行为型模式

1. 定义

描述程序在运行时复杂的流程控制。

描述多个类或对象之间怎样相互协同共同完成无法单独完成的任务。

涉及算法与对象间职责的分配。

1. 模板方法模式

①定义

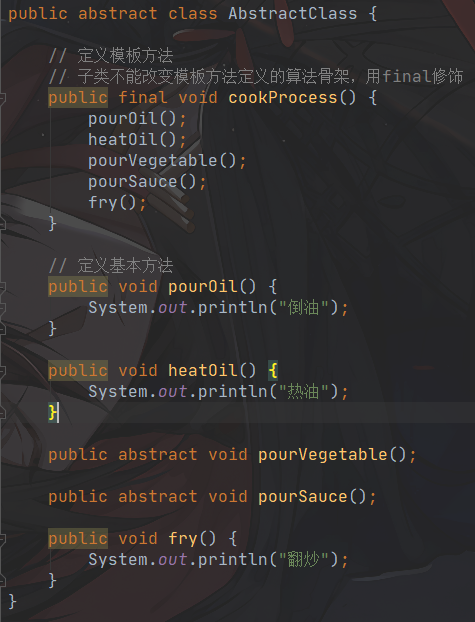
定义一个操作中的算法骨架（取号→排队→办理具体业务→服务评分），将算法的一些步骤延迟到子类中（办理的具体业务，如存款取款等），使得子类可以在不改变算法结构的情况下重定义该算法的某些特定步骤。

