**设计模式**

目录

[设计模式 1](#_Toc3170)

[第一章 设计模式——设计模式概述 7](#_Toc16430)

[1.设计模式由来 8](#_Toc6195)

[2.设计模式的优点 8](#_Toc14677)

[第二章 设计模式（二）——UML类图介绍 9](#_Toc4107)

[1.UML是什么？ 9](#_Toc31879)

[1.1.UML类图常用关系 9](#_Toc18536)

[1.2.UML模型结构 9](#_Toc18235)

[1.3.UML类的表示 9](#_Toc8204)

[2.类之间的关系 10](#_Toc24492)

[2.1.泛化(Generalization) 10](#_Toc9412)

[2.2.实现(Realization) 10](#_Toc25350)

[2.3.关联(Association) 10](#_Toc29523)

[2.4.聚合(Aggregation) 12](#_Toc17950)

[2.5.组合(Composition) 12](#_Toc21463)

[2.6.依赖(Dependency) 12](#_Toc9427)

[2.7.注意事项 12](#_Toc24088)

[第三章 设计模式（三）——面向对象设计原则 12](#_Toc7561)

[1.单一职责原则 13](#_Toc3498)

[2.开闭原则 13](#_Toc8537)

[3.里氏代换原则 13](#_Toc11210)

[4.依赖倒转原则 13](#_Toc9964)

[5.接口隔离原则 14](#_Toc25939)

[6.合成复用原则 14](#_Toc23846)

[7.迪米特法则 14](#_Toc21833)

[第四章 再不学简单工厂模式，就真的要去工厂搬砖啦！ 14](#_Toc9211)

[1.简单工厂模式简介 14](#_Toc20679)

[1.1.简单工厂应用举例 15](#_Toc1323)

[1.2.简单工厂基本实现流程 15](#_Toc26453)

[1.3.简单工厂定义 15](#_Toc29634)

[2.简单工厂模式结构 15](#_Toc25328)

[3.简单工厂模式代码实例 17](#_Toc4219)

[3.1.定义抽象产品类AbstractProduct，抽象方法不提供实现 17](#_Toc11591)

[3.2.定义三个具体产品类 17](#_Toc25226)

[3.3.定义工厂类和工厂方法 18](#_Toc27647)

[3.4.客户端使用方法示例 19](#_Toc12112)

[3.5.效果 19](#_Toc29694)

[4.简单工厂模式总结 19](#_Toc12544)

[第五章 不知道工厂方法模式？你可能真的应该去工厂！ 20](#_Toc29700)

[1.工厂方法模式简介 20](#_Toc11087)

[2.工厂方法模式结构 20](#_Toc31005)

[3.工厂方法模式代码实例 21](#_Toc5780)

[3.1.定义抽象产品类AbstractSportProduct，方法不提供实现 22](#_Toc17339)

[3.2.定义三个具体产品类 22](#_Toc7437)

[3.3.定义抽象工厂类AbstractFactory，方法为纯虚方法 23](#_Toc23156)

[3.4.定义三个具体工厂类 23](#_Toc17184)

[3.5.客户端使用方法示例 24](#_Toc28585)

[3.6.效果 25](#_Toc6865)

[4.工厂方法模式总结 25](#_Toc6587)

[第六章工厂搬砖也要带脑壳！Jungle学抽象工厂模式 26](#_Toc9162)

[1.抽象工厂模式简介 26](#_Toc6504)

[2.抽象工厂模式结构 26](#_Toc17564)

[3.抽象工厂模式代码实例 27](#_Toc2182)

[3.1.定义产品类 27](#_Toc28459)

[3.2.定义工厂类 29](#_Toc23360)

[3.3.客户端使用方法示例 30](#_Toc20838)

[3.4.效果 30](#_Toc12405)

[第七章 建一栋房子总共分几步？建造者模式告诉你答案！ 31](#_Toc15537)

[1.建造者模式简介 31](#_Toc4599)

[2.建造者模式结构 31](#_Toc427)

[3.建造者模式代码实例 32](#_Toc2987)

[3.1.定义产品类House 32](#_Toc11073)

[3.2.定义建造者 33](#_Toc30953)

[3.3.定义指挥者 34](#_Toc6950)

[3.4.客户端代码示例 35](#_Toc8578)

[3.5.效果 36](#_Toc23329)

[4.建造者模式总结 36](#_Toc19547)

[第八章 克隆以后我就成了你——探究原型模式 36](#_Toc20957)

[1.原型模式简介 36](#_Toc23421)

[2.原型模式结构 37](#_Toc16150)

[3.关于克隆方法：浅拷贝/深拷贝 37](#_Toc26124)

[3.1.浅拷贝 37](#_Toc16989)

[3.2.深拷贝 37](#_Toc9855)

[4.原型模式代码实例 37](#_Toc2249)

[4.1.定义原型类和克隆方法 38](#_Toc2061)

[4.2.客户端使用代码示例 39](#_Toc24903)

[5.原型模式总结 41](#_Toc13370)

[第九章 “天上天下，唯我独尊”——单例模式 42](#_Toc13423)

[1.单例模式简介 42](#_Toc18899)

[2.单例模式结构 42](#_Toc1035)

[3.单例模式代码及效果 42](#_Toc28905)

[3.1.单例模式代码及验证 42](#_Toc30633)

[3.2.多线程环境测试单例模式 43](#_Toc23435)

[3.3.线程安全的单例模式代码实现 45](#_Toc26356)

[4.单例模式总结 46](#_Toc322)

[第十章 代码还可以这么复用——适配器模式 46](#_Toc6057)

[1.适配器模式简介 46](#_Toc5037)

[2.适配器模式结构 46](#_Toc21605)

[3.适配器模式代码实例 47](#_Toc23718)

[3.1.目标抽象类 47](#_Toc27287)

[3.2.适配者类 47](#_Toc31444)

[3.3.适配器类 48](#_Toc20834)

[3.4.客户端代码示例 48](#_Toc22428)

[3.5.效果 49](#_Toc2099)

[4.适配器模式总结 49](#_Toc18893)

[第十一章 号称最难理解的设计模式，只需看这篇解析——桥接模式 49](#_Toc29521)

[1.桥接模式简介 50](#_Toc12925)

[2.桥接模式结构 50](#_Toc28091)

[3.桥接模式代码实例 50](#_Toc3565)

[3.1.实现类 51](#_Toc17154)

[3.2.抽象类和扩充抽象类 51](#_Toc113)

[3.3.客户端代码示例 52](#_Toc6415)

[3.4.效果 53](#_Toc26536)

[4.桥接模式总结 53](#_Toc20718)

[第十二章 逆袭成为主管和高级经理！——组合模式 54](#_Toc31385)

[1.组合模式简介 54](#_Toc24339)

[2.组合模式结构 55](#_Toc5984)

[2.1.透明组合模式 55](#_Toc24252)

[2.2.安全组合模式 55](#_Toc5006)

[3.组合模式代码实例 55](#_Toc341)

[3.1.抽象构件 56](#_Toc19579)

[3.2.叶子构件 56](#_Toc16906)

[3.3.容器构件 57](#_Toc24331)

[3.4.客户端代码示例 58](#_Toc18160)

[3.5.效果 59](#_Toc27320)

[4.组合模式总结 59](#_Toc25705)

[第十三章人靠衣装，美靠靓装——装饰模式，教你成为最靓的仔！ 60](#_Toc8678)

[1.装饰模式简介 60](#_Toc7344)

[2.装饰模式结构 61](#_Toc14787)

[3.装饰模式代码实例 61](#_Toc1193)

[3.1.抽象构件类 62](#_Toc7749)

[3.2.具体构件类 62](#_Toc118)

[3.3.装饰类 62](#_Toc13063)

[3.4.客户端代码示例 64](#_Toc7512)

[3.5.效果 64](#_Toc28686)

[4.总结 65](#_Toc15326)

[第十四章细粒度对象的大面积复用！——实例分析享元模式 65](#_Toc25027)

[1.享元模式简介 65](#_Toc5253)

[2.享元模式结构 66](#_Toc3487)

[3.享元模式代码实例 66](#_Toc12832)

[3.1.抽象享元类 67](#_Toc2400)

[3.2.具体享元类 67](#_Toc6345)

[3.3.享元工厂类 67](#_Toc16576)

[3.4.客户端代码示例 68](#_Toc17716)

[3.5.效果 69](#_Toc1538)

[3.6.有外部状态的享元模式 70](#_Toc20361)

[4.总结 71](#_Toc11221)

[第十五章如何吃到回锅肉？找厨师啊——外观模式实例解析 71](#_Toc5690)

[1.外观模式简介 72](#_Toc7005)

[2.外观模式结构 72](#_Toc14453)

[3.外观模式代码实例 73](#_Toc12121)

[3.1.子系统类 73](#_Toc9496)

[3.2.外观类设计 74](#_Toc9674)

[3.3.客户端代码示例 74](#_Toc27220)

[3.4.效果 75](#_Toc17287)

[4.总结 75](#_Toc4965)

[第十六章 双十一天猫购物找代理？先学会代理模式 75](#_Toc24985)

[1.代理模式简介 75](#_Toc3332)

[2.代理模式结构 76](#_Toc6746)

[3.代理模式代码实例 76](#_Toc3718)

[3.1.抽象主题角色 77](#_Toc15529)

[3.2.真实主题角色 77](#_Toc13500)

[3.3.代理角色和Log类 77](#_Toc7608)

[3.4.客户端代码示例 78](#_Toc9494)

[3.5.效果  79](#_Toc26167)

[4.总结 79](#_Toc661)

[第十七章 “欲戴王冠，必承其重”——深度解析职责链模式 79](#_Toc7658)

[1.职责链模式简介 79](#_Toc2956)

[2.职责链模式结构 80](#_Toc4340)

[3.职责链模式代码实例 80](#_Toc11849)

[3.1.票据类 81](#_Toc17488)

[3.2.抽象处理者 81](#_Toc5143)

[3.3.具体处理者 82](#_Toc7002)

[3.5.客户端代码示例 84](#_Toc23866)

[3.6.效果 84](#_Toc7763)

[4.总结 85](#_Toc24013)

[第十八章命令模式 85](#_Toc4671)

[1.命令模式简介 85](#_Toc27318)

[2. 命令模式结构 85](#_Toc21658)

[3.命令模式代码实例 86](#_Toc32517)

[3.1.接收者类：电灯和风扇 86](#_Toc16811)

[3.2.抽象命令类 87](#_Toc1401)

[3.3.具体命令类  88](#_Toc25217)

[3.3.调用者：Button 88](#_Toc12033)

[3.4.客户端代码示例 89](#_Toc18407)

[3.5.效果 90](#_Toc18035)

[4.命令队列 90](#_Toc25031)

[5.命令模式其他应用 92](#_Toc13009)

[5.1.记录请求日志 92](#_Toc24278)

[5.2.宏命令 92](#_Toc9420)

[6.总结 92](#_Toc16006)

[第十九章 读懂老板的暗语，你需要知道解释器模式！ 92](#_Toc4467)

[1.解释器模式概述 92](#_Toc20360)

[2.解释器模式结构 93](#_Toc27430)

[3.解释器模式代码实例 93](#_Toc26473)

[表达式及输出结果部分实例表 94](#_Toc21311)

[3.1.抽象表达式 94](#_Toc17421)

[3.2.终结符表达式角色——值节点 95](#_Toc4642)

[3.3.终结符表达式角色——运算符节点 95](#_Toc19993)

[3.4.非终结符表达式角色——句子节点 96](#_Toc20611)

[3.5.上下文角色——处理者 96](#_Toc27527)

[3.6.客户端代码示例和结果  97](#_Toc9253)

[4.总结 99](#_Toc19448)

[第二十章 迭代器模式，给你更高大上的遍历体验！ 99](#_Toc21354)

[1.迭代器模式概述 99](#_Toc9347)

[2.迭代器模式结构 99](#_Toc3676)

[3.迭代器模式代码实例 100](#_Toc16146)

[3.1.抽象聚合类和具体聚合类 100](#_Toc4386)

[3.2.抽象迭代器 102](#_Toc8586)

[3.3.具体迭代器：RemoteControl 102](#_Toc765)

[3.4.客户端代码示例及结果  103](#_Toc15933)

[4.总结 104](#_Toc15346)

[第二十一章 中介者模式，说一说贝壳找房 105](#_Toc20688)

[1.中介者模式简介 105](#_Toc11248)

[2.中介者模式结构 105](#_Toc16984)

[3.中介者模式代码实例 106](#_Toc26874)

[3.0.公共头文件 106](#_Toc21487)

[3.1.中介者 107](#_Toc24881)

[3.2.同事类 108](#_Toc20219)

[3.3.客户端代码示例及效果 110](#_Toc1303)

[4.总结 111](#_Toc25546)

[第二十二章 我用备忘录模式设计了简易的版本控制系统 112](#_Toc1619)

[1.备忘录模式简介 112](#_Toc24001)

[2.备忘录模式结构 112](#_Toc27016)

[3.备忘录模式代码实例 114](#_Toc25197)

[3.1.备忘录Memento 115](#_Toc27582)

[3.2.原生器CodeVersion 116](#_Toc22969)

[3.3.管理者CodeManager  117](#_Toc48)

[3.4.客户端代码示例及效果 118](#_Toc20742)

[4.总结 119](#_Toc4482)

[第二十三章 “牵一发而动全身”——我用观察者模式简单模拟吃鸡 119](#_Toc30731)

[1.观察者模式简介 119](#_Toc18853)

[2.观察者模式结构 120](#_Toc3489)

[3.观察者模式代码实例 122](#_Toc32255)

[3.0.公共头文件 122](#_Toc991)

[3.1.观察者 122](#_Toc2437)

[3.2.目标类  124](#_Toc7735)

[3.3.客户端代码示例及效果 125](#_Toc27877)

[4.观察者模式的应用 126](#_Toc8570)

[5.总结 127](#_Toc3337)

[第二十四章 状态模式——从斗地主开始说起 127](#_Toc11495)

[1.状态模式简介 127](#_Toc27232)

[2.状态模式结构 127](#_Toc19626)

[3.状态模式代码实例 129](#_Toc16348)

[3.1.上下文类：游戏账户类 131](#_Toc6531)

[3.2.状态类 133](#_Toc11916)

[3.3.客户端代码示例及结果 137](#_Toc5733)

[4.总结 138](#_Toc813)

[第二十五章 如何管理和维护算法族？只需知道策略模式 138](#_Toc17486)

[1.策略模式简介 138](#_Toc19759)

[2.策略模式结构 138](#_Toc26029)

[3.策略模式代码实例 139](#_Toc31491)

[3.1.排序策略类 140](#_Toc30212)

[3.2.上下文类 142](#_Toc3495)

[3.3.客户端代码示例及结果 143](#_Toc16310)

[4.总结 143](#_Toc17872)

[第二十六 不知不觉就在使用的一种设计模式——模板方法模式 144](#_Toc2331)

[1.模板方法模式简介 144](#_Toc32545)

[2.模板方法结构 144](#_Toc23371)

[3.模板方法模式代码实例 146](#_Toc21066)

[3.1.基类 146](#_Toc14409)

[3.2.派生类 147](#_Toc7436)

[3.3.客户端代码实例及效果 148](#_Toc878)

[4.总结 149](#_Toc18294)

[第二十七章 访问者模式，从双十一购物开始说起 150](#_Toc13894)

[1.访问者模式简介 150](#_Toc6094)

[2.访问者模式结构 150](#_Toc3391)

[3.访问者模式代码实例 152](#_Toc13061)

[3.1.元素类 153](#_Toc29395)

[3.2.访问者 155](#_Toc5557)

[3.3.购物车ShoppingCart 156](#_Toc25887)

[3.4.客户端代码示例及结果 157](#_Toc29156)

[4.总结 158](#_Toc12539)

**第一章 设计模式——设计模式概述**

软件设计模式（Design pattern），又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。设计模式主要是为了解决某类重复出现的问题而出现的一套成功或有效的解决方案。设计模式提供一种讨论软件设计的公共语言，使得熟练设计者的设计经验可以被初学者和其他设计者掌握。设计模式还为[软件重构](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E9%87%8D%E6%9E%84)提供了目标。

**1.设计模式由来**

软件设计模式源自于建筑学。积淀了几千年的建筑学工程总结出一系列值得工程领域（包括软件工程领域）借鉴和学习的经验。哈佛大学建筑学博士、美国加州大学伯克利分校建筑学教授Christopher Alexander博士及其研究团队，经过对住宅和城市环境大量的调查研究，发现大众对住宅和城市环境存在部分共同的认同规律。Christopher Alexander在著作《A Pattern Language: Towns, Buildings, Construction》中把这些认同规律归纳为253个模式，对每一个模式(Pattern)都从Context（前提条件）、Theme或Problem（目标问题）、 Solution（解决方案）三个方面进行了描述，并给出了从用户需求分析到建筑环境结构设计直至经典实例的过程模型。在他的经典著作《建筑的永恒之道》中，他给出了关于模式的定义：

**每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心，通过这种方式，我们可以无数次地重用那些已有的成功的解决方案，无须再重复相同的工作。**

总结而言，即本文第一段的黑体字： 设计模式主要是为了解决某类重复出现的问题而出现的一套成功或有效的解决方案。

1991年，四人组GoF（Gang of Four，Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides）最早将模式应用于软件工程方法学，他们归纳发表了23种在软件开发中使用频率较高的设计模式，旨在用模式来统一沟通面向对象方法在分析、设计和实现间的鸿沟，由此，软件设计模式诞生了！

设计模式一般包含模式名称、问题、目的、解决方案、效果等要素：

（1）**模式名称(Pattern Name)**：通过一两个词来描述模式的问题、解决方案和效果，以便更好地理解模式并方便开发人员之间的交流，绝大多数模式都是根据其功能或模式结构来命名的；

（2）**问题(Problem)**：描述应该在何时使用模式，包含了设计中存在的问题以及问题存在的原因；

（3）**解决方案(Solution)**：描述一个设计模式的组成成分，以及这些组成成分之间的相互关系，各自的职责和协作方式，通常解决方案通过UML类图和核心代码来进行描述；

（4）**效果(Consequences)：**描述了模式的优缺点以及在使用模式时应权衡的问题。

**2.设计模式的优点**

设计模式是从众多优秀的软件系统中总结出的成功的、能够实现可维护性复用的设计方案，使用这些方案将避免一些重复性工作，高效设计出高质量的软件系统。总的来说，设计模式主要有以下优点：

（1）设计模式融合了众多专家的经验，以一种标准的形式供广大开发人员使用，通俗的设计词汇和通用的语言方便开发人员交流和学习；

（2）设计模式使人们可以更简单方便地复用成功的设计，使新开发者更容易理解设计思路；

（3）设计模式使设计方案更加灵活、易于修改；

（4）设计模式的使用将提高软件系统的开发效率和软件质量，节约开发成本；

（5）设计模式有助于初学者深入理解面向对象思想。

**第二章 设计模式（二）——UML类图介绍**

在正式介绍设计模式之前，我们有必要学习一下UML类图，因为我们需要一种直观的表示方法来描述设计模式，这种方法即是UML类图。在Jungle之前的一篇博客里，已经介绍到UML类图[UML类图关系与符号简介](https://blog.csdn.net/sinat_21107433/article/details/84580218)。这里，不妨我们再次学习一遍。

**1.UML是什么？**

UML-Unified Model Language [统一建模语言](https://baike.baidu.com/item/%E7%BB%9F%E4%B8%80%E5%BB%BA%E6%A8%A1%E8%AF%AD%E8%A8%80/3160571)，是在开发阶段，说明、可视化、构建和书写一个[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1/2262089)软件的开放方法。统一建模语言（UML）是一种模型化语言，通过一系列标准的图形符号来描述系统。一份典型的建模图表通常包含几个块或框，连接线和作为模型附加信息之用的文本。UML类图是用户将所希望描述的事物抽象为类，描述类的内部结构和类之间关系的设计视图。

**1.1.UML类图常用关系**

在UML类图中，常有以下几种关系：泛化(Generalization)、实现(Realization)、关联(Association)、聚合(Aggregation)、组合(Composition)和依赖(Dependency)。按照关系的强弱顺序：泛化≥实现>关联>聚合>组合>依赖。

**1.2.UML模型结构**

UML是一种主要由图形符号表达的建模语言，其结构主要包括：

（1）视图（View）：从不同角度来表示待建模系统，视图是由许多图形组成的一个抽象集合；一个系统模型可能有多个视图，每个视图从特定角度来描述系统的特征，比如用户视图、结构视图、行为视图、实现视图等。

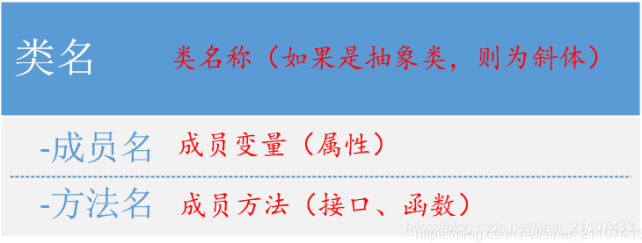
（2）图（Diagram）：UML图是UML视图内容的图形，比如用例图、类图、对象图、包图等。

（3）模型元素（Model Element）：指UML图中使用的一些概念，如类、对象、消息，以及概念之间的关系，如关联关系、以来关系等。

（4）通用机制（General Mechanism）：UML提供的通用机制为模型元素提供额外的注释、语义和扩展机制，允许用户对UML进行扩展。

**1.3.UML类的表示**

如图，类由一个矩形框表示，矩形框分为3层：



（1）第一层：类的名称；如果该类是抽象类，则使用斜体；

（2）第二层：类的属性，即成员变量，[可见性]名称：类型[=默认值]；

（3）第三层：类的方法，即成员函数，[可见性]名称（[参数列表]）[：返回类型]。

成员变量和成员方法前面有权限修饰符：

（1）“-”：private——私有

（2）“+”：public——公有

（3）“#”：protected——保护性



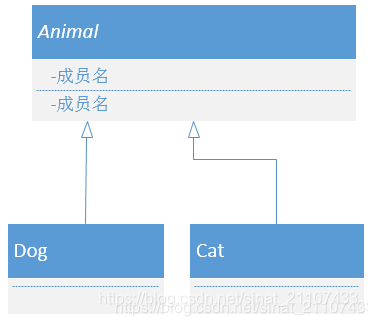
**2.类之间的关系**

**2.1.泛化(Generalization)**

（1）含义：类的继承关系。

（2）表示：用空心三角和实线，空心三角指向父类。

（3）举例：狗和猫都是一种动物。

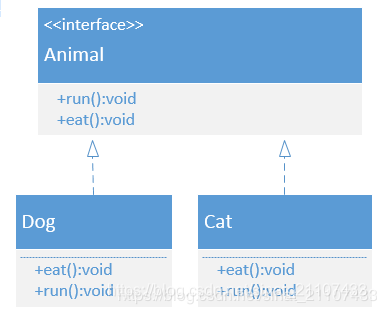


**2.2.实现(Realization)**

（1）含义：类与接口的关系，表示类是接口所有特征和行为的实现。

（2）表示：用空心三角和虚线，空心三角指向接口。

（3）举例：狗和猫实现了动物的“eat()”和"run()"的接口。



**2.3.关联(Association)**

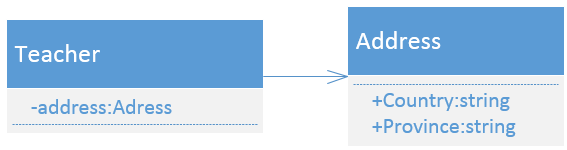
关联是一种拥有关系（has），一个类可以调用另一个类的公有的属性和方法。在类中以成员变量的方式表示。比如老师有自己的学生，知道学生的姓名学号成绩；学生有自己的老师，也知道老师的姓名和所教的科目。关联分为单向关联、双向关联和自关联。

**2.3.1.单向关联**

（1）含义：

（2）表示：带箭头的直线表示。

（3）举例：老师（Teacher类）有（has）自己的地址（Address类）



**2.3.2.双向关联**

（1）含义：指双方都知道对方的存在，都可以调用对方的公共属性和方法。

（2）表示：用一条直线连接两个类，也可以用双向箭头。

（3）举例：老师（Teacher类）有自己的学生（Student类），学生也有自己的老师。

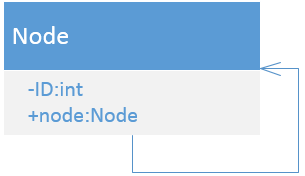


**2.3.3.自关联**

（1）含义：自己引用自己

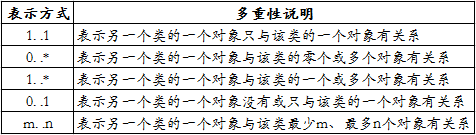
（2）表示：带箭头的直线，指向自己。

（3）举例：二叉树结构体（刷题时经常碰见吧）



**2.3.4.多重性关联**

（1）含义：对象之间存在多种多重性关联关系，如下表：



（2）表示：带箭头的直线，指向有关系的对象，可以在关联直线上用一个数字或数字范围表示。

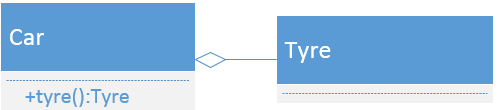
（3）举例：一个窗体界面对象有多个按钮对象

**2.4.聚合(Aggregation)**

（1）含义：整体与部分的关系，部分离开整体后可以单独存在。常用于类的成员变量。

（2）表示：带空心菱形和箭头的直线（或没有箭头），菱形挨着整体，箭头指向部分。

（3）举例：汽车和轮胎，轮胎是汽车的一部分，但轮胎也可以单独存在。

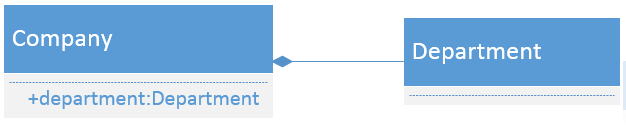


**2.5.组合(Composition)**

（1）含义：整体与部分的关系，部分离开整体后不可以单独存在，代表整体的对象负责代表部分的对象的生命周期。常用于类的成员变量。

（2）表示：带实心菱形和箭头的直线（或没有箭头），菱形挨着整体，箭头指向部分。

（3）举例：公司和部门，部门是公司的一部分，但部门不可以单独存在。



**2.6.依赖(Dependency)**

（1）含义：是一种使用的关系，即一个类的实现需要另一个类的协助，常用于类方法的局部变量、方法参数等。

（2）表示：带箭头的虚线，箭头指向协助的类（下面例子中箭头指向Food类）。

（3）举例：Animal的eat()方法的参数是Food

**2.7.注意事项**

注意：关联、聚合和组合、依赖的区别

（1）关联代表类与类的关系；

（2）聚合和组合代表整体与部分的关系；

（3）依赖是一种弱的关系，表示一个类的方法的实现需要用到另一个类，但二者之间并没有明显的关系。

**第三章 设计模式（三）——面向对象设计原则**

设计模式需要遵循基本的软件设计原则。可维护性（Maintainability）和可复用性（Reusability）是衡量软件质量的重要的两个属性：

（1）可维护性：软件能够被理解、改正、适应及扩展的难易程度

（2）可复用性：软件能够被复用的难易程度

**面向对象设计的原则是支持可维护性复用**，一方面需要实现设计方案或代码的复用，另一方面要保证系统易于扩展和修改，具有良好的可维护性。面向对象设计原则蕴含在各个设计模式中，是学习设计模式的基石，也是用于评价某个设计模式效果（Consequence）的重要指标。常见的面向对象设计原则包括：单一职责原则、开闭原则、里氏代换原则、依赖倒转原则、接口隔离原则、合成复用原则、迪米特法则。

**1.单一职责原则**

**单一职责原则：**

**定义1：一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。**

**定义2：就一个类而言，应该仅有一个引起它变化的原因。**

首先需要知道两个原则：高内聚和低耦合

**高内聚：内聚是对软件系统中元素职责相关性和集中度的度量。如果元素具有高度相关的职责，除了这些职责内的任务，没有其它过多的工作，那么该元素就具有高内聚性；反之则成为低内聚性。**

**低耦合：耦合是软件结构中各模块之间相互连接的一种度量，耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据**

单一职责原则用于控制类的力度大小。软件设计过程中，如果一个类承担的职责越多，那么它被复用的可能性越小。（为什么？想想，如果一个类有许多接口，另一个类想复用其中一两个接口，还不如重新实现！）。另一方面，如果一个类承担的职责越多，各个职责耦合在一起，修改其中一个职责可能“牵一发而动全身”。因此，应该将这些职责进行分离，不同的职责封装在不同的类中。

**2.开闭原则**

**开闭原则：软件实体应对扩展开放，对修改关闭。**

开闭原则指软件实体（一个软件模块、一个由不同类组成的局部结构或一个独立的类） 应该在不修改原有代码的基础上进行扩展。软件设计过程中，需求可能会随时变化，需要根据需求扩展已有的设计。如果原有的设计符合开闭原则，那么扩展起来就比较安全（不会影响原有功能，稳定）和方便（易于扩展）。开闭原则的关键在于抽象化。可以为系统定义一个相对较为稳定的抽象层，将不同的实现行为放到具体的实现层中完成。

举个例子：要设计一个计算器的类Computer，包含加、减两个功能，很自然的想法是在类Computer里声明并实现Add和Sub两个方法。那么如果要求再增加乘法功能，是不是要在Computer里增加Mul的方法呢？这就违背了开闭原则。

**3.里氏代换原则**

**历史替换原则：所有引用基类的地方必须能透明地使用其子类的对象。**

  在软件中，如果用子类对象来替换基类对象，程序将不会产生任何异常和问题，反过来不成立。为什么？很好理解，子类继承自基类，基类有的成员方法和成员属性，子类全都有；而子类可以增加新的方法和属性，所以反过来不成立。

**里氏代换原则的指导意义在于：尽可能地使用基类类型来对对象进行定义，而在运行时再确定子类类型，然后用子类对象替换父类对象。设计时应将父类设计为抽象类或者接口，子类继承父类并实现在父类中声明的方法；运行时子类实例（对象）替换父类实例（对象），可以很方便地扩展系统功能。**

**4.依赖倒转原则**

**依赖倒转原则：高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。**

  什么是高层，什么是低层呢？它们指的是继承（派生）中的基类子类关系，基类或者越抽象的类，层次越高。简单说，依赖倒转原则要求针对接口编程，不要针对实现编程。

依赖倒转原则要求再程序代码中传递参数时，或在关联关系中，尽量引用层次高的出现层类，即使用接口或抽象类来声明变量类型、参数类型声明、方法返回类型声明，以及数据类型转换等，而不要使用具体类来做这些事情。（其实这一点也符合里氏代换原则的指导意义，即对一个方法而言，返回基类的地方一定可以返回子类）。同样，依赖倒转原则设计的关键也在与抽象化设计。

**5.接口隔离原则**

**接口隔离原则：客户端不应该依赖那些它不需要的接口。**

当一个接口太大时，应该将它根据需要分割成多个更细小的接口，每个接口仅承担一个相对独立的角色或功能，使用该接口的客户端仅需知道与之相关的方法即可。 但是，接口不能过小，否则系统中接口太多，不利于维护。一般而言，在接口中仅包含为某一类用户定制的方法即可。

**6.合成复用原则**

**合成复用原则：优先使用对象组合，而不是通过继承来达到复用的目的。**

 根据UML类图关系，**合成复用原则指导在软件设计时，优先使用关联、聚合和组合关系，尽量少用泛化（继承）。对象组合可以使系统更加灵活（黑箱复用），降低类与类之间的耦合度，一个类的变化尽可能不影响其他类（父类和子类耦合度高不高？）**。如果要使用继承，则需考虑里氏代换原则和依赖倒转原则。继承关系会破坏系统的封装性，会将基类的实现细节暴露给子类（白箱复用），如果基类发生改变，那么子类的实现也不得不改变。

**7.迪米特法则**

**迪米特法则：每一个软件单位对其他单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位**

迪米特法则要求一个软件实体应当尽可能少地与其他实体发生相互作用。如果一个系统负荷迪米特法则，那么当修改其中某一个模块时就会尽量少地影响其他模块。应用迪米特法则可以降低系统的耦合度。**在类的设计上应该注意以下几点：在类的划分上应尽量创建松耦合的类，类之间的耦合度越低，越有利于复用；类的结构设计上，每一个类都应该降低其成员变量和成员函数的访问权限。**

**第四章 再不学简单工厂模式，就真的要去工厂搬砖啦！**

源码和资料地址：<https://github.com/FengJungle/DesignPattern>

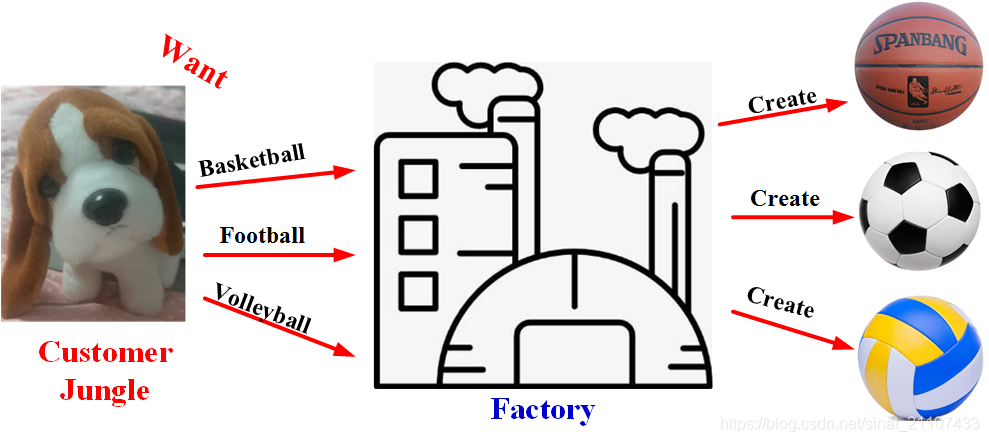
从本文开始，Jungle将开始一一整理介绍23种设计模式。

**1.简单工厂模式简介**

创建型模式关注对象的创建过程，在软件开发中应用非常广泛。创建型模式描述如何将对象的创建和使用分离，让用户在使用对象过程中无须关心对象的创建细节，从而降低系统耦合度，并且让系统易于修改和扩展。

简单工厂模式是最简单的设计模式之一，其实它并不属于Gof的23种设计模式，但应用也十分频繁，同时也是其余创建模式的基础，因此有必要先学习简单工厂模式。

**1.1.简单工厂应用举例**



什么是简单工厂模式呢？举个例子：如上图，一个体育用品生产厂（这即是一个工厂Factory），该工厂可以根据客户需求生产篮球、足球和排球。篮球、足球和排球被成为产品（Product），产品的名称可以被成为参数。客户Jungle需要时可以向工厂提供产品参数，工厂根据产品参数生产对应产品，客户Jungle并不需要关心产品的生产过程细节。

**1.2.简单工厂基本实现流程**

由上述例子，可以很容易总结出简单工厂的实现流程：

（1）设计一个抽象产品类，它包含一些公共方法的实现；

（2）从抽象产品类中派生出多个具体产品类，如篮球类、足球类、排球类，具体产品类中实现具体产品生产的相关代码；

（3）设计一个工厂类，工厂类中提供一个生产各种产品的工厂方法，该方法根据传入参数（产品名称）创建不同的具体产品类对象；

（4）客户只需调用工厂类的工厂方法，并传入具体产品参数，即可得到一个具体产品对象。

**1.3.简单工厂定义**

**简单工厂模式：**

**定义一个简单工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类**

 **2.简单工厂模式结构**

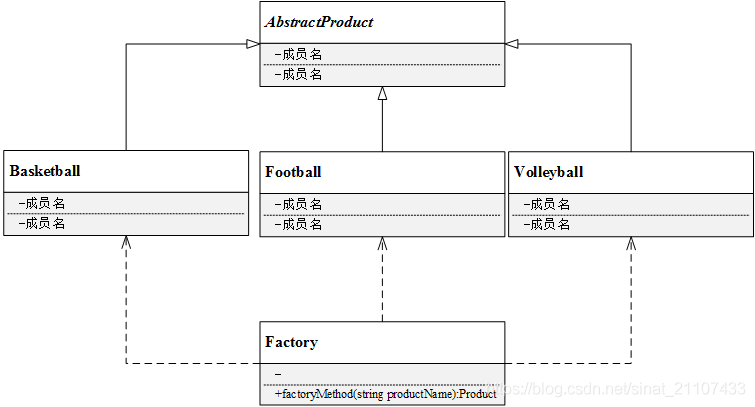
从简单工厂模式的定义和例子可以看出，在简单工厂模式中，大体上有3个角色：

**（1）工厂（Factory）**：根据客户提供的具体产品类的参数，创建具体产品实例；

**（2）抽象产品（AbstractProduct）**：具体产品类的基类，包含创建产品的公共方法；

**（3）具体产品（ConcreteProduct）**：抽象产品的派生类，包含具体产品特有的实现方法，是简单工厂模式的创建目标。

简单工厂模式UML类图如下：



代码结构如下：

//抽象产品类AbstractProduct

class AbstractProduct

{

public:

//抽象方法：

};

//具体产品类Basketball

class ConcreteProduct :public AbstractProduct

{

public:

//具体实现方法

};

class Factory

{

public:

AbstractProduct createProduct(string productName)

{

AbstractProduct pro = NULL;

if (productName == "ProductA"){

pro = new ProductA();

}

else if (productName == "ProductB"){

pro = new ProductB();

}

...

