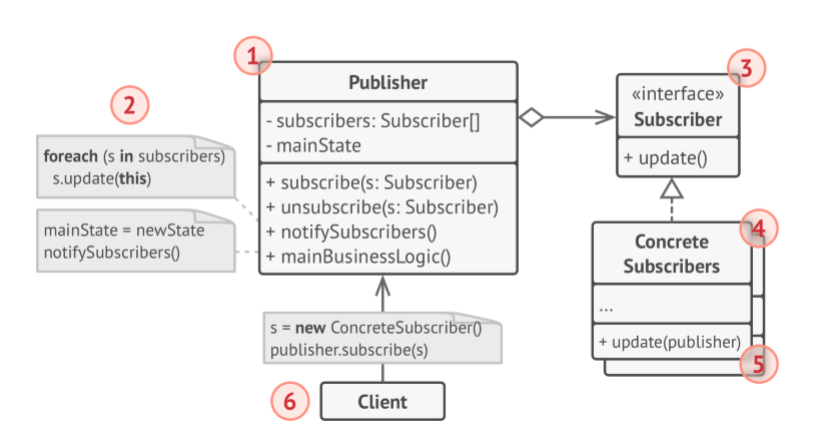
Memento 本质上是用来解决私有域的保存问题的。首先，由于类的功能专一性，我们需要把不同功能分解成若干个具体并且独立的类。在这里，我们打算保存一个类的状态信息，那么就需要另外做一个类来执行“保存”这个行为。但是问题是，我们需要保存的状态有一些是私有的，也就是说，负责保存的类无法获得需要保存的信息，因此无法正常实现功能。由于此，Memento 提供了一种解决方案，如下图所示：



Memento 作为 Originator 的内部类，能够被 Originator 访问所有内部的信息，也就是说可以在 Originator 内部将 state 信息转移到 Memento 内。又因为 Memento 内部的 state 是私有的，因此传给 Caretaker 时，Caretaker 并不能读取内部的 state 信息，只能充当一个对象管理的作用。这就实现了对状态的储存功能的同时保持合理的封装。

## 5.6 观察者（Observer）

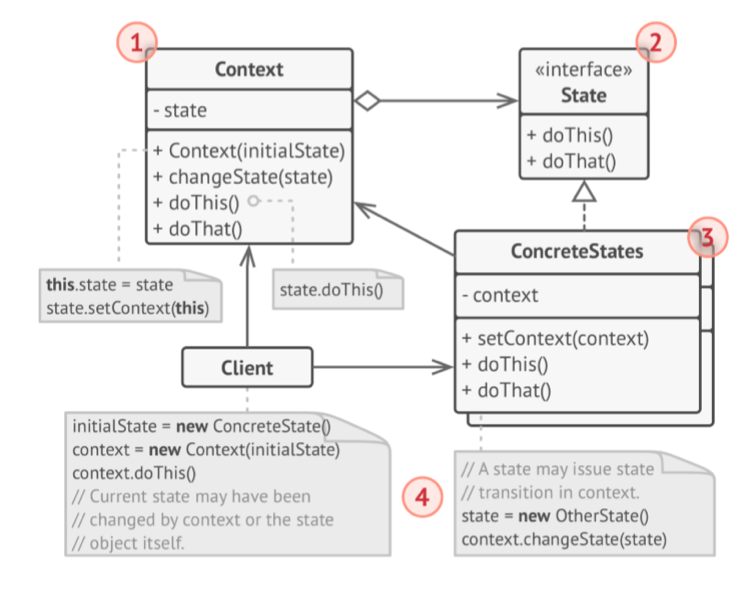
Observer 可以用于类之间高效的消息通讯。在传统应用场景下，如果一个界面上需要实现实时数据播报，他就可能隔一段时间主动向数据源发送请求，将新数据更新显示在界面上。但是由于数据源不一定发生了更新，那么多数的查询行为就变成了一种对时钟和算力资源的浪费。因此，Observer 模式下，界面首先向数据源注册，如果发生数据的更新，由信息源主动向界面报告信息的更新。因此可以节省不必要的资源消耗。



在基础信息模块中，我们对前端的信息使用了 Observer 的数据更新方式，即 View 会对 ViewModel 中的某些属性进行 Binding 或者叫做 subscribe。当 ViewModel 的某个属性发生了改变，那么将会触发 publisher，publisher 也就会遍历所有已注册的 View 中的 Objects 发送更新讯号，View 就会自动发生更新。通过这种方式，我们就不会在 View 中漏掉任何的数据更新，就是说，能够一举解决数据不同步的问题。