②组成

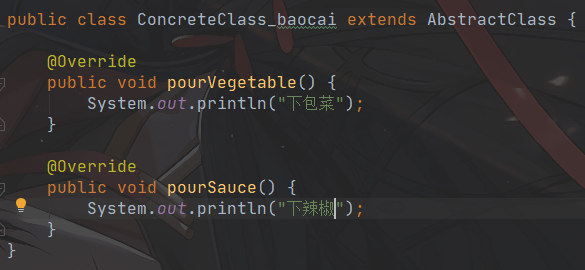
以“炒菜”为例。

（1）抽象类角色：给出算法的轮廓和骨架。包含模板方法（算法骨架，按顺序调用基本方法）和基本方法（算法的各个步骤，包含抽象方法、具体方法和钩子方法）。

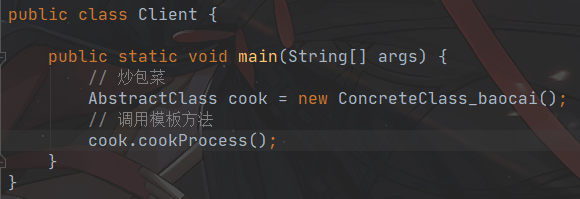
钩子方法：判断的逻辑方法，返回值一般是布尔类型。



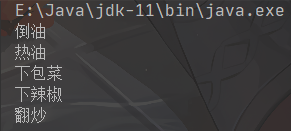
（2）具体子类



（3）客户类



输出：



③优点

（1）提高代码的复用性

（2）实现了反向控制：父类调用子类中的方法，符合开闭原则。

④缺点

（1）类的个数增加

（2）反向控制的结构，提高了代码阅读的难度

⑤适用场景

（1）算法的整体架构固定，但个别部分易变

（2）需要实现子类反向控制父类

1. 策略模式

①定义

定义一系列算法，并将每个算法封装起来，使其可以相互替换，且算法的变化不会影响使用算法的客户。

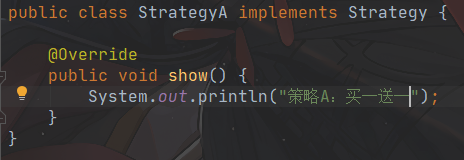
②组成

以百货公司促销活动为例。

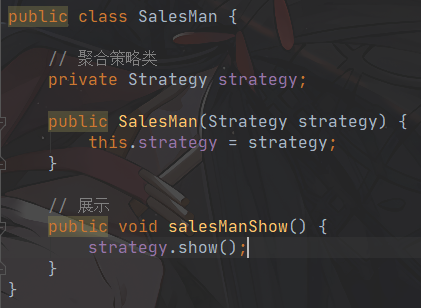
1. 抽象策略类：给出所有具体策略类所需的接口



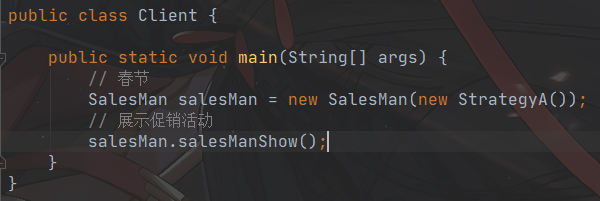
1. 具体策略类：实现抽象策略类的接口



1. 环境类：持有一个策略类的引用，最终给客户端调用



1. 客户类



③优点

1. 策略类间可以自由替换（实现了相同的接口）；
2. 易于扩展；
3. 避免使用多重条件选择语句。

④缺点

1. 客户端必须知道所有的策略类，并自行决定使用哪个策略类；
2. 结合享元模式能够减少对象的数量。

⑤适用场景

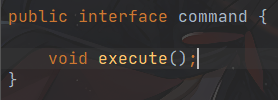
1. 一个系统需要动态地在几个算法中选择一种；
2. 代替大量的条件分支语句；
3. 各个算法彼此完全独立，要求对客户隐藏具体算法的实现细节；
4. 隐藏与算法相关的数据结构；
5. 多个类只区别在表现行为不同。
6. 命令模式

①定义

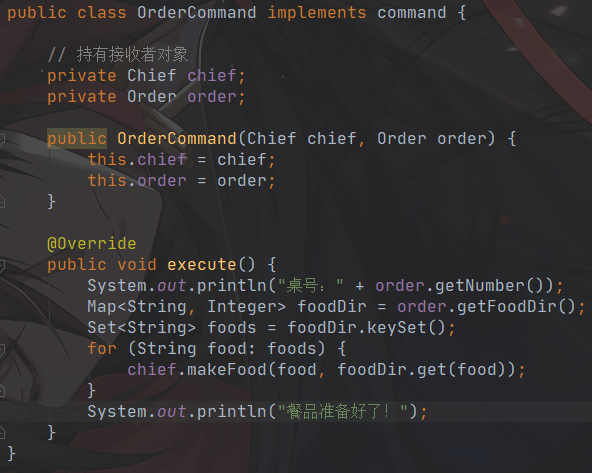
将一个请求封装为一个对象，使发出请求（招待）和执行请求（厨师）的责任分隔开。

②组成

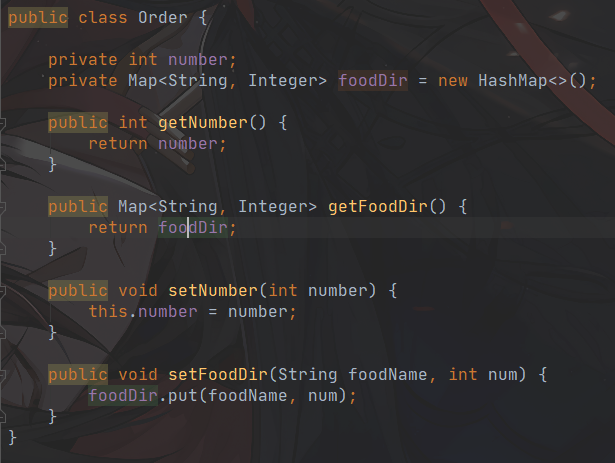
1. 抽象命令类角色：定义命令的接口；



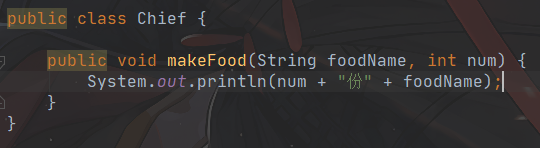
1. 具体命令角色：实现命令接口，通常会持有接收者，调用接收者的功能完成命令要执行的操作；