}

};

客户端在使用时，只需要创建一个工厂对象，调用工厂对象的createProduct方法，并传入所需要的产品参数，即可得到所需产品实例对象，而无需关心产品的创建细节。

**3.简单工厂模式代码实例**

考虑有以下一个场景：

**Jungle想要进行户外运动，它可以选择打篮球、踢足球或者玩排球。它需要凭票去体育保管室拿，票上写着一个具体球类运动的名字，比如“篮球”。体育保管室负责人根据票上的字提供相应的体育用品。然后Jungle就可以愉快地玩耍了。**

我们采用简单工厂模式来实现上述场景。首先，体育保管室是工厂，篮球、足球和排球是具体的产品，而抽象产品可以定义为“运动球类产品SportProduct”.Jungle作为客户只需要提供具体产品名字，工厂就可“生产”出对应产品。

**3.1.定义抽象产品类AbstractProduct，抽象方法不提供实现**

//抽象产品类AbstractProduct

class AbstractSportProduct

{

public:

AbstractSportProduct(){

}

//抽象方法：

void printName(){};

void play(){};

};

**3.2.定义三个具体产品类**

//具体产品类Basketball

class Basketball :public AbstractSportProduct

{

public:

Basketball(){

printName();

play();

}

//具体实现方法

void printName(){

printf("Jungle get Basketball\n");

}

void play(){

printf("Jungle play Basketball\n");

}

};

//具体产品类Football

class Football :public AbstractSportProduct

{

public:

Football(){

printName();

play();

}

//具体实现方法

void printName(){

printf("Jungle get Football\n");

}

void play(){

printf("Jungle play Football\n");

}

};

//具体产品类Volleyball

class Volleyball :public AbstractSportProduct

{

public:

Volleyball(){

printName();

play();

}

//具体实现方法

void printName(){

printf("Jungle get Volleyball\n");

}

void play(){

printf("Jungle play Volleyball\n");

}

};

**3.3.定义工厂类和工厂方法**

class Factory

{

public:

AbstractSportProduct \*getSportProduct(string productName)

{

AbstractSportProduct \*pro = NULL;

if (productName == "Basketball"){

pro = new Basketball();

}

else if (productName == "Football"){

pro = new Football();

}

else if (productName == "Volleyball"){

pro = new Volleyball();

}

return pro;

}

};

**3.4.客户端使用方法示例**

#include <iostream>

#include "SimpleFactory.h"

int main()

{

printf("简单工厂模式\n");

//定义工厂类对象

Factory \*fac = new Factory();

AbstractSportProduct \*product = NULL;

product = fac->getSportProduct("Basketball");

product = fac->getSportProduct("Football");

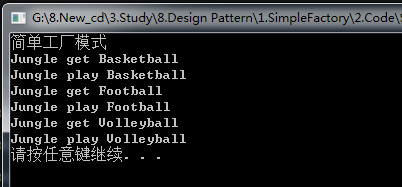
product = fac->getSportProduct("Volleyball");

system("pause");

return 0;

}

**3.5.效果**



可以看到，在客户端使用时，只需要提供产品名称作为参数，传入工厂的方法中，即可得到对应产品。抽象产品类中并没有提供公共方法的实现，而是在各个具体产品类中根据各自产品情况实现。

**4.简单工厂模式总结**

简单工厂模式的优点在于：

（1）工厂类提供创建具体产品的方法，并包含一定判断逻辑，客户不必参与产品的创建过程；

（2）客户只需要知道对应产品的参数即可，参数一般简单好记，如数字、字符或者字符串等。

当然，简单工厂模式存在明显的不足（想想我们之前介绍的面向对象设计原则？？？）。假设有一天Jungle想玩棒球了，该怎么办呢？你肯定会说，这还不容易吗？再从抽象产品类派生出一个Baseball类，并在工厂类的getSportProduct方法中增加“productName == "Baseball”的条件分支即可。的确如此，但是这明显违背了开闭原则（对扩展开放，对修改关闭），即在扩展功能时修改了既有的代码。另一方面，简单工厂模式所有的判断逻辑都在工厂类中实现，一旦工厂类设计故障，则整个系统都受之影响！

那该这么办呢？请看下集！

**第五章 不知道工厂方法模式？你可能真的应该去工厂！**

源码和资料地址：<https://github.com/FengJungle/DesignPattern>

在介绍简单工厂方法模式时Jungle总结出简单工厂模式存在最大的问题是违背了“开闭原则”，每当增加新的产品时，需要修改工厂类的逻辑。为了规避这种不足，同时很好的利用简单工厂模式的优点，本节Jungle将声情并茂地为您奉上工厂方法模式。

**1.工厂方法模式简介**

简单工厂模式中，每新增一个具体产品，就需要修改工厂类内部的判断逻辑。为了不修改工厂类，遵循开闭原则，工厂方法模式中不再使用工厂类统一创建所有的具体产品，而是针对不同的产品设计了不同的工厂，每一个工厂只生产特定的产品。

工厂方法模式定义：

**工厂方法模式：**

**定义一个用于创建对象的接口，但是让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。**

**2.工厂方法模式结构**

从工厂方法模式简介中，可以知道该模式有以下几种角色：

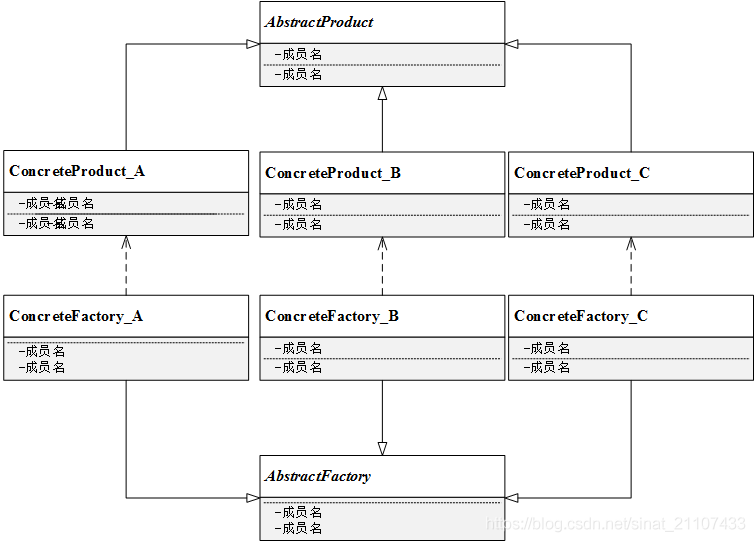
**（1）抽象工厂（AbstractFactory）：**所有生产具体产品的工厂类的基类，提供工厂类的公共方法；

**（2）具体工厂（ConcreteFactory）：**生产具体的产品

**（3）抽象产品（AbstractProduct）：**所有产品的基类，提供产品类的公共方法

**（4）具体产品（ConcreteProduct）：**具体的产品类

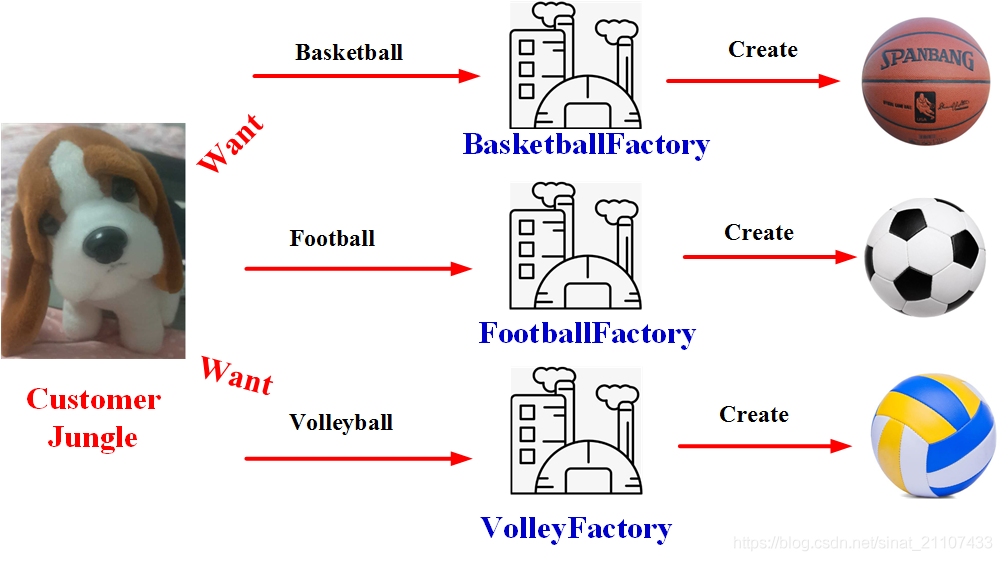
工厂方法模式UML类图如下：



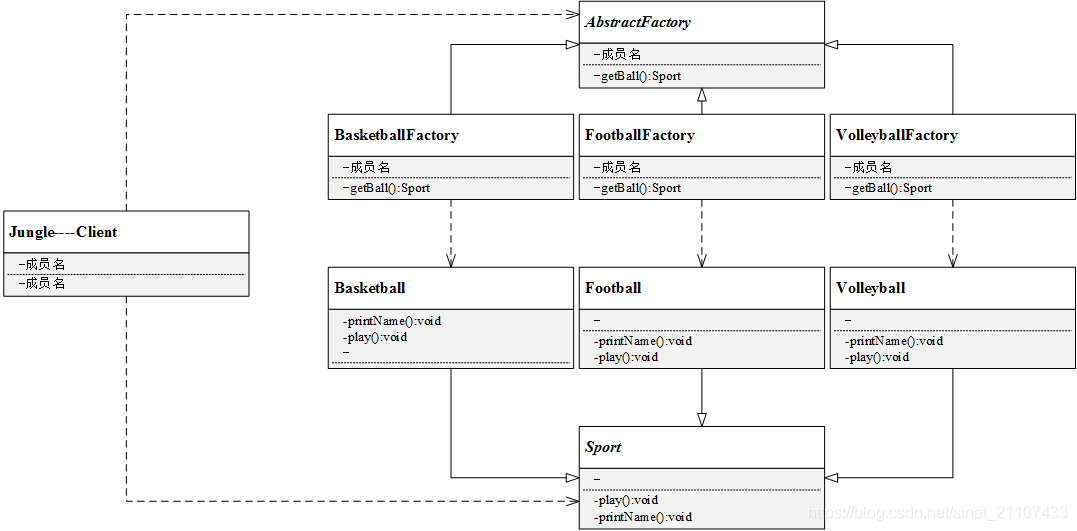
**3.工厂方法模式代码实例**

考虑这样一个场景，如下图：

**Jungle想要进行户外运动，它可以选择打篮球、踢足球或者玩排球。和上一次的体育保管室不同，这次分别由篮球保管室、足球保管室和排球保管室，Jungle只需直接去相应的保管室就可以拿到对应的球！然后Jungle就可以愉快地玩耍了。**



对应的UML实例图如下图：



**3.1.定义抽象产品类AbstractSportProduct，方法不提供实现**

//抽象产品类AbstractProduct

class AbstractSportProduct

{

public:

AbstractSportProduct(){

}

//抽象方法：

void printName(){};

void play(){};

};

**3.2.定义三个具体产品类**

//具体产品类Basketball

class Basketball :public AbstractSportProduct

{

public:

Basketball(){

printName();

play();

}

//具体实现方法

void printName(){

printf("Jungle get Basketball\n");

}

void play(){

printf("Jungle play Basketball\n\n");

}

};

//具体产品类Football

class Football :public AbstractSportProduct

{

public:

Football(){

printName();

play();

}

//具体实现方法

void printName(){

printf("Jungle get Football\n");

}

void play(){

printf("Jungle play Football\n\n");

}

};

//具体产品类Volleyball

class Volleyball :public AbstractSportProduct

{

public:

Volleyball(){

printName();

play();

}

//具体实现方法

void printName(){

printf("Jungle get Volleyball\n");

}

void play(){

printf("Jungle play Volleyball\n\n");

}

};

**3.3.定义抽象工厂类AbstractFactory，方法为纯虚方法**

//抽象工厂类

class AbstractFactory

{

public:

virtual AbstractSportProduct \*getSportProduct() = 0;

};

**3.4.定义三个具体工厂类**

/具体工厂类BasketballFactory

class BasketballFactory :public AbstractFactory

{

public:

BasketballFactory(){

printf("BasketballFactory\n");

}

AbstractSportProduct \*getSportProduct(){

printf("basketball");

return new Basketball();

}

};

//具体工厂类FootballFactory

class FootballFactory :public AbstractFactory

{

public:

FootballFactory(){

printf("FootballFactory\n");

}

AbstractSportProduct \*getSportProduct(){

return new Football();

}

};

//具体工厂类VolleyballFactory

class VolleyballFactory :public AbstractFactory

{

public:

VolleyballFactory(){

printf("VolleyballFactory\n");

}

AbstractSportProduct \*getSportProduct(){

return new Volleyball();

}

};

**3.5.客户端使用方法示例**

#include <iostream>

#include "FactoryMethod.h"

int main()

{

printf("工厂方法模式\n");

//定义工厂类对象和产品类对象

AbstractFactory \*fac = NULL;

AbstractSportProduct \*product = NULL;

fac = new BasketballFactory();

product = fac->getSportProduct();

fac = new FootballFactory();

product = fac->getSportProduct();

fac = new VolleyballFactory();

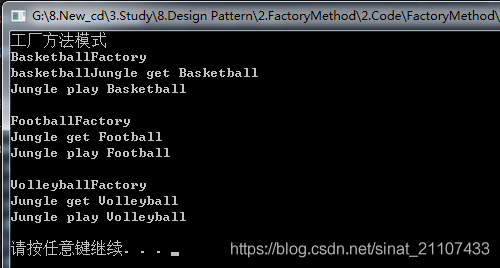
product = fac->getSportProduct();

system("pause");

return 0;

}

**3.6.效果**



**4.工厂方法模式总结**

如果Jungle想玩棒球（Baseball），只需要增加一个棒球工厂（BaseballFacory），然后在客户端代码中修改具体工厂类的类名，而原有的类的代码无需修改。由此可看到，相较简单工厂模式，工厂方法模式更加符合开闭原则。工厂方法是使用频率最高的设计模式之一，是很多开源框架和API类库的核心模式。

**优点：**

（1）工厂方法用于创建客户所需产品，同时向客户隐藏某个具体产品类将被实例化的细节，用户只需关心所需产品对应的工厂；

（2）工厂自主决定创建何种产品，并且创建过程封装在具体工厂对象内部，多态性设计是工厂方法模式的关键；

（3）新加入产品时，无需修改原有代码，增强了系统的可扩展性，符合开闭原则。

**缺点：**

（1）添加新产品时需要同时添加新的产品工厂，系统中类的数量成对增加，增加了系统的复杂度，更多的类需要编译和运行，增加了系统的额外开销；

（2）工厂和产品都引入了抽象层，客户端代码中均使用的抽象层（AbstractFactory和AbstractSportProduct ），增加了系统的抽象层次和理解难度。

**适用环境：**

客户端不需要知道它所需要创建的对象的类；

抽象工厂类通过其子类来指定创建哪个对象（运用多态性设计和里氏代换原则）

**第六章工厂搬砖也要带脑壳！Jungle学抽象工厂模式**

回顾之前的设计模式，简单工厂模式所有逻辑都封装在工厂类中，工厂根据客户提供的产品名字创建对应产品的对象实例；工厂方法模式将产品的创建过程放到了具体工厂类中，每一个工厂可以创建一个具体产品，由此可能会创建许多工厂类。**很多时候，一个工厂不只是生产一种产品，而是生产一类产品，**比如一个体育用品工厂，可以生产篮球、足球、排球等多种产品。此时我们可以把这些相关的产品归纳为一个“产品族”，由同一个工厂来生产，这即是Jungle今天要学习的抽象工厂模式。

**1.抽象工厂模式简介**

抽象工厂模式是常用的创建型设计模式之一。而今天Jungle要学习的抽象工厂模式，其抽象程度更高，每一个具体工厂可以生产一组相关的具体产品对象。

抽象工厂模式定义：

**抽象工厂模式：**

**提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。**

**简言之，一个工厂可以提供创建多种相关产品的接口，而无需像工厂方法一样，为每一个产品都提供一个具体工厂。**

**2.抽象工厂模式结构**

抽象工厂模式结构与工厂方法模式结构类似，不同之处在于，一个具体工厂可以生产多种同类相关的产品：

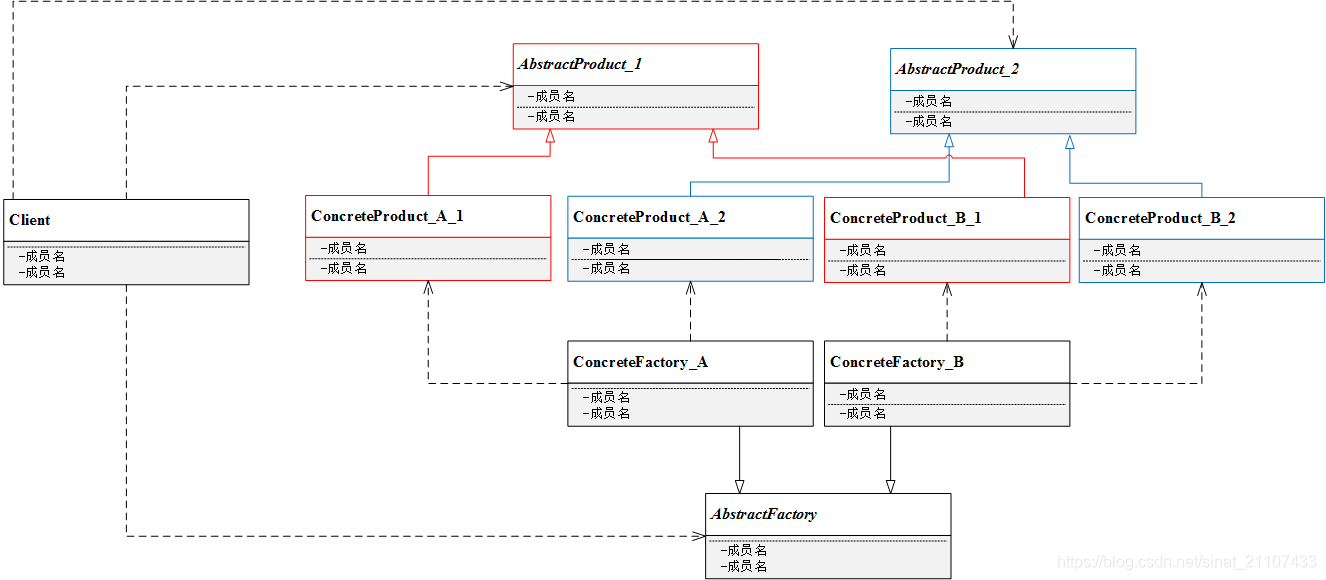
（1）抽象工厂（AbstractFactory）：所有生产具体产品的工厂类的基类，提供工厂类的公共方法；

（2）具体工厂（ConcreteFactory）：生产具体的产品

（3）抽象产品（AbstractProduct）：所有产品的基类，提供产品类的公共方法

（4）具体产品（ConcreteProduct）：具体的产品类

抽象工厂模式UML类图如下：

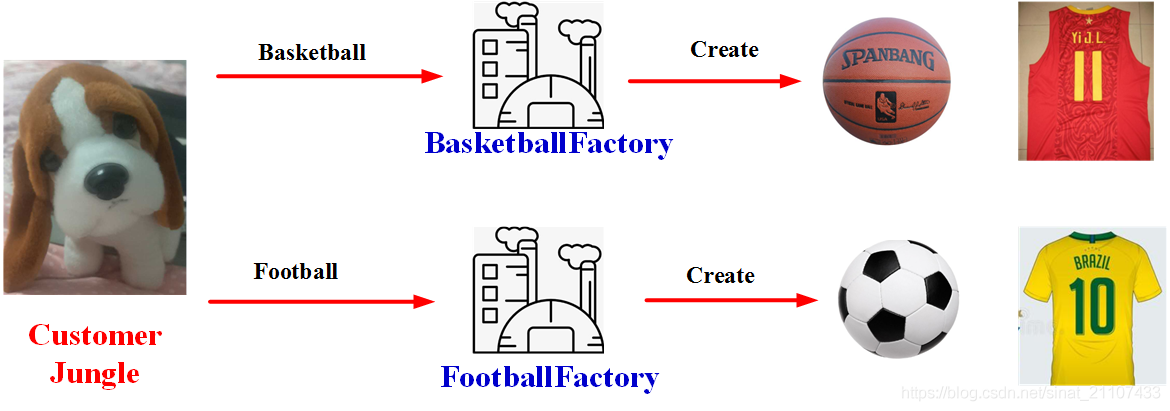


结合抽象工厂模式定义和UML，可以看到具体工厂ConcreteFactory\_A可以生产两种产品，分别是ConcreteProduct\_A\_1和ConcreteProduct\_A\_2，另一个具体工厂ConcreteFactory\_B同理。客户端使用时，**需要声明一个抽象工厂AbstractFactory和两个抽象产品AbstractProduct。**

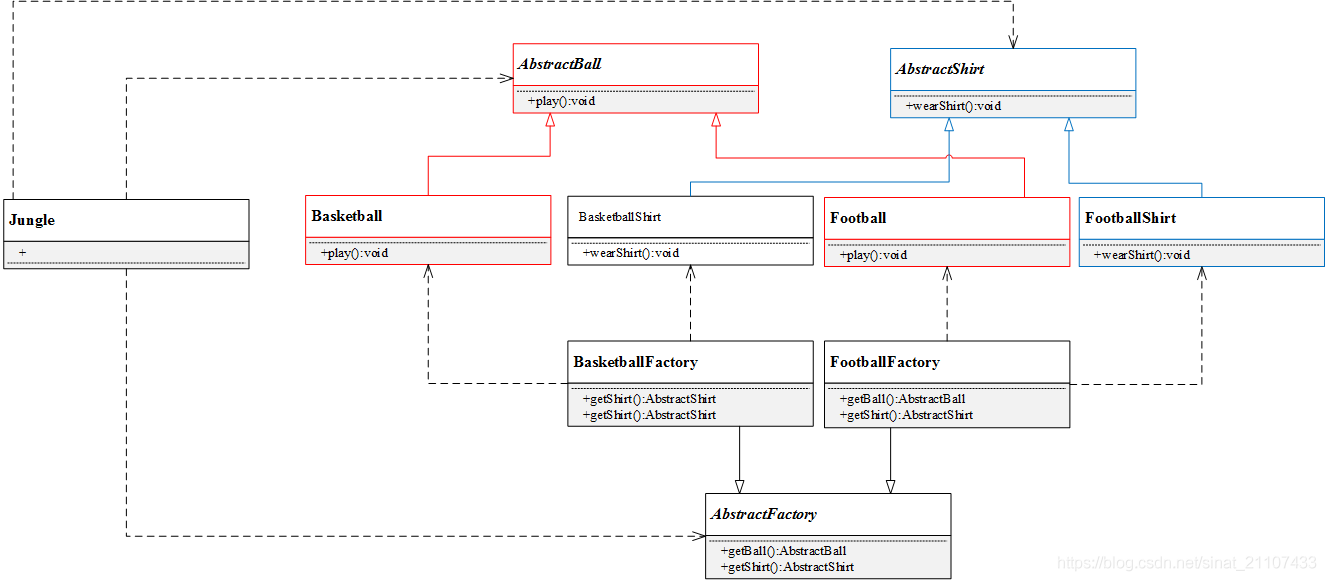
**3.抽象工厂模式代码实例**

考虑这样一个场景，如下图：

**Jungle想要进行户外运动，它可以选择打篮球和踢足球。但这次Jungle不想弄脏原本穿的T恤，所以Jungle还需要穿球衣，打篮球就穿篮球衣，踢足球就穿足球衣。篮球保管室可以提供篮球和篮球衣，足球保管室可以提供足球和足球衣。Jungle只要根据心情去某个保管室，就可以换上球衣、拿上球，然后就可以愉快地玩耍了。**



 对应的UML实例图如下图：



**3.1.定义产品类**

**3.1.1.产品类Ball**

（1）抽象产品类AbstractBall, 球类的基类，定义抽象方法play

//抽象产品类AbstractBall

class AbstractBall

{

public:

AbstractBall(){}

//抽象方法：

void play(){};

};

（2）具体产品类， 分别为Basketball和Football，具体实现方法play

//具体产品类Basketball

class Basketball :public AbstractBall

{

public:

Basketball(){

play();

}

//具体实现方法

void play(){

printf("Jungle play Basketball\n\n");

}

};

//具体产品类Football

class Football :public AbstractBall

{

public:

Football(){

play();

}

//具体实现方法

void play(){

printf("Jungle play Football\n\n");

}

};

**3.1.2.产品类Shirt**

（1）抽象产品类AbstractShirt：球衣类的基类，定义抽象方法wearShirt

//抽象产品类AbstractShirt

class AbstractShirt

{

public:

AbstractShirt(){}

//抽象方法：

void wearShirt(){};

};

（2）具体产品类BasketballShirt和FootballShirt，具体实现方法wearShirt

//具体产品类BasketballShirt

class BasketballShirt :public AbstractShirt

{

public:

BasketballShirt(){

wearShirt();

}

//具体实现方法

void wearShirt(){

printf("Jungle wear Basketball Shirt\n\n");

}

};

//具体产品类FootballShirt

class FootballShirt :public AbstractShirt

{

public:

FootballShirt(){

wearShirt();

}

//具体实现方法

void wearShirt(){

printf("Jungle wear Football Shirt\n\n");

}

};

**3.2.定义工厂类**

（1）定义抽象工厂AbstractFactory，声明两个方法getBall和getShirt

//抽象工厂类

class AbstractFactory

{

public:

virtual AbstractBall \*getBall() = 0;

virtual AbstractShirt \*getShirt() = 0;

};

（2）定义具体工厂BasketballFactory和FootballFactory，重新具体实现两个方法getBall和getShirt

//具体工厂类BasketballFactory

class BasketballFactory :public AbstractFactory

{

public:

BasketballFactory(){

printf("BasketballFactory\n");

}

AbstractBall \*getBall(){

printf("Jungle get basketball\n");

return new Basketball();

}

AbstractShirt \*getShirt(){

printf("Jungle get basketball shirt\n");

return new BasketballShirt();

}

};

//具体工厂类BasketballFactory

class FootballFactory :public AbstractFactory

{

public:

FootballFactory(){

printf("FootballFactory\n");

}

AbstractBall \*getBall(){

printf("Jungle get football\n");

return new Football();

}

AbstractShirt \*getShirt(){

printf("Jungle get football shirt\n");

return new FootballShirt();

}

};

**3.3.客户端使用方法示例**

#include <iostream>

#include "AbstractFactory.h"

int main()

{

printf("抽象工厂模式\n");

//定义工厂类对象和产品类对象

AbstractFactory \*fac = NULL;

AbstractBall \*ball = NULL;

AbstractShirt \*shirt = NULL;

fac = new BasketballFactory();

ball = fac->getBall();

shirt = fac->getShirt();

fac = new FootballFactory();

ball = fac->getBall();

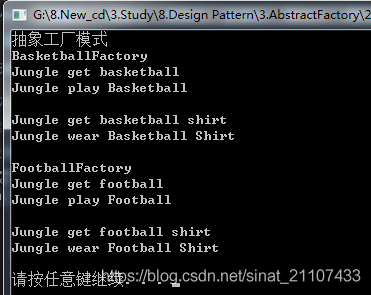
shirt = fac->getShirt();

system("pause");

return 0;

}

**3.4.效果**



**第七章 建一栋房子总共分几步？建造者模式告诉你答案！**

Jungle活了这20多年，全靠这个笑话活着！ 把大象装冰箱竟然只需要三步？那到底是怎么把大象装进冰箱呢？你问我，我问谁？再说，我也不关心这个呀！这……来点实际的吧，如果Jungle要建一栋房子，总共分几步？本文的建造者模式将声情并茂地向您娓娓道来……

**1.建造者模式简介**

建造者模式将客户端与包含多个部件的复杂对象的创建过程分离，客户端不必知道复杂对象的内部组成方式与装配方式（就好像Jungle不知道到底是如何把大象装进冰箱一样），只需知道所需建造者的类型即可。

**建造者模式定义：**

**建造者模式：**

**将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。**

**“同样的构建过程可以创建不同的表示”？？**这句话是什么意思呢？想象一下，建造一栋房子，建造过程无非都是打地基、筑墙、安装门窗等过程，但不同的客户可能希望不同的风格或者过程，最终建造出来的房子当然就呈现不同的风格啦！

**2.建造者模式结构**

建造者模式的结构包含以下几个角色：

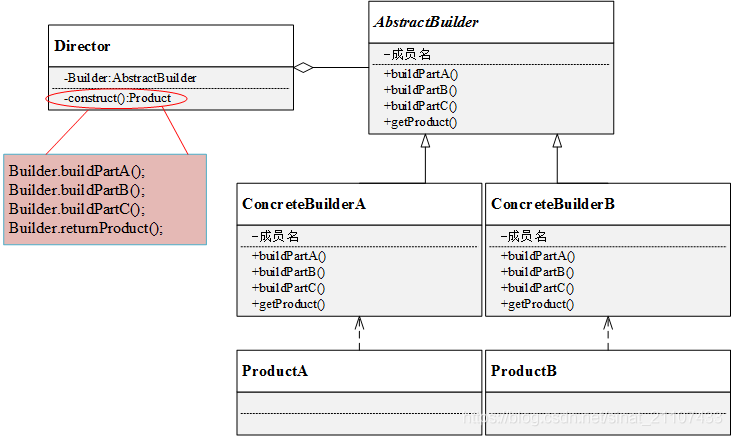
（1）抽象建造者（AbstractBuilder）：创建一个Product对象的各个部件指定的抽象接口；

（2）具体建造者（ConcreteBuilder）：实现AbstractBuilder的接口，实现各个部件的具体构造方法和装配方法，并返回创建结果。

（3）产品（Product）：具体的产品对象

（4）指挥者（Director）： 构建一个使用Builder接口的对象，安排复杂对象的构建过程，**客户端一般只需要与Director交互，指定建造者类型，然后通过构造函数或者setter方法将具体建造者对象传入Director。它主要作用是：隔离客户与对象的生产过程，并负责控制产品对象的生产过程。**

