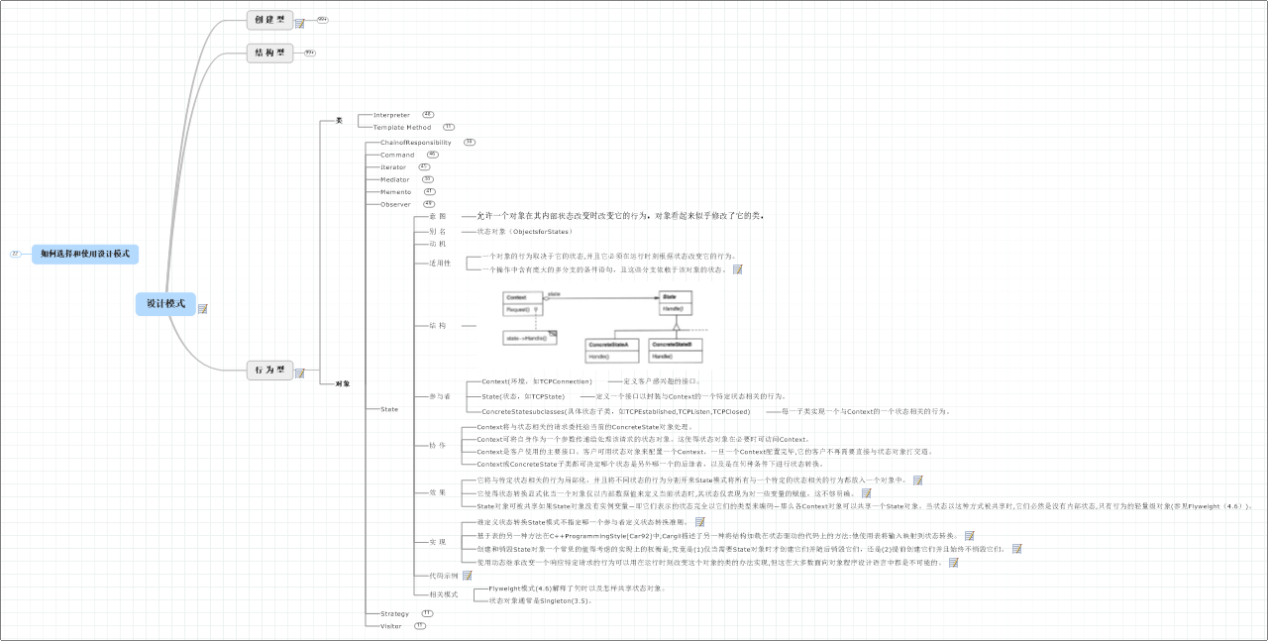
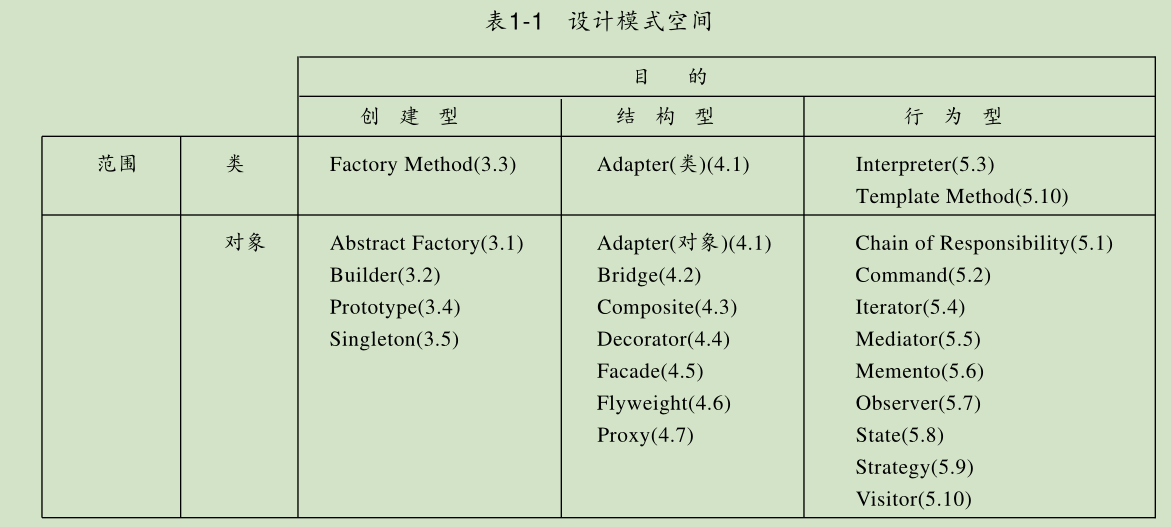
设计模式





创建型类模式将对象的部分创建工作延迟到子类，创建型对象模式则将它延迟到另一个对象中.

设计模式 1

如何选择和使用设计模式 8

如何选择设计模式 8

考虑设计模式是怎样解决设计问题的 8

寻找合适的对象 8

决定对象的粒度 9

指定对象接口 9

描述对象的实现 9

运用复用机制 12

关联运行时刻和编译时刻的结构 15

设计应支持变化 16

浏览模式的意图部分 20

考虑你的设计中哪些是可变的 21

创 建 型 22

类 23

Factory Method 23

代码示例 23

对象 23

Abstract Factory 23

意 图 23

别 名 24

参与者 24

A b s t r a c t F a c t o r y 24

C o n c r e t e F a c t o r y 24

A b s t r a c t P r o d u c t 24

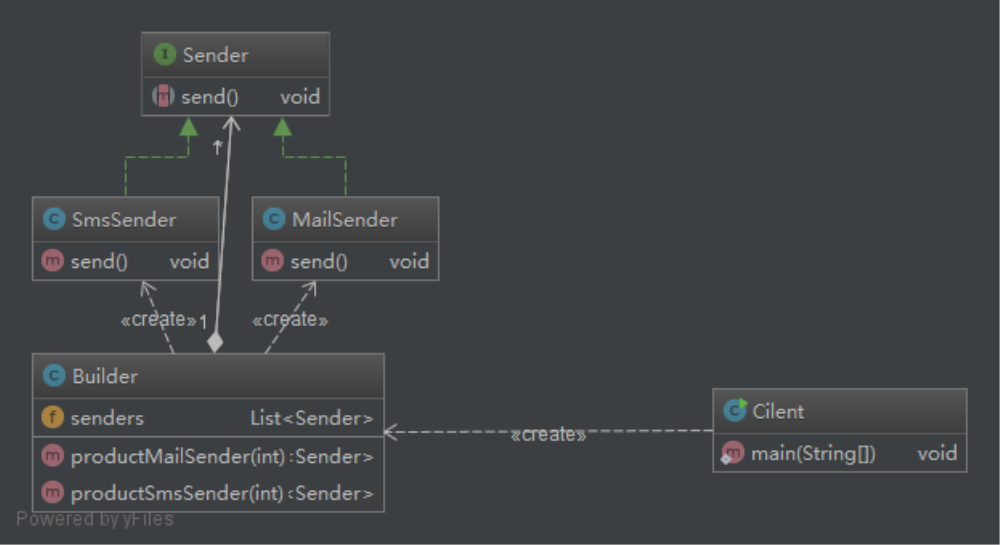
协 作 24

实 现 24

将工厂作为单件 24

代码示例 25

B u i l d e r 26

 代码示例 26

相关模式 27

P r o t o t y p e 28

效 果 28

运行时刻增加和删除产品 28

改变值以指定新对象 28

改变结构以指定新对象 28

减少子类的构造 28

用类动态配置应用 29

是每一个P r o t o t y p e的子类都必须实现C l o n e操作 29

实 现 29

实现克隆操作 30

初始化克隆对象 30

代码示例 31

S i n g l e t o n 32

效 果 32

缩小名空间 32

允许对操作和表示的精化 32

允许可变数目的实例 32

比类操作更灵活 32

实 现 32

创建S i n g l e t o n类的子类 33

代码示例 33

结 构 型 39

类 39

A d a p t e r (类) 39

效 果 39

使用A d a p t e r模式时需要考虑的其他一些因素有： 39

Adapter的匹配程度 39

可插入的Adapter 39

使用双向适配器提供透明操作 39

代码示例 40

类 40

对象 40

A d a p t e r (对象) 42

效 果 42

使用A d a p t e r模式时需要考虑的其他一些因素有： 42

Adapter的匹配程度 42

可插入的Adapter 42

使用双向适配器提供透明操作 42

代码示例 43

类 43

B r i d g e 44

适用性 44

（C + +）你想对客户完全隐藏抽象的实现部分。在 C + +中，类的表示在类接口中是可见的。 44

效 果 44

分离接口及其实现部分 44

提高可扩充性 44

实现细节对客户透明 44

实 现 45

仅有一个Implementor 45

创建正确的I m p l e m e n t o r对象 45

代码示例 46

C o m p o s i t e 48

效 果 48

定义了包含基本对象和组合对象的类层次结构 48

简化客户代码 48

使得更容易增加新类型的组件 48

使你的设计变得更加一般化 49

实 现 49

实现C o m p o s i t e模式时需要考虑以下几个问题 49

显式的父部件引用 49

共享组件 49

最大化C o m p o n e n t接口 50

声明管理子部件的操作 50

Component是否应该实现一个C o m p o n e n t列表 51

子部件排序 51

使用高速缓冲存贮改善性能 51

应该由谁删除Component 52

存贮组件最好用哪一种数据结构 52

代码示例 52

相关模式 54

D e c o r a t o r（4 . 4）模式经常与C o m p o s i t e模式一起使用。 54

D e c o r a t o r 54

效 果 54

优点 54

比静态继承更灵活 54

避免在层次结构高层的类有太多的特征 54

Decorator与它的C o m p o n e n t不一样 55

缺点 55

有许多小对象 55

代码示例 55

F a c a d e 58

适用性 58

当你要为一个复杂子系统提供一个简单接口时。 58

客户程序与抽象类的实现部分之间存在着很大的依赖性。 58

当你需要构建一个层次结构的子系统时，使用 f a c a d e模式定义子系统中每层的入口点。 58

效 果 58

它实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。 58

如果应用需要，它并不限制它们使用子系统类。 59

实 现 59

降低客户-子系统之间的耦合度 59

公共子系统类与私有子系统类 59

代码示例 60

P r o x y 64

适用性 64

远 程 代 理 （ Remote Proxy） 为 一 个 对 象在 不 同 的 地 址 空间 提 供 局 部 代 表。 64

虚代理（Virtual Proxy）根据需要创建开销很大的对象。 64

保护代理（Protection Proxy）控制对原始对象的访问。 64

智能指引（Smart Reference）取代了简单的指针，它在访问对象时执行一些附加操作。 65

效 果 65

代码示例 66

F l y w e i g h t 68

效 果 68

存储节约由以下几个因素决定 68

实 现 69

删除外部状态 69

管理共享对象 69

代码示例 70

结构型模式的讨论 73

Adapter与Bridge 73

Composite、Decorator与Proxy 74

行 为 型 76

类 77

Interpreter 77

效 果 77

复杂的文法难以维护解释器模式为文法中的每一条规则至少定义了一个类(使用BNF定义的文法规则需要更多的类)。因此包含许多规则的文法可能难以管理和维护。 77

实 现 78

与Flyweight模式共享终结符在一些文法中,一个句子可能多次出现同一个终结符。 78

代码示例 78

对象 81

ChainofResponsibility 81

效 果 81

降低耦合度 81

增强了给对象指派职责( R e s p o n s i b i l i t y )的灵活性 82

不保证被接受 82

实 现 82

连接后继者 82

在S m a l l t a l k中自动转发 82

表示请求 82

代码示例 83

Command 86

适用性 86

在不同的时刻指定、排列和执行请求。 86

支持取消操作。 86

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。 86

用构建在原语操作上的高层操作构造一个系统。 86

协 作 87

某Invoker对象存储该ConcreteCommand对象。 87

实 现 87

一个命令对象应达到何种智能程度命令对象的能力可大可小。 87

支持取消（undo）和重做（redo）如果Command提供方法逆转(reverse)它们操作的执行(例如Unexecute或Undo操作)，就可支持取消和重做功能。 87

避免取消操作过程中的错误积累在实现一个可靠的、能保持原先语义的取消/重做机制时，可能会遇到滞后影响问题。 88

代码示例 88

Iterator 91

效 果 91

它支持以不同的方式遍历一个聚合复杂的聚合可用多种方式进行遍历。 91

实 现 91

谁控制该迭代一个基本的问题是决定由哪一方来控制该迭代,是迭代器还是使用该迭代器的客户。 91

谁定义遍历算法迭代器不是唯一可定义遍历算法的地方。 92

迭代器健壮程度如何在遍历一个聚合的同时更改这个聚合可能是危险的。 92

附加的迭代器操作迭代器的最小接口由First、Next、IsDone和CurrentItem操作组成。 93

在C++中使用多态的迭代器使用多态迭代器是有代价的。 93

迭代器可有特权访问迭代器可被看为创建它的聚合的一个扩展。 93

用于复合对象的迭代器在Composite(4.3)模式中的那些递归聚合结构上,外部迭代器可能难以实现,因为在该结构中不同对象处于嵌套聚合的多个不同层次，因此一个外部迭代器为跟踪当前的对象必须存储一条纵贯该Composite的路径。 94