这里具体命令角色还持有Order对象（订单），它属于命令的一部分。



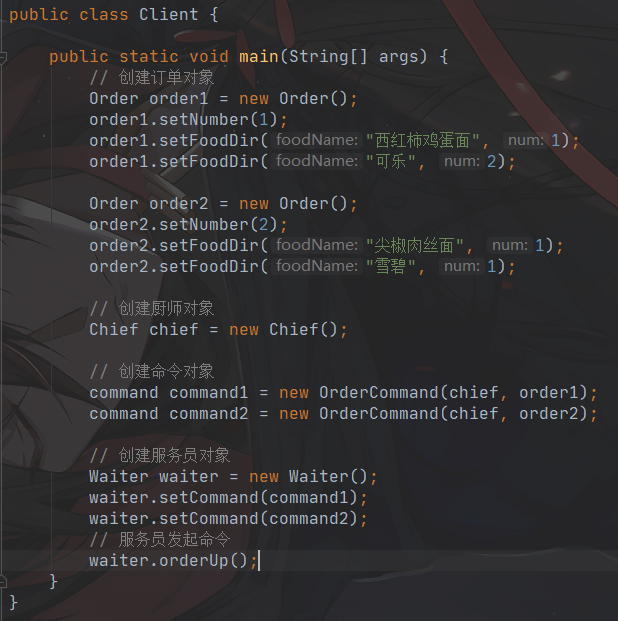
1. 接收者：执行命令的对象；



1. 请求者：发出命令，通常持有命令对象。



1. 客户类



③优点

1. 降低系统的耦合度；
2. 增加命令非常方便，扩展灵活；
3. 可以实现宏命令（将多个命令装配成一个组合命令）；
4. 可以实现Undo和Redo，与备忘录模式结合。

④缺点

1. 可能导致某些系统有过多的命令类；
2. 系统结构更加复杂。

⑤适用场景

1. 系统希望请求者和接收者解耦；
2. 请求排队；
3. 系统希望支持撤销和恢复操作。
4. 责任链模式

① 定义

一个请求有多个对象可以处理，但每个对象的处理条件或权限不同。

为了避免请求发送者和多个请求处理者耦合，将所有请求的处理者通过前一对象记住其下一个对象的引用，连成一条链。当有请求发生时，可将请求沿着这条链传递，直到有对象处理为止。