## 5.7 状态（State）

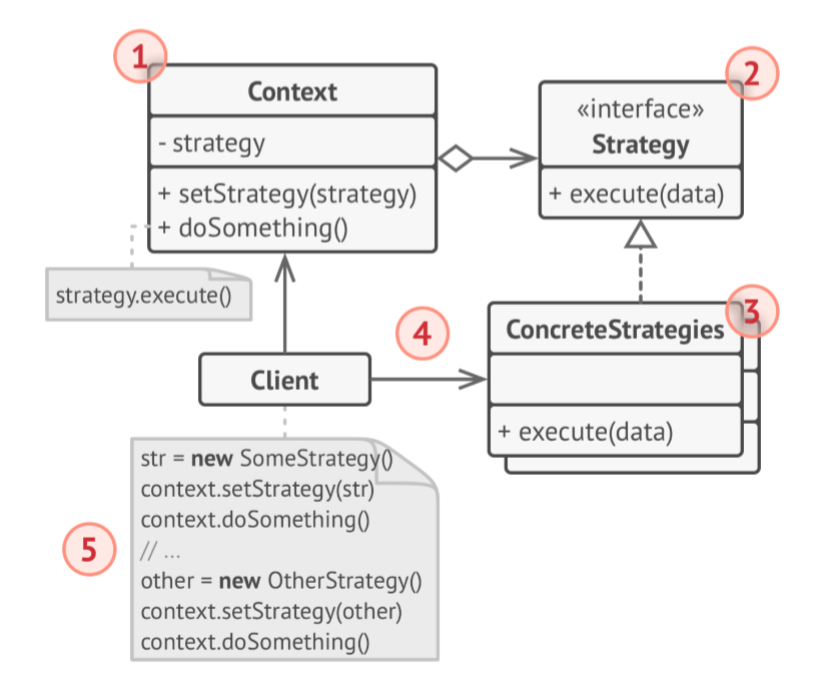
State 的概念首先是从计算理论中的状态自动机 FA 中出现的。状态自动机其实就是执行的指针在不同的 states 之间转移，在不同 state 内做相对应的行为。State 设计模式同样利用到了相同逻辑，此时，类是通过其中的 state 进行行为控制的。也就是说，类中的 state 会决定这个类的行为，并且会影响其下一个 state 的转移。



Context 就是需要被 state 驱使的类。对于一些功能，Context 本身并不会去实现，而是交由 ConcreteStates 来定义。在图中就是 doThis() 和 doThat()。又因为 Context 中的 state 是会发生变化的，因此，Context 的 doThis 和 doThat 行为也会由于 state 发生变化而随之发生变化。这样做可以将 Behaviors 与执行者分离，减少耦合，在将来对 state 修改的时候，不至于改动执行者的代码，从而实现行为与执行之间的解耦。

## 5.8 策略（Strategy）

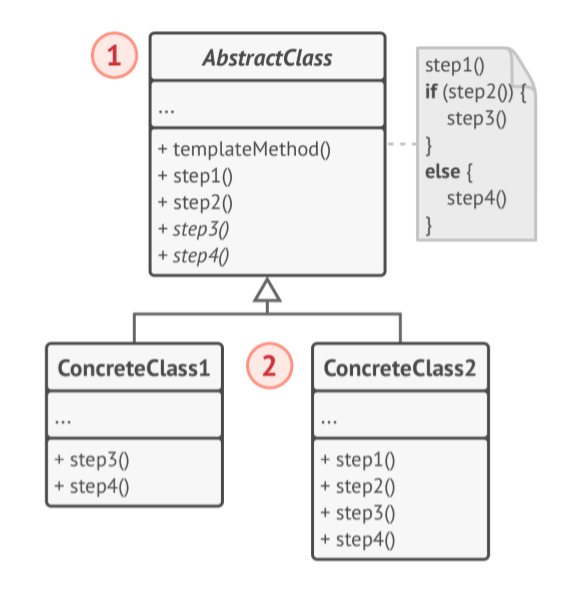
Strategy 与 State 有一个共同之处，就是都以 Behaviors 与执行分离为目标。但是不同的是，Strategy 并不关注 State 变化，只关注行为模块本身的封装性和独立性。



从图中也可以很清晰地发现，execute(data)这个行为的定义已经脱离了 Context，存在于一个类当中。也就是说，行为被转移至类当中，由类来维护一个独立的行为。而调用者 Context 并不需要知道这个行为具体是如何实现的，他只需要使用已经包装好了的行为 Objects 即可。这会使得在行为逻辑的修改的时候，不会影响到 Context 的代码。也就是使行为和调用相互解耦。