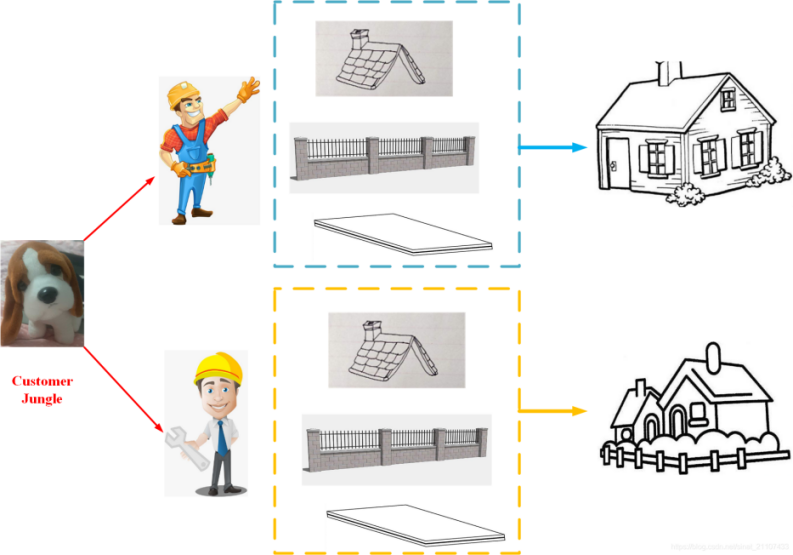
建造者模式UML类图如下：



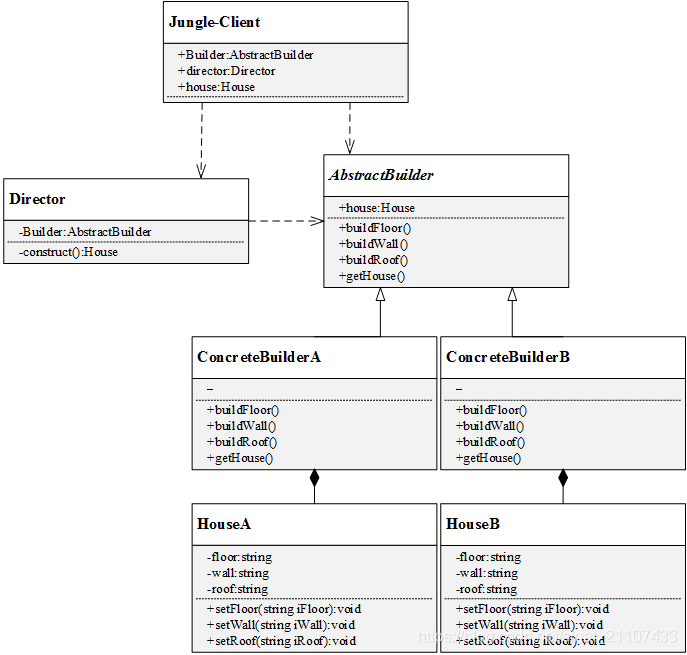
**3.建造者模式代码实例**

考虑这样一个场景，如下图：

Jungle想要建造一栋简易的房子（地板、墙和天花板），两个工程师带着各自的方案找上门来，直接给Jungle看方案和效果图。犹豫再三，Jungle最终选定了一位工程师……交房之日，Jungle满意的看着建好的房子，开始思考：这房子究竟是怎么建成的呢？这地板、墙和天花板是怎么建造的呢？工程师笑着说：“It's none of your business”



 UML图如下：



**3.1.定义产品类House**

//产品类House

class House

{

public:

House(){}

void setFloor(string iFloor){

this->floor = iFloor;

}

void setWall(string iWall){

this->wall = iWall;

}

void setRoof(string iRoof){

this->roof = iRoof;

}

//打印House信息

void printfHouseInfo(){

printf("Floor:%s\t\n", this->floor.c\_str());

printf("Wall:%s\t\n", this->wall.c\_str());

printf("Roof:%s\t\n", this->roof.c\_str());

}

private:

string floor;

string wall;

string roof;

};

 House是本实例中的产品，具有floor、wall和roof三个属性。

**3.2.定义建造者**

**3.2.1.定义抽象建造者AbstractBuilder**

//抽象建造者AbstractBall

class AbstractBuilder

{

public:

AbstractBuilder(){

house = new House();

}

//抽象方法：

virtual void buildFloor() = 0;

virtual void buildWall() = 0;

virtual void buildRoof() = 0;

virtual House \*getHouse() = 0;

House \*house;

};

**3.2.2.定义具体建造者**

//具体建造者ConcreteBuilderA

class ConcreteBuilderA :public AbstractBuilder

{

public:

ConcreteBuilderA(){

printf("ConcreteBuilderA\n");

}

//具体实现方法

void buildFloor(){

this->house->setFloor("Floor\_A");

}

void buildWall(){

this->house->setWall("Wall\_A");

}

void buildRoof(){

this->house->setRoof("Roof\_A");

}

House \*getHouse(){

return this->house;

}

};

//具体建造者ConcreteBuilderB

class ConcreteBuilderB :public AbstractBuilder

{

public:

ConcreteBuilderB(){

printf("ConcreteBuilderB\n");

}

//具体实现方法

void buildFloor(){

this->house->setFloor("Floor\_B");

}

void buildWall(){

this->house->setWall("Wall\_B");

}

void buildRoof(){

this->house->setRoof("Roof\_B");

}

House \*getHouse(){

return this->house;

}

};

**3.3.定义指挥者**

//指挥者Director

class Director

{

public:

Director(){}

//具体实现方法

void setBuilder(AbstractBuilder \*iBuilder){

this->builder = iBuilder;

}

//封装组装流程，返回建造结果

House \*construct(){

builder->buildFloor();

builder->buildWall();

builder->buildRoof();

return builder->getHouse();

}

private:

AbstractBuilder \*builder;

};

**3.4.客户端代码示例**

#include "BuilderPattern.h"

int main()

{

//抽象建造者

AbstractBuilder \*builder;

//指挥者

Director \*director = new Director();

//产品：House

House \*house;

//指定具体建造者A

builder = new ConcreteBuilderA();

director->setBuilder(builder);

house = director->construct();

house->printfHouseInfo();

//指定具体建造者B

builder = new ConcreteBuilderB();

director->setBuilder(builder);

house = director->construct();

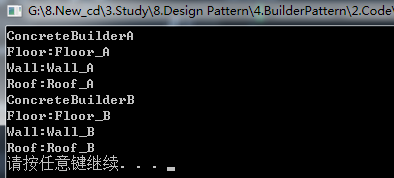
house->printfHouseInfo();

system("pause");

return 0;

}

**3.5.效果**



**4.建造者模式总结**

 从客户端代码可以看到，客户端只需指定具体建造者，并作为参数传递给指挥者，通过指挥者即可得到结果。客户端无需关心House的建造方法和具体流程。如果要更换建造风格，只需更换具体建造者即可，不同建造者之间并无任何关联，方便替换。从代码优化角度来看，其实可以不需要指挥者Director的角色，而直接把construct方法放入具体建造者当中。

**优点：**

（1）建造者模式中，客户端不需要知道产品内部组成细节，将产品本身和产品的创建过程分离，使同样的创建过程可以创建不同的产品对象；

（2）不同建造者相互独立，并无任何挂链，方便替换。

**缺点：**

（1）建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。

（2） 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大

**适用环境：**

（1）需要生成的产品对象有复杂的内部结构（通常包含多个成员变量）；

（2）产品对象内部属性有一定的生成顺序；

（3）同一个创建流程适用于多种不同的产品。

**第八章 克隆以后我就成了你——探究原型模式**

即便Jungle读书少，Jungle也清晰地记得中学生物课本上提到过的克隆羊“多利”。虽然多利寿命不长，但它的出现对“克隆（Clone）”技术意义重大。克隆，直观说就是从原有生物体上取体细胞，然后无性繁殖出有完全相同基因的个体或种群。这么说来中国的克隆技术其实是世界领先的，因为孙悟空拔一根毫毛变出许多一模一样的孙悟空的传说本质上就是克隆！而本文将要介绍的原型模式，将克隆技术应用到了软件设计层面。

**1.原型模式简介**

原型模式通过复制一个已有对象来获取更多相同或者相似的对象。原型模式定义如下：

**原型模式：**

**使用原型实例指定待创建对象的类型，并且通过复制这个原型来创建新的对象。**

原型模式的工作原理是将一个原型对象传给要发动创建的对象（即客户端对象），这个要发动创建的对象通过请求原型对象复制自己来实现创建过程。从工厂方法角度而言，创建新对象的工厂就是原型类自己。软件系统中有些对象的创建过程比较复杂，且有时需要频繁创建，原型模式通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象，这就是原型模式的意图所在。

**2.原型模式结构**

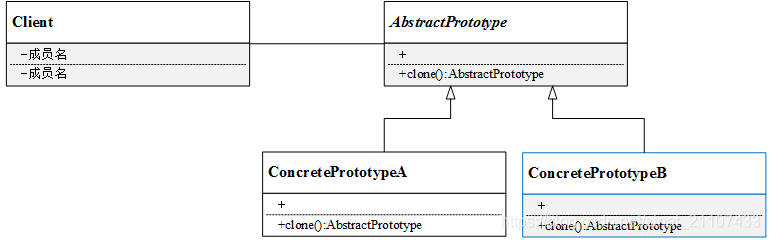
原型式的结构包含以下几个角色：

（1）抽象原型类（AbstractPrototype）：声明克隆clone自身的接口

（2）具体原型类（ConcretePrototype）：实现clone接口

（3）客户端（Client）：客户端中声明一个抽象原型类，根据客户需求clone具体原型类对象实例

原型模式的UML图如下：



**3.关于克隆方法：浅拷贝/深拷贝**

原型模式可以说是“复制”，即克隆，但这个复制不是代码的复制，而是将对象包含的所有属性都创建一份拷贝。但不同的复制操作，可能会产生两种不同的拷贝，即浅拷贝和深拷贝。

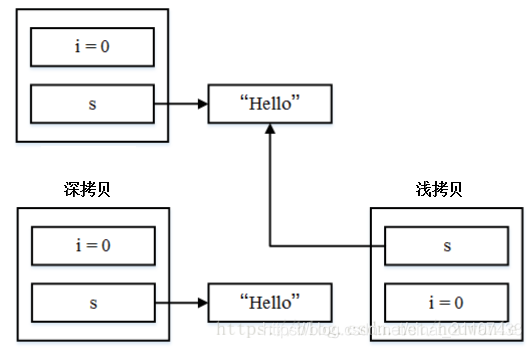
**3.1.浅拷贝**

在浅拷贝中，如果原型对象的成员变量是值类型（如int、double、char等基本数据类型），将复制一份给拷贝对象；如果原型对象的成员变量是引用/指针，则将引用/指针指向的地址拷贝一份给拷贝对象，即原型对象和拷贝对象中的成员变量指向同一个地址。

**3.2.深拷贝**

在深拷贝中，无论原型对象中的成员变量是值类型还是指针/引用类型，都将复制一份给拷贝对象。注意，深拷贝中，指针/引用对象也会被拷贝一份给拷贝对象。

下图举例说明了浅拷贝与深拷贝的区别：



**4.原型模式代码实例**

明天就是周一了，Jungle又陷入了苦恼中，因为作业还没完成。于是Jungle想拿着哥哥Single的作业来抄一份。虽然抄袭作业并不好，但是边抄边学借鉴一下也是可以的。于是乎，Jungle开始动起手来……

 作业包含几个部分：姓名（name）、学号（idNum）、模型(workModel)。首先定义一个workModel类：

//work model类

class WorkModel

{

public:

char \*modelName;

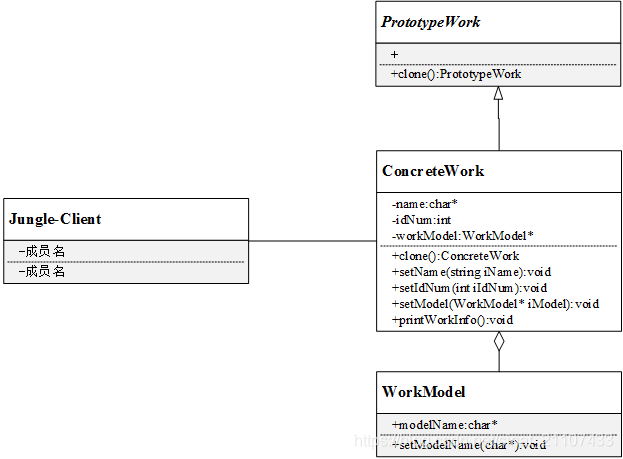
void setWorkModelName(char \*iName){

this->modelName = iName;

}

};

该实例UML图如下：



**4.1.定义原型类和克隆方法**

//抽象原型类PrototypeWork

class PrototypeWork

{

public:

PrototypeWork(){}

virtual PrototypeWork \*clone() = 0;

private:

};

//具体原型类ConcreteWork

class ConcreteWork :public PrototypeWork

{

public:

ConcreteWork(){}

ConcreteWork(char\* iName, int iIdNum, char\* modelName){

this->name = iName;

this->idNum = iIdNum;

this->workModel = new WorkModel();

this->workModel->setWorkModelName(modelName);

}

ConcreteWork \*clone(){

ConcreteWork \*work = new ConcreteWork();

work->setName(this->name);

work->setIdNum(this->idNum);

work->workModel = this->workModel;

return work;

}

void setName(char\* iName){

this->name = iName;

}

void setIdNum(int iIdNum){

this->idNum = iIdNum;

}

void setModel(WorkModel \*iWorkModel){

this->workModel = iWorkModel;

}

//打印work信息

void printWorkInfo(){

printf("name:%s\t\n", this->name);

printf("idNum:%d\t\n", this->idNum);

printf("modelName:%s\t\n", this->workModel->modelName);

}

private:

char\* name;

int idNum;

WorkModel \*workModel;

};

**4.2.客户端使用代码示例**

**4.2.1.示例一：浅拷贝**

#include "PrototypePattern.h"

int main()

{

ConcreteWork \*singleWork = new ConcreteWork("Single",1001,"Single\_Model");

printf("\nSingle的作业：\n");

singleWork->printWorkInfo();

printf("\njungle直接抄作业……\n");

ConcreteWork \*jungleWork = singleWork;

printf("\nJungle的作业：\n");

jungleWork->printWorkInfo();

//抄完改名字和学号，否则会被老师查出来

printf("\njungle抄完改名字和学号，否则会被老师查出来……\n");

jungleWork->setName("jungle");

jungleWork->setIdNum(1002);

WorkModel \*jungleModel = new WorkModel();

jungleModel->setWorkModelName("Jungle\_Model");

jungleWork->setModel(jungleModel);

//检查下是否改对了

printf("\nSingle的作业：\n");

singleWork->printWorkInfo();

printf("\nJungle的作业：\n");

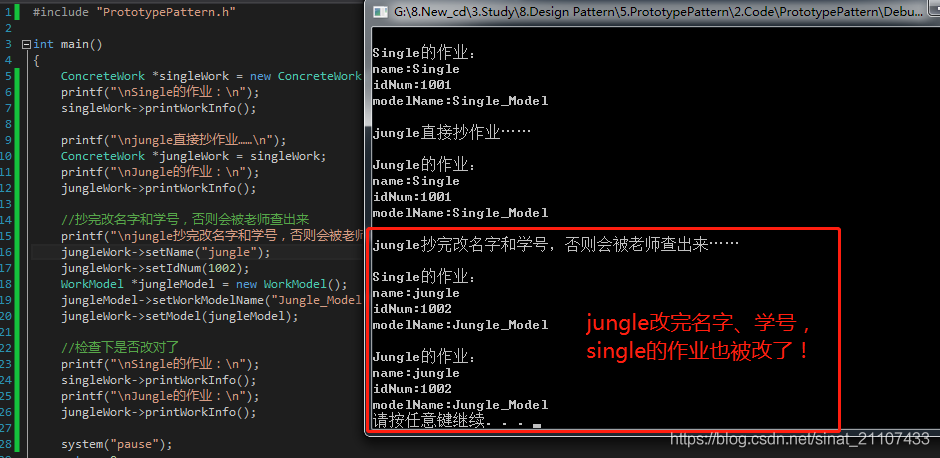
jungleWork->printWorkInfo();

system("pause");

return 0;

}

效果如下图：



显然，这不是我们想要的结果。接下来我们使用clone方法。

**4.2.2.示例二：深拷贝**

#include "PrototypePattern.h"

int main()

{

ConcreteWork \*singleWork = new ConcreteWork("Single", 1001, "Single\_Model");

printf("\nSingle的作业：\n");

ConcreteWork \*jungleWork = singleWork->clone();

printf("\nJungle的作业：\n");

//抄完改名字和学号，否则会被老师查出来

printf("\njungle抄完改名字和学号，否则会被老师查出来……\n");

jungleWork->setName("jungle");

jungleWork->setIdNum(1002);

WorkModel \*jungleModel = new WorkModel();

jungleModel->setWorkModelName("Jungle\_Model");

jungleWork->setModel(jungleModel);

//检查下是否改对了

printf("\nSingle的作业：\n");

singleWork->printWorkInfo();

printf("\nJungle的作业：\n");

jungleWork->printWorkInfo();

system("pause");

return 0;

}

效果如下图：



**5.原型模式总结**

**优点：**

（1）当创建新的对象实例较为复杂时，原型模式可以简化创建过程，提高创建对象的效率；

（2）可扩展：模式中提供了抽象原型类，具体原型类可适当扩展；

（3）创建结构简单：创建工厂即为原型对象本身

**缺点：**

（1）深克隆代码较为复杂；

（2）每一个类都得配备一个clone方法，且该方法位于类的内部，修改时违背开闭原则；

**适用环境：**

（1）当创建新的对象实例较为复杂时，原型模式可以简化创建过程；

（2）结合优点第3条，需要避免使用分层次的工厂类来创建分层次的对象，并且类的实例对象只有一个或很少几个的组合状态，通过复制原型对象得到新实例，比通过使用构造函数创建一个新实例会更加方便。

**第九章 “天上天下，唯我独尊”——单例模式**

**你能在电脑上调出两个Windows任务管理器吗？**

**假设能，如果两个管理器显示的数据相同，那何必要存在两个呢？**

**如果两个管理器显示的数据不同，那我该相信哪一个呢？**

试试看，应该有且仅有一个吧？一个系统里有且仅有一个Windows任务管理器实例供外界访问 。如何保证系统里有且仅有一个实例对象呢？并且能够供外界访问？你可以在系统里定义一个统一的全局变量，但这并不能防止创建多个对象（想一想，为什么？）这就是单例模式的典型应用。

对于一个软件系统中的某些类来说，只有一个实例很重要。假设Windows系统上可以同时调出两个Windows任务管理器，这两个任务管理器显示的都是同样的信息，那势必造成内存资源的浪费；如果这两个任务管理器显示的是不同的信息，这也给用户带来了困惑，到底哪一个才是真实的状态？

**1.单例模式简介**

**单例模式定义：**

**单例模式：**

**确保一个类只有一个实例，并提供一个全局访问点来访问这个唯一实例。**

单例模式有3个要点：

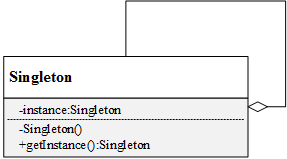
（1）这个类只能有一个实例；

（2）它必须自己创建这个实例；

（3）它必须自己向整个系统提供这个实例。

**2.单例模式结构**

单例模式结构非常简单，其UML图如下所示，只包含一个类，即单例类。为防止创建多个对象，其构造函数必须是私有的（外界不能访问）。另一方面，为了提供一个全局访问点来访问该唯一实例，单例类提供了一个公有方法getInstance来返回该实例。



**3.单例模式代码及效果**

**3.1.单例模式代码及验证**

单例模式代码：

#ifndef \_\_SINGLETON\_H\_\_

#define \_\_SINGLETON\_H\_\_

#include <iostream>

#include <string.h>

using namespace std;

//抽象产品类AbstractBall

class Singleton

{

public:

static Singleton\* getInstance(){

if (instance == NULL){

printf("创建新的实例\n");

instance = new Singleton();

}

return instance;

}

private:

Singleton(){}

static Singleton\* instance;

};

Singleton\* Singleton::instance = NULL;

#endif //\_\_SINGLETON\_H\_\_

可以看到，构造函数是私有的（private），即单例模式对象只能在类内部实例化（这就满足了单例模式的第二个要点）​。同时，实例对象instance是静态static的，也就是全局的，假设客户端实例化了两个Singleton，但instance只有一个（这就满足了第一个要点）​。那第三个要点如何满足呢？即外界如何获取​单例对象呢？上述代码中定义了一个方法（同样也是static的）getInstance(). 接下来看看客户端如何使用​。

客户端代码验证：

int main()

{

Singleton \*s1 = Singleton::getInstance();

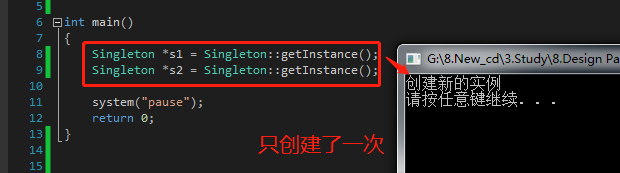
Singleton \*s2 = Singleton::getInstance();

system("pause");

return 0;

}

 效果如下图：



**3.2.多线程环境测试单例模式**

上述的客户端验证似乎说明上述代码实现了单例模式。的确是实现了，但是这样真的安全吗？试想在多线程环境里，当两个线程(甚至更多线程）同时使用，同样存在创建了多个实例的隐患。下面的代码是在多线程环境下进行的测试：

/\*非线程安全 单例模式\*/

#include <process.h>

#include <Windows.h>

#define THREAD\_NUM 5

unsigned int \_\_stdcall CallSingleton(void \*pPM)

{

Singleton \*s = Singleton::getInstance();

int nThreadNum = \*(int \*)pPM;

Sleep(50);

printf("线程编号为%d\n", nThreadNum);

return 0;

}

int main()

{

HANDLE handle[THREAD\_NUM];

//线程编号

int threadNum = 0;

while (threadNum < THREAD\_NUM)

{

handle[threadNum] = (HANDLE)\_beginthreadex(NULL, 0, CallSingleton, &threadNum, 0, NULL);

//等子线程接收到参数时主线程可能改变了这个i的值

threadNum++;

}

//保证子线程已全部运行结束

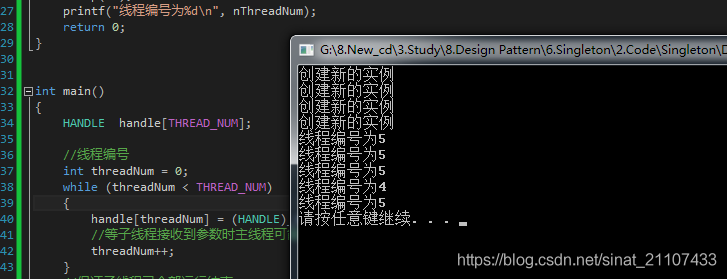
WaitForMultipleObjects(THREAD\_NUM, handle, TRUE, INFINITE);

system("pause");

return 0;

}

一共创建了5个线程，每个线程里面都试图创建一个单例对象。理论上，最终只有第一个线程（第一个被系统调度的线程）才能打印出“创建新的实例”，然而：



 上述结果说明，3.1的单例模式的代码并不是线程安全的。

**3.3.线程安全的单例模式代码实现**

如何做到线程安全呢？多线程同步与互斥有多重方法，这里Jungle介绍互斥锁这种用法。代码如下：

#ifndef \_\_SINGLETON\_H\_\_

#define \_\_SINGLETON\_H\_\_

#include <iostream>

#include <string.h>

#include <mutex>

using namespace std;

class Singleton

{

public:

static Singleton\* getInstance(){

if (instance == NULL){

m\_mutex.lock();

if (instance == NULL){

printf("创建新的实例\n");

instance = new Singleton();

}

m\_mutex.unlock();

}

return instance;

}

private:

Singleton(){}

static Singleton\* instance;

static std::mutex m\_mutex;

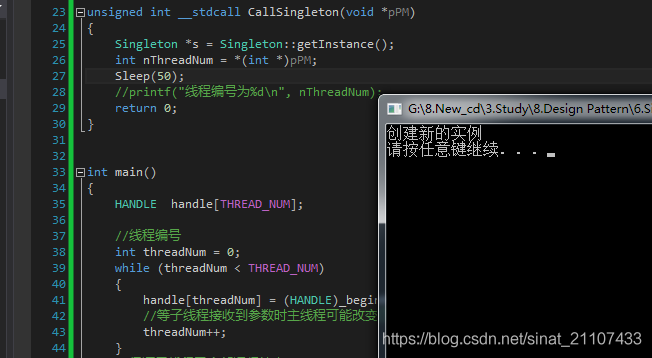
};

Singleton\* Singleton::instance = NULL;

std::mutex Singleton::m\_mutex;

#endif //\_\_SINGLETON\_H\_\_

接下来再在多线程环境下测试，结果显示功能正常！



**4.单例模式总结**

**优点：**

（1）单例模式提供了严格的对唯一实例的创建和访问

（2）单例模式的实现可以节省系统资源

**缺点：**

（1）如果某个实例负责多重职责但又必须实例唯一，那单例类的职责过多，这违背了单一职责原则

（2）多线程下需要考虑线程安全机制

（3）单例模式没有抽象层，不方便扩展

**适用环境：**

（1）系统只需要一个实例对象

（2）某个实例只允许有一个访问接口

**第十章 代码还可以这么复用——适配器模式**

**你一定听过“网络适配器”吧？又叫网卡。它的作用是什么呢？——上网！**

**这样的回答显然不够专业，正确的答案是“网卡的一个重要功能就是要进行串行/并行转换。因为网卡和局域网之间的通信是通过电缆或双绞线以串行传输方式进行，而网卡和计算机之间通信则是通过计算机主板上的[I](https://baike.baidu.com/item/I/O%E6%80%BB%E7%BA%BF/4227070)/O总线以并行传输方式进行。”**

**你肯定要问：“这和我有什么关系？”**

**当然有了，因为你正在学习设计模式，而这就跟本文即将要介绍的适配器模式有关啦！**

**1.适配器模式简介**

除了网卡适配器，你一定还听说过电源适配器吧？我国生活用电电压是220V，但我们的电脑、手机、平板、剃须刀（充电式）不会使用这么高的电压。这就需要电源适配器（充电器、变压器），使各个电子设备和220的供电电压兼容。电源适配器就充当了一个适配器的角色。

在软件系统设计中，当需要组合使用的类不兼容时，也需要类似于变压器一样的适配器来协调这些不兼容者，这就是适配器模式！

那么什么是适配器模式呢？

**适配器模式：**

**将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。**

与电源适配器类似，适配器模式中会设计一个叫做“适配器”的包装类，适配器包装的对象叫做适配者。根据定义，适配器是根据客户的需求，将适配者已有的接口转换成另一个接口，得以使不兼容的类可以协同工作。

**2.适配器模式结构**

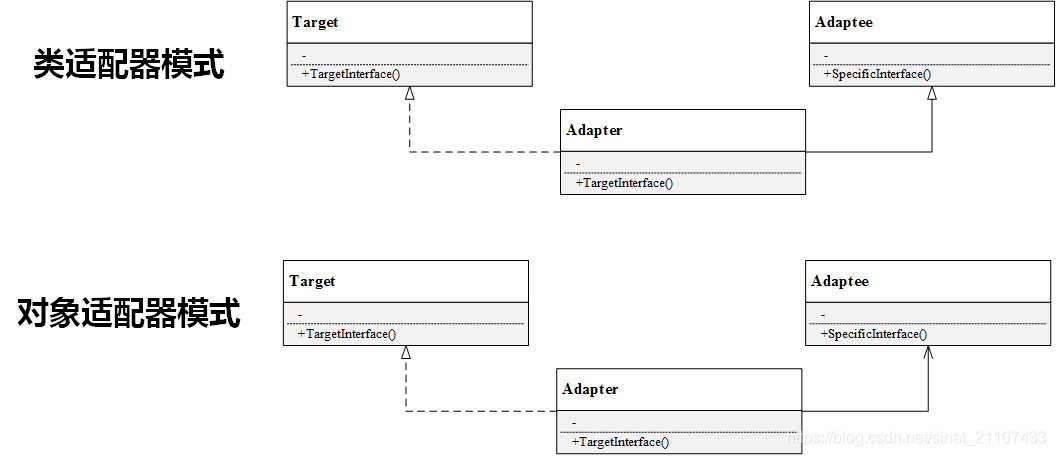
适配器模式分为类适配器和对象适配器。

（1）**适配器类（Adapter）：**适配器与适配者之间是继承或实现关系；

（2）**适配者类（Adaptee）：**适配器与适配者之间是关联关系。

（3）**目标抽象类（Target）：**定义客户所需要的接口。

类适配器和对象适配器的UML图如下。类适配器中，适配器类通过继承适配者类，并重新实现适配者的具体接口来达到适配客户所需要的接口的目的。对象适配器中，适配器类通过在类中实例化一个适配者类的对象，并将其封装在客户所需功能的接口里，达到最终的适配目的。



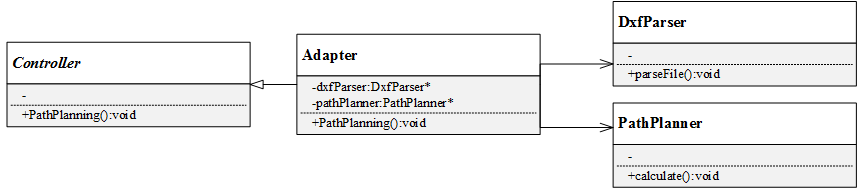
**3.适配器模式代码实例**

Jungle曾经在一个项目里多次使用了适配器模式。这里举个使用对象适配器模式的例子。

**路径规划包括两个阶段：首先读取并解析工程图文件，得到其中的点、直线坐标；其次根据需求计算加工路径。软件控制器（Controller）上，系统点击“路径规划”按钮就自动完成上述过程。**

**Jungle已经封装好一个类DxfParser，该类可以读取后缀名为dxf的工程图文件，并解析其中的点、线，保存到路径列表里。另一个类PathPlanner用于计算加工路径。**

这个例子中，Controller就是目标抽象类，DxfParser和PathPlanner是适配者类，这两个类提供的方法可以用于实现路径规划的需求。我们只需再定义一个适配器类Adapter即可。



**3.1.目标抽象类**

//目标抽象类

class Controller

{

public:

Controller(){}

virtual void pathPlanning() = 0;

private:

};

**3.2.适配者类**

//适配者类DxfParser

class DxfParser

{

public:

DxfParser(){}

void parseFile(){

printf("解析文件提取数据\n");

}

};

//适配者类PathPlanner

class PathPlanner

{

public:

PathPlanner(){}

void calculate(){

printf("计算加工路径\n");

}

};

**3.3.适配器类**

//适配器类Adapter

class Adapter:public Controller

{

public:

Adapter(){

dxfParser = new DxfParser();

pathPlanner = new PathPlanner();

}

void pathPlanning(){

printf("路径规划：\n");

dxfParser->parseFile();

pathPlanner->calculate();

}

private:

DxfParser \*dxfParser;

PathPlanner \*pathPlanner;

};

**3.4.客户端代码示例**

#include <iostream>

#include "AdapterPattern.h"

int main()

{

Controller \*controller = new Adapter();

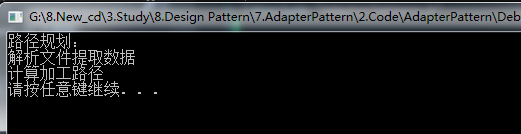
controller->pathPlanning();

system("pause");

return 0;

}

**3.5.效果**



**4.适配器模式总结**

**优点：**

（1）将目标类和适配者类解耦，引入一个适配器类实现代码重用，无需修改原有结构；

（2）增加类的透明和复用，对于客户端而言，适配者类是透明的；

（3）对象适配器可以把不同适配者适配到同一个目标（对象适配器）；

**缺点：**

对编程语言的限制：Java不支持多重继承，一次最多只能适配一个适配者类，不能同时适配多个适配者类；

**适用环境：**

（1）系统需要使用一些现有的类，但这些类的接口不符合系统需要，或者没有这些类的源代码；

（2）想创建一个重复使用的类，用于和一些彼此没有太大关联的类一起工作。

**第十一章 号称最难理解的设计模式，只需看这篇解析——桥接模式**

**Jungle有两个手机，分别是M手机和N手机，M手机上有游戏Game1，N手机上有Game2。每次Jungle想玩Game1时，就使用M手机，想玩Game2时，就玩N手机。要是某天Jungle外出，心情大好，两个游戏都想玩，那Jungle还得带上两个手机？？？麻不麻烦？**

**如果新出一个游戏Game3，那Jungle是不是要再买一个手机呢？**

**同样都是游戏软件，为什么不把所有游戏都装到一个手机上呢？**



**1.桥接模式简介**

如果系统中的某个类存在两个独立变化的维度，通过桥接模式可以将这两个维度分离开来，使两者独立扩展。如同上述实例，Jungle想用手机玩游戏，手机和游戏是两个独立变化的维度，增加一个游戏对手机没有影响，增加一个手机对游戏也没有影响。手机上可以安装游戏，而游戏必须在手机上玩，从这个角度而言，手机和游戏之间存在较强的耦合。

但两者可以很好的解耦，且解耦后扩展灵活：所有游戏安装在一个手机上，新出一个游戏，新安装就ok！买了新手机，同样可以装上所有游戏。这就是桥接模式：

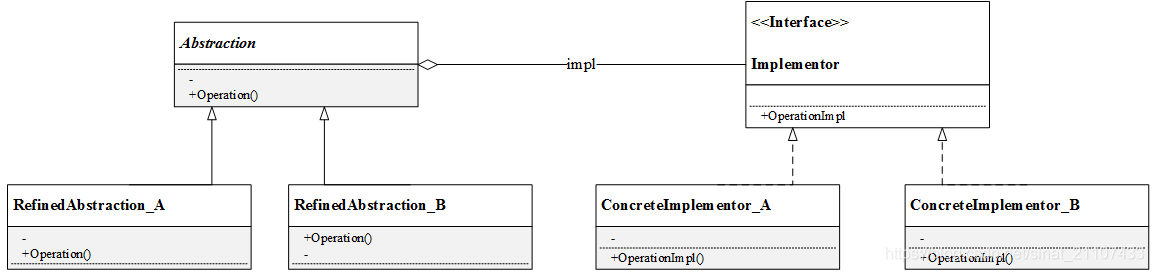
**桥接模式：**

**将抽象部分与它的实现部分解耦，使得两者都能够独立变化。**

桥接模式将两个独立变化的维度设计成两个独立的继承等级结构（而不会将两者耦合在一起形成多层继承结构），在抽象层将二者建立起一个抽象关联，该关联关系类似一座桥，将两个独立的等级结构连接起来，故曰“桥接模式”。