空迭代器一个空迭代器(NullIterator)是一个退化的迭代器,它有助于处理边界条件。 94

代码示例 94

Mediator 99

实 现 99

)忽略抽象的Mediator类当各Colleague仅与一个Mediator一起工作时,没有必要定义一个抽象的Mediator类。 99

Colleague—Mediator通信当一个感兴趣的事件发生时,Colleague必须与其Mediator通信。 99

代码示例 100

相关模式 105

Facade 105

Colleague 105

Memento 105

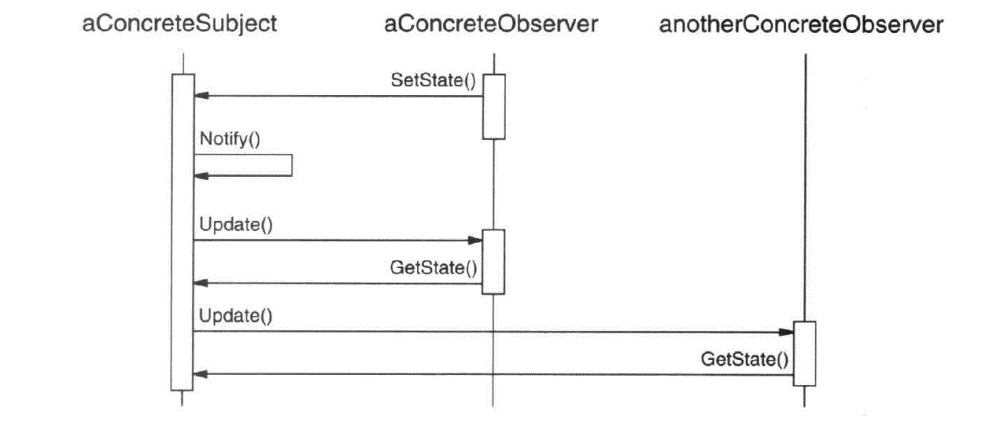
协 作 105

管理器向原发器请求一个备忘录,保留一段时间后,将其送回给原发器 105

代码示例 105

Observer 110

协 作 110

 110

效 果 110

目标和观察者间的抽象耦合一个目标所知道的仅仅是它有一系列观察者,每个都符合抽象的Observer类的简单接口。目标不知道任何一个观察者属于哪一个具体的类。这样目标和观察者之间的耦合是抽象的和最小的。 110

意外的更新因为一个观察者并不知道其它观察者的存在,它可能对改变目标的最终代价一无所知。 111

实 现 111

创建目标到其观察者之间的映射一个目标对象跟踪它应通知的观察者的最简单的方法是显式地在目标中保存对它们的引用。 111

观察多个目标在某些情况下,一个观察者依赖于多个目标可能是有意义的。 111

谁触发更新目标和它的观察者依赖于通知机制来保持一致。 111

对已删除目标的悬挂引用删除一个目标时应注意不要在其观察者中遗留对该目标的悬挂引用。 112

在发出通知前确保目标的状态自身是一致的在发出通知前确保状态自身一致这一点很重要,因为观察者在更新其状态的过程中需要查询目标的当前状态。 112

显式地指定感兴趣的改变你可以扩展目标的注册接口,让各观察者注册为仅对特定事件感兴趣，以提高更新的效率。 113

结合目标类和观察者类用不支持多重继承的语言(如Smalltalk)书写的类库通常不单独定义Subject和Observer类,而是将它们的接口结合到一个类中。 114

代码示例 114

State 119

适用性 119

一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。 119

效 果 120

它将与特定状态相关的行为局部化，并且将不同状态的行为分割开来State模式将所有与一个特定的状态相关的行为都放入一个对象中。 120

它使得状态转换显式化当一个对象仅以内部数据值来定义当前状态时,其状态仅表现为对一些变量的赋值，这不够明确。 120

实 现 121

谁定义状态转换State模式不指定哪一个参与者定义状态转换准则。 121

基于表的另一种方法在C++ProgrammingStyle[Car92]中,Cargil描述了另一种将结构加载在状态驱动的代码上的方法:他使用表将输入映射到状态转换。 121

创建和销毁State对象一个常见的值得考虑的实现上的权衡是,究竟是(1)仅当需要State对象时才创建它们并随后销毁它们，还是(2)提前创建它们并且始终不销毁它们。 122

使用动态继承改变一个响应特定请求的行为可以用在运行时刻改变这个对象的类的办法实现,但这在大多数面向对象程序设计语言中都是不可能的。 122

代码示例 122

如何选择和使用设计模式

如何选择设计模式

考虑设计模式是怎样解决设计问题的

寻找合适的对象

面向对象程序由对象组成，对象包括数据和对数据进行操作的过程，过程通常称为方法或操作。对象在收到客户的请求(或消息)后，执行相应的操作。客户请求是使对象执行操作的唯一方法，操作又是对象改变内部数据的唯一方法。由于这些限制，对象的内部状态是被封装的，它不能被直接访问，它的表示对于对象外部是不可见的。面向对象设计最困难的部分是将系统分解成对象集合。因为要考虑许多因素：封装、粒度、依赖关系、灵活性、性能、演化、复用等等，它们都影响着系统的分解，并且这些因素通常还是互相冲突的。面向对象设计方法学支持许多设计方法。你可以写出一个问题描述，挑出名词和动词，进而创建相应的类和操作；或者，你可以关注于系统的协作和职责关系；或者，你可以对现实世界建模，再将分析时发现的对象转化至设计中。至于哪一种方法最好，并无定论。设计的许多对象来源于现实世界的分析模型。但是，设计结果所得到的类通常在现实世界中并不存在，有些是像数组之类的低层类，而另一些则层次较高。例如，Composite(4.3)模式引入了统一对待现实世界中并不存在的对象的抽象方法。严格反映当前现实世界的模型并不能产生也能反映将来世界的系统。设计中的抽象对于产生灵活的设计是至关重要的。设计模式帮你确定并不明显的抽象和描述这些抽象的对象。例如，描述过程或算法的对象现实中并不存在，但它们却是设计的关键部分。Strategy(5.9)模式描述了怎样实现可互换的算法族。State(5.8)模式将实体的每一个状态描述为一个对象。这些对象在分析阶段，甚至在设计阶段的早期都并不存在，后来为使设计更灵活、复用性更好才将它们发掘出来。决定对象的粒度

决定对象的粒度

对象在大小和数目上变化极大。它们能表示下自硬件或上自整个应用的任何事物。那么我们怎样决定一个对象应该是什么呢？设计模式很好地讲述了这个问题。Facade(4.5)模式描述了怎样用对象表示完整的子系统，Flyweight(4.6)模式描述了如何支持大量的最小粒度的对象。其他一些设计模式描述了将一个对象分解成许多小对象的特定方法。AbstractFactory(3.1)和Builder(3.2)产生那些专门负责生成其他对象的对象。Visitor(5.10)和Command(5.2)生成的对象专门负责实现对其他对象或对象组的请求。

指定对象接口

对象声明的每一个操作指定操作名、作为参数的对象和返回值，这就是所谓的操作的型构(signature)。对象操作所定义的所有操作型构的集合被称为该对象的接口(interface)。对象接口描述了该对象所能接受的全部请求的集合，任何匹配对象接口中型构的请求都可以发送给该对象。

类型(type)是用来标识特定接口的一个名字。如果一个对象接受“Window”接口所定义的所有操作请求，那么我们就说该对象具有“Window”类型。一个对象可以有许多类型，并且不同的对象可以共享同一个类型。对象接口的某部分可以用某个类型来刻画，而其他部分则可用其他类型刻画。两个类型相同的对象只需要共享它们的部分接口。接口可以包含其他接口作为子集。当一个类型的接口包含另一个类型的接口时，我们就说它是另一个类型的子

类型(subtype)，另一个类型称之为它的超类型(supertype)。我们常说子类型继承了它的超类型

的接口。

描述对象的实现

至此，我们很少提及到实际上怎么去定义一个对象。对象的实现

是由它的类决定的，类指定了对象的内部数据和表示，也定义了对象

所能完成的操作，如右图所示。

我们基于O M T的表示法，将类描述成一个矩形，其中的类名以黑

体表示的。操作在类名下面，以常规字体表示。类所定义的任何数据

都在操作的下面。类名与操作之间以及操作与数据之间用横线分割。

返回类型和实例变量类型是可选的，因为我们并未假设一定要用具有静态类型的实现语

言。

对象通过实例化类来创建，此对象被称为该类的实例。当实例化类时，要给对象的内部

数据(由实例变量组成)分配存储空间，并将操作与这些数据联系起来。对象的许多类似实例是

由实例化同一个类来创建的。

下图中的虚箭头线表示一个类实例化另一个类的对象，箭头指向被实例化的对象的类。

新的类可以由已存在的类通过类继承(class inheritance)来定义。当子类( s u b c l a s s )继承父类

(parent class)时，子类包含了父类定义的所有数据和操作。子类的实例对象包含所有子类和父

类定义的数据，且它们能完成子类和父类定义的所有操作。我们以竖线和三角表示子类关系，

如下图所示。

抽象类(abstract class)的主要目的是为它的子类定义公共接口。一个抽象类将把它的部分

或全部操作的实现延迟到子类中，因此，一个抽象类不能被实例化。在抽象类中定义却没有

实现的操作被称为抽象操作(abstract operation)。非抽象类称为具体类(concrete class)。

子类能够改进和重新定义它们父类的操作。更具体地说，类能够重定义( o v e r r i d e )父类定

义的操作，重定义使得子类能接管父类对请求的处理操作。类继承允许你只需简单的扩展其

他类就可以定义新类，从而可以很容易地定义具有相近功能的对象族。

抽象类的类名以斜体表示，以与具体类相区别。抽象操作也用斜体表示。图中可以包括

实现操作的伪代码，如果这样，则代码将出现在带有摺角的框中，并用虚线将该摺角框与代

码所实现的操作相连，图示如下。

混入类(mixin class)是给其他类提供可选择的接口或功能的类。它与抽象类一样不能实例

化。混入类要求多继承，图示如下。

1. 类继承与接口继承的比较

理解对象的类( c l a s s )与对象的类型( t y p e )之间的差别非常重要。

一个对象的类定义了对象是怎样实现的，同时也定义了对象的内部状态和操作的实现。

但是对象的类型只与它的接口有关，接口即对象能响应的请求的集合。一个对象可以有多个

类型，不同类的对象可以有相同的类型。

当然，对象的类和类型是有紧密关系的。因为类定义了对象所能执行的操作，也定义了

对象的类型。当我们说一个对象是一个类的实例时，即指该对象支持类所定义的接口。

C + +和E i ff e l语言的类既指定对象的类型又指定对象的实现。 S m a l l t a l k程序不声明变量的

类型，所以编译器不检查赋给变量的对象类型是否是该变量的类型的子类型。发送消息时需

要检查消息接收者是否实现了该消息，但不检查接收者是否是某个特定类的实例。

理解类继承和接口继承(或子类型化)之间的差别也十分重要。类继承根据一个对象的实现

定义了另一个对象的实现。简而言之，它是代码和表示的共享机制。然而，接口继承 (或子类

型化)描述了一个对象什么时候能被用来替代另一个对象。

因为许多语言并不显式地区分这两个概念，所以容易被混淆。在 C + +和E i ff e l语言中，继

承既指接口的继承又指实现的继承。 C + +中接口继承的标准方法是公有继承一个含 (纯)虚成员

函数的类。C + +中纯接口继承接近于公有继承纯抽象类，纯实现继承或纯类继承接近于私有

继承。S m a l l t a l k中的继承只指实现继承。只要任何类的实例支持对变量值的操作，你就可以

将这些实例赋给变量。

尽管大部分程序设计语言并不区分接口继承和实现继承的差别，但使用中人们还是分别

对待它们的。S m a l l t a l k程序员通常将子类当作子类型 (尽管有一些熟知的例外情况 [ C o o 9 2 ] )，

C + +程序员通过抽象类所定义的类型来操纵对象。

很多设计模式依赖于这种差别。例如， Chain of Responsibility(5.1)模式中的对象必须有一

个公共的类型，但一般情况下它们不具有公共的实现。在 C o m p o s i t e ( 4 . 3 )模式中，构件定义了

一个公共的接口，但 C o m p o s i t e通常定义一个公共的实现。 C o m m a n d ( 5 . 2 )、O b s e r v e r ( 5 . 7 )、

S t a t e ( 5 . 8 )和S t r a t e g y ( 5 . 9 )通常纯粹作为接口的抽象类来实现。

2. 对接口编程，而不是对实现编程

类继承是一个通过复用父类功能而扩展应用功能的基本机制。它允许你根据旧对象快速

定义新对象。它允许你从已存在的类中继承所需要的绝大部分功能，从而几乎无需任何代价

就可以获得新的实现。

然而，实现的复用只是成功的一半，继承所拥有的定义具有相同接口的对象族的能力也

是很重要的(通常可以从抽象类来继承)。为什么？因为多态依赖于这种能力。

当继承被恰当使用时，所有从抽象类导出的类将共享该抽象类的接口。这意味着子类仅

仅添加或重定义操作，而没有隐藏父类的操作。这时，所有的子类都能响应抽象类接口中的

请求，从而子类的类型都是抽象类的子类型。

只根据抽象类中定义的接口来操纵对象有以下两个好处：

1) 客户无须知道他们使用对象的特定类型，只须对象有客户所期望的接口。

2) 客户无须知道他们使用的对象是用什么类来实现的，他们只须知道定义接口的抽象类。

这将极大地减少子系统实现之间的相互依赖关系，也产生了可复用的面向对象设计的如

下原则：

针对接口编程，而不是针对实现编程。

不将变量声明为某个特定的具体类的实例对象，而是让它遵从抽象类所定义的接口。这

是本书设计模式的一个常见主题。

当你不得不在系统的某个地方实例化具体的类 (即指定一个特定的实现 )时，创建型模式

(Abstract Factory(3.1)，B u i l d e r ( 3 . 2 )，Factory Method(3.3)，P r o t o t y p e ( 3 . 4 )和S i n g l e t o n ( 3 . 5 ) )可