## 5.9 模板（Template Method）

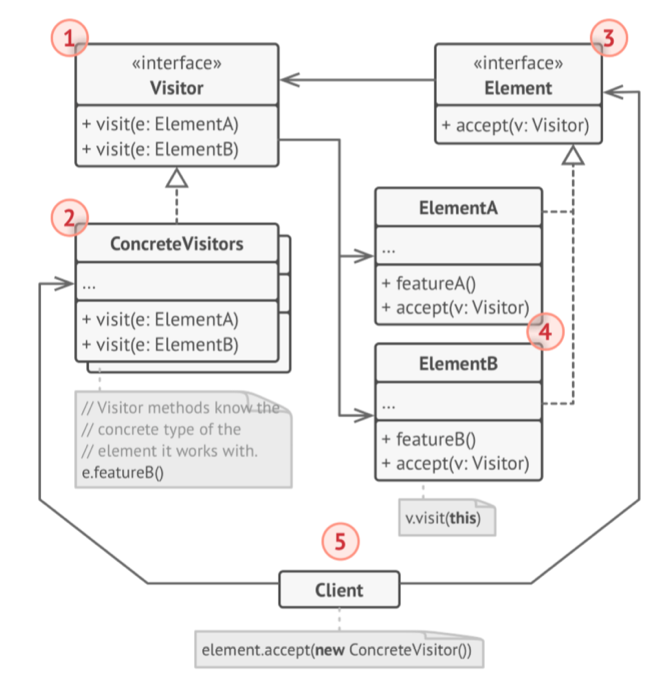
Template Method 主要就是把具体的 methods 抽象成没有被具体实现的的 virtual methods，这些 methods 并不需要对具体行为进行定义，而继承于他的类就需要把这些抽象的 methods 逐个 overrides 并且详细定义。目的是统一整个骨架，但是允许继承他的 subclasses 能够做出一些限制范围内的特异化。



虽然说 AbstractClass 是 Abstract，但是其中的 templateMethod 却是已经定义好了的，这是骨架，不能被改变。因此，继承于它的 ConcreteClass 就需要把抽象的步骤实现，在不影响骨架结构的情况下实现自定义的行为。

## 5.10 浏览者（Visitor）

Visitor 本质上是一个类，这个类里面包含了对不同 objects 中公开信息的格式转换输出。



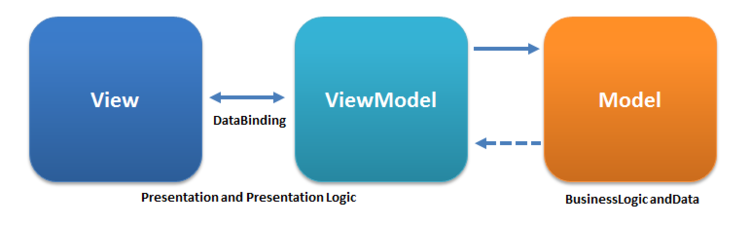
值得注意的是，Visitor 中的 Method 是由其他 objects 主动触发的。而作为最外层，程序员通过 objects 内的 accept 函数来间接调用 Visitor。

# 6基础信息模块设计模式分析

我们在总体上采用的是 MVVM 架构 (Model-View-ViewModel)。这种架构是基于 MVC 架构之上发展而来的。它是由在 Microsoft 工作的架构师 Ken Cooper and Ted Peters 发明的。这种设计主要针对的是事件驱动性的程序的架构优化。在实际产品中，WPF (Windows Presentation Foundation) 有着十分优异的表现。

MVVM 的设计目的是为了进一步解决 MVC 等架构中 View 和 Model 的耦合性问题。在 MVC 中，如果 View 需要了解 Model 的更新情况，他需要在 Controller 【显式地】把信息呈递过来；并且 Controller 并非与 View 一一对应，因此就会出现一个 Controller 服务很多个 View 的情况，代码也会变得十分冗杂，Controller 内部逻辑也会变得十分耦合。在 MVVM 中，加入了 ViewModel。他就是用来解决 Controller 复杂度爆炸问题的。

* Model 本质上是一个独立的 class，这个 class 主要承担的是不同模块之间交换信息的媒介。也就是说，如果两个模块之间需要传递一个信息，他就需要使用 Model 对信息进行包装，才能进行交换。又因为多个模块会用到相同的 Model，Model 就成为了一种标准化工具，用于标准化接口。在我们的实战经验中，Model 大多是用于表达数据库中的 table 结构，每个 instance of Model 分别代表着一个 row 的信息。
* View 就是用户能够看得到摸得着的图形界面。他是跟用户直接打交道的界面。View 中不会存在关于数据信息任何业务逻辑，最多只会有针对动画效果等方面的代码逻辑。当然，一个程序当然是可以在无 View 的情况下工作的，这种形式在 Windows 中被称为 Services。View 需要知道 ViewModel 的存在，当然也包括 Model。
* ViewModel 与 View 一一对应，但是 ViewModel 完全不知道 View 的存在，他只依赖 Model。因此，ViewModel 可以完全独立于 View 而存在。ViewModel 会暴露一些 attributes，方便 View 捕获，并且 Binding 在 View 自己的 Controls 上。Binding 就是 ViewModel 存在的精髓所在。这种 Binding 可以让 View 直接改动 ViewModel 中 attributes 的值，也可以在 ViewModel 改变了 attributes 的值之后同步 View 上显示的数据信息。因此，这是一种 【隐式的】 数据传递，可以杜绝信息不同步等问题。