② 组成

1. 抽象处理者：定义处理请求的接口，包含抽象处理方法和一个后继连接；

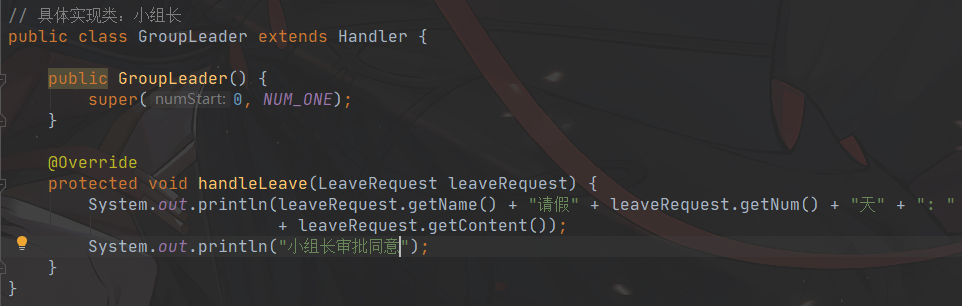


把构造方法在抽象类中实现，子类只需要调用抽象类的构造方法。

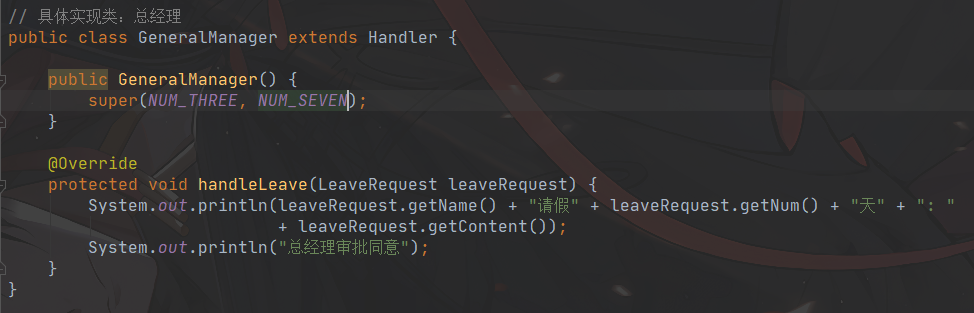
setNextHandler为了让客户类构造责任链。

submitLeave方法处理责任链的逻辑：当前对象处理能够处理的请求，不能处理的部分交给接下来的对象处理。

1. 具体处理者：实现处理方法；



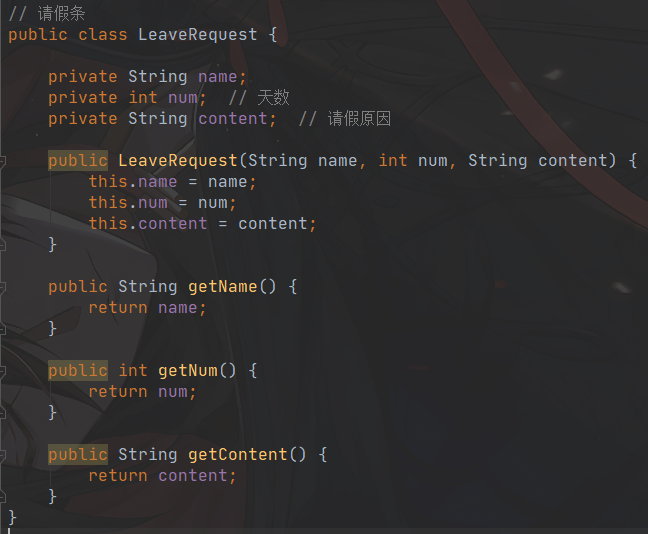




这个例子中，责任链上的每一个对象都对请求进行处理，但是可能只能处理一部分，剩下的部分交给接下来的对象处理。

1. 客户类：创建责任链，将请求发送给链头的处理者，不关心请求的具体处理方式。

请求类：



③ 优点

1. 降低了对象之间的耦合度；
2. 增强了系统的可扩展性；
3. 增强了给对象指派职责的灵活性；

对于上述（2）（3），只需要换一种方式创建责任链即可。

1. 简化了对象之间的连接；
2. 职责分担。

④ 缺点

1. 不能保证每一个请求一定能被处理；
2. 对于较长的责任链，系统性能将受到一定影响；
3. 责任链的合理性要靠客户端保证。
4. 状态模式

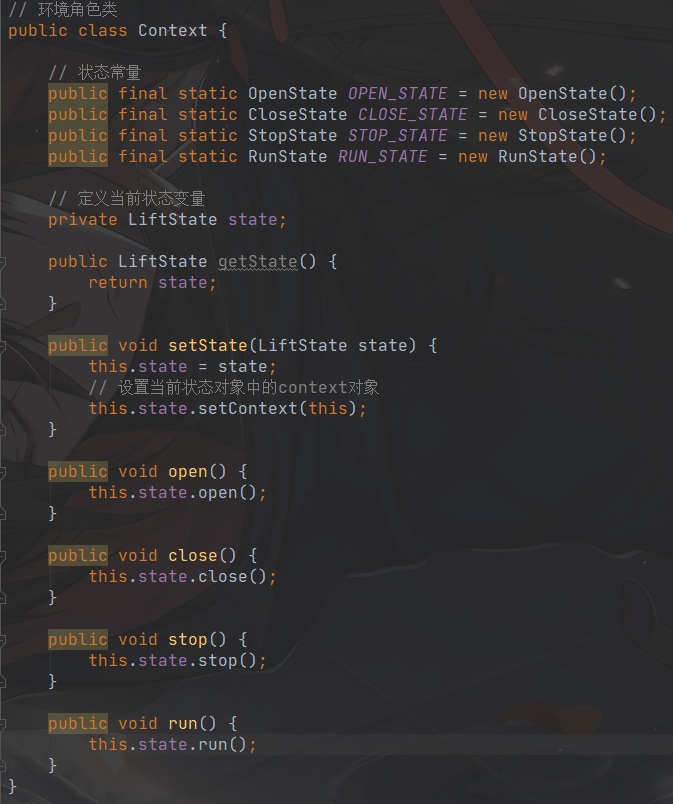
① 定义

允许状态对象在其内部状态发生改变时改变其行为。

② 组成

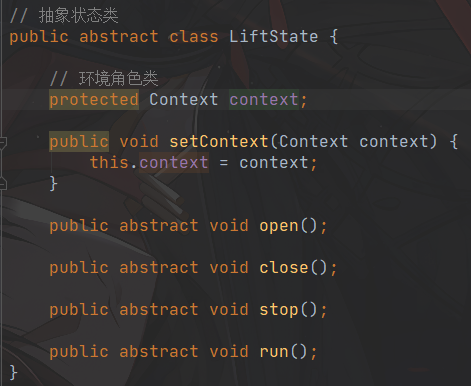
1. 环境角色：定义客户程序需要的接口，维护一个当前状态，将与状态相关的操作委托给当前状态的对象处理；