**2.桥接模式结构**

桥接模式UML图如下图。由图可知，桥接模式包含以下角色：



**（1）Abstraction（抽象类）：**定义抽象类的接口（抽象接口），由聚合关系可知，抽象类中包含一个Implementor类型的对象，它与Implementor之间有关联关系，既可以包含抽象业务方法，也可以包含具体业务方法；

**（2）Implementor（实现类接口）：**定义实现类的接口，这个接口可以与Abstraction类的接口不同。一般而言，实现类接口只定义基本操作，而抽象类的接口还可能会做更多复杂的操作。

**（3）RefinedAbstraction（扩充抽象类）：**具体类，实现在抽象类中定义的接口，可以调用在Implementor中定义的方法；

**（4）ConcreteImplementor（具体实现类）：**具体实现了Implementor接口，在不同的具体实现类中实现不同的具体操作。运行时ConcreteImplementor将替换父类。

简言之，在Abstraction类中维护一个Implementor类指针，需要采用不同的实现方式的时候只需要传入不同的Implementor派生类就可以了。

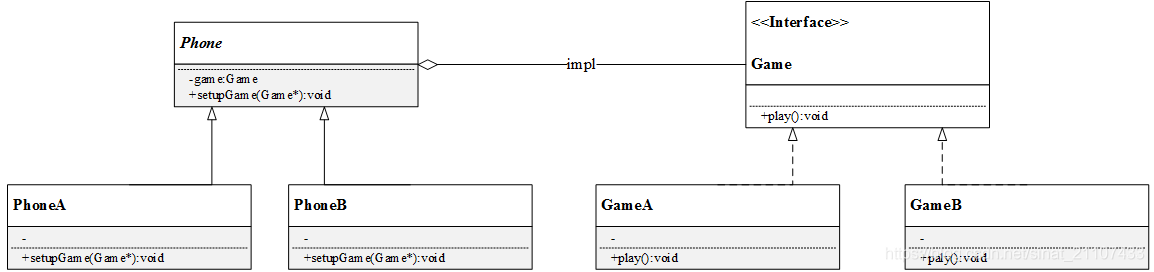
**3.桥接模式代码实例**

以引言中的故事为例，Jungle学习了桥接模式后大受启发，想实现如下操作：

**新手机上能够迅速在新手机上安装（setup）并玩（play）游戏**

**新增加一个游戏时Jungle能够在已有手机上安装并play**

在这个实例里，手机是抽象类Abstraction，具有玩游戏这样的实现类接口Implementor，不同的手机品牌扩充抽象类RefinedAbstraction，多个不同的游戏则是具体实现类ConcreteImplementor。



**3.1.实现类**

//实现类接口

class Game

{

public:

Game(){}

virtual void play() = 0;

private:

};

//具体实现类GameA

class GameA:public Game

{

public:

GameA(){}

void play(){

printf("Jungle玩游戏A\n");

}

};

//具体实现类GameB

class GameB :public Game

{

public:

GameB(){}

void play(){

printf("Jungle玩游戏B\n");

}

};

实现类Game中声明了play的接口，不过它是一个虚方法，其实现在具体实现类GameA和GameB中定义。

**3.2.抽象类和扩充抽象类**

//抽象类Phone

class Phone

{

public:

Phone(){

}

//安装游戏

virtual void setupGame(Game \*igame) = 0;

virtual void play() = 0;

private:

Game \*game;

};

//扩充抽象类PhoneA

class PhoneA:public Phone

{

public:

PhoneA(){

}

//安装游戏

void setupGame(Game \*igame){

this->game = igame;

}

void play(){

this->game->play();

}

private:

Game \*game;

};

//扩充抽象类PhoneB

class PhoneB :public Phone

{

public:

PhoneB(){

}

//安装游戏

void setupGame(Game \*igame){

this->game = igame;

}

void play(){

this->game->play();

}

private:

Game \*game;

};

抽象类Phone中也声明了两个虚方法，并且定义了一个实现类的对象，使抽象和实现具有关联关系。而对象的实例化则放在客户端使用时进行。

**3.3.客户端代码示例**

#include <iostream>

#include "BridgePattern.h"

int main()

{

Game \*game;

Phone \*phone;

//Jungle买了PhoneA品牌的手机，想玩游戏A

phone = new PhoneA();

game = new GameA();

phone->setupGame(game);

phone->play();

printf("++++++++++++++++++++++++++++++++++\n");

//Jungle想在这个手机上玩游戏B

game = new GameB();

phone->setupGame(game);

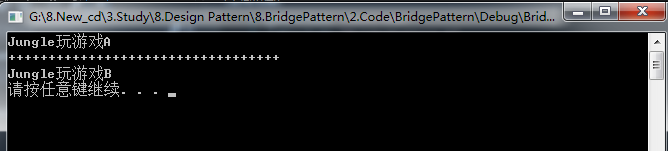
phone->play();

system("pause");

return 0;

}

**3.4.效果**



**4.桥接模式总结**

**优点：**

（1）分离抽象接口与实现部分，使用对象间的关联关系使抽象与实现解耦；

（2）桥接模式可以取代多层继承关系，多层继承违背单一职责原则，不利于代码复用；

（3）桥接模式提高了系统可扩展性，某个维度需要扩展只需增加实现类接口或者具体实现类，而且不影响另一个维度，符合开闭原则。

**缺点：**

（1）桥接模式难以理解，因为关联关系建立在抽象层，需要一开始就设计抽象层；

（2）如何准确识别系统中的两个维度是应用桥接模式的难点。

**适用场景：**

（1）如果一个系统需要在抽象化和具体化之间增加灵活性，避免在两个层次之间增加继承关系，可以使用桥接模式在抽象层建立关联关系；

（2）抽象部分和实现部分可以各自扩展而互不影响；

（3）一个类存在多个独立变化的维度，可采用桥接模式。

**第十二章 逆袭成为主管和高级经理！——组合模式**

**今天Jungle又是被压榨的一天：**

**同事让帮忙做个测试，一测就是一上午；主管叫去开会，尽管他是三级主管，但也是Jungle的主管呀！更磨人的在后头：**

**Jungle由于项目需要，要申请某个IT权限，需要三级主管、二级主管、成都区总经理层层批复……**

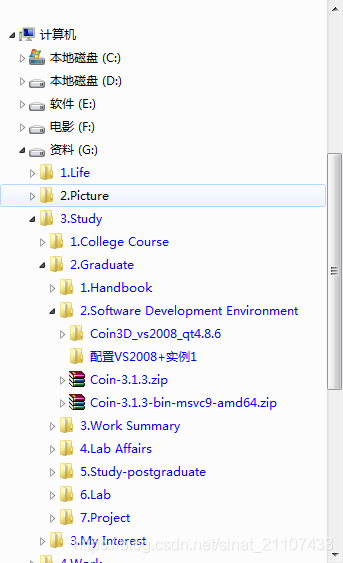
**这公司哟，简直就像一棵树，Jungle只是一片绿叶，绿叶前面有树枝，树枝前面还有树枝，直到树的躯干……Jungle何时才能成为大树枝呢？**

这一幕，相信大家都感同身受吧！整个公司的组织结构就像是一个树形结构，公司分成几个区，每个区分成几个产品部门，一个产品部门又会分成几个组，组里可能还会有细分，分到最后，就是类似于Jungle这样的小喽喽了。

看到了吗？这就是组合模式的典型应用场景！什么是组合模式呢？且听Jungle一一道来。

**1.组合模式简介**

组合模式关注包含叶子节点和容器节点的结构以及他们构成的组织形式。这样的组织形式的特点在于：叶子节点不再包含成员对象，而容器节点可以包含成员对象，这些对象可以是叶子节点，也可以是容器节点。这些节点通过不同的递归组合形成一个树形结构。好比Windows系统的目录结构，文件夹里包含文件和子文件夹，子文件夹里还可以包含文件和文件夹。



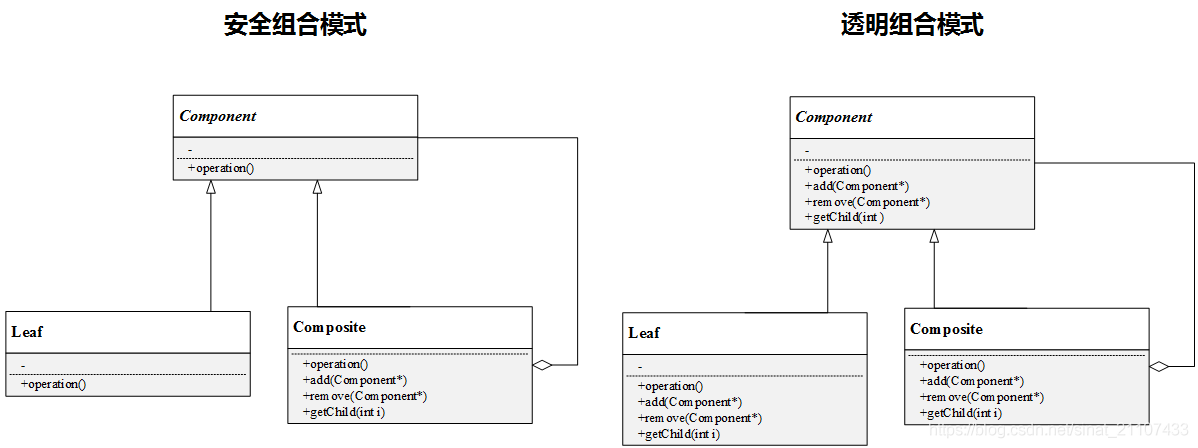
组合模式为叶子节点和容器节点提供了公共的抽象构建类，客户端无需关心所操作的对象是叶子节点还是容器节点，只需针对抽象构建类编程处理即可。

**组合模式定义：**

**组合多个对象形成树形结构以表示具有部分-整体关系的层次结构。组合模式让客户端可以统一对待单个对象和组合对象。**

**2.组合模式结构**

组合模式结构图如下图：



结合引言及组合模式UML图可知组合模式有如下角色：

（1）Component（抽象构件）：Component是一个抽象类，定义了构件的一些公共接口，这些接口是管理或者访问它的子构件的方法（如果有子构件），具体的实现在叶子构件和容器构件中进行。

（2）Leaf（叶子构件）：它代表树形结构中的叶子节点对象，叶子构件没有子节点，它实现了在抽象构件中定义的行为。对于抽象构件定义的管理子构件的方法，叶子构件可以通过抛出异常、提示错误等方式进行处理。

（3）Composite（容器构件） ：容器构件一方面具体实现公共接口，另一方面通过聚合关系包含子构件，子构件可以是容器构件，也可以是叶子构件。

结合公司组织架构的例子，各个分级部门是容器构件，类似于Jungle的员工是叶子构件；结合Windows目录结构，文件夹是容器构件，可以包含子文件夹和文件，而文件则是叶子构件，不能再包含子构件。

**2.1.透明组合模式**

如UML图所示，组合模式分为透明组合模式和安全组合模式。在透明组合模式中，抽象构件Component声明了add、remove、getChild等所有管理和访问子构件的方法，不论是叶子构件还容器构件都具有相同的接口。客户端在使用时可以一致地对待所有对象，即具体是叶子构件还是容器构件，对客户端而言是透明的，因为它们都暴露相同的接口。

但是，叶子构件是没有子构件的，所有就没有add、remove和getChild方法的，所以必须在叶子构件的实现中提供相应的错误处理代码，否则代码会出错。

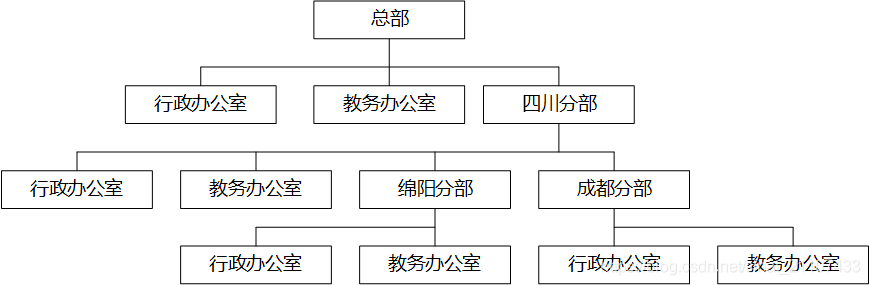
**2.2.安全组合模式**

在安全组合模式中，抽象构件Component没有声明任何管理和访问子构件的方法，在具体的实现类中才去定义这些行为。之所以叫“安全”，是因为叶子构件没有子构件，也就不必定义管理访问子构件的方法，对客户端而言，当它处理子构件时，不可能调用到类似透明组合模式中的子构件的add、remove等方法，因此也就不会出错。

**安全模式的不足在于不够透明，客户端使用时必须有区别的对待叶子构件和容器构件。**

**3.组合模式代码实例**

如下图，某个公司的组织结构分为总部、省级分部、市级分部和各个分部的行政办公室和教务办公室：



这是一个典型的树形结构，本例将采用透明这模式来实现上述结构。

**3.1.抽象构件**

//抽象构件

class Component

{

public:

Component(){}

Component(string iName){

this->name = iName;

}

//增加一个部门或办公室

virtual void add(Component\*) = 0;

//撤销一个部门或办公室

virtual void remove(Component\*) = 0;

//

virtual Component\* getChild(int) = 0;

//各部门操作

virtual void operation() = 0;

string getName(){

return name;

}

private:

string name;

};

**3.2.叶子构件**

叶子构件定义了行政办公室和教务办公室，都集成自Office。

//叶子构件：办公室

class Office :public Component

{

public:

Office(string iName){

this->name = iName;

}

Office(){}

void add(Component\* c){

printf("not support!\n");

}

void remove(Component\* c){

printf("not support!\n");

}

Component\* getChild(int i){

printf("not support!\n");

return NULL;

}

private:

string name;

};

//叶子构件：行政办公室

class AdminOffice :public Office

{

public:

AdminOffice(string iName){

this->name = iName;

}

void operation(){

printf("-----Administration Office:%s\n", name.c\_str());

}

private:

string name;

};

//叶子构件：教务办公室

class DeanOffice :public Office

{

public:

DeanOffice(string iName){

this->name = iName;

}

void operation(){

printf("-----Dean Office:%s\n", name.c\_str());

}

private:

string name;

};

**3.3.容器构件**

//容器构件SubComponent

class SubComponent :public Component

{

public:

SubComponent(string iName){

this->name = iName;

}

void add(Component \*c){

componentList.push\_back(c);

}

void remove(Component \*c){

for (int i = 0; i < componentList.size(); i++){

if (componentList[i]->getName() == c->getName()){

componentList.erase(componentList.begin() + i);

break;

}

}

}

Component\* getChild(int i){

return (Component\*)componentList[i];

}

void operation(){

printf("%s\n", this->name.c\_str());

for (int i = 0; i < componentList.size(); i++){

((Component\*)componentList[i])->operation();

}

}

private:

string name;

//构件列表

vector<Component\*>componentList;

};

**3.4.客户端代码示例**

#include <iostream>

#include "CompositePattern.h"

int main()

{

Component \*head, \*sichuanBranch, \*cdBranch, \*myBranch, \*office1, \*office2, \*office3,

\*office4, \*office5, \*office6, \*office7, \*office8;

head = new SubComponent("总部");

sichuanBranch = new SubComponent("四川分部");

office1 = new AdminOffice("行政办公室");

office2 = new DeanOffice("教务办公室");

cdBranch = new SubComponent("成都分部");

myBranch = new SubComponent("绵阳分部");

office3 = new AdminOffice("行政办公室");

office4 = new DeanOffice("教务办公室");

office5 = new AdminOffice("行政办公室");

office6 = new DeanOffice("教务办公室");

office7 = new AdminOffice("行政办公室");

office8 = new DeanOffice("教务办公室");

cdBranch->add(office5);

cdBranch->add(office6);

myBranch->add(office7);

myBranch->add(office8);

sichuanBranch->add(office3);

sichuanBranch->add(office4);

sichuanBranch->add(cdBranch);

sichuanBranch->add(myBranch);

head->add(office1);

head->add(office2);

head->add(sichuanBranch);

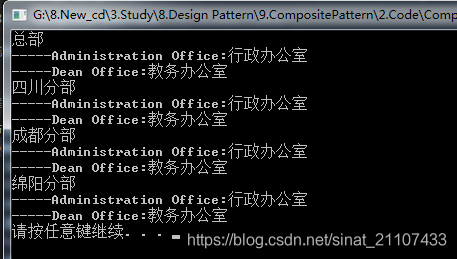
head->operation();

system("pause");

return 0;

}

**3.5.效果**



**4.组合模式总结**

**优点：**

**（1）**清楚地定义分层次的复杂对象，表示出复杂对象的层次结构，让客户端忽略层次的差异；

（2）客户端可以一致地使用层次结构中各个层次的对象，而不必关心其具体构件的行为如何实现；

（3）在组合模式中增加新的叶子构件和容器构件非常方便，易于扩展，符合开闭原则；

（4）为树形结构的案例提供了解决方案。

**缺点：**

子构件或容器构件的行为受限制，因为它们来自相同的抽象层。如果要定义某个容器或者某个叶子节点特有的方法，那么要求在运行时判断对象类型，增加了代码的复杂度。

**适用场景：**

（1）系统中需要用到树形结构；

（2）系统中能够分离出容器节点和叶子节点；

（3）具有整体和部门的层次结构中，能够通过某种方式忽略层次差异，使得客户端可以一致对待。

**第十三章人靠衣装，美靠靓装——装饰模式，教你成为最靓的仔！**

**成都的气温说降就降啊！还没立冬呢，温度骤然下降，只有10度了。扛不住了，Jungle要穿一件毛衣！**

**额，出门还是很冷！那再带个围巾！**

**天，上半身不冷了，下半身还那么冷~~~穿上秋裤！**

**出门前想了想，这种天气骑摩拜单车一定吹着手冷，于是Jungle又带上了手套……**

**看到了吗，天冷了，Jungle想加什么衣裤即加什么衣裤，各种装饰让Jungle不再感到寒冷。不过，不管Jungle穿再多衣服，Jungle始终还是那个Jungle，那个生活在成都的狗子，这一点是不变的！**

看到上面的例子了吗？这就是装饰模式的典型应用，穿戴再多服饰并不会改变Jungle的本质，但穿戴会让Jungle更加温暖和帅气，满足Jungle冷天出行的需求。在软件设计中，对已有对象的功能进行扩展，以获得更加符合用户需求的对象，使得对象具有更加强大的功能，这就是装饰模式。

**1.装饰模式简介**

装饰模式可以在不改变一个对象本身功能的基础上给对象增加额外的新行为，比如手机，为防止摔坏，可以给手机贴膜或者带上保护套；为美观，可以在保护套上贴卡通贴纸；为便于携带可以增加挂绳，如下图。但这并不改变手机的本质。



**装饰模式：**

**动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案。**

  装饰模式是一种用于替代继承的技术。通过一种无须定义子类的方式给对象动态增加职责，使用对象之间的关联关系取代类之间的继承关系。装饰模式中引入了装饰类，在装饰类中既可以调用待装饰的原有对象的方法，还可以增加新的方法，以扩充原有类的功能。

**2.装饰模式结构**

装饰模式中有如下角色：

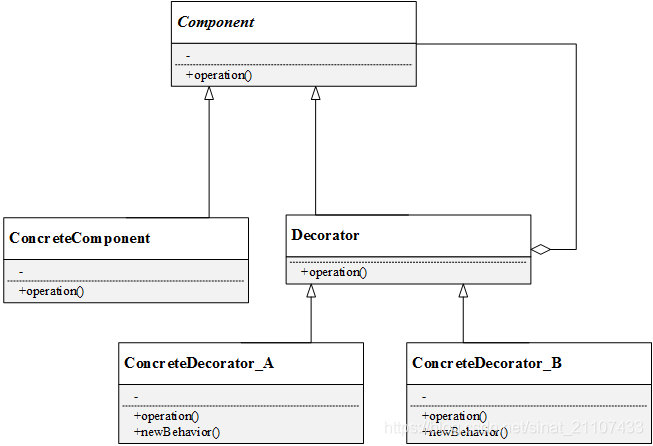
（1）Component（抽象构件）：是具体构件类和装饰类的共同基类，声明了在具体构件中定义的方法，客户端可以一致的对待使用装饰前后的对象；

（2）ConcreteComponent（具体构件）：具体构件定义了构件具体的方法，装饰类可以给它增加更多的功能；

（3）Decorator（抽象装饰类）：用于给具体构件增加职责，但具体职责在其子类中实现。抽象装饰类通过聚合关系定义一个抽象构件的对象，通过该对象可以调用装饰之前构件的方法，并通过其子类扩展该方法，达到装饰的目的；

（4）ConcreteDecorator（具体装饰类）： 向构件增加新的功能。

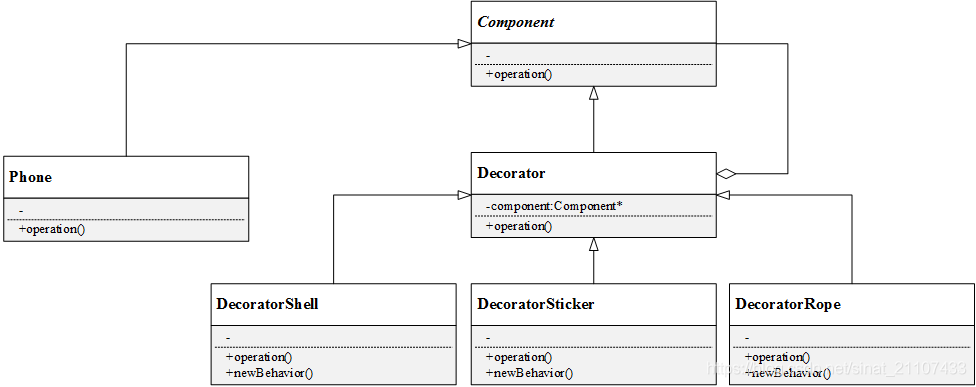
装饰模式的结构如图所示：



以上面提到的手机为例，手机就是一个具体构件，而手机壳、手机贴纸和手机挂绳，都是具体的装饰类。以Jungle在冷天出门前精心打扮为例，Jungle本人是一个具体构件对象，各类衣裤围巾手套都是具体的装饰类对象。

**3.装饰模式代码实例**

本节以给手机带上手机壳、贴上手机贴纸、系上手机挂绳为例，展示装饰模式的代码。该例的UML图如下所示：



**3.1.抽象构件类**

//抽象构件

class Component

{

public:

Component(){}

virtual void operation() = 0;

};

**3.2.具体构件类**

//具体构件类

class Phone :public Component

{

public:

Phone(){}

void operation(){

printf("手机\n");

}

};

**3.3.装饰类**

**3.3.1.抽象装饰类**

//抽象装饰类

class Decorator :public Component

{

public:

Decorator(){}

Decorator(Component \*c){

this->component = c;

}

void operation(){

this->component->operation();

}

Component \*getComponent(){

return this->component;

}

void setComponent(Component \*c){

this->component = c;

}

private:

Component \*component;

};

抽象装饰类中有一个成员对象component，以及setter和getter方法。

**3.3.2.具体装饰类**

具体装饰类一共有三个，分别是手机壳装饰类DecoratorShell、贴纸装饰类DecoratorSticker和挂绳装饰类DecoratorRope。每一个具体装饰类都增加了各自新的职责newBehavior。

//具体装饰类：手机壳

class DecoratorShell:public Decorator

{

public:

DecoratorShell(){}

DecoratorShell(Component \*c){

this->setComponent(c);

}

void operation(){

this->getComponent()->operation();

this->newBehavior();

}

void newBehavior(){

printf("装手机壳\n");

}

};

//具体装饰类：手机贴纸

class DecoratorSticker :public Decorator

{

public:

DecoratorSticker(){}

DecoratorSticker(Component \*c){

this->setComponent(c);

}

void operation(){

this->getComponent()->operation();

this->newBehavior();

}

void newBehavior(){

printf("贴卡通贴纸\n");

}

};

//具体装饰类：手机挂绳

class DecoratorRope :public Decorator

{

public:

DecoratorRope(){}

DecoratorRope(Component \*c){

this->setComponent(c);

}

void operation(){

this->getComponent()->operation();

this->newBehavior();

}

void newBehavior(){

printf("系手机挂绳\n");

}

};

**3.4.客户端代码示例**

客户端展示了三段代码，分别为三个手机配上不同的装饰：

#include <iostream>

#include "DecoratorPattern.h"

int main()

{

printf("\nJungle的第一个手机：\n");

Component \*c;

Component \*com\_Shell;

c = new Phone();

com\_Shell = new DecoratorShell(c);

com\_Shell->operation();

printf("\nJungle的第二个手机：\n");

Component \*c2;

Component \*com\_Shell2;

c2 = new Phone();

com\_Shell2 = new DecoratorShell(c2);

Component \*com\_Sticker;

com\_Sticker = new DecoratorSticker(com\_Shell2);

com\_Sticker->operation();

printf("\nJungle的第三个手机：\n");

Component \*c3;

Component \*com\_Shell3;

c3 = new Phone();

com\_Shell3 = new DecoratorShell(c3);

Component \*com\_Sticker2;

com\_Sticker2 = new DecoratorSticker(com\_Shell3);

Component \*com\_Rope;

com\_Rope = new DecoratorRope(com\_Sticker2);

com\_Rope->operation();

printf("\n\n");

system("pause");

return 0;

}

**3.5.效果**



由上述代码示例及运行结果可以看到，客户可以根据个人需要，动态增加构件的职责，但并不影响构件最基本的功能，不会修改原有的设计。同时，如果要增加新的职责（装饰），非常容易进行扩展。

**4.总结**

**优点：**

（1）对于扩展一个类的新功能，装饰模式比继承更加灵活；

（2）动态扩展一个对象的功能；

（3）可以对一个对象进行多次装饰（如上述例子第二个手机和第三个手机）；

（4）具体构件类和具体装饰类可以独立变化和扩展，符合开闭原则。

**缺点：**

（1）装饰模式中会增加很多小的对象，对象的区别主要在于各种装饰的连接方式不同，而并不是职责不同，大量小对象的产生会占用较多的系统资源；

（2）装饰模式比继承模式更灵活，但也更容易出错，更难于排错。

**适用场景：**

（1）在不影响其他对象的情况下，给单个对象动态扩展职责；

（2）不适宜采用继承的方式进行扩展的时候，可以考虑使用装饰模式。

**第十四章细粒度对象的大面积复用！——实例分析享元模式**

**知道围棋吗？不会下围棋那总见过吧？四四方方的棋盘上，白色棋子和黑色棋子分布在棋盘的各个位置上。**

**棋子与棋子之间的区别是什么？除了颜色和位置，好像没什么不同了吧！也就是说，每个棋子对象的大部分状态都是一样的（形状、材料、质地等）。如果我们要设计一个程序来实现下围棋的功能，该如何来创建或者表示这上百个棋子对象呢？**

类似的，你想输入一段英文段落，无论每个单词再长再复杂，也无非都是由26个字母中的几个组成的。上述两个示例的共同点在于，整个环境中存在大量相同或者相似的、需要重复使用的对象。针对这样的场景，面向对象设计中有一类值得借鉴的设计模式是不错的解决方案——享元模式。

**1.享元模式简介**

如果一个系统在运行时创建太多相同或者相似的对象，会占用大量内存和资源，降低系统性能。享元模式通过共享技术实现相同或相似的细粒度对象的复用，提供一个享元池存储已经创建好的对象，并通过享元工厂类将享元对象提供给客户端使用。

**享元模式：  
运用共享技术有效地支持大量细粒度对象的复用。**

**享元模式要求被共享的对象必须是细粒度对象。**如上面提到的输入英文段落的例子，26个字母可能随时被客户重复使用。尽管每个字母可能出现的位置不一样，但在物理上它们共享同一个对象（同一个实例）。利用享元模式，可以创建一个存储26个字母对象的享元池，需要时从享元池中取出。

享元对象能够做到共享的关键在于区分了内部状态和外部状态：

**（1）内部状态：**存储在享元对象内部，不会随着环境的改变而改变的，内部状态可以共享。比如围棋中棋子的形状、大小，不会随着外部变化而变化；比如字母A，无论谁使用，都是A，不会变化；

**（2）外部状态：**随环境变化而变化、不可以共享的状态，如棋子的位置、颜色，如每个字母的位置。外部状态一般由客户端保存，在使用时再传入到享元对象内部。不同的外部状态之间是相互独立的，棋子A和棋子B的位置可以不同，并且不会相互影响。

**2.享元模式结构**

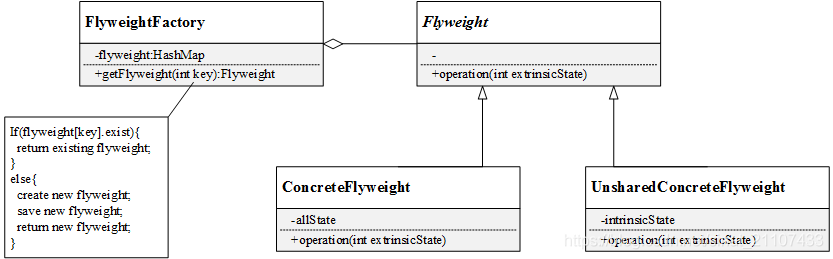
享元模式常常结合工厂模式一起使用，其结构包含抽象享元类、具体享元类、非共享具体享元类和享元工厂类：

**（1）Flyweight（抽象享元类）：**是一个抽象类，声明了具体享元类公共的方法，这些方法可以向外部提供享元对象的内部状态数据，也可以通过这些方法设置外部状态；

**（2）ConcreteFlyweight（具体享元类）：**具体实现抽象享元类声明的方法，具体享元类中为内部状态提供存储空间。具体享元类常常结合[单例模式](https://blog.csdn.net/sinat_21107433/article/details/102649056)来设计实现，保证每个享元类对象只被创建一次，为每个具体享元类提供唯一的享元对象。

**（3）UnsharedConcreteFlyweight（非共享具体享元类）：**并不是所有抽象享元类的子类都需要被共享，可以将这些类设计为非共享具体享元类；

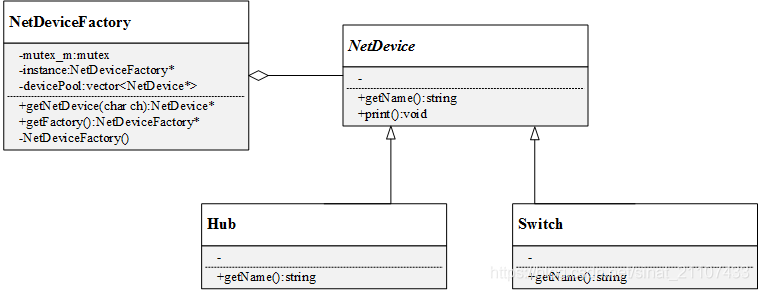
**（4）FlyweightFactory（享元工厂类）：**用于创建并管理享元对象，针对抽象享元类编程，将各种具体享元类对象存储在一个享元池中，享元池一般设计为一个存储键值对的集合（或者其他类型的集合），可结合工厂模式设计。客户需要某个享元对象时，如果享元池中已有该对象实例，则返回该实例，否则创建一个新的实例，给客户返回新的实例，并将新实例保存在享元池中。

****

**3.享元模式代码实例**

**很多网络设备都是支持共享的，如交换机（switch）、集线器（hub）等。多台中断计算机可以连接同一台网络设备，并通过网络设备进行数据转发。本节Jungle将使用享元模式来模拟共享网络设备的实例。**

**本例中，交换机（switch）和集线器（hub）是具体享元对象。UML图如下所示：**

****

**3.1.抽象享元类**

// 抽象享元类

class NetDevice

{

public:

NetDevice(){}

virtual string getName() = 0;

void print(){

printf("NetDevice :%s\n",getName().c\_str());

}

};

**3.2.具体享元类**

具体享元类有集线器和交换机，实现了抽象享元类声明的方法。

// 具体享元类:集线器

class Hub :public NetDevice

{

public:

Hub(){}

string getName(){

return "集线器";

}

};

// 具体享元类:交换机

class Switch :public NetDevice

{

public:

Switch(){}

string getName(){

return "交换机";

}

};

**3.3.享元工厂类**

享元工厂类采用了单例模式，保证工厂实例的唯一性。采用一个vector作为共享池。

// 享元工厂类

class NetDeviceFactory

{

public:

NetDevice\* getNetDevice(char ch){

if (ch == 'S'){

return devicePool[1];

}

else if (ch == 'H'){

return devicePool[0];

}

else{

printf("wrong input!\n");

}

return NULL;

}

// 单例模式：返回享元工厂类的唯一实例

static NetDeviceFactory\* getFactory(){

if (instance == NULL){

m\_mutex.lock();

if (instance == NULL){

instance = new NetDeviceFactory();

}

m\_mutex.unlock();

}

return instance;

}

private:

NetDeviceFactory(){

Hub \*hub = new Hub();

Switch \*switcher = new Switch();

devicePool.push\_back(hub);

devicePool.push\_back(switcher);

}

static NetDeviceFactory\* instance;

static std::mutex m\_mutex;

// 共享池：用一个vector来表示

vector<NetDevice\*> devicePool;

};

NetDeviceFactory\* NetDeviceFactory::instance = NULL;

std::mutex NetDeviceFactory::m\_mutex;

**3.4.客户端代码示例**

#include <iostream>

#include "FlyweightPattern.h"

int main()

{

NetDeviceFactory \*factory = NetDeviceFactory::getFactory();

NetDevice \*device1, \*device2, \*device3, \*device4;

// 客户端1获取一个hub

device1 = factory->getNetDevice('H');

device1->print();

// 客户端2获取一个hub

device2 = factory->getNetDevice('H');

device2->print();

// 判断两个hub是否是同一个

printf("判断两个hub是否是同一个:\n");

printf("device1:%p\ndevice2:%p\n", device1, device2);

printf("\n\n\n\n");

// 客户端3获取一个switch

device3 = factory->getNetDevice('S');

device3->print();

// 客户端4获取一个switch

device4 = factory->getNetDevice('S');

device4->print();

// 判断两个switch是否是同一个

printf("判断两个switch是否是同一个:\n");

printf("device3:%p\ndevice4:%p\n", device3, device4);

printf("\n\n");

system("pause");

return 0;