以帮你。通过抽象对象的创建过程，这些模式提供不同方式以在实例化时建立接口和实现的

透明连接。创建型模式确保你的系统是采用针对接口的方式书写的，而不是针对实现而书写

的。

运用复用机制

理解对象、接口、类和继承之类的概念对大多数人来说并不难，问题的关键在于如何运

用它们写出灵活的、可复用的软件。设计模式将告诉你怎样去做。

1. 继承和组合的比较

面向对象系统中功能复用的两种最常用技术是类继承和对象组合(object composition)。正

如我们已解释过的，类继承允许你根据其他类的实现来定义一个类的实现。这种通过生成子

类的复用通常被称为白箱复用(white-box reuse)。术语“白箱”是相对可视性而言：在继承方

式中，父类的内部细节对子类可见。

对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组装或组合对象

来获得。对象组合要求被组合的对象具有良好定义的接口。这种复用风格被称为黑箱复用

(black-box reuse)，因为对象的内部细节是不可见的。对象只以“黑箱”的形式出现。

继承和组合各有优缺点。类继承是在编译时刻静态定义的，且可直接使用，因为程序设

计语言直接支持类继承。类继承可以较方便地改变被复用的实现。当一个子类重定义一些而

不是全部操作时，它也能影响它所继承的操作，只要在这些操作中调用了被重定义的操作。

但是类继承也有一些不足之处。首先，因为继承在编译时刻就定义了，所以无法在运行

时刻改变从父类继承的实现。更糟的是，父类通常至少定义了部分子类的具体表示。因为继

承对子类揭示了其父类的实现细节，所以继承常被认为“破坏了封装性” [ S n y 8 6 ]。子类中的

实现与它的父类有如此紧密的依赖关系，以至于父类实现中的任何变化必然会导致子类发生

变化。

当你需要复用子类时，实现上的依赖性就会产生一些问题。如果继承下来的实现不适合

解决新的问题，则父类必须重写或被其他更适合的类替换。这种依赖关系限制了灵活性并最

终限制了复用性。一个可用的解决方法就是只继承抽象类，因为抽象类通常提供较少的实现。

对象组合是通过获得对其他对象的引用而在运行时刻动态定义的。组合要求对象遵守彼

此的接口约定，进而要求更仔细地定义接口，而这些接口并不妨碍你将一个对象和其他对象

一起使用。这还会产生良好的结果：因为对象只能通过接口访问，所以我们并不破坏封装

性；只要类型一致，运行时刻还可以用一个对象来替代另一个对象；更进一步，因为对象的

实现是基于接口写的，所以实现上存在较少的依赖关系。

对象组合对系统设计还有另一个作用，即优先使用对象组合有助于你保持每个类被封装，

并被集中在单个任务上。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控

制的庞然大物。另一方面，基于对象组合的设计会有更多的对象 (而有较少的类)，且系统的行

为将依赖于对象间的关系而不是被定义在某个类中。

这导出了我们的面向对象设计的第二个原则：

优先使用对象组合，而不是类继承。

理想情况下，你不应为获得复用而去创建新的构件。你应该能够只使用对象组合技术，

通过组装已有的构件就能获得你需要的功能。但是事实很少如此，因为可用构件的集合实际

上并不足够丰富。使用继承的复用使得创建新的构件要比组装旧的构件来得容易。这样，继

承和对象组合常一起使用。

然而，我们的经验表明：设计者往往过度使用了继承这种复用技术。但依赖于对象组合

技术的设计却有更好的复用性(或更简单)。你将会看到设计模式中一再使用对象组合技术。

2. 委托

委托(d e l e g a t i o n)是一种组合方法，它使组合具有与继承同样的复用能力 [ L i e 8 6 , J Z 9 1 ]。在

委托方式下，有两个对象参与处理一个请求，接受请求的对象将操作委托给它的代理者

(d e l e g a t e)。这类似于子类将请求交给它的父类处理。使用继承时，被继承的操作总能引用接

受请求的对象，C + +中通过t h i s成员变量，S m a l l t a l k中则通过s e l f。委托方式为了得到同样的

效果，接受请求的对象将自己传给被委托者（代理人），使被委托的操作可以引用接受请求的

对象。

举例来说，我们可以在窗口类中保存一个矩形类的实例变量来代理矩形类的特定操作，

这样窗口类可以复用矩形类的操作，而不必像继承时那样定义成矩形类的子类。也就是说，

一个窗口拥有一个矩形，而不是一个窗口就是一个矩形。窗口现在必须显式的将请求转发给

它的矩形实例，而不是像以前它必须继承矩形的操作。

下面的图显示了窗口类将它的A r e a操作委托给一个矩形实例。

箭头线表示一个类对另一个类实例的引用关系。引用名是可选的，本例为“ r e c t a n g l e”。

委托的主要优点在于它便于运行时刻组合对象操作以及改变这些操作的组合方式。假定

矩形对象和圆对象有相同的类型，我们只需简单的用圆对象替换矩形对象，则得到的窗口就

是圆形的。

委托与那些通过对象组合以取得软件灵活性的技术一样，具有如下不足之处：动态的、

高度参数化的软件比静态软件更难于理解。还有运行低效问题，不过从长远来看人的低效才

是更主要的。只有当委托使设计比较简单而不是更复杂时，它才是好的选择。要给出一个能

确切告诉你什么时候可以使用委托的规则是很困难的。因为委托可以得到的效率是与上下文

有关的，并且还依赖于你的经验。委托最适用于符合特定程式的情形，即标准模式的情形。

有一些模式使用了委托，如 S t a t e ( 5 . 8 )、S t r a t e g y ( 5 . 9 )和Vi s i t o r ( 5 . 11 )。在S t a t e模式中，一

个对象将请求委托给一个描述当前状态的 S t a t e对象来处理。在S t r a t e g y模式中，一个对象将一

个特定的请求委托给一个描述请求执行策略的对象，一个对象只会有一个状态，但它对不同

的请求可以有许多策略。这两个模式的目的都是通过改变受托对象来改变委托对象的行为。

在Vi s i t o r中，对象结构的每个元素上的操作总是被委托到 Vi s i t o r对象。

其他模式则没有这么多 地用到委托。M e d i a t o r ( 5 . 5 )引进了一个中介其他对象间通信的对

象。有时，M e d i a t o r对象只是简单地将请求转发给其他对象；有时，它沿着指向自己的引用

来传递请求，使用真正意义的委托。 Chain of Responsibility(5.1)通过将请求沿着对象链传递

来处理请求，有时，这个请求本身带有一个接受请求对象的引用，这时该模式就使用了委托。

B r i d g e ( 4 . 2 )将实现和抽象分离开，如果抽象和一个特定实现非常匹配，那么这个实现可以代

理抽象的操作。

委托是对象组合的特例。它告诉你对象组合作为一个代码复用机制可以替代继承。

3. 继承和参数化类型的比较

另一种功能复用技术(并非严格的面向对象技术 )是参数化类型(parameterized type)，也就

是类属( g e n e r i c ) ( A d a、E i ff e l )或模板( t e m p l a t e s ) ( C + + )。它允许你在定义一个类型时并不指定该

类型所用到的其他所有类型。未经指定的类型在使用时以参数形式提供。例如，一个列表类

能够以它所包含元素的类型来进行参数化。如果你想声明一个 I n t e g e r列表，只需将I n t e g e r类

型作为列表参数化类型的参数值；声明一个 S t r i n g列表，只需提供S t r i n g类型作为参数值。语

言的实现将会为各种元素类型创建相应的列表类模板的定制版本。

参数化类型给我们提供除了类继承和对象组合外的第三种方法来组合面向对象系统中的

行为。许多设计可以使用这三种技术中的任何一种来实现。实现一个以元素比较操作为可变

元的排序例程，可有如下方法：

1) 通过子类实现该操作( Template Method(5.10)的一个应用)。

2 ) 实现为传给排序例程的对象的职责 ( S t r a t e g y ( 5 . 9 ) )。

3) 作为C + +模板或A d a类属的参数，以指定元素比较操作的名称。

这些技术存在着极大的不同之处。对象组合技术允许你在运行时刻改变被组合的行为，

但是它存在间接性，比较低效。继承允许你提供操作的缺省实现，并通过子类重定义这些操

作。参数化类型允许你改变类所用到的类型。但是继承和参数化类型都不能在运行时刻改变。

哪一种方法最佳，取决于你设计和实现的约束条件。

本书没有一种模式是与参数化类型有关的，尽管我们在定制一个模式的 C + +实现时用到了

参数化类型。参数化类型在像 S m a l l t a l k那样的编译时刻不进行类型检查的语言中是完全不必

要的。

关联运行时刻和编译时刻的结构

一个面向对象程序运行时刻的结构通常与它的代码结构相差较大。代码结构在编译时刻

就被确定下来了，它由继承关系固定的类组成。而程序的运行时刻结构是由快速变化的通信

对象网络组成。事实上两个结构是彼此独立的，试图由一个去理解另一个就好像试图从静态

的动、植物分类去理解活生生的生态系统的动态性。反之亦然。

考虑对象聚合( a g g r e g a t i o n )和相识( a c q u a i n t a n c e )的差别以及它们在编译和运行时刻的表示

是多么的不同。聚合意味着一个对象拥有另一个对象或对另一个对象负责。一般我们称一个

对象包含另一个对象或者是另一个对象的一部分。聚合意味着聚合对象和其所有者具有相同

的生命周期。

相识意味着一个对象仅仅知道另一个对象。有时相识也被称为“关联”或“引用”关系。

相识的对象可能请求彼此的操作，但是它们不为对方负责。相识是一种比聚合要弱的关系，

它只标识了对象间较松散的耦合关系。

在下图中，普通的箭头线表示相识，尾部带有菱形的箭头线表示聚合：

聚合和相识很容易混淆，因为它们通常以相同的方法实现。 S m a l l t a l k中，所有变量都是

其他对象的引用，程序设计语言中两者并无区别。 C + +中，聚合可以通过定义表示真正实例

的成员变量来实现，但更通常的是将这些成员变量定义为实例指针或引用；相识也是以指针

或引用来实现。

从根本上讲，是聚合还是相识是由你的意图而不是由显式的语言机制决定的。尽管它们

之间的区别在编译时刻的结构中很难看出来，但这些区别还是很大的。聚合关系使用较少且

比相识关系更持久；而相识关系则出现频率较高，但有时只存在于一个操作期间，相识也更

具动态性，使得它在源代码中更难被辨别出来。

程序的运行时刻结构和编译时刻结构存在这么大的差别，很明显代码不可能揭示关于系

统如何工作的全部。系统的运行时刻结构更多地受到设计者的影响，而不是编程语言。对象

和它们的类型之间的关系必须更加仔细地设计，因为它们决定了运行时刻程序结构的好坏。

许多设计模式 (特别是那些属于对象范围的 )显式地记述了编译时刻和运行时刻结构的差

别。C o m p o s i t e ( 4 . 3 )和D e c o r a t o r ( 4 . 4 )对于构造复杂的运行时刻结构特别有用。 O b s e r v e r ( 5 . 7 )也

与运行时刻结构有关，但这些结构对于不了解该模式的人来说是很难理解的。 Chain of

R e s p o n s i b i l i t y ( 5 . 1 )也产生了继承所无法展现的通信模式。总之，只有理解了模式，你才能清

楚代码中的运行时刻结构。

设计应支持变化

获得最大限度复用的关键在于对新需求和已有需求发生变化时的预见性，要求你的系统

设计要能够相应地改进。

为了设计适应这种变化、且具有健壮性的系统，你必须考虑系统在它的生命周期内会发

生怎样的变化。一个不考虑系统变化的设计在将来就有可能需要重新设计。这些变化可能是

类的重新定义和实现，修改客户和重新测试。重新设计会影响软件系统的许多方面，并且未

曾料到的变化总是代价巨大的。

设计模式可以确保系统能以特定方式变化，从而帮助你避免重新设计系统。每一个设计

模式允许系统结构的某个方面的变化独立于其他方面，这样产生的系统对于某一种特殊变化

将更健壮。

下面阐述了一些导致重新设计的一般原因，以及解决这些问题的设计模式：

1) 通过显式地指定一个类来创建对象 在创建对象时指定类名将使你受特定实现的约束

而不是特定接口的约束。这会使未来的变化更复杂。要避免这种情况，应该间接地创建对象。

设计模式：Abstract Factory(3.1)，Factory Method(3.3)，P r o t o t y p e ( 3 . 4 )。

2) 对特殊操作的依赖 当你为请求指定一个特殊的操作时，完成该请求的方式就固定下

来了。为避免把请求代码写死，你将可以在编译时刻或运行时刻很方便地改变响应请求的方

法。

设计模式：Chain of Resposibility(5.1)，C o m m a n d ( 5 . 2 )。

3) 对硬件和软件平台的依赖 外部的操作系统接口和应用编程接口 ( A P I )在不同的软硬件

平台上是不同的。依赖于特定平台的软件将很难移植到其他平台上，甚至都很难跟上本地平

台的更新。所以设计系统时限制其平台相关性就很重要了。

设计模式：Abstract Factory(3.1)，B r i d g e ( 4 . 2 )。

4) 对对象表示或实现的依赖 知道对象怎样表示、保存、定位或实现的客户在对象发生

变化时可能也需要变化。对客户隐藏这些信息能阻止连锁变化。

设计模式：Abstract Factory(3.1)，B r i d g e ( 4 . 2 )，M e m e n t o ( 5 . 6 )，P r o x y ( 4 . 7 )