当前状态：state



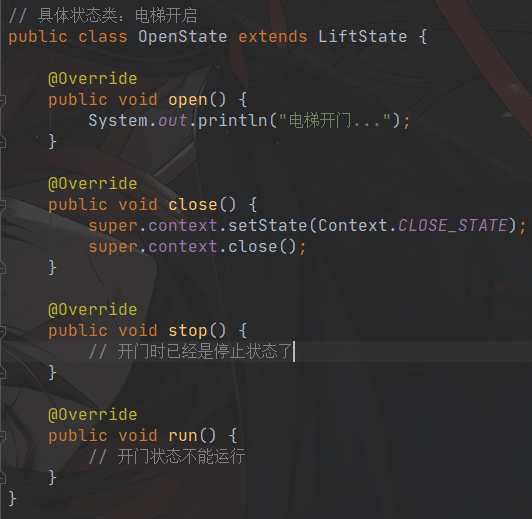
1. 抽象状态角色：接口，封装环境对象中特定状态对应的行为；

这里必须要聚合环境角色的对象，因为具体状态角色需要依靠环境角色的对象修改状态或调用其它具体状态角色实现的方法。



1. 具体状态角色：实现抽象状态角色对应的行为。

以电梯开门为例，每个具体状态角色只实现自己负责的那一部分，其它的方法调用其它具体状态角色的实现。



③ 优点

1. 将所有与某个状态有关的行为放在同一个类中，可以方便地增加新的状态；
2. 避免巨大的if-else代码块。

④ 缺点

1. 增加系统类和对象的个数；
2. 结构与实现都较为复杂；
3. 不能很好地支持“开闭原则”。

⑤ 适用场景

1. 对象的行为取决于它的状态；
2. 操作中存在庞大的分支结构，且这些分支结构决定于对象的状态。
3. 观察者模式

① 定义

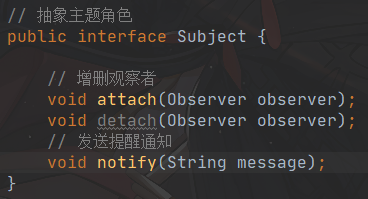
让多个观察者对象同时监听某一个主题对象。当主题对象的状态发生变化时，会通知所有的观察者对象，让它们自动更新自己。

② 组成

以微信公众号为例。

1. 抽象主题角色：将所有观察者对象保存在一个集合里。提供一个接口，可以添加或者删除观察者对象（公众号的抽象类）；

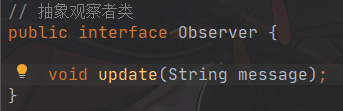
集合在子类中定义，因为抽象主题角色是一个接口。



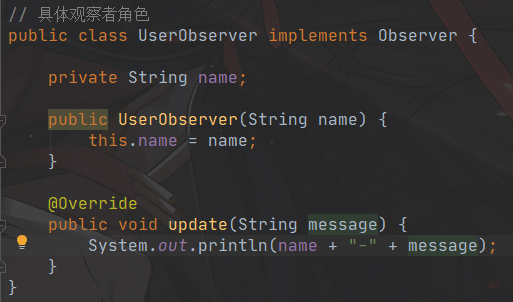
1. 具体主题角色：存入具体观察者对象，在内部状态发生改变时，给所有注册过的观察者发送通知（公众号）；



1. 抽象观察者角色：定义更新接口，在接收到主题改变通知时更新自己（用户的抽象类）；



1. 具体观察者角色：实现抽象观察者的更新接口（用户）。



③ 优点

1. 降低了主题和观察者之间的耦合关系（抽象耦合）；
2. 可以实现广播机制。

④ 缺点

1. 如果观察者的数量特别多，观察者接收到的信息会延时；
2. 如果主题存在循环依赖，会导致观察者循环调用，导致系统崩溃。

⑤ 适用场景

1. 对象间存在一对多关系，且一个对象的状态变化会影响到其它对象；
2. 一个抽象模型有两个方面，其中一个方面依赖于另一方面。
3. 中介者模式

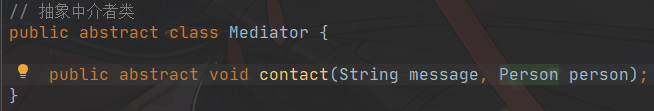
① 定义

封装一系列对象之间的交互，使原有对象间的耦合松散，且可以独立地改变它们之间的交互。

② 组成

以租房为例。

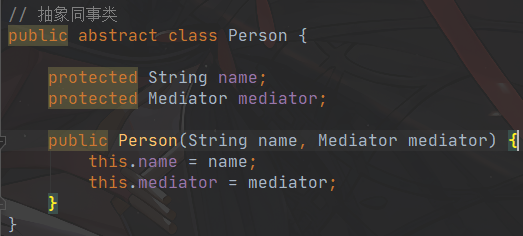
1. 抽象中介者角色：中介者的接口，提供了同事对象注册和转发同事信息的抽象方法（房屋中介的抽象类）；