}

客户端代码中，两个客户端分别获取集线器，Jungle打印出两个集线器的地址，来判断是否是同一个对象。同理，对交换机，Jungle也进行类似的判断。

**3.5.效果**



由测试结果可以看出，两个集线器对象的地址是相同的，说明它们都是同一个实例对象，两个交换机也都指向同一个交换机实例对象。由此说明本例的代码实现了网络设备的共享。

**3.6.有外部状态的享元模式**

进一步，尽管不同的终端计算机可能会共享同一个集线器（交换机），但是每个计算机接入的端口（port）是不一样的，端口就是每个享元对象的外部状态。 在享元模式的使用过程中，内部状态可以作为具体享元类的成员对象，而外部状态可以通过外部注入的方式添加到具体享元类中。

“通过外部注入”，因此，客户端可以通过函数传参的方式将“端口”号注入具体享元类：

// 抽象享元类

class NetDevice

{

public:

NetDevice(){}

virtual string getName() = 0;

/\*void print(){

printf("NetDevice :%s\n",getName().c\_str());

}\*/

void print(int portNum){

printf("NetDevice :%s port: %d\n", getName().c\_str(), portNum);

}

};

那么客户端的使用方式将变为：

#include <iostream>

#include "FlyweightPattern.h"

int main()

{

NetDeviceFactory \*factory = NetDeviceFactory::getFactory();

NetDevice \*device1, \*device2, \*device3, \*device4;

// 客户端2获取一个hub

device1 = factory->getNetDevice('H');

device1->print(1);

// 客户端2获取一个hub

device2 = factory->getNetDevice('H');

device2->print(2);

// 判断两个hub是否是同一个

printf("判断两个hub是否是同一个:\n");

printf("device1:%p\ndevice2:%p\n", device1, device2);

printf("\n\n\n\n");

// 客户端3获取一个switch

device3 = factory->getNetDevice('S');

device3->print(1);

// 客户端4获取一个hub

device4 = factory->getNetDevice('S');

device4->print(2);

// 判断两个hub是否是同一个

printf("判断两个switch是否是同一个:\n");

printf("device3:%p\ndevice4:%p\n", device3, device4);

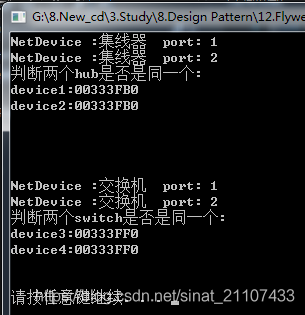
printf("\n\n");

system("pause");

return 0;

}

 效果如下：



**4.总结**

**优点：**

（1）享元模式通过享元池存储已经创建好的享元对象，实现相同或相似的细粒度对象的复用，大大减少了系统中的对象数量，节约了内存空间，提升了系统性能；

（2）享元模式通过内部状态和外部状态的区分，外部状态相互独立，客户端可以根据需求任意使用。

**缺点**：

享元模式需要增加逻辑来取分出内部状态和外部状态，增加了编程的复杂度；

**适用环境：**

（1）当一个系统中有大量重复使用的相同或相似对象时，使用享元模式可以节约系统资源；

（2）对象的大部分状态都可以外部化，可以将这些状态传入对象中。

**第十五章如何吃到回锅肉？找厨师啊——外观模式实例解析**

**好不容易盼到周末啦！Jungle想吃点好的犒劳自己！吃什么呢？回锅肉！是的，吃回锅肉！**

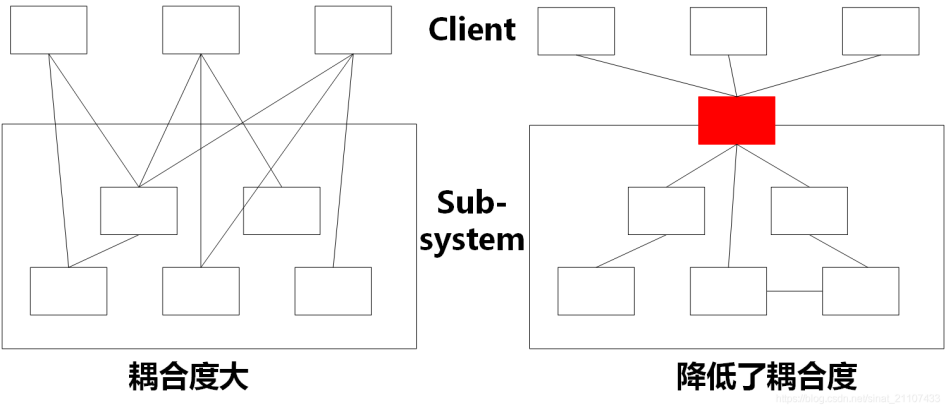
**可是这过程好麻烦啊，先得去市场里买肉，买回来得洗好，然后切好，再炒肉，最后才能吃上！不仅过程繁杂，而且Jungle还得跟市场、厨房打交道，想想都头大。**

**如果有个厨师就好了，Jungle直接告诉厨师“我要吃回锅肉”，20分钟后厨师直接端上来就开吃。而中间那些买肉洗肉切肉的过程Jungle统统不关心了，而且Jungle也不必再关心市场和厨房，直接和厨师说句话就ok！真是方便！**

在这个例子中，厨师整合了一系列复杂的过程，外界（Jungle）只需与厨师交互即可。在软件设计模式中，有一类设计模式正式如此——外观模式。

**1.外观模式简介**

外观模式是一种使用频率较高的设计模式，它提供一个外观角色封装多个复杂的子系统，简化客户端与子系统之间的交互，方便客户端使用。外观模式可以降低系统的耦合度。如果没有外观类，不同的客户端在需要和多个不同的子系统交互，系统中将存在复杂的引用关系，如下图。引入了外观类，原有的复杂的引用关系都由外观类实现，不同的客户端只需要与外观类交互。



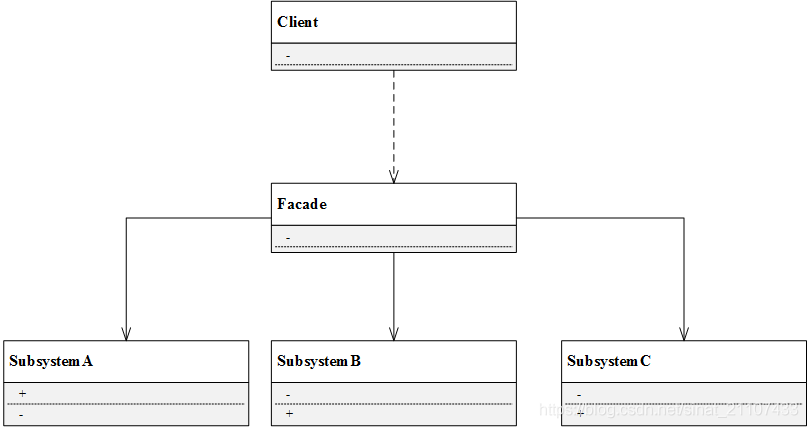
**外观模式：**

**为子系统中的一组接口提供一个统一的入口。外观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。**

 外观模式的应用很多，比如浏览器，用户要查找什么东西，不论是浏览知乎、腾讯或者CSDN，用户都只需要打开浏览器即可，剩下的搜索工作由浏览器完成。

**2.外观模式结构**

外观模式的UML结构图如下所示：



外观模式一共有以下角色：

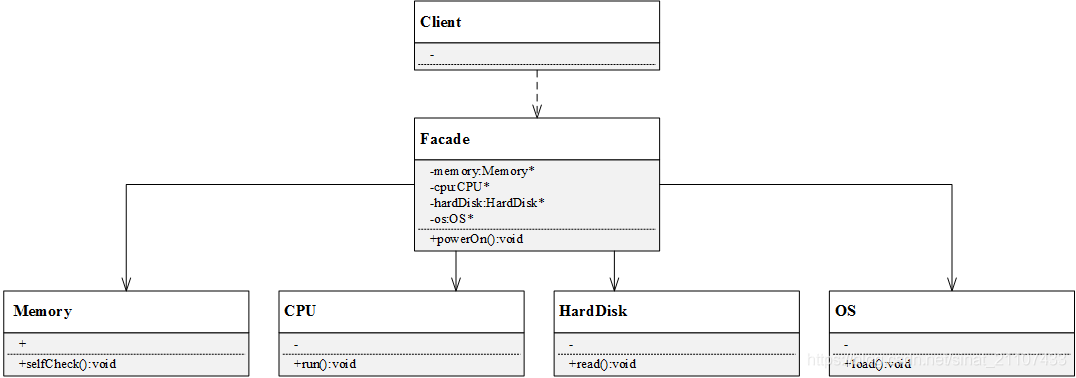
（1）Facade（外观角色）：外观角色可以知道多个相关子系统的功能，它将所有从客户端发来的请求委派给相应的子系统，传递给相应的子系统处理。

（2）SubSystem（子系统角色）：子系统是一个类，或者由多个类组成的类的集合，它实现子系统具体的功能。

**3.外观模式代码实例**

**电脑主机（Mainframe）中只需要按下主机的开机按钮（powerOn），即可调用其他硬件设备和软件的启动方法，如内存（Memory）的自检（selfCheck）、CPU的运行（run）、硬盘（HardDisk）的读取（read）、操作系统（OS）的载入（load）等。如果某一过程发生错误则电脑开机失败。**

这里Jungle用外观模式来模拟该过程，该例子UML图如下：



**3.1.子系统类**

本例中一共有4个子系统，因此设计4个类：Memory、CPU、HardDisk和OS，并且每个子系统都有自己独立的流程。

//子系统：内存

class Memory

{

public:

Memory(){}

void selfCheck(){

printf("…………内存自检……\n");

}

};

//子系统：CPU

class CPU

{

public:

CPU(){}

void run(){

printf("…………运行CPU运行……\n");

}

};

//子系统：硬盘

class HardDisk

{

public:

HardDisk(){}

void read(){

printf("…………读取硬盘……\n");

}

};

//子系统：操作系统

class OS

{

public:

OS(){}

void load(){

printf("…………载入操作系统……\n");

}

};

**3.2.外观类设计**

//外观类

class Facade

{

public:

Facade(){

memory = new Memory();

cpu = new CPU();

hardDisk = new HardDisk();

os = new OS();

}

void powerOn(){

printf("正在开机……\n");

memory->selfCheck();

cpu->run();

hardDisk->read();

os->load();

printf("开机完成！\n");

}

private:

Memory \*memory;

CPU \*cpu;

HardDisk \*hardDisk;

OS \*os;

};

**3.3.客户端代码示例**

#include <iostream>

#include "FacadePattern.h"

int main()

{

Facade \*facade = new Facade();

facade->powerOn();

printf("\n\n");

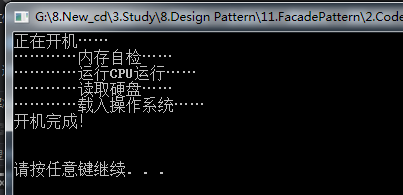
system("pause");

return 0;

}

看到了吗，客户端的代码就是如此简单，跟子系统无关！

**3.4.效果**



**4.总结**

**优点：**

（1）外观模式使得客户端不必关心子系统组件，减少了与客户端交互的对象的数量，简化了客户端的编程；

（2）外观模式可以大大降低系统的耦合度；

（3）子系统的变化并不需要修改客户端，只需要适当修改外观类即可；

（4）子系统之间不会相互影响。

**缺点：**

（1）如果需要增加或者减少子系统，需要修改外观类，违反开闭原则；

（2）并不能限制客户端直接与子系统交互，但如果加强限制，又使得系统灵活度降低。

**适用场景：**

（1）为访问一系列复杂的子系统提供一个统一的、简单的入口，可以使用外观模式；

（2）客户端与多个子系统之间存在很大依赖，但在客户端编程，又会增加系统耦合度，且使客户端编程复杂，可以使用外观模式。

**第十六章 双十一天猫购物找代理？先学会代理模式**

在软件设计模式中，也有一种模式可以提供与代购网站类似的功能。当客户端不能或者不便直接访问一个对象时，可以通过一个称为“代理”的第三方来间接访问，这样的设计模式称为代理模式。

**1.代理模式简介**

代理模式在软件设计中广泛应用，而且产生的变种很多，如远程代理、虚拟代理、缓冲代理、保护代理等。

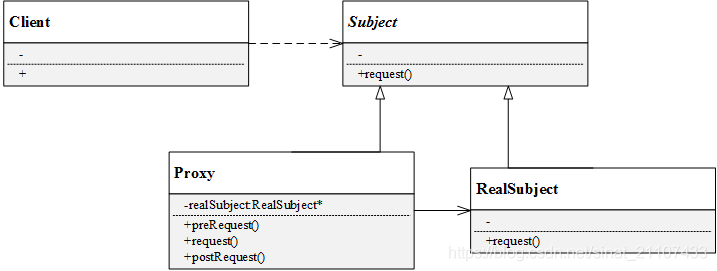
**代理模式：**

**给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。**

代理模式是一种对象结构型模式，在该模式中引入了一个代理对象，在客户端和目标访问对象之间起到中介的作用。代理对象可以屏蔽或删除客户不想访问的内容和服务，也可以根据客户需求增加新的内容和服务。

**2.代理模式结构**

代理模式的关键是代理类（Proxy）。代理模式中引入了抽象层，客户端针对抽象层编程，这样使得客户端可以一致对待真实对象和代理对象。代理模式主要有抽象主题角色（Subject）、代理主题角色（Proxy）和真实主题角色（RealSubject）组成，其UML图如下：



（1）抽象主题角色（Subject）：声明了代理主题角色和真实主题角色共同的一些接口，因此在任何可以使用真实主题对象的地方都可以使用代理主题角色（想一想代购是不是也是这样？），客户端通常针对抽象主题编程；

（2）代理主题角色（Proxy）：代理主题角色通过关联关系引用真实主题角色，因此可以控制和操纵真实主题对象；代理主题角色中提供一个与真实主题角色相同的接口（以在需要时代替真实主题角色），同时还可以在调用对真实主题对象的操作之前或之后增加新的服务和功能；

（3）真实主题角色（RealSubject）：真实主题角色是代理角色所代表的真实对象，提供真正的业务操作，客户端可以通过代理主题角色间接地调用真实主题角色中定义的操作。

**在实际开发过程中，代理模式产生了很多类型：**

（1）远程代理（Remote Proxy）：为一个位于不同地址空间的对象提供一个本地的代理对象。不同的地址空间可以在相同或不同的主机中。

（2）虚拟代理（Virtual Proxy）：当创建一个对象需要消耗大量资源时，可以先创建一个消耗较少资源的虚拟代理来表示，当真正需要时再创建。

（3）保护代理（Protect Proxy）：给不同的用户提供不同的对象访问权限。

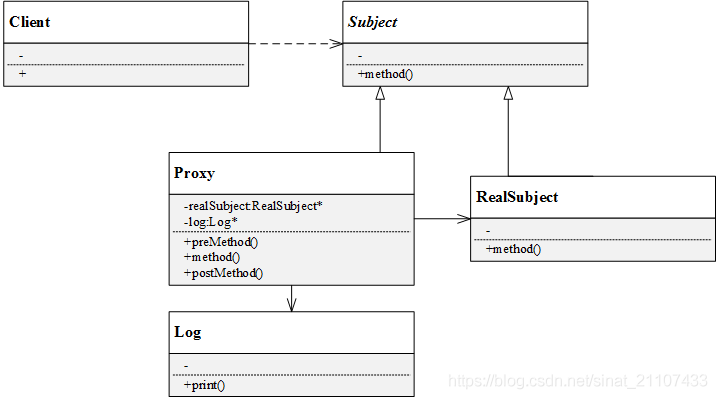
（4）缓冲代理（Cache Proxy）：为某一个目标操作的结果提供临时存储空间，以使更多用户可以共享这些结果。

（5）智能引用代理（Smart Reference Proxy）：当一个对象被引用时提供一些额外的操作，比如将对象被调用的次数记录下来等。

**3.代理模式代码实例**

**在某应用软件中需要记录业务方法的调用日志，在不修改现有业务的基础上位每个类提供一个日志记录代理类，在代理类中输出日志，例如在业务方法method()调用之前输出“方法method()被调用，调用时间为2019-10-28 07:33:30”，调用之后输出“方法method()”调用成功。在代理类中调用真实业务类的业务方法，使用代理模式设计该日志记录模块的结构。**

在这个案例中，真实主题角色是真实业务类，在代理类中调用真实主题角色的method()方法。该实例的UML图如下：



**3.1.抽象主题角色**

声明抽象方法method():

// 抽象主题角色

class Subject

{

public:

Subject(){}

virtual void method() = 0;

};

**3.2.真实主题角色**

实现具体业务方法method()：

// 真实主题角色

class RealSubject :public Subject

{

public:

RealSubject(){}

void method(){

printf("调用业务方法\n");

}

};

**3.3.代理角色和Log类**

// Log类

class Log

{

public:

Log(){}

string getTime(){

time\_t t = time(NULL);

char ch[64] = { 0 };

//年-月-日 时:分:秒

strftime(ch, sizeof(ch)-1, "%Y-%m-%d %H:%M:%S", localtime(&t));

return ch;

}

};

// 代理类

class Proxy:public Subject

{

public:

Proxy(){

realSubject = new RealSubject();

log = new Log();

}

void preCallMethod(){

printf("方法method()被调用，调用时间为%s\n",log->getTime().c\_str());

}

void method(){

preCallMethod();

realSubject->method();

postCallMethod();

}

void postCallMethod(){

printf("方法method()调用调用成功!\n");

}

private:

RealSubject \*realSubject;

Log\* log;

};

**3.4.客户端代码示例**

#include <iostream>

#include "ProxyPattern.h"

int main()

{

Subject \*subject;

subject = new Proxy();

subject->method();

printf("\n\n");

system("pause");

return 0;

}

**3.5.效果**



**4.总结**

**优点：**

（1）代理模式能够协调调用者和被调用者，降低系统耦合度；

（2）客户端针对抽象主题角色编程，如果要增加或替换代理类，无需修改源代码，符合开闭原则，系统扩展性好；

（3）远程代理优点：为两个位于不同地址空间的对象的访问提供解决方案，可以将一些资源消耗较多的对象移至性能较好的计算机上，提高系统整体性能；

（4）虚拟代理优点：通过一个资源消耗较少的对象来代表一个消耗资源较多的对象，节省系统运行开销；

（5）缓冲代理优点：为某一个操作结果提供临时的存储空间，可以在后续操作中使用这些结果，缩短了执行时间；

（6）保护代理优点：:控制对一个对象的访问权限，为不同客户提供不同的访问权限。

**缺点：**

（7）增加了代理类和代理对象，增加了代理对象中的某些处理流程，可能会使得系统响应变慢；

（8）有的代理模式（如远程代理）实现代码较为复杂。

**适用环境：**

（1）当客户端对象需要访问远程主机中的对象——可以使用远程代理；

（2）当需要用一个资源消耗较少的对象来代表一个资源消耗较多的对象——虚拟代理；

（3）当需要限制不同用户对一个独享的访问权限——保护代理；

（4）当需要为一个频繁访问的操作结果提供临时存储空间——缓冲代理；

（5）当需要为一个对象的访问提供一些额外的操作——智能引用代理。

**第十七章 “欲戴王冠，必承其重”——深度解析职责链模式**

**1.职责链模式简介**

职责链模式又叫责任链模式。很多情况下，可以处理某个请求的对象可能不止一个，请求可以沿着某一条对象之间形成的关系一级一级由下家传递到上家，**形成一条链——职责链。职责链可以是直线，也可以是环或树形结构。常见的职责链形式是直线。链上的每一个对象都是请求的处理者，客户端要做的仅仅是发送请求，不需要关心请求的处理细节过程。由此，职责链模式将请求者和请求的接收者解耦。**

职责链模式定义如下：

**职责链模式：**

**避免将一个请求的发送者和接收者耦合在一起，让多个对象都有机会处理请求。将接收请求的对象连接成一条链，并且沿着这条链传递请求，直到有一个对象能够处理它为止。**

**2.职责链模式结构**

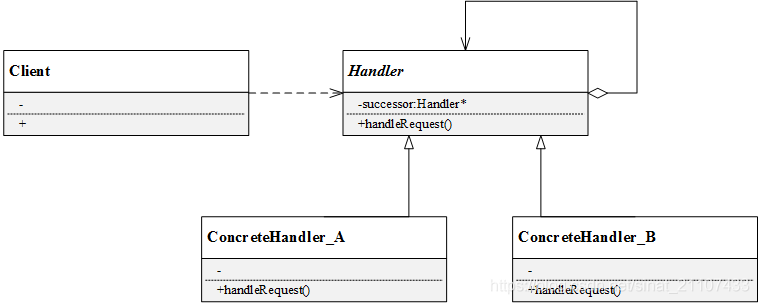
职责链模式中一共包含两个角色:

（1）Handler（抽象处理者）：抽象处理者一般为抽象类，声明了一个处理请求的接口handleRequest()，定义了一个抽象处理者类型的对象，作为其对下家的引用，通过该引用可以形成一条责任链。

（2）ConcreteHandler（具体处理者）： 是抽象处理者的子类，实现了处理请求的接口。在具体的实现中，如果该具体处理者能够处理该请求，就处理它，否则将该请求转发给后继者。具体处理者可以访问下一个对象。

由上述可知，在职责链模式中很多对象由每一个对象对其下家的引用连接起来形成一条链条，请求在这个链条上逐级传递，知道某一级能够处理这个请求为止。客户端不知道也不必知道是哪一级处理者处理了该请求，因为每个处理者都有相同的接口handleRequest()。接下来通过一个实例来进一步认识职责链模式。

职责链模式的UML结构如下图所示，职责链模式的核心在于引入了一个抽象处理者：



**3.职责链模式代码实例**

以引言中的例子为例，对于不同金额的票据，公司不同级别的领导处理情况如下：

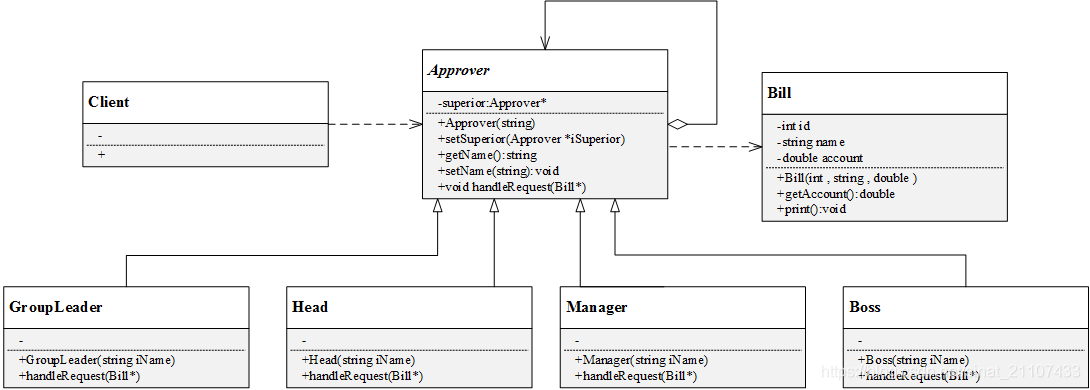
金额0~10万：组长可处理

金额10~30万：主管处理

金额30~60万：经理处理

金额超过60万：老板处理

本节Jungle将用C++模拟该过程。该实例UML图如下：



**3.1.票据类**

// 请求：票据

class Bill

{

public:

Bill(){}

Bill(int iId, string iName, double iAccount){

id = iId;

name = iName;

account = iAccount;

}

double getAccount(){

return this->account;

}

void print(){

printf("\nID:\t%d\n", id);

printf("Name:\t%s\n", name.c\_str());

printf("Account:\t%f\n", account);

}

private:

int id;

string name;

double account;

};

**3.2.抽象处理者**

// 抽象处理者

class Approver

{

public:

Approver(){}

Approver(string iName){

setName(iName);

}

// 添加上级

void setSuperior(Approver \*iSuperior){

this->superior = iSuperior;

}

// 处理请求

virtual void handleRequest(Bill\*) = 0;

string getName(){

return name;

}

void setName(string iName){

name = iName;

}

protected:

Approver \*superior;

private:

string name;

};

**3.3.具体处理者**

**3.3.1.具体处理者：组长**

// 具体处理者：组长

class GroupLeader :public Approver

{

public:

GroupLeader(){}

GroupLeader(string iName){

setName(iName);

}

// 处理请求

void handleRequest(Bill \*bill){

if (bill->getAccount() < 10){

printf("组长 %s 处理了该票据，票据信息：",this->getName().c\_str());

bill->print();

}

else{

printf("组长无权处理，转交上级……\n");

this->superior->handleRequest(bill);

}

}

};

**3.3.2.具体处理者：主管**

// 具体处理者：主管

class Head :public Approver

{

public:

Head(){}

Head(string iName){

setName(iName);

}

// 处理请求

void handleRequest(Bill \*bill){

if (bill->getAccount() >= 10 && bill->getAccount()<30){

printf("主管 %s 处理了该票据，票据信息：", this->getName().c\_str());

bill->print();

}

else{

printf("主管无权处理，转交上级……\n");

this->superior->handleRequest(bill);

}

}

};

**3.3.3.具体处理者：经理**

// 具体处理者：经理

class Manager :public Approver

{

public:

Manager(){}

Manager(string iName){

setName(iName);

}

// 处理请求

void handleRequest(Bill \*bill){

if (bill->getAccount() >= 30 && bill->getAccount()<60){

printf("经理 %s 处理了该票据，票据信息：", this->getName().c\_str());

bill->print();

}

else{

printf("经理无权处理，转交上级……\n");

this->superior->handleRequest(bill);

}

}

};

**3.3.4.具体处理者：老板**

// 具体处理者：老板

class Boss :public Approver

{

public:

Boss(){}

Boss(string iName){

setName(iName);

}

// 处理请求

void handleRequest(Bill \*bill){

printf("老板 %s 处理了该票据，票据信息：", this->getName().c\_str());

bill->print();

}

};

**3.5.客户端代码示例**

客户端创建了四个角色，分别是组长、主管、经理和老板，并设置了上下级关系。然后创建了4张票据，金额不等，都先统一交给组长处理。

#include <iostream>

#include "ChainOfResponsibility.h"

int main()

{

// 请求处理者：组长，兵哥，春总，老板

Approver \*zuzhang, \*bingge, \*chunzong, \*laoban;

zuzhang = new GroupLeader("孙大哥");

bingge = new Head("兵哥");

chunzong = new Manager("春总");

laoban = new Boss("张老板");

zuzhang->setSuperior(bingge);

bingge->setSuperior(chunzong);

chunzong->setSuperior(laoban);

// 创建报销单

Bill \*bill1 = new Bill(1, "Jungle", 8);

Bill \*bill2 = new Bill(2, "Lucy", 14.4);

Bill \*bill3 = new Bill(3, "Jack", 32.9);

Bill \*bill4 = new Bill(4, "Tom", 89);

// 全部先交给组长审批

zuzhang->handleRequest(bill1); printf("\n");

zuzhang->handleRequest(bill2); printf("\n");

zuzhang->handleRequest(bill3); printf("\n");

zuzhang->handleRequest(bill4);

printf("\n\n");

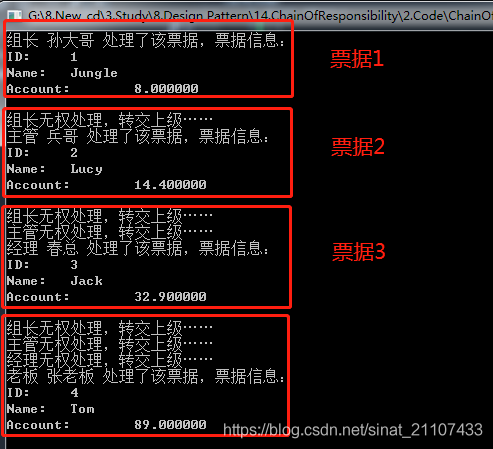
system("pause");

return 0;

}

**3.6.效果**

运行结果如下图，可以看到，针对不同金额的票据，处理请求在不同职级之间层层上报，成功模拟了引言中的过程。



**4.总结**

**优点：**

（1）将请求的接收者和处理者解耦，客户端无需知道具体处理者，只针对抽象处理者编程，简化了客户端编程过程，降低系统耦合度；

（2）在系统中增加一个新的处理者时，只需要继承抽象处理者，重新实现handleRequest()接口，无需改动原有代码，符合开闭原则；

（3）给对象分配职责时，职责链模式赋予系统更多灵活性。

**缺点：**

（1）请求没有一个明确的接收者，有可能遇到请求无法响应的问题；

（2）比较长的职责链，其处理过程会很长。

（3）建立职责链的工作是在客户端进行，如果建立不当，可能导致循环调用或者调用失败。

**适用环境：**

（1）有多个对象处理同一个请求，具体由谁来处理是在运行时决定，客户端只需发出请求到职责链上，而无需关心具体是谁来处理；

（2）可动态指定一组对象处理请求，客户端可以动态创建职责链来处理请求，还可以改变职责链中各个处理者之间的上下级关系。

**第十八章命令模式**

**鼠标点击某个键，就好像用户在向图片浏览器发送指令，图片浏览器内部接收到指令后开始调用相应的函数，最终结果是播放上一张或下一张图片，即执行或响应了用户发出的命令。客户并不知道发出的命令是什么形式，也不知道图片浏览器内部命令是如何执行的；同样，浏览器内部也不知道是谁发送了命令。命令的发送方和接收方（执行方）没有任何关联。在软件设计模式中，有一种将命令的发送者与执行者解耦的设计模式——命令模式。**

**1.命令模式简介**

命令模式可以将请求（命令）的发送者与接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道请求是如何完成的。下面是比较晦涩难懂的命令模式的定义：

**命令模式：**

**将一个请求封装为一个对象，从而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。**

 命令模式的定义比较复杂，也提到一些术语。这些将在下面的阐述和举例中做进一步说明。

1. **命令模式结构**

命令模式一共有以下几种角色：

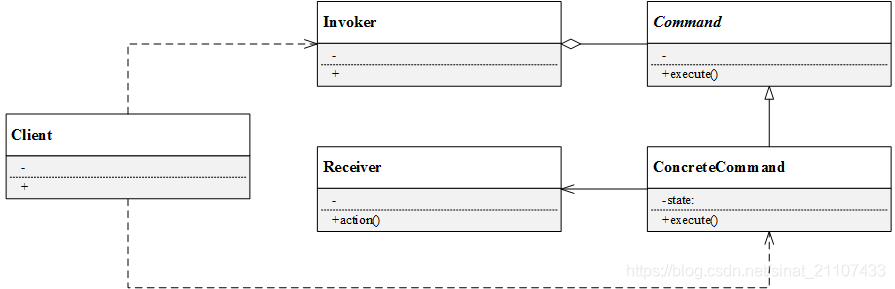
（1）Command（抽象命令类）：是一个抽象类，声明了用于执行命令的接口execute()。

（2）ConcreteCommand（具体命令类）：具体的命令类，实现了执行命令的接口execute()，它对应具体的接收者对象，将接收者（Receiver）的动作action()绑定其中。在execu()方法中将调用接收者的动作action()。（这就是定义中的“将请求封装成一个对象”的体现）

（3）Invoker（调用者）：请求的发送者，通过命令对象来执行请求。一个调用者不需要在设计时确定其接收者，所以调用者通过聚合，与命令类产生关联。具体实现中，可以将一个具体命令对象注入到调用者中，再通过调用具体命令对象的execute()方法，实现简介请求命令执行者（接收者）的操作。

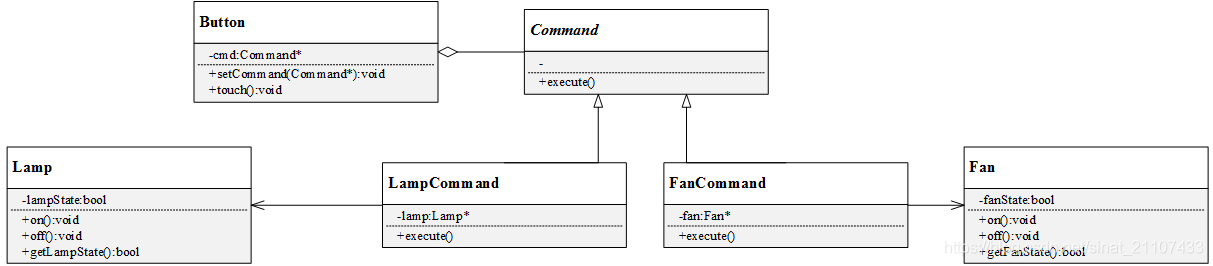
（4）Receiver（接收者）： 实现处理请求的具体操作（action）。

命令模式的UML结构如图，



**3.命令模式代码实例**

房间中的开关（Button）就是命令模式的一个实现，本例使用命令模式来模拟开关功能，可控制的对象包括电灯（Lamp）和风扇（Fan）。用户每次触摸（touch）开关，都可以打开或者关闭电灯或者电扇。



本实例的UML图如上所示。抽象命令类仅声明execute()接口。有两个具体命令类，分别是控制灯的LampCommand和控制风扇的FanCommand类，两个具体类中实现了execute()接口，即执行开关灯/风扇请求。本例中的调用者是按钮Button，每次用户触摸touch()）开关按钮，即是在发送请求。本例具体设计实现过程如下。

**3.1.接收者类：电灯和风扇**

// 接收者：电灯类

class Lamp

{

public :

Lamp(){

this->lampState = false;

}

void on(){

lampState = true;

printf("Lamp is on\n");

}

void off(){

lampState = false;

printf("Lamp is off\n");

}

bool getLampState(){

return lampState;

}

private:

bool lampState;

};

// 接收者：风扇类

class Fan

{

public:

Fan(){

this->fanState = false;

}

void on(){

fanState = true;

printf("Fan is on\n");

}

void off(){

fanState = false;

printf("Fan is off\n");

}

bool getFanState(){

return fanState;

}

private:

bool fanState;

};

**3.2.抽象命令类**

// 抽象命令类 Command

class Command

{

public:

Command(){}

// 声明抽象接口：发送命令

virtual void execute() = 0;

private:

Command \*command;

};

**3.3.具体命令类**

// 具体命令类 LampCommand

class LampCommand :public Command

{

public:

LampCommand(){

printf("开关控制电灯\n");

lamp = new Lamp();

}

// 实现execute()

void execute(){

if (lamp->getLampState()){

lamp->off();

}

else{

lamp->on();

}

}

private:

Lamp \*lamp;

};

// 具体命令类 FanCommand

class FanCommand :public Command

{

public:

FanCommand(){

printf("开关控制风扇\n");

fan = new Fan();

}

// 实现execute()

void execute(){

if (fan->getFanState()){

fan->off();

}

else{

fan->on();

}

}

private:

Fan \*fan;

};

**3.3.调用者：Button**

// 调用者 Button

class Button

{

public:

Button(){}

// 注入具体命令类对象

void setCommand(Command \*cmd){

this->command = cmd;

}

// 发送命令：触摸按钮

void touch(){

printf("触摸开关:");

command->execute();

}

private:

Command \*command;

};

**3.4.客户端代码示例**

#include <iostream>

#include "CommandPattern.h"

int main()

{

// 实例化调用者：按钮

Button \*button = new Button();

Command \*lampCmd, \*fanCmd;

// 按钮控制电灯

lampCmd = new LampCommand();

button->setCommand(lampCmd);

button->touch();

button->touch();

button->touch();

printf("\n\n");

// 按钮控制风扇

fanCmd = new FanCommand();

button->setCommand(fanCmd);

button->touch();

button->touch();

button->touch();

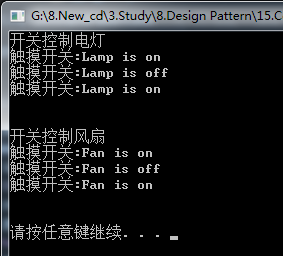
printf("\n\n");

system("pause");

return 0;

}

**3.5.效果**



**可以看到，客户端只需要有一个调用者和抽象命令类，在给调用者注入命令时，再将命令类具体化（这也就是定义中“可用不同的请求对客户进行参数化”的体现）。客户端并不知道命令是如何传递和响应，只需发送命令touch()即可，由此实现命令发送者和接收者的解耦。**

如果系统中增加了新的功能，功能键与新功能对应，只需增加对应的具体命令类，在新的具体命令类中调用新的功能类的action()方法，然后将该具体命令类通过注入的方式加入到调用者，无需修改原有代码，符合开闭原则。

**4.命令队列**

有时候，当请求发送者发送一个请求时，有不止一个请求接收者产生响应（Qt信号槽，一个信号可以连接多个槽），这些请求接收者将逐个执行业务方法，完成对请求的处理，此时可以用命令队列来实现。**比如按钮开关同时控制电灯和风扇，这个例子中，请求发送者是按钮开关，有两个接收者产生响应，分别是电灯和风扇。**

可以参考的命令队列的实现方式是增加一个命令队列类（CommandQueue）来存储多个命令对象，不同命令对象对应不同的命令接收者。调用者也将面对命令队列类编程，增加注入具体命令队列类对象的方法setCommandQueue(CommandQueue \*cmdQueue)。

下面的例子展示了按钮开关请求时，电灯和风扇同时作为请求的接收者。代码如下所示：

#ifdef COMMAND\_QUEUE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 命令队列 \*/

#include <vector>

// 命令队列类

class CommandQueue

{

public:

CommandQueue(){

}

void addCommand(Command \*cmd){

commandQueue.push\_back(cmd);

}

void execute(){

for (int i = 0; i < commandQueue.size(); i++)

{

commandQueue[i]->execute();

}

}

private:

vector<Command\*>commandQueue;

};

// 调用者

class Button2

{

public:

Button2(){}

// 注入具体命令队列类对象

void setCommandQueue(CommandQueue \*cmdQueue){

this->cmdQueue = cmdQueue;

}

// 发送命令：触摸按钮

void touch(){

printf("触摸开关:");

cmdQueue->execute();

}

private:

CommandQueue \*cmdQueue;

};

#endif

 客户端代码如下：

#ifdef COMMAND\_QUEUE

printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

Button2 \*button2 = new Button2();

Command \*lampCmd2, \*fanCmd2;

CommandQueue \*cmdQueue = new CommandQueue();

// 按钮控制电灯

lampCmd2 = new LampCommand();

cmdQueue->addCommand(lampCmd2);

// 按钮控制风扇

fanCmd2 = new FanCommand();

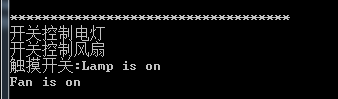
cmdQueue->addCommand(fanCmd2);

button2->setCommandQueue(cmdQueue);

button2->touch();

#endif

效果如下图：



**5.命令模式其他应用**

**5.1.记录请求日志**

将历史请求记录保存在日志里，即请求日志。很多软件系统都提供了日志文件，记录运行过程中的流程。一旦系统发生故障，日志成为了分析问题的关键。日志也可以保存命令队列中的所有命令对象，每执行完一个命令就从日志里删除一个对应的对象。

**5.2.宏命令**

宏命令又叫组合命令，是组合模式和命令模式的结合。宏命令是一个具体命令类，拥有一个命令集合，命令集合中包含了对其他命令对象的引用。宏命令通常不直接与请求者交互，而是通过它的成员来遍历调用接收者的方法。当调用宏命令的execute()方法时，就遍历执行每一个具体命令对象的execute()方法。（类似于前面的命令队列）

**6.总结**

**优点：**

（1）降低系统耦合度，将命令的请求者与接收者分离解耦，请求者和发送者不存在直接关联，各自独立互不影响。

（2）便于扩展：新的命令很容易加入到系统中，且符合开闭原则。

（3）较容易实现命令队列或宏命令。

（4）为请求的撤销和回复操作提供了一种设计实现方案。

**缺点：**

命令模式可能导致系统中有过多的具体命令类，增加了系统中对象的数量。

**适用环境：**

（1）系统需要将请求发送者和接收者解耦，使得发送者和接收者互不影响。

（2）系统需要在不同时间指定请求、将请求排队和执行请求。

（3）系统需要支持命令的撤销和恢复操作。

（4）系统需要将一组操作组合在一起形成宏命令。

**第十九章 读懂老板的暗语，你需要知道解释器模式！**

**1.解释器模式概述**

解释器模式用于描述一个简单的语言解释器，主要应用于使用面向对象语言开发的解释器的设计。当需要开发一个新的语言是，可以使用解释器模式。

**解释器模式：**

**给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。**

**解释器模式需要解决的是，如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构件一个解释器，该解释器通过解释这些句子，来解决该问题。解释器模式描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。**

**2.解释器模式结构**

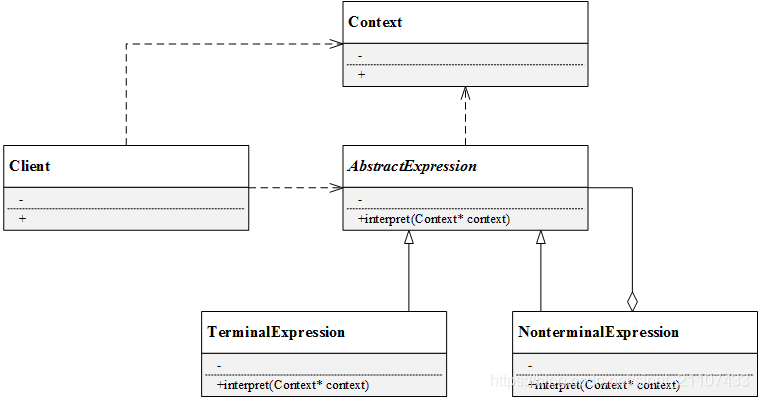
解释器模式的结构由抽象表达式、终结符表达式、非终结符表达式和环境类组成：

（1）AbstractExpression（抽象表达式）：声明了抽象的解释操作interpret()，是所有终结符表达式和非终结符表达式的基类；

（2）TerminalExpression（终结符表达式）：终结符是文法规则的组成元素中最基本的语言单位，不能再分解。终结符表达式实现了与文法规则中终结符相关的解释操作，句子中的每一个终结符都是该类的一个实例。

（3）NonterminalExpression（非终结符表达式）：实现了文法规则中非终结符的解释操作，因为非终结符表达式同样可以包含终结符表达式，所以终结符表达式可以是非终结符表达式的成员。

（4）Context（环境类）：即上下文类，用于存储解释器之外的一些全局信息，通常临时存储需要解释的语句。



解释器模式的UML图如上所示。抽象表达式声明了抽象接口interpret()，终结符表达式和非终结符表达式式具体实现了该接口。其中，终结符表达式的interpret()接口实现了具体的解释操作，**而非终结符表达式中可能包含终结符表达式或者非终结符表达式，所以非终结符表达式的interpret()接口中可能是递归调用每一个组成部分的interpret()方法。**

**3.解释器模式代码实例**

本节Jungle使用解释器模式实现下面一个小功能：

设计一个简单的解释器，使得系统可以解释0和1的或运算和与运算（不考虑或运算和与运算的优先级，即从左往右依次运算），语句表达式和输出结果的几个实例如下表：

表达式及输出结果部分实例表

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 输出结果 | 表达式 | 输出结果 |
| 1 and 1 | 1 | 0 or 0 | 0 |
| 1 or 1 | 1 | 1 and 1 or 0 | 1 |
| 1 or 0 | 1 | 0 or 1 and 0 | 0 |
| 1 and 0 | 0 | 0 or 1 and 1 or 1 | 1 |
| 0 and 0 | 0 | 1 or 0 and 1 and 0 or 0 | 0 |

结合前面叙述的解释器模式的结构和本例，可以划分出以下角色：

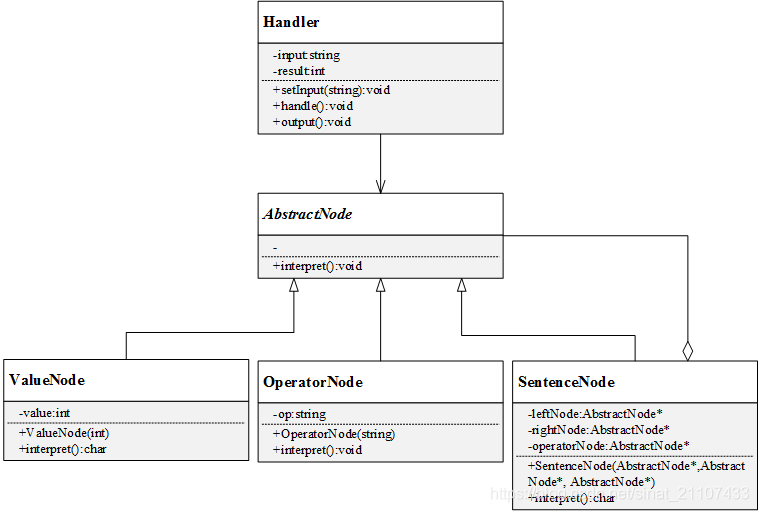
（1）终结符表达式角色——值节点（ValueNode）：0、1，因为它们是表达式的基本组成元素，不可再细分

（2）终结符表达式角色——运算符节点（OperatorNode）：运算符号“and”和“or” ，同样也是表达式的基本组成元素

（3）非终结符表达式角色——句子节点（SentenceNode）：类似于“1 and 1”这样的表达式或者更长的组合表达式

（4）上下文类角色——处理者（Handler）：保存输入的表达式和输出的结果

由此，本例的UML实例图如下：



**3.1.抽象表达式**

// 抽象表达式类

class AbstractNode

{

public:

AbstractNode(){}

// 声明抽象接口

virtual char interpret() = 0;

};

**3.2.终结符表达式角色——值节点**

// 终结符表达式：ValueNode

class ValueNode :public AbstractNode

{

public :

ValueNode(){}

ValueNode(int iValue){

this->value = iValue;

}

// 实现解释操作

char interpret(){

return value;

}

private:

int value;

};

**3.3.终结符表达式角色——运算符节点**

// 终结符表达式：OperationNode

class OperatorNode :public AbstractNode

{

public:

OperatorNode(){}

OperatorNode(string iOp){

this->op = iOp;

}

// 实现解释操作

char interpret(){

if (op == "and"){

return '&';

}

else if (op == "or"){

return '|';

}

return 0;

}

private:

string op;

};

**3.4.非终结符表达式角色——句子节点**

每一个句子节点由“左值节点+运算符节点+右值节点”组成。

// 非终结符表达式：SentenceNode

class SentenceNode :public AbstractNode

{

public:

SentenceNode(){}

SentenceNode(AbstractNode \*iLeftNode,

AbstractNode \*iRightNode, AbstractNode\* iOperatorNode){

this->leftNode = iLeftNode;

this->rightNode = iRightNode;

this->operatorNode = iOperatorNode;

}

char interpret(){

if (operatorNode->interpret() == '&'){

return leftNode->interpret()&rightNode->interpret();

}

else{

return leftNode->interpret()|rightNode->interpret();

}

return 0;

}

private:

AbstractNode \*leftNode;

AbstractNode \*rightNode;

AbstractNode \*operatorNode;

};

**3.5.上下文角色——处理者**

处理者将处理输入的表达式，并解释出表达式最终的结果。

// 处理者

class Handler

{

public:

Handler(){}

void setInput(string iInput){

this->input = iInput;

}

void handle(){

AbstractNode \*left = NULL;

AbstractNode \*right = NULL;

AbstractNode \*op = NULL;

AbstractNode \*sentence = NULL;

string iInput = this->input;

vector<string>inputList;

char\* inputCh = const\_cast<char\*>(iInput.c\_str());

char \*token = strtok(inputCh, " ");

while (token != NULL){

inputList.push\_back(token);

token = strtok(NULL, " ");

}

for (int i = 0; i < inputList.size() - 2; i += 2){

left = new ValueNode(\*(inputList[i].c\_str()));

op = new OperatorNode(inputList[i + 1]);

right = new ValueNode(\*(inputList[i+2].c\_str()));

sentence = new SentenceNode(left, right, op);

inputList[i + 2] = string(1, sentence->interpret());

}

string tmpRes = inputList[inputList.size() - 1];

if (tmpRes == "1"){

result = 1;

}

else if (tmpRes == "0"){

result = 0;

}

else{

result = -1;

}

this->output();

}

void output(){

printf("%s = %d\n", input.c\_str(), result);

}

private:

string input;

char result;

};

**3.6.客户端代码示例和结果**

#include <iostream>

#include "InterpreterPattern.h"

int main()

{

Handler \*handler = new Handler();

string input\_1 = "1 and 1";

string input\_2 = "1 and 0";

string input\_3 = "0 and 1";

string input\_4 = "0 and 0";

string input\_5 = "0 or 0";

string input\_6 = "0 or 1";

string input\_7 = "1 or 0";

string input\_8 = "1 or 1";

string input\_9 = "1 and 0 or 1";

string input\_10 = "0 or 0 and 1";

string input\_11 = "1 or 1 and 1 and 0";

string input\_12 = "0 and 1 and 1 and 1";

string input\_13 = "0 and 1 and 1 and 1 or 1 or 0 and 1";

handler->setInput(input\_1); handler->handle();

handler->setInput(input\_2); handler->handle();

handler->setInput(input\_3); handler->handle();

handler->setInput(input\_4); handler->handle();

handler->setInput(input\_5); handler->handle();

handler->setInput(input\_6); handler->handle();

handler->setInput(input\_7); handler->handle();

handler->setInput(input\_8); handler->handle();

handler->setInput(input\_9); handler->handle();

handler->setInput(input\_10); handler->handle();

handler->setInput(input\_11); handler->handle();

handler->setInput(input\_12); handler->handle();

handler->setInput(input\_13); handler->handle();

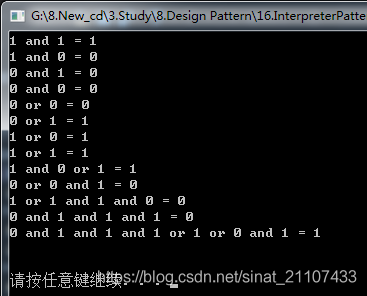
printf("\n\n");

system("pause");

return 0;

}

运行结果如下：



**4.总结**

**优点：**

（1）易于改变和扩展文法，在解释器中使用类表示语言的文法规则，可以通过继承等机制类改变或扩展文法；

（2）每一条文法规则都可以表示为一个类，因此可以方便地实现一个简单的语言；

（3）如果要增加新的解释表达式，只需增加一个新的终结符表达式或非终结符表达式类，无需修改原有代码，符合开闭原则。

**缺点：**

（1）对于复杂文法难以维护。在解释器模式中每一条规则至少需要定义一个类，因此如果一个语言包含太多文法规则，类的个数将会大量增加，导致系统难以管理和维护；

（2）执行效率低，因为解释器模式中有大量循环和递归调用。

**适用环境：**

（1）一些重复出现的问题可以用一种简单的语言进行表达；

（2）一个语言的文法较为简单；

（3）不考虑执行效率的问题时可以使用解释器模式。

**第二十章 迭代器模式，给你更高大上的遍历体验！**

**1.迭代器模式概述**

遍历在日常编码过程中经常使用，通常是需要对一个具有很多对象实例的集合（称为聚合对象）进行访问或获取。比如要取聚合对象的首位元素、判断是否在聚合对象的末尾等。针对聚合对象的遍历，迭代器模式是一种很有效的解决方案，也是一种使用频率很高的设计模式。

**迭代器模式：**

**提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。**

**通过引入迭代器，可以将数据的遍历功能从聚合对象中分离出来，这样一来，聚合对象只需负责存储数据，而迭代器对象负责遍历数据，使得聚合对象的职责更加单一，符合单一职责原则。**

**2.迭代器模式结构**

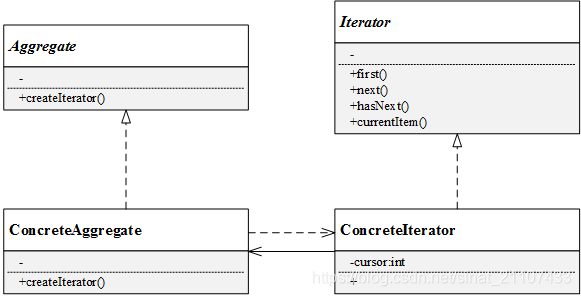
迭代器模式结构中包含聚合和迭代器两个层次的结构。为方便扩展，迭代器模式常常和工厂方法模式结合。迭代器模式的UML图如下。有图可知，迭代器模式有以下几个角色：

（1）Iterator（抽象迭代器）：声明了访问和遍历聚合对象元素的接口，如first()方法用于访问聚合对象中第一个元素，next()方法用于访问下一个元素，hasNext()判断是否还有下一个元素，currentItem()方法用于获取当前元素。

（2）ConcreteIterator（具体迭代器）：实现抽象迭代器声明的方法，通常具体迭代器中会专门用一个变量（称为游标）来记录迭代器在聚合对象中所处的位置。

（3）Aggregate（抽象聚合类）：用于存储和管理元素对象，声明一个创建迭代器的接口，其实是一个抽象迭代器工厂的角色。

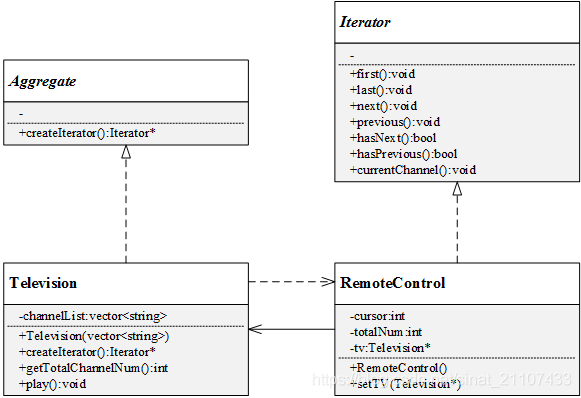
（4）ConcreteAggregate（具体聚合类）：实现了方法createIterator()，该方法返回一个与该具体聚合类对应的具体迭代器ConcreteIterator的实例。



**3.迭代器模式代码实例**

电视机遥控器是迭代器的一个现实应用，通过它可以实现对电视频道集合的遍历操作，电视机可以看成一个存储频道的聚合对象。本例Jungle将采用迭代器模式来模拟遥控器操作电视频道的过程。

很明显，遥控器是一个具体的迭代器，具有上一个频道previous() 、下一个频道next()、当前频道currentChannel()等功能；需要遍历的聚合对象是电视频道的集合，即电视机。本例的UML图如下：



**3.1.抽象聚合类和具体聚合类**

#ifndef \_\_AGGREGATE\_H\_\_

#define \_\_AGGREGATE\_H\_\_

#include <vector>

using namespace std;

// 前向声明，因为两个类互相引用

class Iterator;

class RemoteControl;

// 抽象聚合类 Aggregate

class Aggregate

{

public:

Aggregate(){}

virtual Iterator\* createIterator() = 0;

};

// 具体聚合类 Television

class Television :public Aggregate

{

public:

Television();

Television(vector<string> iChannelList);

// 实现创建迭代器

Iterator\* createIterator();

// 获取总的频道数目

int getTotalChannelNum();

void play(int i);

private:

vector<string> channelList;

};

#endif //\_\_AGGREGATE\_H\_\_

实现：

#include "Iterator.h"

Television::Television(){}

Television::Television(vector<string> iChannelList){

this->channelList = iChannelList;

}

Iterator\* Television::createIterator(){

RemoteControl \*it = new RemoteControl();

it->setTV(this);

return (Iterator\*)it;

}

int Television::getTotalChannelNum(){

return channelList.size();

}

void Television::play(int i){

printf("现在播放：%s……\n", channelList[i].c\_str());

}

**3.2.抽象迭代器**

// 抽象迭代器

class Iterator

{

public:

Iterator(){}

// 声明抽象遍历方法

virtual void first() = 0;

virtual void last() = 0;

virtual void next() = 0;

virtual void previous() = 0;

virtual bool hasNext() = 0;

virtual bool hasPrevious() = 0;

virtual void currentChannel() = 0;

};

**3.3.具体迭代器：RemoteControl**

// 遥控器：具体迭代器

class RemoteControl :public Iterator

{

public:

RemoteControl(){}

void setTV(Television \*iTv){

this->tv = iTv;

cursor = -1;

totalNum = tv->getTotalChannelNum();

}

// 实现各个遍历方法

void first(){

cursor = 0;

}

void last(){

cursor = totalNum - 1;

}

void next(){

cursor++;

}

void previous(){

cursor--;

}

bool hasNext(){

return !(cursor == totalNum);

}

bool hasPrevious(){

return !(cursor == -1);

}

void currentChannel(){

tv->play(cursor);

}

private:

// 游标

int cursor;

// 总的频道数目

int totalNum;

// 电视

Television\* tv;

};

**3.4.客户端代码示例及结果**

#include <iostream>

#include "Iterator.h"

int main()

{

vector<string> channelList = { "新闻频道", "财经频道", "体育频道", "电影频道", "音乐频道", "农业频道", "四川卫视", "成都卫视" };

// 创建电视

Television \*tv = new Television(channelList);

// 创建遥控器

Iterator \*remoteControl = tv->createIterator();

// 顺序遍历

printf("顺序遍历:\n");

remoteControl->first();

// 遍历电视所有频道

while (remoteControl->hasNext()){

remoteControl->currentChannel();

remoteControl->next();

}

printf("\n\n");

// 逆序遍历

printf("逆序遍历:\n");

remoteControl->last();

// 遍历电视所有频道

while (remoteControl->hasPrevious()){

remoteControl->currentChannel();

remoteControl->previous();

}

printf("\n\n");

system("pause");

return 0;

}

结果如下图：



**4.总结**

观察上述代码可发现，迭代器类和聚合类存在相互包含相互引用的关系，因此代码里需要前向声明某个类（具体操作见上，代码资源见<https://github.com/FengJungle/DesignPattern>）。

**优点：**

（1）支持以不同的方式遍历一个聚合对象，在同一个聚合对象上可以定义多个遍历方式。

（2）简化了聚合类，使得聚合类的职责更加单一；

（3）迭代器模式中引入抽象层，易于增加新的迭代器类，便于扩展，符合开闭原则。

**缺点：**

（1）将聚合类中存储对象和管理对象的职责分离，增加新的聚合类时同样需要考虑增加对应的新的迭代器类，类的个数成对增加，不利于系统管理和维护；

（2）设计难度较大，需要充分考虑将来系统的扩展。

**适用环境：**

以下场景可以考虑使用迭代器模式：

（1）访问一个聚合对象而无需暴露它的内部结构；

（2）需要为一个聚合对象提供多种遍历方法。

**第二十一章 中介者模式，说一说贝壳找房**

**1.中介者模式简介**

上述Jungle租房的例子如上图，如果Jungle自己去租房，得和各个房东亲自交互，如果另一个租客贱萌兔也在自己找房，同样也得和很多房东打交道。房东也是一样，得和众多不同的租客联系。如果有中介者了，房东们只需要去中介者那里注册一下，自己的房子在哪儿、什么户型设施、价格多少，就ok了；Jungle和贱萌兔也只需要和一个人打交道，那就是中介。中介的出现使两边都省去了不少事。

软件设计模式中，也有一种类似的解决方案，那就是中介者模式——

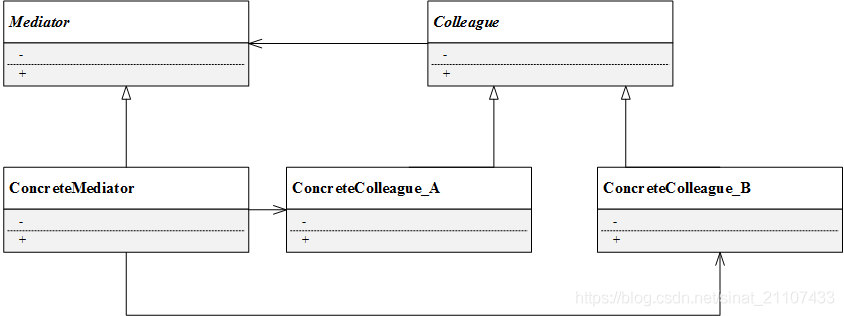
**中介者模式：**

**定义一个对象来封装一系列对象的交互。中介者模式使各个对象之间不需要显示地相互引用，从而使其耦合松散，而且用户可以独立地改变它们之间的交互。**

如果一个系统里各个对象之间存在多对多的相互关系，可以将对象之间的一些交互行为从各个对象中分离出来，集中封装在一个中介者对象中，使其耦合松散，并由中介者统一协调。通过中介者，对象之间的多对多关系就简化了相对更简单的一对多关系。

**2.中介者模式结构**

中介者模式的UML图如下，为了便于扩展，系统引入了抽象中介者。



由图可知，中介者模式主要有以下角色：

（1）Mediator（抽象中介者）：声明一个用于与各个同事对象之间交互的接口，通常声明一个注册方法，用于增加同事对象；

（2）ConcreteMediator（具体中介者）：实现上面的接口，协调各个同事对象来实现协作行为，维持对各个同事对象的引用；

（3）Colleague（抽象同事类）：声明各个同事类公有的接口，同时维持了一个对抽象中介者类的引用；

（4）ConcreteColleague（具体同事类）： 具体实现接口，具体同事类只需与中介者通信，通过中介者完成与其他同事类的通信。

**中介者模式的核心在于引入了中介者类，中介者类承担了两个层次的职责：**

（1）结构上起中转作用：通过中介者的中转，各个同事之间不必再相互显示调用或引用，只需通过中介者实现间接调用的目的；

（2）行为上起协调作用：中介者可以进一步地将同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不必指出中介者具体该如何操作，中介者根据封装在自身内部的协调逻辑对同事的请求进一步处理，将同事成员之间的关系行为进行分离和封装。

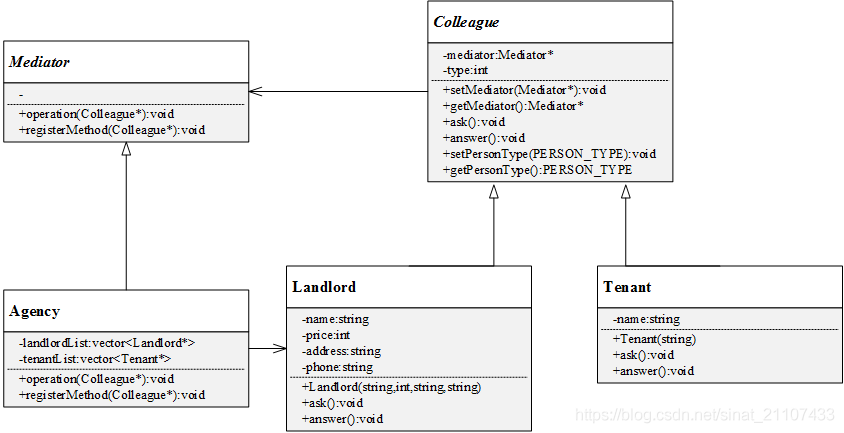
**3.中介者模式代码实例**

本节Jungle将采用中介者模式模拟“租客——房租中介——房东”之间的爱恨情仇！

（本例的代码相对较复杂，具体代码资源请见<https://github.com/FengJungle/DesignPattern>）

Jungle和贱萌兔想要通过房屋中介（Agency）租房，需要去中介处了解房东（Landlord）的信息（姓名，价格，地址和联系方式）；房东们（Landlord）需要在中介处注册自己的房源，同时也可以从中介处了解租客（Tenant）的信息（姓名）。

本例的UML图如下：



**3.0.公共头文件**

为区分房东和租客，Jungle定义了一个枚举类型和对应的setter、getter方法：

#ifndef \_\_COMMON\_H\_\_

#define \_\_COMMON\_H\_\_

// 公共头文件

#include <vector>

using namespace std;

enum PERSON\_TYPE

{

NONE\_PERSON,

LANDLORD,

TENANT

};

#endif //\_\_COMMON\_H\_\_

**3.1.中介者**

**3.1.1.抽象中介者**

// 抽象中介者

class Mediator

{

public:

Mediator(){}

// 声明抽象方法

virtual void operation(Colleague\*) = 0;

// 声明注册方法

virtual void registerMethod(Colleague\*) = 0;

};

**3.1.2.具体中介者Agency**

具体中介者就是真实的中介对象类，他手里有房东的名单（landlordList）和租客名单（tenantList），房东和租客通过registerMethod()在中介处登记注册。同时，房东可以询问中介租客信息，租客也可以向中介询问房东信息。

// 具体中介者

class Agency:public Mediator

{

public:

Agency(){}

void registerMethod(Colleague\* person){

switch (person->getPersonType()){

case LANDLORD:

landlordList.push\_back((Landlord\*)person);

break;

case TENANT:

tenantList.push\_back((Tenant\*)person);

break;

default:

printf("wrong person\n");

}

}

void operation(Colleague\* person){

switch (person->getPersonType()){

case LANDLORD:

for (int i = 0; i < tenantList.size(); i++){

tenantList[i]->answer();

}

break;

case TENANT:

for (int i = 0; i < landlordList.size(); i++){

landlordList[i]->answer();

}

break;

default:

break;

}

}

private:

vector<Landlord\*>landlordList;

vector<Tenant\*>tenantList;

};

**3.2.同事类**

**3.2.1.抽象同事类**

// 前向声明

class Mediator;

class Agency;

// 抽象同事类

class Colleague

{

public:

Colleague(){}

void setMediator(Mediator\* iMediator){

this->mediator = iMediator;

}

Mediator\* getMediator(){

return this->mediator;

}

void setPersonType(PERSON\_TYPE iPersonType){

this->personType = iPersonType;

}

PERSON\_TYPE getPersonType(){

return this->personType;

}

virtual void ask() = 0;

virtual void answer() = 0;

private:

PERSON\_TYPE personType;

Mediator\* mediator;

};

**3.2.2.具体同事类——房东（Landlord）**

声明：

// 具体同事类：房东

class Landlord :public Colleague

{

public:

Landlord();

Landlord(string iName, int iPrice, string iAddress, string iPhoneNum);

void ask();

void answer();

private:

string name;

int price;

string address;

string phoneNumber;

};

实现：

#include "Colleague.h"

#include "Mediator.h"

Landlord::Landlord(){

name = "none";

price = 0;

address = "none";

phoneNumber = "none";

setPersonType(NONE\_PERSON);

}

Landlord::Landlord(string iName, int iPrice,

string iAddress, string iPhoneNum){

name = iName;

price = iPrice;

address = iAddress;

phoneNumber = iPhoneNum;

setPersonType(LANDLORD);

}

void Landlord::answer(){

printf("房东姓名：%s, 房租：%d, 地址：%s, 联系电话：%s\n",

name.c\_str(), price, address.c\_str(), phoneNumber.c\_str());

}

void Landlord::ask(){

printf("房东%s查看租客信息：\n",name.c\_str());

(this->getMediator())->operation(this);

}

**3.2.3.具体同事类——租客（Tenant）**

声明：

// 具体同事类：租客

class Tenant :public Colleague

{

public:

Tenant();

Tenant(string name);

void ask();

void answer();

private:

string name;

};

实现：

#include "Colleague.h"

#include "Mediator.h"

Tenant::Tenant(){

name = "none";

setPersonType(NONE\_PERSON);

}

Tenant::Tenant(string iName){

name = iName;

setPersonType(TENANT);

}

void Tenant::ask(){

printf("租客%s询问房东信息\n", name.c\_str());

(this->getMediator())->operation(this);

}

void Tenant::answer(){

printf("租客姓名：%s\n", name.c\_str());

}

**3.3.客户端代码示例及效果**

#include <iostream>

#include "Mediator.h"

#include "Colleague.h"

int main()

{

// 创建租房中介

Agency \*mediator = new Agency();

// 创建3位房东

Landlord \*fangdong1 = new Landlord("刘备", 1350, "成都市双流区", "1351025");

Landlord \*fangdong2 = new Landlord("关羽", 1500, "成都市武侯区", "1378390");

Landlord \*fangdong3 = new Landlord("张飞", 1000, "成都市龙泉驿", "1881166");

fangdong1->setMediator(mediator);

fangdong2->setMediator(mediator);

fangdong3->setMediator(mediator);

// 房东在中介处登记注册房源信息

mediator->registerMethod(fangdong1);

mediator->registerMethod(fangdong2);

mediator->registerMethod(fangdong3);

// 创建两位租客Jungle和贱萌兔

Tenant \*jungle = new Tenant("Jungle");

Tenant \*jianmengtu = new Tenant("贱萌兔");

jungle->setMediator(mediator);

jianmengtu->setMediator(mediator);

// Jungle和贱萌兔在中介处登记求租信息

mediator->registerMethod(jungle);

mediator->registerMethod(jianmengtu);

jungle->ask();

printf("\n\n");

fangdong1->ask();

printf("\n\n");

system("pause");

return 0;