5) 算法依赖 算法在开发和复用时常常被扩展、优化和替代。依赖于某个特定算法的对

象在算法发生变化时不得不变化。因此有可能发生变化的算法应该被孤立起来。

设计模式： B u i l d e r ( 3 . 2 )，I t e r a t o r ( 5 . 4 )，S t r a t e g y ( 5 . 9 )，Template Method(5.10)，

Vi s i t o r ( 5 . 11 )

6) 紧耦合 紧耦合的类很难独立地被复用，因为它们是互相依赖的。紧耦合产生单块的

系统，要改变或删掉一个类，你必须理解和改变其他许多类。这样的系统是一个很难学习、

移植和维护的密集体。

松散耦合提高了一个类本身被复用的可能性，并且系统更易于学习、移植、修改和扩展。

设计模式使用抽象耦合和分层技术来提高系统的松散耦合性。

设计模式： Abstract Factory(3.1)，C o m m a n d ( 5 . 2 )，F a c a d e ( 4 . 5 )，M e d i a t o r ( 5 . 5 )，

Observer(5.7) ，Chain of Responsibility(5.1)。

7) 通过生成子类来扩充功能 通常很难通过定义子类来定制对象。每一个新类都有固定

的实现开销(初始化、终止处理等)。定义子类还需要对父类有深入的了解。如，重定义一个操

作可能需要重定义其他操作。一个被重定义的操作可能需要调用继承下来的操作。并且子类

方法会导致类爆炸，因为即使对于一个简单的扩充，你也不得不引入许多新的子类。

一般的对象组合技术和具体的委托技术，是继承之外组合对象行为的另一种灵活方法。

新的功能可以通过以新的方式组合已有对象，而不是通过定义已存在类的子类的方式加到应

用中去。另一方面，过多使用对象组合会使设计难于理解。许多设计模式产生的设计中，你

可以定义一个子类，且将它的实例和已存在实例进行组合来引入定制的功能。

设计模式：B r i d g e ( 4 . 2 )，Chain of Responsibility(5.1)，C o m p o s i t e ( 4 . 3 )，D e c o r a t o r ( 4 . 4 )，

O b s e r v e r ( 5 . 7 )，S t r a t e g y ( 5 . 9 )。

8) 不能方便地对类进行修改 有时你不得不改变一个难以修改的类。也许你需要源代码

而又没有(对于商业类库就有这种情况)，或者可能对类的任何改变会要求修改许多已存在的其

他子类。设计模式提供在这些情况下对类进行修改的方法。

设计模式：A d a p t e r ( 4 . 1 )，D e c o r a t o r ( 4 . 4 )，Vi s i t o r ( 5 . 11 )。

这些例子反映了使用设计模式有助于增强软件的灵活性。这种灵活性所具有的重要程度

取决于你将要建造的软件系统。让我们看一看设计模式在开发如下三类主要软件中所起的作

用：应用程序、工具箱和框架。

1. 应用程序

如果你将要建造像文档编辑器或电子制表软件这样的应用程序 (Application Program)，那

么它的内部复用性、可维护性和可扩充性是要优先考虑的。内部复用性确保你不会做多余的

设计和实现。设计模式通过减少依赖性来提高内部复用性。松散耦合也增强了一类对象与其

他多个对象协作的可能性。例如，通过孤立和封装每一个操作，以消除对特定操作的依赖，可使在不同上下文中复用一个操作变得更简单。消除对算法和表示的依赖可达到同样的效果。

当设计模式被用来对系统分层和限制对平台的依赖性时，它们还会使一个应用更具可维

护性。通过显示怎样扩展类层次结构和怎样使用对象复用，它们可增强系统的易扩充性。同

时，耦合程度的降低也会增强可扩充性。如果一个类不过多地依赖其他类，扩充这个孤立的

类还是很容易的。

2. 工具箱

一个应用经常会使用来自一个或多个被称为工具箱( To o l k i t )的预定义类库中的类。工具箱

是一组相关的、可复用的类的集合，这些类提供了通用的功能。工具箱的一个典型例子就是

列表、关联表单、堆栈等类的集合， C + +的I / O流库是另一个例子。工具箱并不强制应用采用

某个特定的设计，它们只是为你的应用提供功能上的帮助。工具箱强调的是代码复用，它们

是面向对象环境下的“子程序库”。

工具箱的设计比应用设计要难得多，因为它要求对许多应用是可用的和有效的。再者，

工具箱的设计者并不知道什么应用使用该工具箱及它们有什么特殊需求。这样，避免假设和

依赖就变得很重要，否则会限制工具箱的灵活性，进而影响它的适用性和效率。

3. 框架

框架( F r a m e w o r k )是构成一类特定软件可复用设计的一组相互协作的类 [ D e u 8 9 , J F 8 8 ]。例

如，一个框架能帮助建立适合不同领域的图形编辑器，像艺术绘画、音乐作曲和机械

C A D [ V L 9 0 , J o h 9 2 ]。另一个框架也许能帮助你建立针对不同程序设计语言和目标机器的编译

器[ J M L 9 2 ]。而再一个也许能帮助你建立财务建模应用 [ B E 9 3 ]。你可以定义框架抽象类的应用

相关的子类，从而将一个框架定制为特定应用。

框架规定了你的应用的体系结构。它定义了整体结构，类和对象的分割，各部分的主要

责任，类和对象怎么协作，以及控制流程。框架预定义了这些设计参数，以便于应用设计者

或实现者能集中精力于应用本身的特定细节。框架记录了其应用领域的共同的设计决策。因

而框架更强调设计复用，尽管框架常包括具体的立即可用的子类。

这个层次的复用导致了应用和它所基于的软件之间的反向控制 (inversion of control)。当

你使用工具箱(或传统的子程序库)时，你需要写应用软件的主体并且调用你想复用的代码。而

当你使用框架时，你应该复用应用的主体，写主体调用的代码。你不得不以特定的名字和调

用约定来写操作地实现，但这会减少你需要做出的设计决策。

你不仅可以更快地建立应用，而且应用还具有相似的结构。它们很容易维护，且用户看

来也更一致。另一方面，你也失去了一些表现创造性的自由，因为许多设计决策无须你来作

出。

如果说应用程序难以设计，那么工具箱就更难了，而框架则是最难的。框架设计者必须

冒险决定一个要适应于该领域的所有应用的体系结构。任何对框架设计的实质性修改都会大

大降低框架所带来的好处，因为框架对应用的最主要贡献在于它所定义的体系结构。因此设

计的框架必须尽可能地灵活、可扩充。

更进一步讲，因为应用的设计如此依赖于框架，所以应用对框架接口的变化是极其敏感

的。当框架演化时，应用不得不随之演化。这使得松散耦合更加重要，否则框架的一个细微

变化都将引起强烈反应。

刚才讨论的主要设计问题对框架设计而言最具重要性。一个使用设计模式的框架比不用设计模式的框架更可能获得高层次的设计复用和代码复用。成熟的框架通常使用了多种设计

模式。设计模式有助于获得无须重新设计就可适用于多种应用的框架体系结构。

当框架和它所使用的设计模式一起写入文档时，我们可以得到另外一个好处 [ B J 9 4 ]。了解

设计模式的人能较快地洞悉框架。甚至不了解设计模式的人也可以从产生框架文档的结构中

受益。加强文档工作对于所有软件而言都是重要的，但对于框架其重要性显得尤为突出。学

会使用框架常常是一个必须克服很多困难的过程。设计模式虽然无法彻底克服这些困难，但

它通过对框架设计的主要元素做更显式的说明可以降低框架学习的难度。

因为模式和框架有些类似，人们常常对它们有怎样的区别和它们是否有区别感到疑惑。

它们最主要的不同在于如下三个方面：

1) 设计模式比框架更抽象 框架能够用代码表示，而设计模式只有其实例才能表示为代

码。框架的威力在于它们能够使用程序设计语言写出来，它们不仅能被学习，也能被直接执

行和复用。而本书中的设计模式在每一次被复用时，都需要被实现。设计模式还解释了它的

意图、权衡和设计效果。

2) 设计模式是比框架更小的体系结构元素 一个典型的框架包括了多个设计模式，而反

之决非如此。

3) 框架比设计模式更加特例化 框架总是针对一个特定的应用领域。一个图形编辑器框

架可能被用于一个工厂模拟，但它不会被错认为是一个模拟框架。而本书收录的设计模式几

乎能被用于任何应用。当然比我们的模式更特殊的设计模式也是可能的 (如，分布式系统和并

发程序的设计模式)，尽管这些模式不会像框架那样描述应用的体系结构。

框架变得越来越普遍和重要。它们是面向对象系统获得最大复用的方式。较大的面向对

象应用将会由多层彼此合作的框架组成。应用的大部分设计和代码将来自于它所使用的框架

或受其影响。

浏览模式的意图部分

Abstract Factory( 3 . 1 )：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它

们具体的类。

A d a p t er ( 4 . 1 )：将一个类的接口转换成客户希望的另外一个接口。 A d a p t e r模式使得原本

由于接口不兼容而不能一起工作的那些类可以一起工作。

B r i d g e( 4 . 2 )：将抽象部分与它的实现部分分离，使它们都可以独立地变化。

B u i l d e r( 3 . 2 )：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。

Chain of Responsibility( 5 . 1 )：为解除请求的发送者和接收者之间耦合，而使多个对象都

有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象

处理它。

C o m m a n d( 5 . 2 )：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数

化；对请求排队或记录请求日志，以及支持可取消的操作。

C o m p o s i t e( 4 . 3 )：将对象组合成树形结构以表示“部分 -整体”的层次结构。C o m p o s i t e使

得客户对单个对象和复合对象的使用具有一致性。

D e c o r a t o r( 4 . 4 )：动态地给一个对象添加一些额外的职责。就扩展功能而言， D e c o r a t o r模

式比生成子类方式更为灵活。

F a c a d e( 4 . 5 )：为子系统中的一组接口提供一个一致的界面， F a c a d e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。

Factory Method( 3 . 3 )：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。

Factory Method使一个类的实例化延迟到其子类。

F l y w e i g h t( 4 . 6 )：运用共享技术有效地支持大量细粒度的对象。

I n t e r p r e t e r( 5 . 3 )：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 , 该解释

器使用该表示来解释语言中的句子。

I t e r a t o r( 5 . 4 )：提供一种方法顺序访问一个聚合对象中各个元素 , 而又不需暴露该对象的

内部表示。

M e d i a t o r( 5 . 5 )：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式

地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

M e m e n t o( 5 . 6 )：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外

保存这个状态。这样以后就可将该对象恢复到保存的状态。

O b s e r v e r( 5 . 7 )：定义对象间的一种一对多的依赖关系 ,以便当一个对象的状态发生改变时 ,

所有依赖于它的对象都得到通知并自动刷新。

P r o t o t y p e( 3 . 4 )：用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对

象。

P r o x y( 4 . 7 )：为其他对象提供一个代理以控制对这个对象的访问。

S i n g l e t o n( 3 . 5 )：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

S t a t e( 5 . 8 )：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

所属的类。

S t r a t e g y (5 . 9 )：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模

式使得算法的变化可独立于使用它的客户。

Template Method( 5 . 1 0 )：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。

Template Method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。

Vi s i t o r( 5 . 11 )：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元

素的类的前提下定义作用于这些元素的新操作。

考虑你的设计中哪些是可变的

这个方法与关注引起重新设计的原因刚好相反。它不是

考虑什么会迫使你的设计改变，而是考虑你想要什么变化却又不会引起重新设计。最主

要的一点是封装变化的概念，这是许多设计模式的主题。

目 的 设 计 模 式 可变的方面

创 建 Abstract Factory(3.1) 对产品对象家族

B u i l d e r ( 3 . 2 ) 对如何创建一个组合对象

Factory Method(3.3) 对被实例化的子类

P r o t o t y p e ( 3 . 4 ) 对被实例化的类

S i n g l e t o n ( 3 . 5 ) 对一个类的唯一实例

结构 A d a p t e r ( 4 . 1 ) 对对象的接口

B r i d g e ( 4 . 2 ) 对对象的实现

C o m p o s i t e ( 4 . 3 ) 对一个对象的结构和组成

D e c o r a t o r ( 4 . 4 ) 对对象的职责，不生成子类

F a c a d e ( 4 . 5 ) 对一个子系统的接口

F l y w e i g h t ( 4 . 6 ) 对对象的存储开销

P r o x y ( 4 . 7 ) 对如何访问一个对象；该对象的位置

行 为 Chain of Responsibility(5.1) 对满足一个请求的对象

C o m m a n d ( 5 . 2 ) 对何时、怎样满足一个请求

I n t e r p r e t e r ( 5 . 3 ) 对一个语言的文法及解释

I t e r a t o r ( 5 . 4 ) 对如何遍历、访问一个聚合的各元素

M e d i a t o r ( 5 . 5 ) 对对象间怎样交互、和谁交互

M e m e n t o ( 5 . 6 ) 对一个对象中哪些私有信息存放在该对象之外，以及在

对什么时候进行存储

O b s e r v e r ( 5 . 7 ) 对多个对象依赖于另外一个对象，而这些对象又如何保

持一致

S t a t e ( 5 . 8 ) 对对象的状态

S t r a t e g y ( 5 . 9 ) 对算法

Template Method(5.10) 对算法中的某些步骤

Vi s i t o r ( 5 . 11 ) 对某些可作用于一个（组）对象上的操作，但不修改这

些对象的类

创 建 型

在这些模式中有两个不断出现的主旋律。第一，它们都将关于该系统使用哪些具体的类

的信息封装起来。第二，它们隐藏了这些类的实例是如何被创建和放在一起的。整个系统关

于这些对象所知道的是由抽象类所定义的接口。因此，创建型模式在什么被创建，谁创建它，

它是怎样被创建的，以及何时创建这些方面给予你很大的灵活性。它们允许你用结构和功能

差别很大的“产品”对象配置一个系统。配置可以是静态的（即在编译时指定），也可以是动

态的（在运行时）。

类

Factory Method

代码示例

public class StaticFactoryMethod {

static Sender mailSenderFactory(){

return new MailSender();