## 架构具体实现方法

### Model

由于 Model 的数量过多，此处仅展示部分典型案例。

public class Person

{

[Key]

[Required]

[Column(TypeName = "varchar(30)")]

[StringLength(30)]

public string Username { get; set; }

[Required]

public int Status { get; set; }

[Required]

[Column(TypeName = "varchar(20)")]

[StringLength(20)]

public string Name { get; set; }

[Required]

[Column(TypeName = "varchar(1)")]

[StringLength(1)]

public string Gender { get; set; } *//*

*// [Required]*

[Column(TypeName = "varchar(50)")]

[StringLength(50)]

public string IdentityCardNumber { get; set; }

*// \_\_2nd\_\_*

*// [Required]*

[Column(TypeName = "varchar(50)")]

[StringLength(50)]

public string PhoneNumber { get; set; }

[Required]

[Column(TypeName = "varchar(50)")]

[StringLength(50)]

public string Email { get; set; }

*// [Required]*

[DataType(DataType.Date)]

public DateTime? UpdateTime { get; set; } *// program generated*

*// [Required]*

[DataType(DataType.Date)]

public DateTime? CreationTime { get; set; } *// program generated*

[Required]

[Column(TypeName = "varchar(30)")]

[StringLength(30)]

public string Password { get; set; }

}

从中不难看出，Model 主要承担的是数据的定义和限制。在每个 attribute 上都有相对应的 tags，这些 tag 从字面上就能够理解他们所起到的限定性作用。每个 attribute 的类型同样也是具体的限定方式。

此处的 Model 主要是用于 ViewModel 与 Database 的交互数据，并且用于 View 与 ViewModel 之间的 Binding 对象。

### ViewModel

由于 View 与 ViewModel 为一一对应关系，并且 View 依赖于 ViewModel，因此首先介绍 ViewModel。下面举一个例子：

public class CreateModel : PageModel

{

private readonly ControlPanel.Data.BackdoorContext \_context;

public CreateModel(ControlPanel.Data.BackdoorContext context)

{

\_context = context;

}

public IActionResult OnGet()

{

return Page();

}

[BindProperty]

public Person Person { get; set; }

#region snippet\_OnPostAsync

public async Task<IActionResult> OnPostAsync()

{

#region snippet\_TryUpdateModelAsync

var emptyPerson = new Person();

if (await TryUpdateModelAsync<Person>(

emptyPerson,

"person", *// Prefix for form value.*

s => s.Username, s => s.Name, s => s.Gender, s => s.IdentityCardNumber,

s => s.Status, s => s.PhoneNumber, s => s.Email, s => s.Password,

s => s.Address, s => s.HealthStatus, s => s.HealthCode, s => s.Visitedplaces,

s => s.PaymentInformation, s => s.PersonalCenterLink))

{

emptyPerson.CreationTime = DateTime.Now;

emptyPerson.UpdateTime = DateTime.Now;

\_context.PersonalUserInfo.Add(emptyPerson);

await \_context.SaveChangesAsync();

return RedirectToPage("./Index");

}

#endregion

return Page();

}

#endregion

}

这个 class 主要维护 create person 的业务逻辑。可以看到，他利用到了我们刚才提到的 Model 作为 attribute，让 View 能够直接对 这个 Model 作修改。在下面也定义了一个 Command，这个 command 并非依赖于任何一个或一种 Control，他是独立存在的，只有一份 copy。

在例子当中，当我们触发 OnPost 行为，他会把表单中的 Model 从 BindProperty 中提取。接着对 Model 进行简单的合法性检查以后，把合法的 Model 数据传递给数据库管理模块，使数据库持久化该 Model。

### View

View 就是我们所说的 GUI 了。View 提供一切用户需要看到和交互的工具和 Controls。他们共同组成 View。View 的形式并无任何限制，只要稍加改动，View 就可以为手机应用；若使用 HTML 定义 View，那么就变成了一个 Web 应用了。而在此过程中，ViewModel 和 Model 无需发生任何一个字的改动，也就可以说，整个过程中只需要 View 去配合 ViewModel 中暴露的 attributes 即可，可以实现仅仅单向的依赖，十分解耦。

# 7参考资料

Alexander, C. (1996). "The Pattern of Streets". *JOURNAL OF THE AIP*.

Bass, L., Clements, P., & Kazman, R. (2015). *Software architecture in practice*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Shvets, A. (2019). *Dive Into Design Patterns*.