}

运行结果如下：



**4.总结**

**优点：**

（1）简化了对象之间的交互，通过中介者，对象之间的多对多关系就简化了相对更简单的一对多关系；

（2）可将各个同事对象解耦，利于各个同事之间的松散耦合，可独立地改变和复用每一个同事对象，增加新的中介者和同事都比较方便，符合开闭原则；

（3）可减少子类生成，将原本分布于多个对象之间的行为封装在一起，只需生成新的具体中介者类就可以改变这些行为。

**缺点：**

具体中介者类中包含了大量与同事之间交互的细节和逻辑，可能使得中介者类很复杂以至于难以管理维护。

**适用环境：**

系统中的对象之间存在复杂的交互关系，使得系统内逻辑错综复杂，难以管理；

一个对象引用了其他很多对象，并直接和这些对象交互，导致该对象难以复用。

**第二十二章 我用备忘录模式设计了简易的版本控制系统**

**1.备忘录模式简介**

类似于上述引言的例子，在软件系统的操作过程中，难免会出现一些不当的操作，使得系统状态出现某些故障。如果能够有一种机制——能够保存系统每个阶段的状态，当用户操作失误的时候，可以撤销不当的操作，回到历史某个阶段——那么软件系统将更加灵活和人性化。

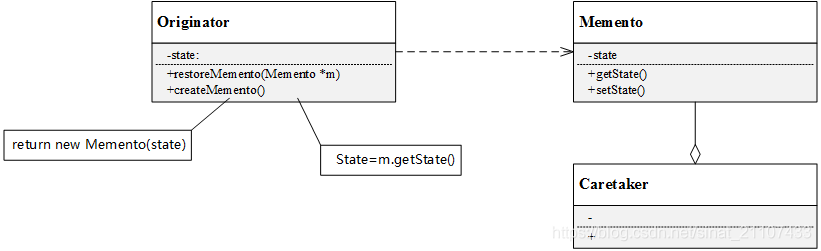
有没有这样的一种解决方案呢？有！那就是备忘录模式。备忘录模式提供了一种状态恢复的机制，用户可以方便地回到指定的某个历史状态。很多软件的撤销操作，就使用了备忘录模式。

**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。**

**2.备忘录模式结构**

备忘录模式的UML图如下所示：



备忘录模式主要有以下角色：

（1）Originator（原发器）：通过创建一个备忘录类存储当前的内部状态，也可以使用备忘录来恢复其内部状态，通常是将系统中需要保存内部状态的类设计为原发器；

（2）Memento（备忘录）：用于存储原发器的内部状态。备忘录的设计可以参考原发器的设计，根据需要确定备忘录类中的属性；除了原发器类对象，不允许其他对象修改备忘录。

（3）Caretaker（负责人）：负责保存备忘录，可以存储一个或多个备忘录对象，但是负责人只负责保存对象，不能修改对象，也不必知道对象的实现细节。（看好了，负责人可以存储多个备忘录对象，想一想这有什么用？是不是可以保存多个历史状态？实现多步撤销操作了）

备忘录模式的关键是备忘录类和负责人类的设计，以下是上述三个角色的典型实现：

#ifndef \_\_DEMO\_H\_\_

#define \_\_DEMO\_H\_\_

// 前向声明

class Memento;

// 原发器 典型实现

class Originator

{

public:

Originator(){

state = "";

}

Originator(String iState){

state = iState;

}

// 创建备忘录对象

Memento\* createMemento(){

return new Memento(this);

}

// 利用备忘录对象恢复原发器状态

void restoreMemento(Memento\* m){

state = m->getState();

}

void setState(string iState){

state = iState;

}

string getState(){

return state;

}

private:

string state;

};

// 备忘录 典型实现（仿照原生器的设计）

class Memento

{

public:

Memento(){

state = "";

}

Memento(Originator\* o){

state = o->getState();

}

void setState(String iState){

state = iState;

}

string getState(){

return state;

}

private:

String state;

};

// 负责人 典型实现

class Caretaker

{

public:

Caretaker(){}

Memento\* getMemento(){

return memento;

}

void setMemento(Memento \*m){

memento = m;

}

private:

Memento\* memento;

};

// 客户端 示例代码

int main()

{

// 创建原发器对象

Originator o = new Originator("状态1");

// 创建负责人对象

Caretaker \*c = new Caretaker();

c->setMemento(o->createMemento());

o->setState("状态2");

// 从负责人对象中取出备忘录对象，实现撤销

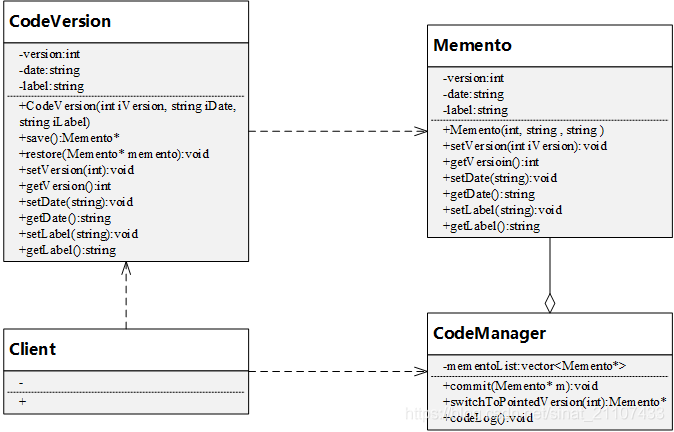
o->restoreMemento(c->getMemento());

return 0;

}

#endif

**3.备忘录模式代码实例**



Jungle正在为代码版本管理苦恼，有时候为了尝试某个功能就去修改代码，导致原有的健壮的代码被破坏。所以Jungle希望能够设计一个代码保存和版本回退功能的demo，方便代码的管理。

本实例中，原生器为CodeVersion，具有版本号version、提交日期date和标签label三个状态需要备忘录Memento保存；管理者是CodeManager，具有提交代码commit（即保存一个版本）、回退到指定版本switchToPointedVersion（即撤销操作）和查看提交历史codeLog的功能。该实例的UML图如下图，具体代码如下。（完整代码资源见<https://github.com/FengJungle/DesignPattern>）

**3.1.备忘录Memento**

#ifndef \_\_MEMENTO\_H\_\_

#define \_\_MEMENTO\_H\_\_

class Memento

{

public:

Memento(){}

Memento(int iVersion, string iDate, string iLabel){

version = iVersion;

date = iDate;

label = iLabel;

}

void setVersion(int iVersion){

version = iVersion;

}

int getVersion(){

return version;

}

void setLabel(string iLabel){

label = iLabel;

}

string getLabel(){

return label;

}

void setDate(string iDate){

date = iDate;

}

string getDate(){

return date;

}

private:

int version;

string date;

string label;

};

#endif

**3.2.原生器CodeVersion**

#ifndef \_\_CODEVERSION\_H\_\_

#define \_\_CODEVERSION\_H\_\_

#include <iostream>

using namespace std;

#include "Memento.h"

// 原生器：CodeVersion

class CodeVersion

{

public:

CodeVersion(){

version = 0;

date = "1900-01-01";

label = "none";

}

CodeVersion(int iVersion, string iDate, string iLabel){

version = iVersion;

date = iDate;

label = iLabel;

}

// 保存代码

Memento\* save(){

return new Memento(this->version, this->date, this->label);

}

// 回退版本

void restore(Memento\* memento){

setVersion(memento->getVersion());

setDate(memento->getDate());

setLabel(memento->getLabel());

}

void setVersion(int iVersion){

version = iVersion;

}

int getVersion(){

return version;

}

void setLabel(string iLabel){

label = iLabel;

}

string getLabel(){

return label;

}

void setDate(string iDate){

date = iDate;

}

string getDate(){

return date;

}

private:

// 代码版本

int version;

// 代码提交日期

string date;

// 代码标签

string label;

};

#endif

**3.3.管理者CodeManager**

#ifndef \_\_CODEMANAGER\_H\_\_

#define \_\_CODEMANAGER\_H\_\_

#include "Memento.h"

#include <vector>

using namespace std;

// 管理者

class CodeManager

{

public:

CodeManager(){}

void commit(Memento\* m){

printf("提交：版本-%d, 日期-%s, 标签-%s\n", m->getVersion(), m->getDate().c\_str(), m->getLabel().c\_str());

mementoList.push\_back(m);

}

// 切换到指定的版本，即回退到指定版本

Memento\* switchToPointedVersion(int index){

mementoList.erase(mementoList.begin() + mementoList.size() - index, mementoList.end());

return mementoList[mementoList.size() - 1];

}

// 打印历史版本

void codeLog(){

for (int i = 0; i < mementoList.size(); i++){

printf("[%d]：版本-%d, 日期-%s, 标签-%s\n", i, mementoList[i]->getVersion(),

mementoList[i]->getDate().c\_str(), mementoList[i]->getLabel().c\_str());

}

}

private:

vector<Memento\*> mementoList;

};

#endif

**3.4.客户端代码示例及效果**

#include "Originator.h"

#include "Memento.h"

#include "CodeManager.h"

int main()

{

CodeManager \*Jungle = new CodeManager();

CodeVersion\* codeVer = new CodeVersion(1001, "2019-11-03", "Initial version");

// 提交初始版本

printf("提交初始版本:\n");

Jungle->commit(codeVer->save());

// 修改一个版本，增加了日志功能

printf("\n提交一个版本，增加了日志功能:\n");

codeVer->setVersion(1002);

codeVer->setDate("2019-11-04");

codeVer->setLabel("Add log funciton");

Jungle->commit(codeVer->save());

// 修改一个版本，增加了Qt图片浏览器

printf("\n提交一个版本，增加了Qt图片浏览器:\n");

codeVer->setVersion(1003);

codeVer->setDate("2019-11-05");

codeVer->setLabel("Add Qt Image Browser");

Jungle->commit(codeVer->save());

// 查看提交历史

printf("\n查看提交历史\n");

Jungle->codeLog();

// 回退到上一个版本

printf("\n回退到上一个版本\n");

codeVer->restore(Jungle->switchToPointedVersion(1));

// 查看提交历史

printf("\n查看提交历史\n");

Jungle->codeLog();

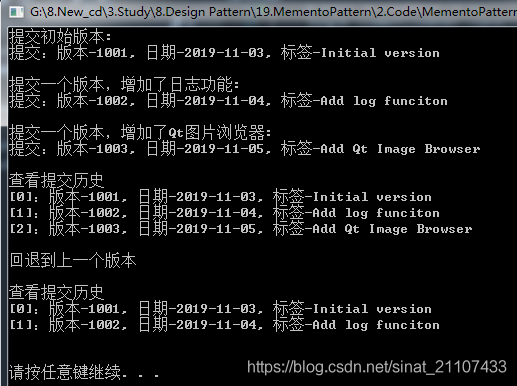
printf("\n\n");

system("pause");

return 0;

}

代码运行结果如下：



 这是不是像一个超级简易版本的代码版本控制系统？？？哈哈哈！

**4.总结**

**优点：**

（1）实现状态恢复、撤销操作的功能，用户可以恢复到指定的历史状态，让软件系统更加人性化；

（2）备忘录封装了信息，除了原生器以外，其他对象访问不了备忘录的代码；

**缺点：**

资源消耗大。如果需要保存原生器对象的多个历史状态，那么将创建多个备忘录对象；或者如果原生器对象的很多状态都需要保存，也将消耗大量存储资源。

**适用环境：**

（1）保存一个对象的历史状态，系统需要设计回退或者撤销功能；

（2）备忘录类可以封装一个对象的历史状态，避免对象的历史状态被外界修改。

**第二十三章 “牵一发而动全身”——我用观察者模式简单模拟吃鸡**

**1.观察者模式简介**

软件系统中的对象并不是孤立存在的，一个对象行为的改变可能会引起其他所关联的对象的状态或行为也发生改变，即“牵一发而动全身”。观察者模式建立了一种一对多的联动，一个对象改变时将自动通知其他对象，其他对象将作出反应。观察者模式中，发生改变的对象称为“观察目标”，被通知的对象称为“观察者”。一个观察目标可以有很多个观察者。

**观察者模式定义如下：**

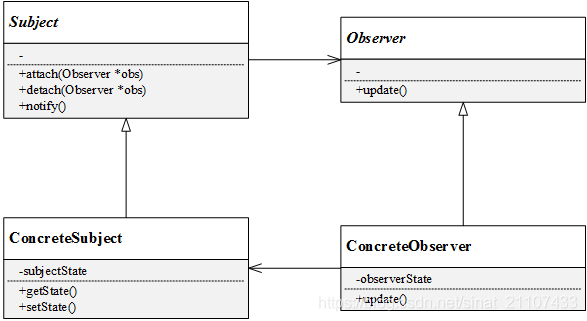
**观察者模式：**

**定义对象之间的一种一对多的依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象都得到通知并被自动更新。**

观察者模式又被称为发布-订阅模式（Publish-Subscribe）、模型-视图模式（Model-View）、源-监听器模式（Source-Listener）、从属者模式（Dependents）。

**2.观察者模式结构**

观察者模式由观察者和观察目标组成，为便于扩展，两个角色都设计了抽象层。观察者模式的UML图如下：



（1）Subject（目标）：是被观察的对象，目标中定义了一个观察者的集合，即一个目标可能会有多个观察者，通过attach()和detach()方法来增删观察者对象。目标声明了通知方法notify()，用于在自身状态发生改变时通知观察者。

（2）ConcreteSubject（具体目标）：具体目标实现了通知方法notify()，同时具体目标有记录自身状态的属性和成员方法；

（3）Observer（观察者）：观察者将对接收到的目标发生改变的通知做出自身的反应，抽象层声明了更新方法update()；

（4）ConcreteObserver（具体观察者）： 实现了更新方法update()，具体观察者中维护了一个具体目标对象的引用（指针），用于存储目标的状态。

下述是观察者模式的典型实现：

#ifndef \_\_DEMO\_H\_\_

#define \_\_DEMO\_H\_\_

// 抽象观察者

class Observer

{

public:

// 声明响应更新方法

virtual void update() = 0;

};

// 具体观察者

class ConcreteObserver:public Observer

{

public:

// 实现响应更新方法

void update(){

// 具体操作

}

};

// 抽象目标

class Subject

{

public:

// 添加观察者

void attach(Observer\* obs){

obsList.push\_back(obs);

}

// 移除观察者

void detach(Observer\* obs){

obsList.remove(obs);

}

// 声明通知方法

virtual void notify() = 0;

protected:

// 观察者列表

list<Observer\*>obsList;

};

// 具体目标

class ConcreteSubject :public Subject

{

public:

// 实现通知方法

void notify(){

// 具体操作

// 遍历通知观察者对象

for (int i = 0; i < obsList.size(); i++){

obsList[i]->update();

}

}

};

// 客户端代码示例

int main()

{

Subject \*sub = new ConcreteSubject();

Observer \*obs = new ConcreteObserver();

sub->attach(obs);

sub->notify();

return 0;

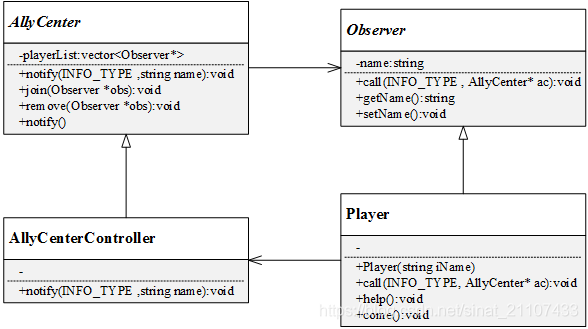
}

#endif //\_\_DEMO\_H\_\_

**3.观察者模式代码实例**

玩过和平精英这款游戏吗？四人组队绝地求生，当一个队友发现物资时，可以发消息“我这里有物资”，其余三个队友听到后可以去取物资；当一个队友遇到危险时，也可以发消息“救救我”，其余三个队友得到消息后便立马赶去营救。本例Jungle将用观察者模式来模拟这个过程。

本例的UML图如下：



本例中，抽象观察者是Observer，声明了发现物资或者需要求救时的呼叫的方法call()，具体观察者是Player，即玩家，Player实现了呼叫call()方法，并且还定义了取物资come()和支援队友help()的方法。本例定义了AllyCenter作为抽象目标，它维护了一个玩家列表playerList，并且定义了加入战队和剔除玩家的方法。具体目标是联盟中心控制器AllyCenterController，它实现了通知notify()方法，该方法将队友call的消息传达给玩家列表里的其余队友，并作出相应的响应。源代码见<https://github.com/FengJungle/DesignPattern>。

**3.0.公共头文件**

通过一个枚举类型来定义两种消息类型，即发现物资和求助

#ifndef \_\_COMMON\_H\_\_

#define \_\_COMMON\_H\_\_

enum INFO\_TYPE{

NONE,

RESOURCE,

HELP

};

#endif //\_\_COMMON\_H\_\_

**3.1.观察者**

3.1.1.抽象观察者Observer

// 抽象观察者 Observer

class Observer

{

public:

Observer(){}

// 声明抽象方法

virtual void call(INFO\_TYPE infoType, AllyCenter\* ac) = 0;

string getName(){

return name;

}

void setName(string iName){

this->name = iName;

}

private:

string name;

};

**3.1.2.具体观察者Player**

// 具体观察者

class Player :public Observer

{

public:

Player(){

setName("none");

}

Player(string iName){

setName(iName);

}

// 实现

void call(INFO\_TYPE infoType, AllyCenter\* ac){

switch (infoType){

case RESOURCE:

printf("%s :我这里有物资\n", getName().c\_str());

break;

case HELP:

printf("%s :救救我\n", getName().c\_str());

break;

default:

printf("Nothing\n");

}

ac->notify(infoType, getName());

}

// 实现具体方法

void help(){

printf("%s:坚持住，我来救你！\n", getName().c\_str());

}

void come(){

printf("%s:好的，我来取物资\n", getName().c\_str());

}

};

**3.2.目标类**

**3.2.1.抽象目标AllyCenter**

声明

// 抽象目标：联盟中心

class AllyCenter

{

public:

AllyCenter();

// 声明通知方法

virtual void notify(INFO\_TYPE infoType, std::string name) = 0;

// 加入玩家

void join(Observer\* player);

// 移除玩家

void remove(Observer\* player);

protected:

// 玩家列表

std::vector<Observer\*>playerList;

};

实现

#include "AllyCenter.h"

#include "Observer.h"

AllyCenter::AllyCenter(){

printf("大吉大利，今晚吃鸡!\n");

}

// 加入玩家

void AllyCenter::join(Observer\* player){

if (playerList.size() == 4){

printf("玩家已满\n");

return;

}

printf("玩家 %s 加入\n", player->getName().c\_str());

playerList.push\_back(player);

if (playerList.size() == 4){

printf("组队成功，不要怂，一起上！\n");

}

}

// 移除玩家

void AllyCenter::remove(Observer\* player){

printf("玩家 %s 退出\n", player->getName().c\_str());

//playerList.remove(player);

}

**3.2.2.具体目标AllyCenterController**

声明：

// 具体目标

class AllyCenterController :public AllyCenter

{

public:

AllyCenterController();

// 实现通知方法

void notify(INFO\_TYPE infoType, std::string name);

};

实现：

AllyCenterController::AllyCenterController(){}

// 实现通知方法

void AllyCenterController::notify(INFO\_TYPE infoType, std::string name){

switch (infoType){

case RESOURCE:

for each (Observer\* obs in playerList){

if (obs->getName() != name){

((Player\*)obs)->come();

}

}

break;

case HELP:

for each (Observer\* obs in playerList){

if (obs->getName() != name){

((Player\*)obs)->help();

}

}

break;

default:

printf("Nothing\n");

}

}

**3.3.客户端代码示例及效果**

#include "Observer.h"

#include "AllyCenter.h"

int main()

{

// 创建一个战队

AllyCenterController\* controller = new AllyCenterController();

// 创建4个玩家，并加入战队

Player\* Jungle = new Player("Jungle");

Player\* Single = new Player("Single");

Player\* Jianmengtu = new Player("贱萌兔");

Player\* SillyDog = new Player("傻子狗");

controller->join(Jungle);

controller->join(Single);

controller->join(Jianmengtu);

controller->join(SillyDog);

printf("\n\n");

// Jungle发现物资，呼叫队友

Jungle->call(RESOURCE, controller);

printf("\n\n");

// 傻子狗遇到危险，求救队友

SillyDog->call(HELP, controller);

printf("\n\n");

system("pause");

return 0;

}

 上述代码运行结果如下图：



**4.观察者模式的应用**

观察者模式是一种使用频率非常高的设计模式，几乎无处不在。凡是涉及一对一、一对多的对象交互场景，都可以使用观察者会模式。比如购物车，浏览商品时，往购物车里添加一件商品，会引起UI多方面的变化（购物车里商品数量、对应商铺的显示、价格的显示等）；各种编程语言的GUI事件处理的实现；所有的浏览器事件（mouseover，keypress等）都是使用观察者模式的例子。

**5.总结**

**优点：**

（1）观察者模式实现了稳定的消息更新和传递的机制，通过引入抽象层可以扩展不同的具体观察者角色；

（2）支持广播通信，所有已注册的观察者（添加到目标列表中的对象）都会得到消息更新的通知，简化了一对多设计的难度；

（3）符合开闭原则，增加新的观察者无需修改已有代码，在具体观察者与观察目标之间不存在关联关系的情况下增加新的观察目标也很方便。

**缺点：**

（1）代码中观察者和观察目标相互引用，存在循环依赖，观察目标会触发二者循环调用，有引起系统崩溃的风险；

（2）如果一个观察目标对象有很多直接和简介观察者，将所有的观察者都通知到会耗费大量时间。

**适用环境：**

（1）一个对象的改变会引起其他对象的联动改变，但并不知道是哪些对象会产生改变以及产生什么样的改变；

（2）如果需要设计一个链式触发的系统，可是使用观察者模式；

（3）广播通信、消息更新通知等场景。

**第二十四章 状态模式——从斗地主开始说起**

**1.状态模式简介**

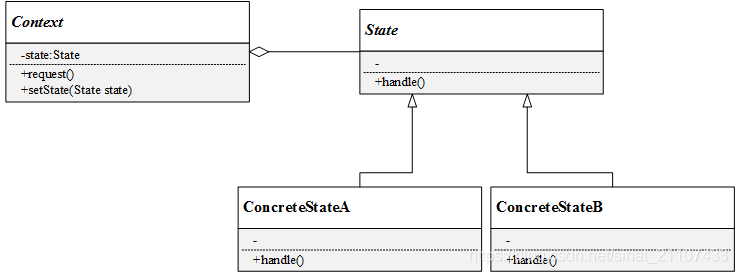
状态模式将一个对象的状态从对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化。对于客户端而言，无需关心对象转态的转换以及对象所处的当前状态，无论处于何种状态的对象，客户端都可以一致处理。

**状态模式：**

**允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。**

**2.状态模式结构**

状态模式的UML图如下。



状态模式引入了抽象层，具有抽象状态类和具体状态类，还包括一个上下文境类:

（1）Context（上下文类）：是拥有多种状态的对象。上下文类的状态存在多样性，并且在不同的状态下，对象表现出不同的行为。在上下文类中，维护了一个抽象状态类的实例。

（2）State（抽象状态类）：声明了一个接口，用于封装与在上下文类中的一个特定状态相关的行为，在子类中实现在各种不同状态对应的方法。不同的子类可能存在不同的实现方法，相同的方法可以写在抽象状态类中。

（3）ConcreteState（具体状态类）：实现具体状态下的方法，每一个具体状态类对应一个具体的状态。

值得注意的是，上下文中维护了一个状态类的指针或者引用，可以由上下文类来觉得具体实例化为哪一个具体的状态对象，也可以由具体的状态类来决定转换为哪一个实例，所以，上下文类和状态类之间存在依赖甚至相互引用的关系：

// 1.由环境类来决定实例化为哪一个具体状态类对象

class Context

{

public:

void convertState(){

if (condition1){

this->state = new ConcreteStateA();

}

else if (condition2){

this->state = new ConcreteStateB();

}

else{

// do something

}

}

private:

// 引用状态对象

State \*state;

};

// 2.有具体状态类来决定转换成哪一个具体状态类对象

class ConcreteState :public State

{

public:

void convertState(Context\* ctx){

if (condition1){

ctx->setState(new ConcreteStateA());

}

else if (condition2){

ctx->setState(new ConcreteStateB());

}

else{

// do something

}

}

};

下面是状态模式的典型用法：

#ifndef \_\_DEMO\_H\_\_

#define \_\_DEMO\_H\_\_

// 抽象状态类

class State

{

public:

// 声明抽象方法

virtual void handle() = 0;

};

// 具体状态类

class ConcreteState :public State

{

public:

// 实现

void handle(){

// ……

}

};

// 上下文类

class Context

{

public:

// set方法设置状态对象

void setState(State\* iState){

this->state = iState;

}

// 对外封装的方法

void request(){

// do something

state->handle();

}

private:

// 引用状态对象

State \*state;

};

#endif //\_\_DEMO\_H\_\_

**3.状态模式代码实例**

接下来Jungle用一个实例来应用状态模式。

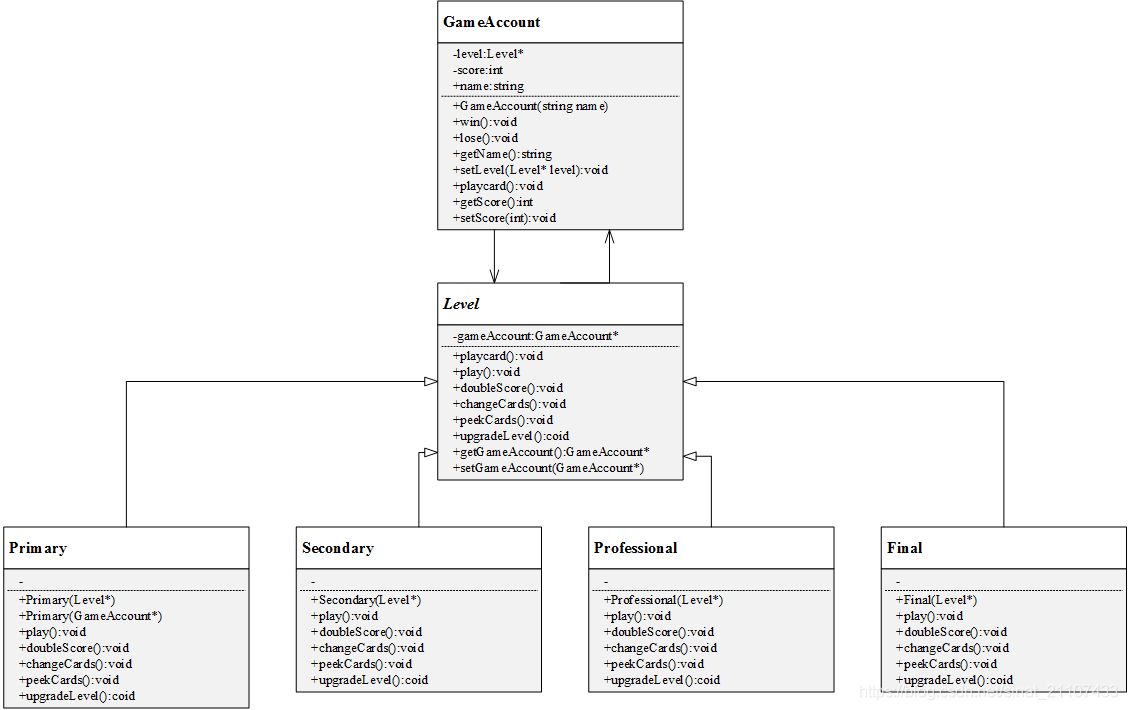
在某纸牌游戏中，游戏人物分为入门级（primary）、熟练级（Secondary）、高手级（Professional）和骨灰级（Final）四种级别，由人物的积分来划分角色等级，游戏胜利将增加积分，失败将扣除积分。入门级有最基本的游戏功能play()，熟练级增加了游戏胜利积分加倍功能doubleScore()，高手级在熟练级的基础上增加了换牌功能changeCards()，骨灰级在高手级的基础上再增加了偷看他人纸牌的功能peekCards()。

积分规则如下：

基础分：100，游戏胜利+50分，游戏失败+30分；

入门级：0~150；熟练级150~200；高手级：200~250；骨灰级：250以上





本例设计游戏账户GameAccount为上下文类，维护了一个级别类（Level）的对象实例。GameAccount中定义了一个代表积分的score整型和统一封装的方法playcard()，在该方法中再调用具体级别的各个技能方法。采用随机数的方式来随机判定牌局的输赢，以增减积分。

级别类Level为抽象类，声明了play()、doubleScore()、changeCards()、seekCards()的抽象方法，在四个具体级别类Primary、Secondary、Professional和Final类中具体实现了该方法，具体来说是根据该级别是否有权利使用该技能来打印一行话。upgradeLevel()方法用于判断每局牌结束后该游戏账户的积分是否可以升级或者降级，通过setLevel()方法改变当前账户的游戏级别。

该实例的UML图如上：

**3.1.上下文类：游戏账户类**

//头文件

#ifndef \_\_GAMEACCOUNT\_H\_\_

#define \_\_GAMEACCOUNT\_H\_\_

using namespace std;

#include <iostream>

// 前向声明

class Level;

class GameAccount

{

public:

GameAccount();

GameAccount(string iName);

string getName();

void win();

void lose();

void playCard();

void setLevel(Level\*);

int getScore();

void setScore(int);

private:

Level\* level;

int score;

string name;

};

#endif

//源文件

#include "GameAccount.h"

#include "Level.h"

#include <Windows.h>

#include <time.h>

#define random(x) (rand()%x)

GameAccount::GameAccount(){

printf("创立游戏角色，积分：100，级别：PRIMARY\n");

score = 100;

name = "none";

setLevel(new Primary(this));

}

GameAccount::GameAccount(string iName){

printf("创立游戏角色，积分：100，级别：PRIMARY\n");

score = 100;

name = iName;

setLevel(new Primary(this));

}

void GameAccount::setLevel(Level\* iLevel){

this->level = iLevel;

}

string GameAccount::getName(){

return name;

}

void GameAccount::playCard(){

this->level->playCard();

Sleep(100);

srand((int)time(0));

int res = random(2);

if (res % 2 == 0){

this->win();

}

else{

this->lose();

}

this->level->upgradeLevel();

}

void GameAccount::win(){

if (this->getScore() < 200){

setScore(getScore() + 50);

}

else{

setScore(getScore() + 100);

}

printf("\n\t胜利，最新积分为 %d\n", score);

}

void GameAccount::lose(){

setScore(getScore() + 30);

printf("\n\t输牌，最新积分为 %d\n", score);

}

int GameAccount::getScore(){

return this->score;

}

void GameAccount::setScore(int iScore){

this->score = iScore;

}

**3.2.状态类**

**3.2.1.抽象状态类：Level**

头文件：

#include "GameAccount.h"

class Level

{

public :

Level();

// 声明方法

void playCard();

void play();

virtual void doubleScore() = 0;

virtual void changeCards() = 0;

virtual void peekCards() = 0;

// 升级

virtual void upgradeLevel() = 0;

GameAccount\* getGameAccount();

void setGameAccount(GameAccount\* iGameAccount);

private:

GameAccount\* gameAccount;

};

源文件：

Level::Level(){}

void Level::playCard(){

this->play();

this->doubleScore();

this->changeCards();

this->peekCards();

}

void Level::play(){

printf("\t使用基本技能,");

}

void Level::setGameAccount(GameAccount\* iGameAccount){

this->gameAccount = iGameAccount;

}

GameAccount\* Level::getGameAccount(){

return gameAccount;

}

**3.2.2.具体状态类：Primary**

头文件：

class Primary :public Level

{

public:

Primary();

Primary(Level\* level);

Primary(GameAccount\* ga);

void doubleScore();

void changeCards();

void peekCards();

// 升级

void upgradeLevel();

};

源文件：

Primary::Primary(){}

Primary::Primary(GameAccount\* iGameAccount){

this->setGameAccount(iGameAccount);

}

Primary::Primary(Level\* level){

getGameAccount()->setLevel(level);

}

void Primary::doubleScore(){

return;

}

void Primary::changeCards(){

return;

}

void Primary::peekCards(){

return;

}

void Primary::upgradeLevel(){

if (this->getGameAccount()->getScore() > 150){

this->getGameAccount()->setLevel(new Secondary(this));

printf("\t升级！ 级别：SECONDARY\n\n");

}

else{

printf("\n");

}

}

**3.2.3.具体状态类：Secondary**

头文件：

class Secondary :public Level

{

public:

Secondary();

Secondary(Level\* level);

void doubleScore();

void changeCards();

void peekCards();

// 升级

void upgradeLevel();

};

源文件：

Secondary::Secondary(){

}

Secondary::Secondary(Level\* level){

this->setGameAccount(level->getGameAccount());

getGameAccount()->setLevel(level);

}

void Secondary::doubleScore(){

printf("使用胜利双倍积分技能");

}

void Secondary::changeCards(){

return;

}

void Secondary::peekCards(){

return;

}

void Secondary::upgradeLevel(){

if (this->getGameAccount()->getScore() < 150){

this->getGameAccount()->setLevel(new Primary(this));

printf("\t降级！ 级别：PRIMARY\n\n");

}

else if (this->getGameAccount()->getScore() > 200){

this->getGameAccount()->setLevel(new Professional(this));

printf("\t升级！ 级别：PROFESSIONAL\n\n");

}

}

**3.2.4.具体状态类：Professional**

头文件：

class Professional :public Level

{

public:

Professional();

Professional(Level\* level);

void doubleScore();

void changeCards();

void peekCards();

// 升级

void upgradeLevel();

};

源文件：

Professional::Professional(){

}

Professional::Professional(Level\* level){

this->setGameAccount(level->getGameAccount());

getGameAccount()->setLevel(level);

}

void Professional::doubleScore(){

printf("使用胜利双倍积分技能,");

}

void Professional::changeCards(){

printf("使用换牌技能");

}

void Professional::peekCards(){

return;

}

void Professional::upgradeLevel(){

if (this->getGameAccount()->getScore() < 200){

this->getGameAccount()->setLevel(new Secondary(this));

printf("\t降级！ 级别：SECONDARY\n\n");