}

static Sender smsSenderFactory(){

return new SmsSender();

}

}

public class Cilent {

public static void main(String[] args) {

Sender mailSender = StaticFactoryMethod.mailSenderFactory();

Sender smsSender = StaticFactoryMethod.smsSenderFactory();

mailSender.send();

smsSender.send();

}

}

对象

Abstract Factory

ABSTRACT FACTORY（抽象工厂） — 对象创建型模式

意 图

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类

别 名

K i t

参与者

A b s t r a c t F a c t o r y

声明一个创建抽象产品对象的操作接口

C o n c r e t e F a c t o r y

实现创建具体产品对象的操作

A b s t r a c t P r o d u c t

为一类产品对象声明一个接口

协 作

通常在运行时刻创建一个C o n c r e t e F a c t r o y类的实例。这一具体的工厂创建具有特定实现的产品对象。为创建不同的产品对象，客户应使用不同的具体工厂。

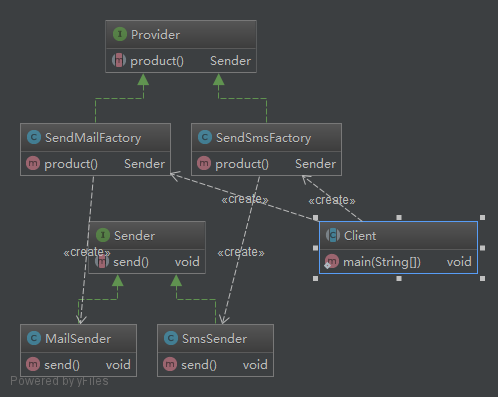
AbstractFactory将产品对象的创建延迟到它的C o n c r e t e F a c t o r y子类

实 现

将工厂作为单件

一个应用中一般每个产品系列只需一个 C o n c r e t e F a c t o r y的实例。因此工厂通常最好实现为一个S i n g l e t o n

代码示例



public abstract class Client {

public static void main(String[] args) {

Sender mailSender = new SendMailFactory().product();

Sender smsSender = new SendSmsFactory().product();

mailSender.send();

smsSender.send();

}

}

public class MailSender implements Sender {

public void send() {

System.out.println("this is a mailSender!");

}

}

public interface Provider {

Sender product();

}

public interface Provider {

Sender product();

}

public class SendMailFactory implements Provider {

public Sender product() {

return new MailSender();

}

}

public class SendSmsFactory implements Provider{

public Sender product() {

return new SmsSender();

}

}

public class SmsSender implements Sender {

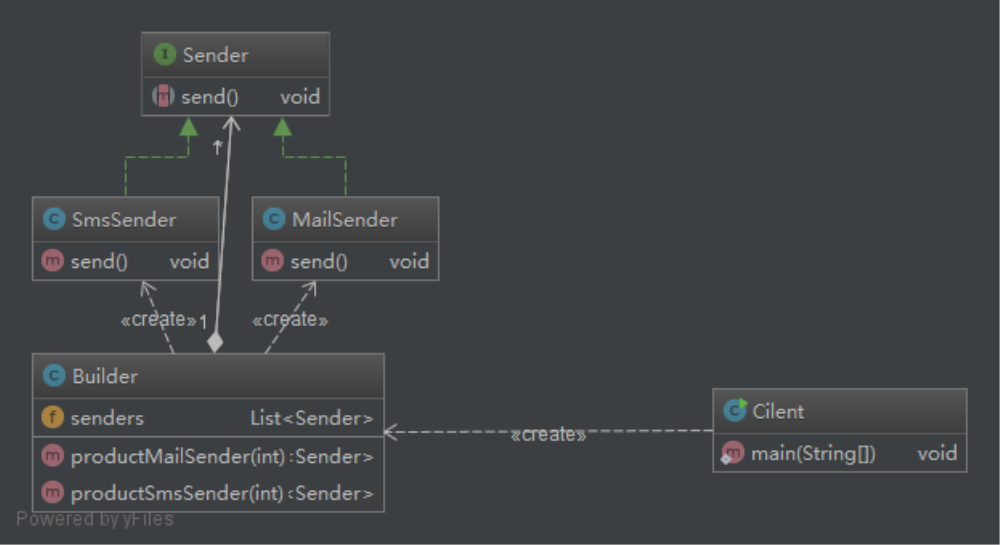
public void send() {

System.out.println("this is a SmsSender");

}

}

B u i l d e r

  
代码示例

public class Builder {

private List<Sender> senders = new ArrayList<Sender>();

List<Sender> productMailSender(int count){

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

senders.add(new MailSender());

}

return senders;

}

List<Sender> productSmsSender(int count){

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

senders.add(new SmsSender());

}

return senders;

}

public class Cilent {

public static void main(String[] args) {

Builder builder = new Builder();

List<Sender> senders = builder.productMailSender(3);

for (Sender sender : senders) {

sender.send();

}

}

}

相关模式

Abstract Factory（3 . 1）与B u i l d e r相似，因为它也可以创建复杂对象。主要的区别是

B u i l d e r模式着重于一步步构造一个复杂对象。而 Abstract Factory着重于多个系列的产品对象

（简单的或是复杂的）。B u i l d e r在最后的一步返回产品，而对于 Abstract Factory来说，产品是

立即返回的。

C o m p o s i t e（4 . 3）通常是用B u i l d e r生成的

P r o t o t y p e

效 果

运行时刻增加和删除产品

P r o t o t y p e允许只通过客户注册原型实例就可以将一个新的

具体产品类并入系统。它比其他创建型模式更为灵活，因为客户可以在运行时刻建立和删除

原型。

改变值以指定新对象

高度动态的系统允许你通过对象复合定义新的行为 — 例如，通

过为一个对象变量指定值 — 并且不定义新的类。你通过实例化已有类并且将这些实例注册

为客户对象的原型，就可以有效定义新类别的对象。客户可以将职责代理给原型，从而表现

出新的行为。这种设计使得用户无需编程即可定义新“类”。实际上，克隆一个原型类似于实例化一个

类。P r o t o t y p e模式可以极大的减少系统所需要的类的数目。在我们的音乐编辑器中，一个

G r a p h i c To o l类可以创建无数种音乐对象。

改变结构以指定新对象

许多应用由部件和子部件来创建对象。例如电路设计编辑器

就是由子电路来构造电路的 。为方便起见，这样的应用通常允许你实例化复杂的、用户定

义的结构，比方说，一次又一次的重复使用一个特定的子电路。

P r o t o t y p e模式也支持这一点。我们仅需将这个子电路作为一个原型增加到可用的电路元

素选择板中。只要复合电路对象将 C l o n e实现为一个深拷贝（deep copy），具有不同结构的电

路就可以是原型了。

减少子类的构造

Factory Method（3 . 3）经常产生一个与产品类层次平行的 C r e a t o r类

层次。P r o t o t y p e模式使得你克隆一个原型而不是请求一个工厂方法去产生一个新的对象。因

此你根本不需要 C r e a t o r类层次。这一优点主要适用于像 C + +这样不将类作为一级类对象的语

言。像S m a l l t a l k和Objective C这样的语言从中获益较少，因为你总是可以用一个类对象作为

生成者。在这些语言中，类对象已经起到原型一样的作用了。

用类动态配置应用

一些运行时刻环境允许你动态将类装载到应用中。在像 C + +这样的

语言中，P r o t o t y p e模式是利用这种功能的关键。

一个希望创建动态载入类的实例的应用不能静态引用类的构造器。而应该由运行环境在

载入时自动创建每个类的实例，并用原型管理器来注册这个实例（参见实现一节）。这样应用

就可以向原型管理器请求新装载的类的实例，这些类原本并没有和程序相连接。 E T + +应用框

架[ W G M 8 8 ]有一个运行系统就是使用这一方案的。

是每一个P r o t o t y p e的子类都必须实现C l o n e操作

P r o t o t y p e的主要缺陷是每一个P r o t o t y p e的子类都必须实现C l o n e操作，这可能很困难。例

如，当所考虑的类已经存在时就难以新增 C l o n e操作。当内部包括一些不支持拷贝或有循环引

用的对象时，实现克隆可能也会很困难的。

实 现

当一个系统中原型数目不固定时（也就是说，它们可以动态创

建和销毁），要保持一个可用原型的注册表。客户不会自己来管理原型，但会在注册表中存储

和检索原型。客户在克隆一个原型前会向注册表请求该原型。我们称这个注册表为原型管理

器（prototype manager）。

原型管理器是一个关联存储器（ associative store），它返回一个与给定关键字相匹配的原

型。它有一些操作可以用来通过关键字注册原型和解除注册。客户可以在运行时更改甚或浏

览这个注册表。这使得客户无需编写代码就可以扩展并得到系统清单。

实现克隆操作

P r o t o t y p e模式最困难的部分在于正确实现 C l o n e操作。当对象结构包含

循环引用时，这尤为棘手。

大多数语言都对克隆对象提供了一些支持。例如， S m a l l t a l k提供了一个c o p y的实现，它

被所有O b j e c t的子类所继承。C + +提供了一个拷贝构造器。但这些设施并不能解决“浅拷贝和

深拷贝”问题[ G R 8 3 ]。也就是说，克隆一个对象是依次克隆它的实例变量呢，或者还是由克

隆对象和原对象共享这些变量？

浅拷贝简单并且通常也足够了，它是 S m a l l t a l k所缺省提供的。C + +中的缺省拷贝构造器实

现按成员拷贝，这意味着在拷贝的和原来的对象之间是共享指针的。但克隆一个结构复杂的

原型通常需要深拷贝，因为复制对象和原对象必须相互独立。因此你必须保证克隆对象的构

件也是对原型的构件的克隆。克隆迫使你决定如果所有东西都被共享了该怎么办。

如果系统中的对象提供了 S a v e和L o a d操作，那么你只需通过保存对象和立刻载入对象，

就可以为C l o n e操作提供一个缺省实现。 S a v e操作将该对象保存在内存缓冲区中，而 L o a d则通

过从该缓冲区中重构这个对象来创建一个复本。

初始化克隆对象

当一些客户对克隆对象已经相当满意时，另一些客户将会希望使用

他们所选择的一些值来初始化该对象的一些或是所有的内部状态。一般来说不可能在 C l o n e操

作中传递这些值，因为这些值的数目由于原型的类的不同而会有所不同。一些原型可能需要

多个初始化参数，另一些可能什么也不要。在 C l o n e操作中传递参数会破坏克隆接口的统一

性。

可能会这样，原型的类已经为（重）设定一些关键的状态值定义好了操作。如果这样的

话，客户在克隆后马上就可以使用这些操作。否则，你就可能不得不引入一个 I n i t i a l i z e操作

（参见代码示例一节），该操作使用初始化参数并据此设定克隆对象的内部状态。注意深拷贝

C l o n e操作 — 一些复制在你重新初始化它们之前可能必须要被删除掉（删除可以显式地做也

可以在I n i t i a l i z e内部做）。

代码示例

public class Prototype implements Cloneable,Serializable{

private static final long serialVersionUID = 1L;

private String string;

private SerializableObject serializableObject;

public Object clone() throws CloneNotSupportedException {

return (Prototype)super.clone();

}

public Object deepClone() throws IOException, ClassNotFoundException {

ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();

ObjectOutputStream objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);

objectOutputStream.writeObject(this);

ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(byteArrayOutputStream.toByteArray());

ObjectInputStream objectInputStream = new ObjectInputStream(byteArrayInputStream);

return objectInputStream.readObject();

}

public String getString() {

return string;

}

public Prototype setString(String string) {

this.string = string;

return this;

}

public SerializableObject getSerializableObject() {

return serializableObject;

}

public Prototype setSerializableObject(SerializableObject serializableObject) {

this.serializableObject = serializableObject;

return this;