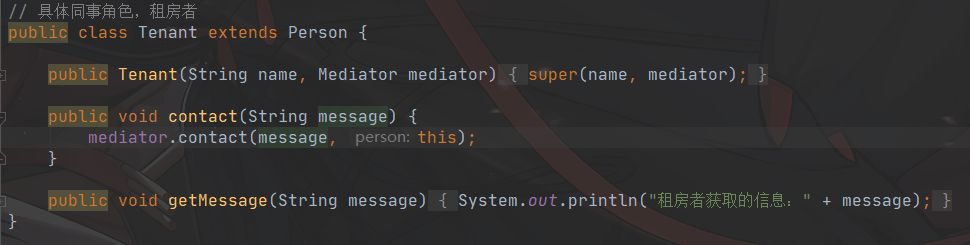
1. 具体中介者角色：实现中介者接口，定义一个集合管理同事角色，协调各个同事角色之间的交互关系（房屋中介）；

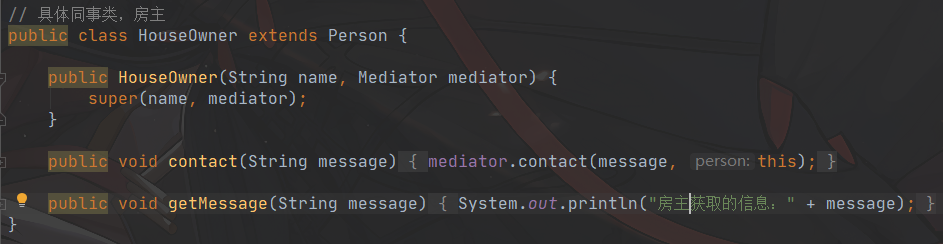


1. 抽象同事类角色：同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有公共功能（租房者和房主的抽象类）；



1. 具体同事类角色：实现同事类接口。当需要与其它同事对象交互时，由中介者对象负责后续的交互（租房者和房主）。





整体逻辑：

租房者发送信息（contact）→调用中介者的contact方法→中介者调用房主的getMessage方法→房主的getMessage方法打印信息。

③ 优点

1. 松散耦合：同事对象之间松散耦合；
2. 集中控制交互：交互被封装在中介者中（租房者发送信息给房主，房主返回信息给租房者）；
3. 一对多关联转变为一对一关联：同事对象和中介者呈现星形结构。

④ 缺点

当同事类太多时，中介者的职责将会很大。

⑤ 适用场景

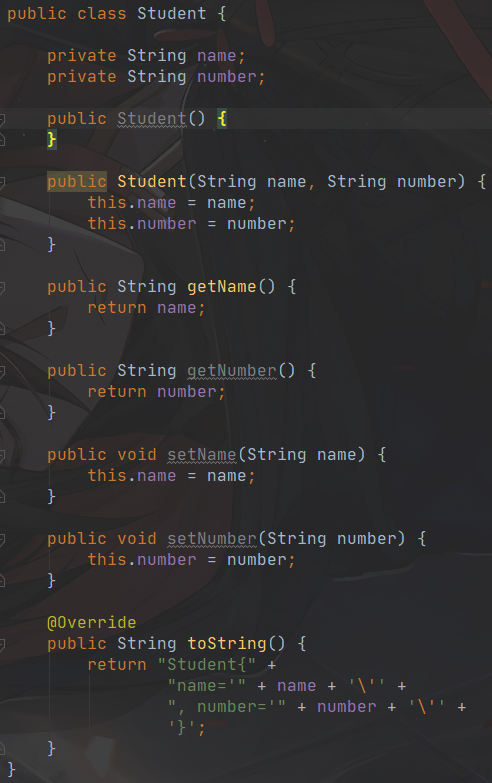
1. 系统中对象存在复杂的引用关系；
2. 想创建一个运行于多个类之间的对象。
3. 迭代器模式