}

else if (this->getGameAccount()->getScore() > 250){

this->getGameAccount()->setLevel(new Final(this));

printf("\t升级！ 级别：FINAL\n\n");

}

}

**3.2.5.具体状态类：Final**

头文件：

class Final :public Level

{

public:

Final();

Final(Level\* level);

void doubleScore();

void changeCards();

void peekCards();

// 升级

void upgradeLevel();

};

源文件：

Final::Final(){

}

Final::Final(Level\* level){

this->setGameAccount(level->getGameAccount());

getGameAccount()->setLevel(level);

}

void Final::doubleScore(){

printf("使用胜利双倍积分技能,");

}

void Final::changeCards(){

printf("使用换牌技能,");

}

void Final::peekCards(){

printf("使用偷看卡牌技能");

}

void Final::upgradeLevel(){

if (this->getGameAccount()->getScore() < 250){

this->getGameAccount()->setLevel(new Professional(this));

printf("\t降级！ 级别：PROFESSIONAL\n\n");

}

else{

printf("\t%s 已经是最高级\n\n", this->getGameAccount()->getName().c\_str());

}

}

**3.3.客户端代码示例及结果**

客户端代码创建了一个游戏账户Jungle，初始积分为100分，级别为Primary，即入门级，Jungle一共玩了5局牌。

#include "GameAccount.h"

#include "Level.h"

int main()

{

GameAccount \*jungle = new GameAccount("Jungle");

for (int i = 0; i < 5; i++){

printf("第%d局：\n", i + 1);

jungle->playCard();

}

printf("\n\n");

system("pause");

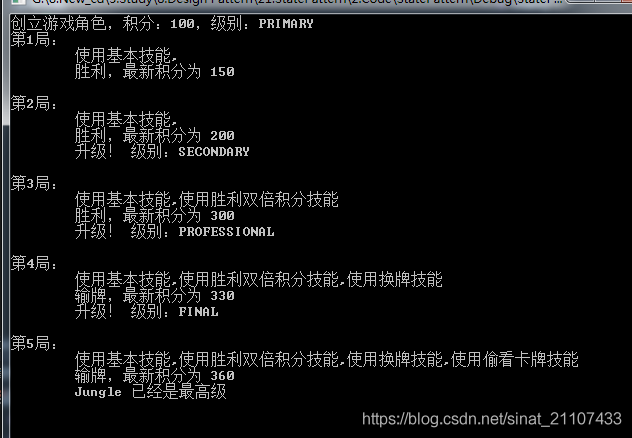
return 0;

}

结果如下：

 上面的代码不管Jungle当前是什么级别，都统一地调用了上下文类封装好的方法playcard()，即外界并不知道不同级别内部的具体实现细节。运行结果显示，Jungle的在不同的状态（级别）下能够表现不同的行为（不同的技能），并且能够不断改变自身的状态（升级或降级）。

上述代码源码请访问：<https://github.com/FengJungle/DesignPattern>

****

**4.总结**

**优点：**

（1）状态模式封装了状态转换的规则，只给外界暴露了统一的接口，客户端可以无差别地调用该接口（如上述实例的客户端代码）

（2）状态模式将所有与具体状态有关的行为放到一个类（具体状态类）中，只需要注入（依赖）不同的状态类对象到上下文类中，即可使上下文中拥有不同的行为

**缺点：**

（1）状态模式增加了系统中类的个数（不同的具体状态类）

（2）结构相对复杂（如前述实例的UML图），代码逻辑也较复杂

（3）如果要增加新的状态，需要修改负责状态转换的代码，不符合开闭原则（如上述实例，（4）如果增加了一个中间级别，是不是得修改很多状态转换的逻辑？）

**适用环境：**

（1）对象的行为根据它的状态的改变而不同

（2）代码中含有大量与对象状态有关的判断逻辑（if……else……或switch……case……）

**第二十五章 如何管理和维护算法族？只需知道策略模式**

**1.策略模式简介**

策略模式用于算法的自由切换和扩展，对应于解决某一问题的一个算法族，允许用户从该算法族中任意选择一个算法解决问题，同时还可以方便地更换算法或者增加新的算法。策略模式将算法族中的每一个算法都封装成一个类，每一个类称为一个策略（Strategy）。

**策略模式：**

**定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法可以独立于使用它的客户而变化。**

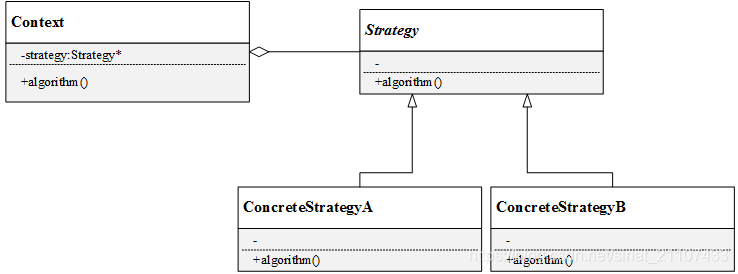
**2.策略模式结构**

为了方便算法族中的不同算法在使用中具有一致性，在策略模式中会提供一个抽象层来声明公共接口，在具体的策略类中实现各个算法。策略模式由上下文类和策略类组成，其UML结构如下图：

（1）Context（上下文类） ：上下文类是使用算法的角色，可以在解决不同具体的问题时实例化不同的具体策略类对象；

（2）Strategy（抽象策略类）：声明算法的方法，抽象层的设计使上下文类可以无差别的调用不同的具体策略的方法；

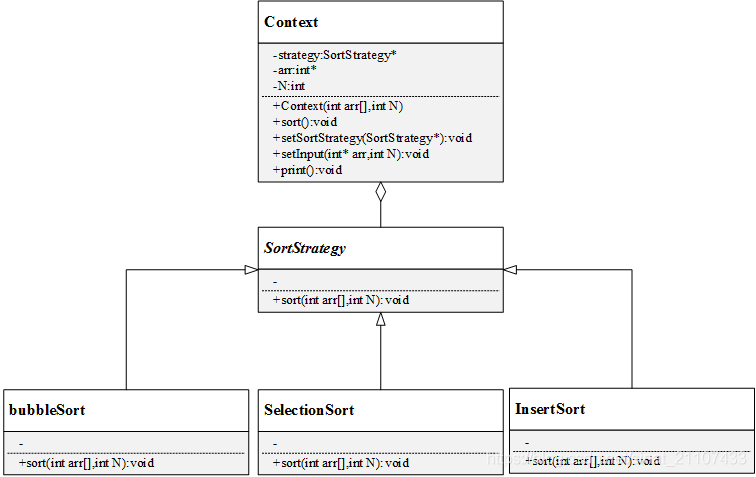
（3）ConcreteStrategy（具体策略类）：实现具体的算法。



**3.策略模式代码实例**

某系统提供了一个用于对数组进行操作的类，该类封装了对数组的常见操作，现以排序操作为例，使用策略模式设计该数组操作类，使得客户端可以动态更换排序算法，可以根据需要选择冒泡排序或者选择排序或者插入排序，也能够灵活增加新的排序算法 。

 显然，在该实例中，可以冒泡排序、选择排序和插入排序分别封装为3个具体策略类，它们有共同的基类SortStrategy。还需要一个上下文类Context，Context中维护了一个SortStrategy的指针，在客户端需要的时候，通过Context的setSortStrategy()方法来实例化具体的排序类对象。该实例的UML结构图如下：



**3.1.排序策略类**

**3.1.1.抽象排序策略类**

// 抽象策略类

class Strategy

{

public:

Strategy(){}

virtual void sort(int arr[], int N) = 0;

};

**3.1.2.具体策略类：冒泡排序类**

// 具体策略：冒泡排序

class BubbleSort :public Strategy

{

public:

BubbleSort(){

printf("冒泡排序\n");

}

void sort(int arr[], int N){

for (int i = 0; i<N; i++)

{

for (int j = 0; j<N - i - 1; j++)

{

if (arr[j]>arr[j + 1]){

int tmp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = tmp;

}

}

}

}

};

**3.1.3.具体策略类：选择排序类**

// 具体策略：选择排序

class SelectionSort :public Strategy

{

public:

SelectionSort(){

printf("选择排序\n");

}

void sort(int arr[], int N){

int i, j, k;

for (i = 0; i<N; i++)

{

k = i;

for (j = i + 1; j<N; j++)

{

if (arr[j] < arr[k]){

k = j;

}

}

int temp = arr[i];

arr[i] = arr[k];

arr[k] = temp;

}

}

};

**3.1.4.具体策略类：插入排序类**

// 具体策略：插入排序

class InsertSort :public Strategy

{

public:

InsertSort(){

printf("插入排序\n");

}

void sort(int arr[], int N){

int i, j;

for (i = 1; i<N; i++)

{

for (j = i - 1; j >= 0; j--)

{

if (arr[i]>arr[j]){

break;

}

}

int temp = arr[i];

for (int k = i - 1; k > j; k--){

arr[k + 1] = arr[k];

}

arr[j + 1] = temp;

}

}

};

**3.2.上下文类**

#ifndef \_\_CONTEXT\_H\_\_

#define \_\_CONTEXT\_H\_\_

#include "Strategy.h"

#include <stdio.h>

// 上下文类

class Context

{

public:

Context(){

arr = NULL;

N = 0;

}

Context(int iArr[], int iN){

this->arr = iArr;

this->N = iN;

}

void setSortStrategy(Strategy\* iSortStrategy){

this->sortStrategy = iSortStrategy;

}

void sort(){

this->sortStrategy->sort(arr, N);

printf("输出： ");

this->print();

}

void setInput(int iArr[], int iN){

this->arr = iArr;

this->N = iN;

}

void print(){

for (int i = 0; i < N; i++){

printf("%3d ", arr[i]);

}

printf("\n");

}

private:

Strategy\* sortStrategy;

int\* arr;

int N;

};

#endif // \_\_CONTEXT\_H\_\_

**3.3.客户端代码示例及结果**

#include "Context.h"

#include <stdio.h>

#include <stdlib.h>

int main()

{

Context\* ctx = new Context();

int arr[] = { 10, 23, -1, 0, 300, 87, 28, 77, -32, 2 };

ctx->setInput(arr, sizeof(arr)/sizeof(int));

printf("输入：");

ctx->print();

// 冒泡排序

ctx->setSortStrategy(new BubbleSort());

ctx->sort();

// 选择排序

ctx->setSortStrategy(new SelectionSort());

ctx->sort();

// 插入排序

ctx->setSortStrategy(new InsertSort());

ctx->sort();

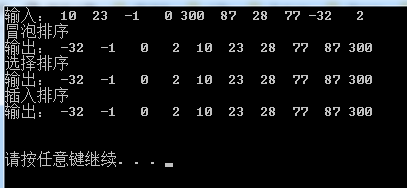
printf("\n\n");

system("pause");

return 0;

}

代码运行结果如下：



从客户端代码可以看到，客户端无需关心具体排序算法的细节，都是统一的调用上下文的sort()接口。另外，如果要增加新的排序算法，比如快速排序QuickSort，只需要从基类SortStrategy在派生一个类QuickSort，在QuickSort类中实现具体的sort()算法即可，扩展起来非常方便。

**4.总结**

**优点：**

（1）符合开闭原则，策略模式易于扩展，增加新的算法时只需继承抽象策略类，新设计实现一个具体策略类即可；

（2）客户端可以无差别地通过公共接口调用，利用里式替换原则，灵活使用不同的算法策略；

（3）提供了一个算法族管理机制和维护机制。

**缺点：**

（1）客户端必须要知道所有的策略，以便在使用时按需实例化具体策略；

（2）系统会产生很多单独的类，增加系统中类的数量；

（3）客户端在同一时间只能使用一种策略。

**适用环境：**

（1）系统需要在一个算法族中动态选择一种算法，可以将这些算法封装到多个具体算法类中，这些算法类都有共同的基类，即可以通过一个统一的接口调用任意一个算法，客户端可以使用任意一个算法；

（2）不希望客户端知道复杂的、与算法相关的数据结构，在具体策略类中封装与算法相关的数据结构，可以提高算法的安全性。

**第二十六 不知不觉就在使用的一种设计模式——模板方法模式**

**1.模板方法模式简介**

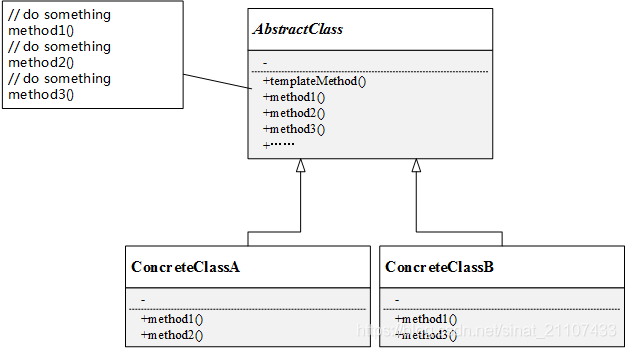
模板方法模式是较简单且常用的一种设计模式，是基于类的继承的一种代码复用技术，其结构只存在基类和派生类之间的继承关系。模板方法是一个具体的方法，给出了一个顶层逻辑流程框架。

**模板方法模式：**

**定义一个操作中的算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。**

**2.模板方法结构**

模板方法模式的UML图如下：



模板方法的结构很简单，只有基类和派生类两个角色：

（1）AbstractClass（基类）：即抽象类，在基类中定义或声明了一系列基本操作method，这些操作是具体或者抽象的，每一个操作都对应算法的一个步骤，在其派生类中可以重定义。基类中定义了一个模板方法（template method），它规定了算法的流程框架，模板方法由基类定义或声明的一系列基本操作按照一定流程实现。

（2）ConcreteClass（派生类）：实现在基类中声明的抽象方法，也可以覆盖在基类中已经实现的方法。

模板方法模式的典型代码如下：

#ifndef \_\_DEMO\_H\_\_

#define \_\_DEMO\_H\_\_

// 抽象类（基类）

class AbstractClass

{

public:

// 模板方法，定义一个算法的框架流程

void templateMethod(){

// do something

method1();

method2();

method3();

}

// 基本方法——公共方法

void mehtod1(){

// do something

}

// 基本方法2

virtual void method2() = 0;

// 基本方法3——默认实现

void mehtod3(){

// do something

}

};

// 具体类（派生类)

class ConcreteClass :public AbstractClass

{

public:

// 实现基本方法2

void method2(){

// do something

}

// 重定义基本方法3，覆盖基类的方法3

void method3(){

// do something

}

};

#endif

**3.模板方法模式代码实例**

某个指纹处理模块可以在两种模式下处理算法，即安全模式和非安全模式。在安全模式下，为了保证数据安全，某个指纹识别流程需要对采得的指纹图像进行加密，在处理图像之前再对加密数据进行解密。而非安全模式这不需要加密解密过程。指纹算法流程如下：采图——加密——解密——算法处理指纹——处理结果。现用模板方法模式模拟上述过程。

在这个实例中，Jungle首先定义了基类FingerprintModule，声明了基本方法：采图getImage()、判断是否在安全模式isSafeMode()、加密encrypt()、解密decrypt()、处理指纹图像processImage()、输出结果output()，在基类中定义了一个模板方法algorithm()，该方法里定义了指纹算法流程。

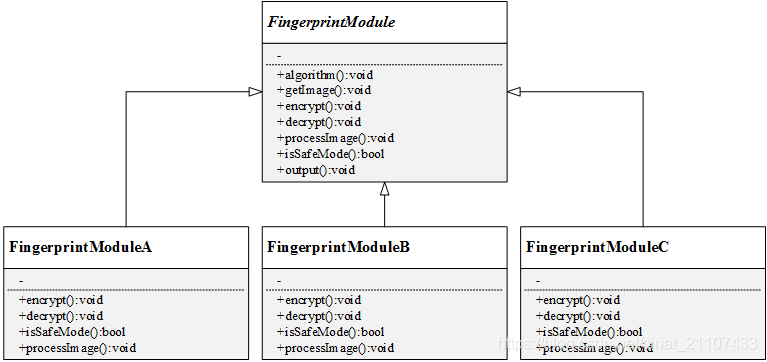
从基类FingerprintModule派生出3个子类，分别是FingerprintModuleA、FingerprintModuleB和FingerprintModuleC，三个子类的特点在于：

FingerprintModuleA：安全模式，采用RSA秘钥加解密，采用第一代版本算法处理指纹图像；

FingerprintModuleB：非安全模式，采用第二代版本算法处理指纹图像；

FingerprintModuleC：安全模式，采用DH秘钥加解密，采用第一代版本算法处理指纹图像；

该实例的UML图如下：



代码如下：

**3.1.基类**

// 基类

class FingerprintModule

{

public:

FingerprintModule(){}

void getImage(){

printf("采指纹图像\n");

}

void output(){

printf("指纹图像处理完成!\n");

}

virtual bool isSafeMode() = 0;

virtual void processImage() = 0;

// 加解密

virtual void encrypt() = 0;

virtual void decrypt() = 0;

// 模板方法

void algorithm(){

// 1.采图

getImage();

// 2.安全模式下加密和解密

if (isSafeMode()){

// 2.1. 加密

encrypt();

// 2.2. 解密

decrypt();

}

// 3.处理Image

processImage();

// 4.处理结果

output();

}

};

**3.2.派生类**

// 派生类

class FingerprintModuleA :public FingerprintModule

{

public:

FingerprintModuleA(){}

void processImage(){

printf("使用 第一代版本算法 处理指纹图像\n");

}

bool isSafeMode(){

printf("安全模式\n");

return true;

}

void encrypt(){

printf("使用RSA密钥加密\n");

}

void decrypt(){

printf("使用RSA密钥解密\n");

}

};

// 派生类

class FingerprintModuleB :public FingerprintModule

{

public:

FingerprintModuleB(){}

void processImage(){

printf("使用 第二代版本算法 处理指纹图像\n");

}

bool isSafeMode(){

printf("非安全模式\n");

return false;

}

void encrypt(){}

void decrypt(){}

};

// 派生类

class FingerprintModuleC :public FingerprintModule

{

public:

FingerprintModuleC(){}

void processImage(){

printf("使用 第一代版本算法 处理指纹图像\n");

}

bool isSafeMode(){

printf("安全模式\n");

return true;

}

void encrypt(){

printf("使用DH密钥加密\n");

}

void decrypt(){

printf("使用DH密钥解密\n");

}

};

**3.3.客户端代码实例及效果**

#include "FingerprintModule.h"

#include <Windows.h>

int main()

{

FingerprintModule \*fp = new FingerprintModuleA();

fp->algorithm();

fp = new FingerprintModuleB();

fp->algorithm();

fp = new FingerprintModuleC();

fp->algorithm();

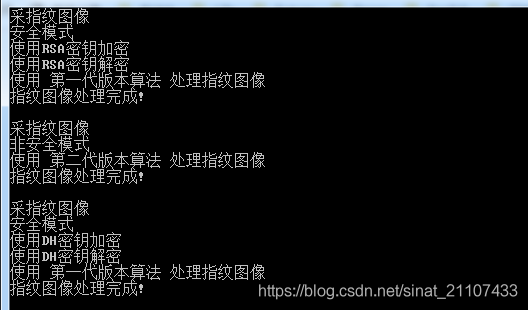
printf("\n\n");

system("pause");

return 0;

}

上述代码运行结果如下：



**4.总结**

模板方法模式是基于类的继承的一种设计模式，使用非常频繁，被广泛应用于框架设计。

**优点：**

（1）在基类中定义算法的框架，并声明一些流程方法，由具体派生类实现细节，派生类中的实现并不会影响基类定义的算法的框架流程；

（2）公共行为在基类中提供实现，有利于代码复用；

（3）派生类可以覆盖基类的方法，重新实现某些方法，具有灵活性；

（4）可以很方便的扩展和更换派生类而不影响基类和其他派生类，符合开闭原则和单一职责原则。

**缺点：**

（1）模板方法模式要为每一个不同的基本方法提供一个派生类，如果基类中基本方法很多，那系统中会定义很多个派生类，导致类的个数很多，系统更加庞大。

**适用环境：**

（1）分割复杂算法，可以将算法的框架流程定义在基类中，设计为模板方法；而具体的细节由派生类设计实现；

（2）各个派生类的公共部分提取到基类中，以实现代码复用；

（3）派生类需要覆盖基类的某些方法。

**第二十七章 访问者模式，从双十一购物开始说起**

**1.访问者模式简介**

类似于上述的习题册，软件设计中也需要这样的类似于习题册的对象结构，不同的对象对应不同的处理。设计模式中，访问者模式就是为了以不同的方式来操作复杂的对象结构。

访问者模式是一种较为复杂的行为型设计模式，具有访问者和被访问元素两个主要的角色。被访问的元素常常有不同的类型，不同的访问者可以对它们提供不同的访问方式。被访问元素通常不是单独存在，而是以集合的形式存在于一个对象结构中，访问者可以遍历该对象结构，以逐个访问其中的每一个元素。

**访问者模式：**

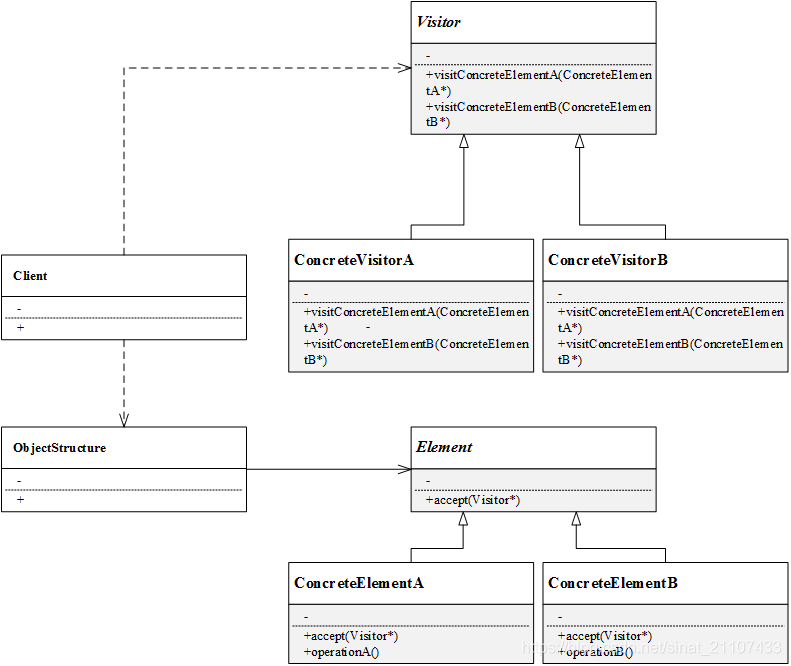
**表示一个作用于某对象结构中的各个元素的操作。访问者模式让用户可以在不改变各元素的前提下定义作用于这些元素的新操作。**

**2.访问者模式结构**

访问者模式的UML结构图如上：

访问者模式的结构相对较复杂，角色有如下几个：

（1）Visitor（抽象访问者）：抽象类，声明了访问对象结构中不同具体元素的方法，由方法名称可知该方法将访问对象结构中的某个具体元素；

（2）ConcreteVisitor（具体访问者）：访问某个具体元素的访问者，实现具体的访问方法；

Element（抽象元素）：抽象类，一般声明一个accept()的方法，用于接受访问者的访问，accept()方法常常以一个抽象访问者的指针作为参数；

（3）ConcreteElement（具体元素）：针对具体被访问的元素，实现accept()方法；

ObjectStructure（对象结构）：元素的集合，提供了遍历对象结构中所有元素的方法。对象结构存储了不同类型的元素对象，以供不同的访问者访问。

**从上图和前述可以看出，访问者模式中有两个层次结构：**

（1）访问者的层次结构：抽象访问者和具体访问者，不同的具体访问者有不同的访问方式（visit()方式）；

（2）被访问元素的层次结构：抽象元素和具体元素，不同的具体元素有不同的被访问方式（accept()方式）

正式由于有这两个层次结构，在增加新的访问者时，不必修改已有的代码，通过继承抽象访问者即可实现扩展，符合开闭原则，系统扩展性较好。但是在增加新的元素时，既要修改抽象访问者类（增加访问新增元素方法的声明），又要修改具体访问者（增加新的具体访问者类），不符合开闭原则。

访问者模式的示例代码如下：

#ifndef \_\_DEMO\_H\_\_

#define \_\_DEMO\_H\_\_

// 抽象访问者 Visitor

class Visitor

{

public:

virtual void visit(ConcreteElementA\*) = 0;

virtual void visit(ConcreteElementB\*) = 0;

};

// 具体访问者 ConcreteVisitor

class ConcreteVisitor :public Visitor

{

public:

// 实现一种针对特定元素的访问操作

void visit(ConcreteElementA\*){

// 元素A的访问操作代码

}

void visit(ConcreteElementB\*){

// 元素B的访问操作代码

}

};

// 抽象元素

class Element

{

public:

// 声明抽象方法，以一个抽象访问者的指针作为函数参数

virtual void accept(Visitor\*) = 0;

};

// 具体元素

class ConcreteElement :public Element

{

public:

void accept(Visitor\* visitor){

visitor->visit(this);

}

};

// 对象结构

class ObjectStructure

{

public:

// 提供接口接受访问者访问

void accept(Visitor\* visitor){

// 遍历访问对象结构中的元素

for (){

elementList[i]->accept(visitor);

}

}

void addElement(){}

void removeElement(){}

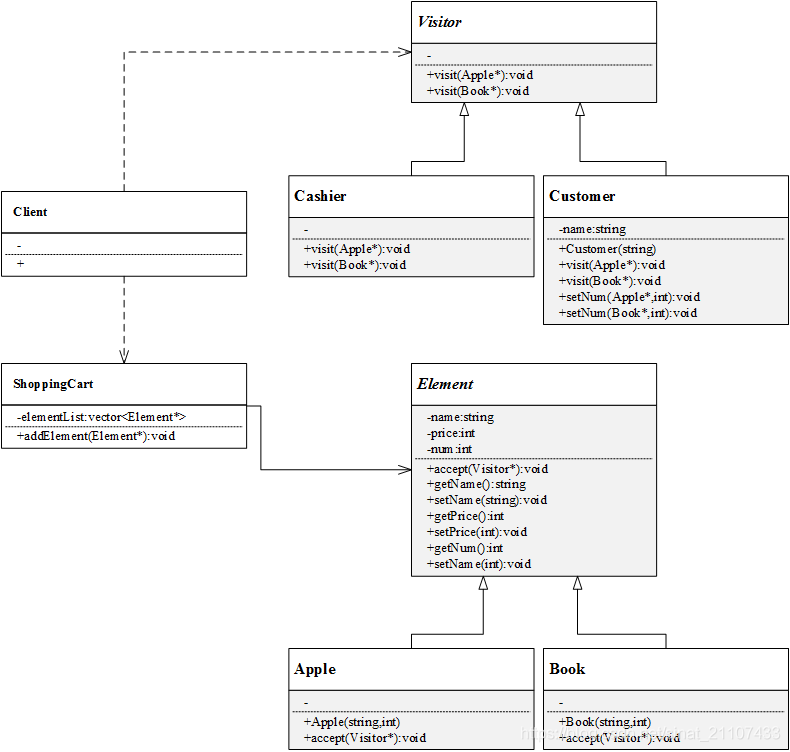
private:

lsit<Element\*>elementList;

};

#endif

**3.访问者模式代码实例**



Jungle作为一名顾客，去超市购物，加入购物车的商品包括两种苹果和两本书，结账时收银员需要计算各个商品的的价格。本例Jungle采用访问者模式来模拟该过程。

本例中，客户Jungle和收银员都会去访问商品，但关心的地方不同：Jungle关心的是苹果和书的单价、品牌等，收银员关注的是商品的价格。因此，客户Customer和收银员Cashier是具体访问者，而苹果Apple和书Book是具体被访问元素；而购物车则是对象结构。本例的UML图如下：

**3.1.元素类**

**3.1.1.抽象元素**

// 抽象元素

class Element

{

public:

Element(){};

virtual void accept(Visitor\*) = 0;

void setPrice(int iPrice){

this->price = iPrice;

}

int getPrice(){

return this->price;

}

void setNum(int iNum){

this->num = iNum;

}

int getNum(){

return num;

}

void setName(string iName){

this->name = iName;

}

string getName(){

return this->name;

}

private:

int price;

int num;

string name;

};

**3.1.2.具体元素Apple**

// 具体元素：Apple

class Apple :public Element

{

public:

Apple();

Apple(string name, int price);

void accept(Visitor\*);

};

实现：

Apple::Apple(){

setPrice(0);

setNum(0);

setName("");

}

Apple::Apple(string name, int price){

setPrice(price);

setNum(0);

setName(name);

}

void Apple::accept(Visitor\* visitor){

visitor->visit(this);

}

**3.1.3.具体元素Book**

// 具体元素：Book

class Book :public Element

{

public:

Book();

Book(string name, int price);

void accept(Visitor\*);

};

实现：

Book::Book(){

setPrice(0);

setNum(0);

setName("");

}

Book::Book(string iName, int iPrice){

setPrice(iPrice);

setNum(0);

setName(iName);

}

void Book::accept(Visitor\* visitor){

visitor->visit(this);

}

**3.2.访问者**

**3.2.1.抽象访问者**

// 抽象访问者

class Visitor

{

public:

Visitor(){};

// 声明一组访问方法

virtual void visit(Apple\*) = 0;

virtual void visit(Book\*) = 0;

};

**3.2.2.具体访问者Customer**

// 具体访问者：顾客

class Customer :public Visitor

{

public:

Customer();

Customer(string iName);

void setNum(Apple\*, int);

void setNum(Book\*, int);

void visit(Apple\* apple);

void visit(Book\* book);

private:

string name;

};

实现：

Customer::Customer(){

this->name = "";

}

Customer::Customer(string iName){

this->name = iName;

}

void Customer::setNum(Apple\* apple, int iNum){

apple->setNum(iNum);

}

void Customer::setNum(Book\* book, int iNum){

book->setNum(iNum);

}

void Customer::visit(Apple\* apple){

int price = apple->getPrice();

printf(" %s \t单价: \t%d 元/kg\n", apple->getName().c\_str(), apple->getPrice());

}

void Customer::visit(Book\* book){

int price = book->getPrice();

string name = book->getName();

printf(" 《%s》\t单价: \t%d 元/本\n", book->getName().c\_str(), book->getPrice());

}

**3.2.3.具体访问者Cashier**

class Cashier :public Visitor

{

public:

Cashier();

void visit(Apple\* apple);

void visit(Book\* book);

};

实现：

Cashier::Cashier(){

}

void Cashier::visit(Apple\* apple){

string name = apple->getName();

int price = apple->getPrice();

int num = apple->getNum();

int total = price\*num;

printf(" %s 总价： %d 元\n", name.c\_str(), total);

}

void Cashier::visit(Book\* book){

int price = book->getPrice();

string name = book->getName();

int num = book->getNum();

int total = price\*num;

printf(" 《%s》 总价： %d 元\n", name.c\_str(), total);

}

**3.3.购物车ShoppingCart**

class ShoppingCart

{

public:

ShoppingCart(){}

void addElement(Element\* element){

printf(" 商品名：%s, \t数量：%d, \t加入购物车成功！\n", element->getName().c\_str(), element->getNum());

elementList.push\_back(element);

}

void accept(Visitor\* visitor){

for (int i = 0; i < elementList.size(); i++){

elementList[i]->accept(visitor);

}

}

private:

vector<Element\*>elementList;

};

**3.4.客户端代码示例及结果**

#include "Element.h"

#include "Visitor.h"

#include "ShoppingCart.h"

#include <Windows.h>

int main()

{

Apple \*apple1 = new Apple("红富士苹果", 7);

Apple \*apple2 = new Apple("花牛苹果", 5);

Book \*book1 = new Book("红楼梦", 129);

Book \*book2 = new Book("终结者", 49);

Cashier\* cashier = new Cashier();

Customer\* jungle = new Customer("Jungle");

jungle->setNum(apple1, 2);

jungle->setNum(apple2, 4);

jungle->setNum(book1, 1);

jungle->setNum(book2, 3);

ShoppingCart\* shoppingCart = new ShoppingCart();

shoppingCart->addElement(apple1);

shoppingCart->addElement(apple2);

shoppingCart->addElement(book1);

shoppingCart->addElement(book2);

printf("\n\n");

shoppingCart->accept(jungle);

printf("\n\n");

shoppingCart->accept(cashier);

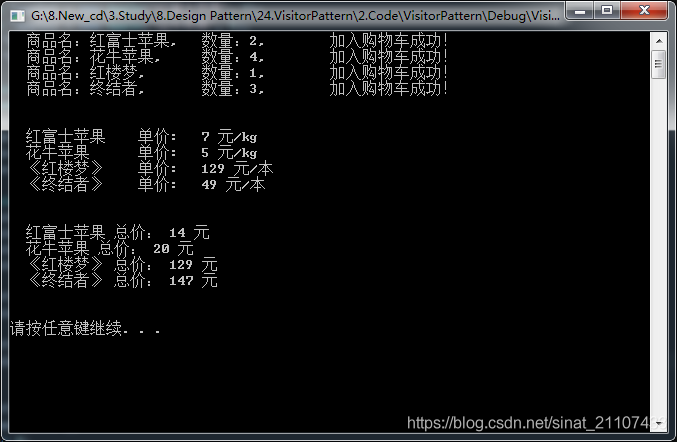
printf("\n\n");

system("pause");

return 0;

}

上述代码运行结果如下：



 上述代码资源见<https://github.com/FengJungle/DesignPattern>

**4.总结**

访问者模式的结构相对较复杂，在实际应用中使用频率较低。如果系统中存在一个复杂的对象结构，且不同的访问者对其具有不同的操作，那么可以考虑使用访问者模式。访问者模式的特点总结如下：

**优点：**

（1）增加新的访问者很方便，即增加一个新的具体访问者类，定义新的访问方式，无需修改原有代码，符合开闭原则；

（2）被访问元素集中在一个对象结构中，类的职责更清晰，利于对象结构中元素对象的复用；

**缺点：**

（1）增加新的元素类很困难，增加新的元素时，在抽象访问者类中需要增加一个对新增的元素方法的声明，即要修改抽象访问者代码；此外还要增加新的具体访问者以实现对新增元（2）素的访问，不符合开闭原则；

（3）破坏了对象的封装性，访问者模式要求访问者对象访问并调用每一个元素对象的操作，那么元素对象必须暴露自己的内部操作和状态，否则访问者无法访问。