}

}

S i n g l e t o n

效 果

因为S i n g l e t o n类封装它的唯一实例，所以它可以严格的控制客户怎样以及何时访问它。

缩小名空间

S i n g l e t o n模式是对全局变量的一种改进。它避免了那些存储唯一实例的

全局变量污染名空间。

允许对操作和表示的精化

S i n g l e t o n类可以有子类，而且用这个扩展类的实例来配置

一个应用是很容易的。你可以用你所需要的类的实例在运行时刻配置应用。

允许可变数目的实例

这个模式使得你易于改变你的想法，并允许 S i n g l e t o n类的多个

实例。此外，你可以用相同的方法来控制应用所使用的实例的数目。只有允许访问 S i n g l e t o n

实例的操作需要改变。

比类操作更灵活

另一种封装单件功能的方式是使用类操作（即 C + +中的静态成员函数

或者是S m a l l t a l k中的类方法）。但这两种语言技术都难以改变设计以允许一个类有多个实例。

此外，C + +中的静态成员函数不是虚函数，因此子类不能多态的重定义它们。

实 现

S i n g l e t o n模式使得这个唯一实例是类的一般实例，但该类被写

成只有一个实例能被创建。做到这一点的一个常用方法是将创建这个实例的操作隐藏在一个

类操作（即一个静态成员函数或者是一个类方法）后面，由它保证只有一个实例被创建。这

个操作可以访问保存唯一实例的变量，而且它可以保证这个变量在返回值之前用这个唯一实

例初始化。这种方法保证了单件在它的首次使用前被创建和使用。

创建S i n g l e t o n类的子类

主要问题与其说是定义子类不如说是建立它的唯一实例，这

样客户就可以使用它。事实上，指向单件实例的变量必须用子类的实例进行初始化。最简单

的技术是在S i n g l e t o n的I n s t a n c e操作中决定你想使用的是哪一个单件。代码示例一节中的一个

例子说明了如何用环境变量实现这一技术。

代码示例

首先我们写一个简单的单例类：

public class Singleton {

/\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/

private static Singleton instance = null;

/\* 私有构造方法，防止被实例化 \*/

private Singleton() {

}

/\* 静态工程方法，创建实例 \*/

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

/\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/

public Object readResolve() {

return instance;

}

}

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

[java] view plaincopy

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

[java] view plaincopy

public static Singleton getInstance() {

if (instance == null) {

synchronized (instance) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

似乎解决了之前提到的问题，将synchronized关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创 建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在Java指令中创建对象和赋值操作是分开进行的，也 就是说instance = new Singleton();语句是分两步执行的。但是JVM并不保证这两个操作的先后顺序，也就是说有可能JVM会为新的Singleton实例分配空间， 然后直接赋值给instance成员，然后再去初始化这个Singleton实例。这样就可能出错了，我们以A、B两个线程为例：

a>A、B线程同时进入了第一个if判断

b>A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton();

c>由于JVM内部的优化机制，JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance成员（注意此时JVM没有开始初始化这个实例），然后A离开了synchronized块。

d>B进入synchronized块，由于instance此时不是null，因此它马上离开了synchronized块并将结果返回给调用该方法的程序。

e>此时B线程打算使用Singleton实例，却发现它没有被初始化，于是错误发生了。

所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：

private static class SingletonFactory{

private static Singleton instance = new Singleton();

}

public static Singleton getInstance(){

return SingletonFactory.instance;

}

实际情况是，单例模式使用内部类来维护单例的实现，JVM内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用 getInstance的时候，JVM能够帮我们保证instance只被创建一次，并且会保证把赋值给instance的内存初始化完毕，这样我们就不 用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式：

[java] view plaincopy

public class Singleton {

/\* 私有构造方法，防止被实例化 \*/

private Singleton() {

}

/\* 此处使用一个内部类来维护单例 \*/

private static class SingletonFactory {

private static Singleton instance = new Singleton();

}

/\* 获取实例 \*/

public static Singleton getInstance() {

return SingletonFactory.instance;

}

/\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/

public Object readResolve() {

return getInstance();

}

}

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适 合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加 synchronized关键字，也是可以的：

[java] view plaincopy

public class SingletonTest {

private static SingletonTest instance = null;

private SingletonTest() {

}

private static synchronized void syncInit() {

if (instance == null) {

instance = new SingletonTest();

}

}

public static SingletonTest getInstance() {

if (instance == null) {

syncInit();

}

return instance;

}

}

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。

补充：采用"影子实例"的办法为单例对象的属性同步更新

[java] view plaincopy

public class SingletonTest {

private static SingletonTest instance = null;

private Vector properties = null;

public Vector getProperties() {

return properties;

}

private SingletonTest() {

}

private static synchronized void syncInit() {

if (instance == null) {

instance = new SingletonTest();

}

}

public static SingletonTest getInstance() {

if (instance == null) {

syncInit();

}

return instance;

}

public void updateProperties() {

SingletonTest shadow = new SingletonTest();

properties = shadow.getProperties();

}

}

结 构 型

类

A d a p t e r (类)

效 果

使用A d a p t e r模式时需要考虑的其他一些因素有：

Adapter的匹配程度

对A d a p t e e的接口与Ta rg e t的接口进行匹配的工作量各个A d a p t e r可

能不一样。工作范围可能是，从简单的接口转换(例如改变操作名 )到支持完全不同的操作集

合。A d a p t e r的工作量取决于Ta rg e t接口与A d a p t e e接口的相似程度。

可插入的Adapter

当其他的类使用一个类时，如果所需的假定条件越少，这个类就更

具可复用性。如果将接口匹配构建为一个类，就不需要假定对其他的类可见的是一个相同的

接口。也就是说，接口匹配使得我们可以将自己的类加入到一些现有的系统中去，而这些系

统对这个类的接口可能会有所不同。 O b j e c t - Wo r k / S m a l l t a l k [ P a r 9 0 ]使用pluggable adapter一词

描述那些具有内部接口适配的类。

使用双向适配器提供透明操作

使用适配器的一个潜在问题是，它们不对所有的客户

都透明。被适配的对象不再兼容 A d a p t e e的接口，因此并不是所有 A d a p t e e对象可以被使用的

地方它都可以被使用。双向适配器提供了这样的透明性。在两个不同的客户需要用不同的方

式查看同一个对象时，双向适配器尤其有用。

代码示例

类

public class Source {

public void method1() {

System.out.println("this is original method!");

}

}

public interface Targetable {

/\* 与原类中的方法相同 \*/

public void method1();

/\* 新类的方法 \*/

public void method2();

}

public class Adapter extends Source implements Targetable {

@Override

public void method2() {

System.out.println("this is the targetable method!");

}

}

Adapter类继承Source类，实现Targetable接口，下面是测试类：

public class AdapterTest {

public static void main(String[] args) {

Targetable target = new Adapter();

target.method1();

target.method2();

}

}

对象

结构型模式涉及到如何组合类和对象以获得更大的结构。结构型类模式采用继承机制来

组合接口或实现。一个简单的例子是采用多重继承方法将两个以上的类组合成一个类，结果

这个类包含了所有父类的性质。这一模式尤其有助于多个独立开发的类库协同工作。另外一

个例子是类形式的A d a p t e r ( 4 . 1 )模式。一般来说，适配器使得一个接口 ( a d a p t e e的接口)与其他

接口兼容，从而给出了多个不同接口的统一抽象。为此，类适配器对一个 a d a p t e e类进行私有

继承。这样，适配器就可以用a d a p t e e的接口表示它的接口。

结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从

而实现新功能的一些方法。因为可以在运行时刻改变对象组合关系，所以对象组合方式具有

更大的灵活性，而这种机制用静态类组合是不可能实现的。

Composite (4.3) 模式是结构型对象模式的一个实例。它描述了如何构造一个类层次式结

构，这一结构由两种类型的对象（基元对象和组合对象）所对应的类构成 . 其中的组合对象使

得你可以组合基元对象以及其他的组合对象，从而形成任意复杂的结构。在 Proxy (4.7) 模式

中，p r o x y对象作为其他对象的一个方便的替代或占位符。它的使用可以有多种形式。例如它

可以在局部空间中代表一个远程地址空间中的对象，也可以表示一个要求被加载的较大的对

象，还可以用来保护对敏感对象的访问。 P r o x y模式还提供了对对象的一些特有性质的一定程

度上的间接访问，从而它可以限制、增强或修改这些性质。

F l y w e i g h t ( 4 . 6 )模式为了共享对象定义了一个结构。至少有两个原因要求对象共享：效率

和一致性。F l y w e i g h t的对象共享机制主要强调对象的空间效率。使用很多对象的应用必需考

虑每一个对象的开销。使用对象共享而不是进行对象复制，可以节省大量的空间资源。但是

仅当这些对象没有定义与上下文相关的状态时，它们才可以被共享。 F l y w e i g h t的对象没有这

样的状态。任何执行任务时需要的其他一些信息仅当需要时才传递过去。由于不存在与上下

文相关的状态，因此F l y w e i g h t对象可以被自由地共享。

如果说F l y w e i g h t模式说明了如何生成很多较小的对象，那么 F a c a d e ( 4 . 5 )模式则描述了如

何用单个对象表示整个子系统。模式中的 f a c a d e用来表示一组对象，f a c a d e的职责是将消息转

发给它所表示的对象。 B r i d g e ( 4 . 2 )模式将对象的抽象和其实现分离，从而可以独立地改变它

们。

D e c o r a t o r ( 4 . 4 )模式描述了如何动态地为对象添加职责。 D e c o r a t o r模式是一种结构型模式。

这一模式采用递归方式组合对象，从而允许你添加任意多的对象职责。例如，一个包含用户

界面组件的D e c o r a t o r对象可以将边框或阴影这样的装饰添加到该组件中，或者它可以将窗口

滚动和缩放这样的功能添加的组件中。我们可以将一个 D e c o r a t o r对象嵌套在另外一个对象中

就可以很简单地增加两个装饰，添加其他的装饰也是如此。因此，每个 D e c o r a t o r对象必须与

其组件的接口兼容并且保证将消息传递给它。 D e c o r a t o r模式在转发一条信息之前或之后都可

以完成它的工作（比如绘制组件的边框）。

A d a p t e r (对象)

效 果

使用A d a p t e r模式时需要考虑的其他一些因素有：

Adapter的匹配程度

对A d a p t e e的接口与Ta rg e t的接口进行匹配的工作量各个A d a p t e r可

能不一样。工作范围可能是，从简单的接口转换(例如改变操作名 )到支持完全不同的操作集

合。A d a p t e r的工作量取决于Ta rg e t接口与A d a p t e e接口的相似程度。

可插入的Adapter

当其他的类使用一个类时，如果所需的假定条件越少，这个类就更

具可复用性。如果将接口匹配构建为一个类，就不需要假定对其他的类可见的是一个相同的

接口。也就是说，接口匹配使得我们可以将自己的类加入到一些现有的系统中去，而这些系

统对这个类的接口可能会有所不同。 O b j e c t - Wo r k / S m a l l t a l k [ P a r 9 0 ]使用pluggable adapter一词

描述那些具有内部接口适配的类。

使用双向适配器提供透明操作

使用适配器的一个潜在问题是，它们不对所有的客户

都透明。被适配的对象不再兼容 A d a p t e e的接口，因此并不是所有 A d a p t e e对象可以被使用的

地方它都可以被使用。双向适配器提供了这样的透明性。在两个不同的客户需要用不同的方

式查看同一个对象时，双向适配器尤其有用。

代码示例

类

public class Source {

public void method1() {

System.out.println("this is original method!");

}

}

public interface Targetable {

/\* 与原类中的方法相同 \*/

public void method1();

/\* 新类的方法 \*/

public void method2();

}

public class Adapter extends Source implements Targetable {

@Override

public void method2() {

System.out.println("this is the targetable method!");

}

}

Adapter类继承Source类，实现Targetable接口，下面是测试类：

public class AdapterTest {

public static void main(String[] args) {

Targetable target = new Adapter();

target.method1();

target.method2();

}

}

B r i d g e

适用性

（C + +）你想对客户完全隐藏抽象的实现部分。在 C + +中，类的表示在类接口中是可见的。

。

效 果

分离接口及其实现部分

一个实现未必不变地绑定在一个接口上。抽象类的实现可以

在运行时刻进行配置，一个对象甚至可以在运行时刻改变它的实现。

将A b s t r a c t i o n与I m p l e m e n t o r分离有助于降低对实现部分编译时刻的依赖性，当改变一个

实现类时，并不需要重新编译 A b s t r a c t i o n类和它的客户程序。为了保证一个类库的不同版本

之间的二进制兼容性，一定要有这个性质。

另外，接口与实现分离有助于分层，从而产生更好的结构化系统，系统的高层部分仅需

知道A b s t r a c t i o n和I m p l e m e n t o r即可。

提高可扩充性

你可以独立地对A b s t r a c t i o n和I m p l e m e n t o r层次结构进行扩充。

实现细节对客户透明

你可以对客户隐藏实现细节，例如共享 I m p l e m e n t o r对象以及相

应的引用计数机制（如果有的话）。

实 现

仅有一个Implementor

在仅有一个实现的时候，没有必要创建一个抽象的 I m p l e m e n t o r

类。这是B r i d g e模式的退化情况；在 A b s t r a c t i o n与I m p l e m e n t o r之间有一种一对一的关系。尽

管如此，当你希望改变一个类的实现不会影响已有的客户程序时，模式的分离机制还是非常

有用的 — 也就是说，不必重新编译它们，仅需重新连接即可。

C a r o l a n [ C a r 8 9 ]用“常露齿嘻笑的猫”（Cheshire Cat）描述这一分离机制。在 C + +中，

I m p l e m e n t o r类的类接口可以在一个私有的头文件中定义，这个文件不提供给客户。这样你就

对客户彻底隐藏了一个类的实现部分。

创建正确的I m p l e m e n t o r对象

当存在多个I m p l e m e n t o r类的时候，你应该用何种方法，

在何时何处确定创建哪一个I m p l e m e n t o r类呢？

如果A b s t r a c t i o n知道所有的C o n c r e t e I m p l e m e n t o r类，它就可以在它的构造器中对其中的