① 定义

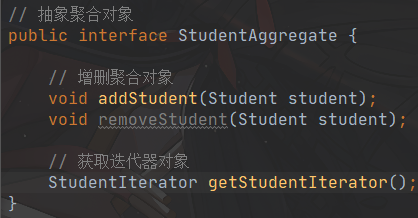
提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

② 组成

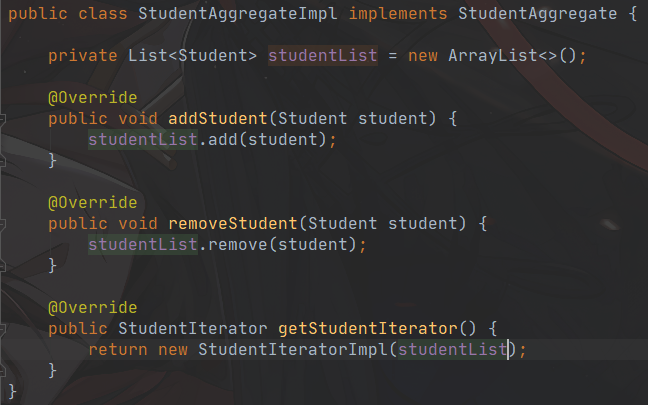
以存储学生对象的容器对象为例。



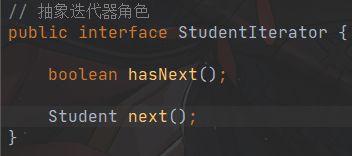
1. 抽象聚合角色：定义存储、添加、删除聚合元素，创建迭代器对象的接口；



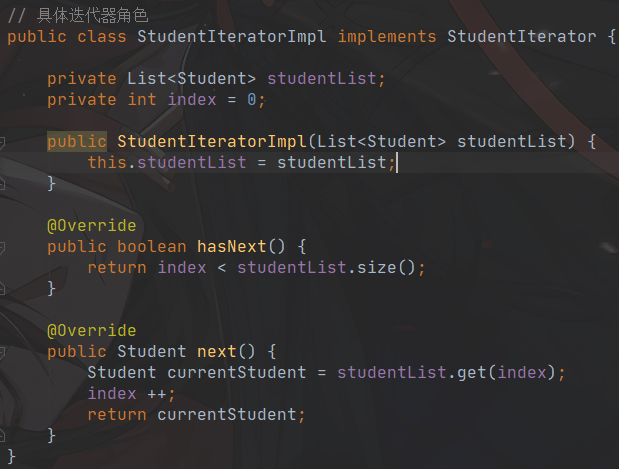
1. 具体聚合角色：实现抽象聚合角色，返回一个具体迭代器的实例；



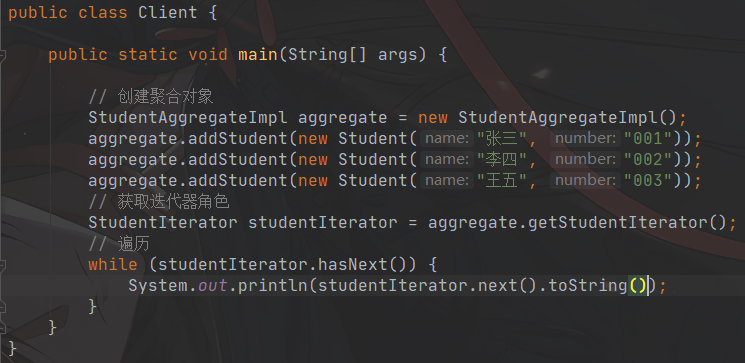
1. 抽象迭代器角色：定义访问和遍历聚合元素的接口；



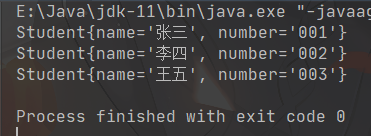
1. 具体迭代器角色：实现抽象迭代器角色，完成对聚合对象的遍历，记录遍历的当前位置。



1. 客户类



输出结果：



③ 优点

1. 支持以不同的方式遍历一个聚合对象；
2. 在原有的聚合对象中不需要自行提供数据遍历等方法；
3. 可以很方便地增加新的聚合类和迭代器类，满足“开闭原则”。

④ 缺点

增加了类的个数，在一定程度上增加了系统的复杂性。

⑤ 适用场景

1. 需要为聚合对象提供多种遍历方式；
2. 需要为不同的聚合结构提供一个统一的接口；
3. 访问一个聚合对象而无须暴露其内部细节。