一个类进行实例化，它可以通过传递给构造器的参数确定实例化哪一个类。例如，如果一个c o l l e c t i o n类支持多重实现，就可以根据 c o l l e c t i o n的大小决定实例化哪一个类。链表的实现可

以用于较小的c o l l e c t i o n类，而h a s h表则可用于较大的c o l l e c t i o n类。

另外一种方法是首先选择一个缺省的实现，然后根据需要改变这个实现。例如，如果一

个c o l l e c t i o n的大小超出了一定的阈值时，它将会切换它的实现，使之更适用于表目较多的

c o l l e c t i o n。

也可以代理给另一个对象，由它一次决定。在 Wi n d o w / Wi n d o w I m p的例子中，我们可以

引入一个f a c t o r y对象（参见Abstract Factory(3.1)），该对象的唯一职责就是封装系统平台的细

节。这个对象知道应该为所用的平台创建何种类型的 Wi n d o w I m p对象；Wi n d o w仅需向它请求

一个Wi n d o w I m p，而它会返回正确类型的Wi n d o w I m p对象。这种方法的优点是 Abstraction 类

不和任何一个I m p l e m e n t o r类直接耦合。

代码示例

view plaincopy

public interface Sourceable {

public void method();

}

分别定义两个实现类：

public class SourceSub1 implements Sourceable {

@Override

public void method() {

System.out.println("this is the first sub!");

}

}

public class SourceSub2 implements Sourceable {

@Override

public void method() {

System.out.println("this is the second sub!");

}

}

定义一个桥，持有Sourceable的一个实例：

public abstract class Bridge {

private Sourceable source;

public void method(){

source.method();

}

public Sourceable getSource() {

return source;

}

public void setSource(Sourceable source) {

this.source = source;

}

}

[java] view plaincopy

public class MyBridge extends Bridge {

public void method(){

getSource().method();

}

}

测试类：

[java] view plaincopy

public class BridgeTest {

public static void main(String[] args) {

Bridge bridge = new MyBridge();

/\*调用第一个对象\*/

Sourceable source1 = new SourceSub1();

bridge.setSource(source1);

bridge.method();

/\*调用第二个对象\*/

Sourceable source2 = new SourceSub2();

bridge.setSource(source2);

bridge.method();

}

}

C o m p o s i t e

效 果

定义了包含基本对象和组合对象的类层次结构

基本对象可以被组合成更复杂的组合对

象，而这个组合对象又可以被组合，这样不断的递归下去。客户代码中，任何用到基本

对象的地方都可以使用组合对象。

简化客户代码

客户可以一致地使用组合结构和单个对象。通常用户不知道 (也不关心)

处理的是一个叶节点还是一个组合组件。这就简化了客户代码 , 因为在定义组合的那些

类中不需要写一些充斥着选择语句的函数。

使得更容易增加新类型的组件

新定义的C o m p o s i t e或L e a f子类自动地与已有的结构和客

户代码一起工作，客户程序不需因新的 C o m p o n e n t类而改变。

使你的设计变得更加一般化

容易增加新组件也会产生一些问题，那就是很难限制组合

中的组件。有时你希望一个组合只能有某些特定的组件。使用 C o m p o s i t e时，你不能依

赖类型系统施加这些约束，而必须在运行时刻进行检查。

实 现

实现C o m p o s i t e模式时需要考虑以下几个问题

显式的父部件引用

保持从子部件到父部件的引用能简化组合结构的遍历和管理。父部

件引用可以简化结构的上移和组件的删除，同时父部件引用也支持 Chain of Responsibility(5.2)

模式。

通常在C o m p o n e n t类中定义父部件引用。 L e a f和C o m p o s i t e类可以继承这个引用以及管理

这个引用的那些操作。对于父部件引用，必须维护一个不变式，即一个组合的所有子节点以这个组合为父节点，

而反之该组合以这些节点为子节点。保证这一点最容易的办法是，仅当在一个组合中增加或

删除一个组件时，才改变这个组件的父部件。如果能在 C o m p o s i t e类的Add 和R e m o v e操作中

实现这种方法，那么所有的子类都可以继承这一方法，并且将自动维护这一不变式。

共享组件

共享组件是很有用的，比如它可以减少对存贮的需求。但是当一个组件只

有一个父部件时，很难共享组件。

一个可行的解决办法是为子部件存贮多个父部件，但当一个请求在结构中向上传递时，

这种方法会导致多义性。 F l y w e i g h t ( 4 . 6 )模式讨论了如何修改设计以避免将父部件存贮在一起

的方法。如果子部件可以将一些状态 (或是所有的状态)存储在外部，从而不需要向父部件发送

请求，那么这种方法是可行的。

最大化C o m p o n e n t接口

C o m p o s i t e模式的目的之一是使得用户不知道他们正在使用的

具体的Leaf 和C o m p o s i t e类。为了达到这一目的，C o m p o s i t e类应为Leaf 和C o m p o s i t e类尽可能

多定义一些公共操作。 C o m p o s i t e类通常为这些操作提供缺省的实现，而 Leaf 和C o m p o s i t e子

类可以对它们进行重定义。

然而，这个目标有时可能会与类层次结构设计原则相冲突，该原则规定：一个类只能定

义那些对它的子类有意义的操作。有许多C o m p o n e n t所支持的操作对L e a f类似乎没有什么意义，

那么C o m p o n e n t怎样为它们提供一个缺省的操作呢？

有时一点创造性可以使得一个看起来仅对 C o m p o s i t e才有意义的操作，将它移入

C o m p o n e n t类中，就会对所有的 C o m p o n e n t都适用。例如，访问子节点的接口是 C o m p o s i t e类

的一个基本组成部分，但对 L e a f类来说并不必要。但是如果我们把一个 L e a f看成一个没有子

节点的Component, 就可以为在C o m p o n e n t类中定义一个缺省的操作，用于对子节点进行访问，

这个缺省的操作不返回任何一个子节点。Leaf 类可以使用缺省的实现，而C o m p o s i t e类则会重

新实现这个操作以返回它们的子类。

管理子部件的操作比较复杂，我们将在下一项中予以讨论。

声明管理子部件的操作

虽然C o m p o s i t e类实现了Add 和R e m o v e操作用于管理子部件，

但在C o m p o s i t e模式中一个重要的问题是：在 C o m p o s i t e类层次结构中哪一些类声明这些操作。

我们是应该在C o m p o n e n t中声明这些操作，并使这些操作对 L e a f类有意义呢，还是只应该在

C o m p o s i t e和它的子类中声明并定义这些操作呢？

这需要在安全性和透明性之间做出权衡选择。

• 在类层次结构的根部定义子节点管理接口的方法具有良好的透明性，因为你可以一致地

使用所有的组件，但是这一方法是以安全性为代价的，因为客户有可能会做一些无意义

的事情，例如在Leaf 中增加和删除对象等。

• 在C o m p o s i t e类中定义管理子部件的方法具有良好的安全性，因为在象 C + +这样的静态

类型语言中，在编译时任何从Leaf 中增加或删除对象的尝试都将被发现。但是这又损失

了透明性，因为Leaf 和C o m p o s i t e具有不同的接口。

在这一模式中，相对于安全性，我们比较强调透明性。如果你选择了安全性，有时你可

能会丢失类型信息，并且不得不将一个组件转换成一个组合。这样的类型转换必定不是类型

安全的。

Component是否应该实现一个C o m p o n e n t列表

你可能希望在C o m p o n e n t类中将子节点

集合定义为一个实例变量，而这个 C o m p o n e n t类中也声明了一些操作对子节点进行访问和管理。但是在基类中存放子类指针，对叶节点来说会导致空间浪费，因为叶节点根本没有子节

点。只有当该结构中子类数目相对较少时，才值得使用这种方法。

子部件排序

许多设计指定了C o m p o s i t e的子部件顺序。在前面的 G r a p h i c s例子中，排

序可能表示了从前至后的顺序。如果 C o m p o s i t e表示语法分析树， C o m p o s i t e子部件的顺序必

须反映程序结构，而组合语句就是这样一些 C o m p o s i t e的实例。

如果需要考虑子节点的顺序时，必须仔细地设计对子节点的访问和管理接口，以便管理

子节点序列。I t e r a t o r模式( 5 . 4 )可以在这方面给予一些定的指导。

使用高速缓冲存贮改善性能

如果你需要对组合进行频繁的遍历或查找， C o m p o s i t e类

可以缓冲存储对它的子节点进行遍历或查找的相关信息。 C o m p o s i t e可以缓冲存储实际结果或

者仅仅是一些用于缩短遍历或查询长度的信息。例如，动机一节的例子中 P i c t u r e类能高速缓

冲存贮其子部件的边界框，在绘图或选择期间，当子部件在当前窗口中不可见时，这个边界

框使得P i c t u r e不需要再进行绘图或选择。

一个组件发生变化时，它的父部件原先缓冲存贮的信息也变得无效。在组件知道其父部

件时，这种方法最为有效。因此，如果你使用高速缓冲存贮，你需要定义一个接口来通知组

合组件它们所缓冲存贮的信息无效。

应该由谁删除Component

在没有垃圾回收机制的语言中，当一个 C o m p o s i t e被销毁时，

通常最好由C o m p o s i t e负责删除其子节点。但有一种情况除外，即 L e a f对象不会改变，因此可

以被共享。

存贮组件最好用哪一种数据结构

C o m p o s i t e可使用多种数据结构存贮它们的子节点，

包括连接列表、树、数组和 h a s h表。数据结构的选择取决于效率。事实上，使用通用数据结

构根本没有必要。有时对每个子节点， C o m p o s i t e都有一个变量与之对应，这就要求

C o m p o s i t e的每个子类都要实现自己的管理接口。参见 I n t e r p r e t e r ( 5 . 3 )模式中的例子。

代码示例

public class TreeNode {

private String name;

private TreeNode parent;

private Vector<TreeNode> children = new Vector<TreeNode>();

public TreeNode(String name){

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public TreeNode getParent() {

return parent;

}

public void setParent(TreeNode parent) {

this.parent = parent;

}

//添加孩子节点

public void add(TreeNode node){

children.add(node);

}

//删除孩子节点

public void remove(TreeNode node){

children.remove(node);

}

//取得孩子节点

public Enumeration<TreeNode> getChildren(){

return children.elements();

}

}

[java] view plaincopy

public class Tree {

TreeNode root = null;

public Tree(String name) {

root = new TreeNode(name);

}

public static void main(String[] args) {

Tree tree = new Tree("A");

TreeNode nodeB = new TreeNode("B");

TreeNode nodeC = new TreeNode("C");

nodeB.add(nodeC);

tree.root.add(nodeB);

System.out.println("build the tree finished!");

}

}

相关模式

D e c o r a t o r（4 . 4）模式经常与C o m p o s i t e模式一起使用。

当装饰和组合一起使用时，它们

通常有一个公共的父类。因此装饰必须支持具有 A d d、R e m o v e和GetChild 操作的C o m p o n e n t接口。

D e c o r a t o r

效 果

优点

比静态继承更灵活

与对象的静态继承（多重继承）相比， D e c o r a t o r模式提供了更加

灵活的向对象添加职责的方式。可以用添加和分离的方法，用装饰在运行时刻增加和删除职

责。相比之下，继承机制要求为每个添加的职责创建一个新的子类（例如， B o r d e r S c r o l l a b l e

Te x t Vi e w, BorderedTe x t Vi e w）。这会产生许多新的类，并且会增加系统的复杂度。此外，为一

个特定的C o m p o n e n t类提供多个不同的 D e c o r a t o r类，这就使得你可以对一些职责进行混合和

匹配。

使用D e c o r a t o r模式可以很容易地重复添加一个特性，例如在 Te x t Vi e w上添加双边框时，

仅需将添加两个B o r d e r D e c o r a t o r即可。而两次继承B o r d e r类则极容易出错的。

避免在层次结构高层的类有太多的特征

D e c o r a t o r模式提供了一种“即用即付”的方

法来添加职责。它并不试图在一个复杂的可定制的类中支持所有可预见的特征，相反，你可

以定义一个简单的类，并且用 D e c o r a t o r类给它逐渐地添加功能。可以从简单的部件组合出复

杂的功能。这样，应用程序不必为不需要的特征付出代价。同时也更易于不依赖于 D e c o r a t o r

所扩展（甚至是不可预知的扩展）的类而独立地定义新类型的 D e c o r a t o r。扩展一个复杂类的

时候，很可能会暴露与添加的职责无关的细节。

Decorator与它的C o m p o n e n t不一样

D e c o r a t o r是一个透明的包装。如果我们从对象标

识的观点出发，一个被装饰了的组件与这个组件是有差别的，因此，使用装饰时不应该依赖

对象标识。

缺点

有许多小对象

采用D e c o r a t o r模式进行系统设计往往会产生许多看上去类似的小对象，

这些对象仅仅在他们相互连接的方式上有所不同，而不是它们的类或是它们的属性值有所不

同。尽管对于那些了解这些系统的人来说，很容易对它们进行定制，但是很难学习这些系统，

排错也很困难。

代码示例

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：

Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

[java] view plaincopy

public interface Sourceable {

public void method();

}

[java] view plaincopy

public class Source implements Sourceable {

@Override

public void method() {

System.out.println("the original method!");

}

}

[java] view plaincopy

public class Decorator implements Sourceable {

private Sourceable source;

public Decorator(Sourceable source){

super();

this.source = source;

}

@Override

public void method() {

System.out.println("before decorator!");

source.method();

System.out.println("after decorator!");

}

}

测试类：

[java] view plaincopy

public class DecoratorTest {

public static void main(String[] args) {

Sourceable source = new Source();

Sourceable obj = new Decorator(source);

obj.method();

}

}

输出：

before decorator!

the original method!

after decorator!

装饰器模式的应用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

F a c a d e

适用性

当你要为一个复杂子系统提供一个简单接口时。

子系统往往因为不断演化而变得越来越

复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容

易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。

F a c a d e可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需

要更多的可定制性的用户可以越过 f a c a d e层。

客户程序与抽象类的实现部分之间存在着很大的依赖性。

引入 f a c a d e将这个子系统与客

户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

当你需要构建一个层次结构的子系统时，使用 f a c a d e模式定义子系统中每层的入口点。

如果子系统之间是相互依赖的，你可以让它们仅通过 f a c a d e进行通讯，从而简化了它们

之间的依赖关系。

效 果

它实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。

松耦合关系使得子系统的组件变化不会影响到它的客户。 F a c a d e模式有助于建立层次结构系

统，也有助于对对象之间的依赖关系分层。 F a c a d e模式可以消除复杂的循环依赖关系。这一

点在客户程序与子系统是分别实现的时候尤为重要。

在大型软件系统中降低编译依赖性至关重要。在子系统类改变时，希望尽量减少重编译

工作以节省时间。用 F a c a d e可以降低编译依赖性，限制重要系统中较小的变化所需的重编译

工作。F a c a d e模式同样也有利于简化系统在不同平台之间的移植过程，因为编译一个子系统

一般不需要编译所有其他的子系统。

如果应用需要，它并不限制它们使用子系统类。

因此你可以在系统易用性和通用性之

间加以选择。

实 现

降低客户-子系统之间的耦合度

用抽象类实现F a c a d e而它的具体子类对应于不同的子

系统实现，这可以进一步降低客户与子系统的耦合度。这样，客户就可以通过抽象的 F a c a d e

类接口与子系统通讯。这种抽象耦合关系使得客户不知道它使用的是子系统的哪一个实现。

除生成子类的方法以外，另一种方法是用不同的子系统对象配置 F a c a d e对象。为定制

f a c a d e，仅需对它的子系统对象（一个或多个）进行替换即可。

公共子系统类与私有子系统类

一个子系统与一个类的相似之处是，它们都有接口并

且它们都封装了一些东西 — 类封装了状态和操作，而子系统封装了一些类。考虑一个类的

公共和私有接口是有益的，我们也可以考虑子系统的公共和私有接口。

子系统的公共接口包含所有的客户程序可以访问的类；私有接口仅用于对子系统进行扩

充。当然，F a c a d e类是公共接口的一部分，但它不是唯一的部分，子系统的其他部分通常也

是公共的。例如，编译子系统中的 P a r s e r类和S c a n n e r类就是公共接口的一部分。

私有化子系统类确实有用，但是很少有面向对象的编程语言支持这一点。 C + +和S m a l l t a l k

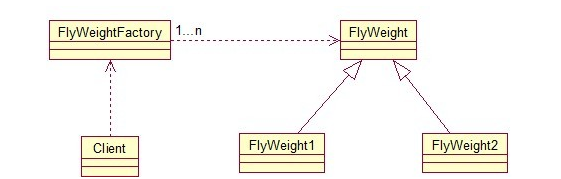
语言仅在传统意义下为类提供了一个全局名空间。然而，最近 C + +标准化委员会在C + +语言中

增加了一些名字空间[ S t r 9 4 ]，这些名字空间使得你可以仅暴露公共子系统类。

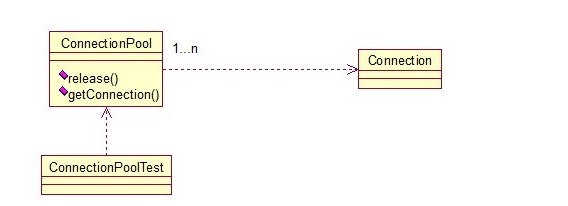
代码示例

外观模式（Facade）

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）



我们先看下实现类：



[java] view plaincopy

public class CPU {

public void startup(){

System.out.println("cpu startup!");

}

public void shutdown(){

System.out.println("cpu shutdown!");

}

}

[java] view plaincopy

public class Memory {

public void startup(){

System.out.println("memory startup!");

}

public void shutdown(){

System.out.println("memory shutdown!");

}

}

[java] view plaincopy

public class Disk {

public void startup(){

System.out.println("disk startup!");

}

public void shutdown(){

System.out.println("disk shutdown!");

}

}

[java] view plaincopy

public class Computer {

private CPU cpu;

private Memory memory;

private Disk disk;

public Computer(){

cpu = new CPU();

memory = new Memory();

disk = new Disk();

}

public void startup(){

System.out.println("start the computer!");

cpu.startup();

memory.startup();

disk.startup();

System.out.println("start computer finished!");

}

public void shutdown(){

System.out.println("begin to close the computer!");

cpu.shutdown();

memory.shutdown();

disk.shutdown();

System.out.println("computer closed!");

}

}

User类如下：

[java] view plaincopy

public class User {

public static void main(String[] args) {

Computer computer = new Computer();

computer.startup();

computer.shutdown();

}

}

输出：

start the computer!

cpu startup!

memory startup!

disk startup!

start computer finished!

begin to close the computer!

cpu shutdown!

memory shutdown!

disk shutdown!

computer closed!

如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会 带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观 模式！

P r o x y

适用性

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用 P r o x y模式。下面是一

些可以使用P r o x y模式常见情况：

远 程 代 理 （ Remote Proxy） 为 一 个 对 象在 不 同 的 地 址 空间 提 供 局 部 代 表。

NEXTSTEP[Add94] 使用N X P r o x y类实现了这一目的。Coplien[Cop92] 称这种代理为“大使”

（A m b a s s a d o r）。

虚代理（Virtual Proxy）根据需要创建开销很大的对象。

在动机一节描述的 I m a g e P r o x y

就是这样一种代理的例子。

保护代理（Protection Proxy）控制对原始对象的访问。

保护代理用于对象应该有不同

的访问权限的时候。例如，在 C h o i c e s操作系统[ C I R M 9 3 ]中K e m e l P r o x i e s为操作系统对象提供

了访问保护。

智能指引（Smart Reference）取代了简单的指针，它在访问对象时执行一些附加操作。

它的典型用途包括：

• 对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它 (也称为S m a r t

P o i n t e r s[ E d e 9 2 ] )。

• 当第一次引用一个持久对象时，将它装入内存。

• 在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

效 果

P r o x y模式在访问对象时引入了一定程度的间接性。根据代理的类型，附加的间接性有多

种用途：

P r o x y模式还可以对用户隐藏另一种称之为 c o p y - o n - w r i t e的优化方式，该优化与根据需要

创建对象有关。拷贝一个庞大而复杂的对象是一种开销很大的操作，如果这个拷贝根本没有

被修改，那么这些开销就没有必要。用代理延迟这一拷贝过程，我们可以保证只有当这个对

象被修改的时候才对它进行拷贝。

在实现C o p y - o n - w r i t e时必须对实体进行引用计数。拷贝代理仅会增加引用计数。只有当

用户请求一个修改该实体的操作时，代理才会真正的拷贝它。在这种情况下，代理还必须减

少实体的引用计数。当引用的数目为零时，这个实体将被删除。

C o p y - o n - Wr i t e可以大幅度的降低拷贝庞大实体时的开销。

代码示例

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该 地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有 专长，可以替我们进行操作，表达我们的想法。先来看看关系图：

根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

[java] view plaincopy

public interface Sourceable {

public void method();

}

[java] view plaincopy

public class Source implements Sourceable {

@Override

public void method() {

System.out.println("the original method!");

}

}

[java] view plaincopy

public class Proxy implements Sourceable {

private Source source;

public Proxy(){

super();

this.source = new Source();

}

@Override

public void method() {

before();

source.method();

atfer();

}

private void atfer() {

System.out.println("after proxy!");

}

private void before() {

System.out.println("before proxy!");

}

}

测试类：

[java] view plaincopy

public class ProxyTest {

public static void main(String[] args) {

Sourceable source = new Proxy();

source.method();

}

}

输出：

before proxy!

the original method!

after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

F l y w e i g h t

效 果

使用F l y w e i g h t模式时，传输、查找和 /或计算外部状态都会产生运行时的开销，尤其当

f l y w e i g h t原先被存储为内部状态时。然而，空间上的节省抵消了这些开销。共享的 f l y w e i g h t

越多，空间节省也就越大。

存储节约由以下几个因素决定

共享的F l y w e i g h t越多，存储节约也就越多。节约量随着共享状态的增多而增大。当对象

使用大量的内部及外部状态，并且外部状态是计算出来的而非存储的时候，节约量将达到最

大。所以，可以用两种方法来节约存储：用共享减少内部状态的消耗，用计算时间换取对外

部状态的存储。

F l y w e i g h t模式经常和C o m p o s i t e（4 . 3）模式结合起来表示一个层次式结构，这一层次式

结构是一个共享叶节点的图。共享的结果是， F l y w e i g h t的叶节点不能存储指向父节点的指针。

而父节点的指针将传给 F l y w e i g h t作为它的外部状态的一部分。这对于该层次结构中对象之间

相互通讯的方式将产生很大的影响。

实 现

删除外部状态

该模式的可用性在很大程度上取决于是否容易识别外部状态并将它从

共享对象中删除。如果不同种类的外部状态和共享前对象的数目相同的话，删除外部状态不

会降低存储消耗。理想的状况是，外部状态可以由一个单独的对象结构计算得到，且该结构

的存储要求非常小。

例如，在我们的文档编辑器中，我们可以用一个单独的结构存储排版布局信息，而不是

存储每一个字符对象的字体和类型信息，布局图保持了带有相同排版信息的字符的运行轨迹。

当某字符绘制自己的时候，作为绘图遍历的副作用它接收排版信息。因为通常文档使用的字

体和类型数量有限，将该信息作为外部信息来存储，要比内部存储高效得多。

管理共享对象

因为对象是共享的，用户不能直接对它进行实例化，因此 F l y w e i g h t -

F a c t o r y可以帮助用户查找某个特定的 F l y w e i g h t对象。F l y w e i g h t F a c t o r y对象经常使用关联存储

帮助用户查找感兴趣的F l y w e i g h t对象。例如，在这个文档编辑器一例中的F l y w e i g h t工厂就有一

个以字符代码为索引的F l y w e i g h t表。管理程序根据所给的代码返回相应的F l y w e i g h t，若不存在，

则创建一个F l y w e i g h t。

共享还意味着某种形式的引用计数和垃圾回收，这样当一个 F l y w e i g h t不再使用时，可以

回收它的存储空间。然而，当 F l y w e i g h t的数目固定而且很小的时候（例如，用于 A C S I I码的

F l y w e i g h t），这两种操作都不必要。在这种情况下， F l y w e i g h t完全可以永久保存。

代码示例

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在 的对象，如果没有，则创建一个新对象，FlyWeight是超类。一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我 们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、 password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的 作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。

看个例子：

看下数据库连接池的代码：

[java] view plaincopy

public class ConnectionPool {

private Vector<Connection> pool;

/\*公有属性\*/

private String url = "jdbc:mysql://localhost:3306/test";

private String username = "root";

private String password = "root";

private String driverClassName = "com.mysql.jdbc.Driver";

private int poolSize = 100;

private static ConnectionPool instance = null;

Connection conn = null;

/\*构造方法，做一些初始化工作\*/

private ConnectionPool() {

pool = new Vector<Connection>(poolSize);

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

try {

Class.forName(driverClassName);

conn = DriverManager.getConnection(url, username, password);

pool.add(conn);

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

/\* 返回连接到连接池 \*/

public synchronized void release() {

pool.add(conn);

}

/\* 返回连接池中的一个数据库连接 \*/

public synchronized Connection getConnection() {

if (pool.size() > 0) {

Connection conn = pool.get(0);

pool.remove(conn);

return conn;

} else {

return null;

}

}

}

通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！本章讲解了7种结构型模式，因为篇幅的问题，剩下的11种行为型模式，

本章是关于设计模式的最后一讲，会讲到第三种设计模式——行为型模式，共11种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模 式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。这段时间一直在写关于设计模式的东西，终于写到一半了，写博文是个很费时间的东西，因为 我得为读者负责，不论是图还是代码还是表述，都希望能尽量写清楚，以便读者理解，我想不论是我还是读者，都希望看到高质量的博文出来，从我本人出发，我会 一直坚持下去，不断更新，源源动力来自于读者朋友们的不断支持，我会尽自己的努力，写好每一篇文章！希望大家能不断给出意见和建议，共同打造完美的博文！

先来张图，看看这11中模式的关系：

第一类：通过父类与子类的关系进行实现。第二类：两个类之间。第三类：类的状态。第四类：通过中间类

结构型模式的讨论

你可能已经注意到了结构型模式之间的相似性，尤其是它们的参与者和协作之间的相似

性。这可能是因为结构型模式依赖于同一个很小的语言机制集合构造代码和对象：单继承和

多重继承机制用于基于类的模式，而对象组合机制用于对象式模式。但是这些相似性掩盖了

这些模式的不同意图。在本节中，我们将对比这些结构型模式，使你对它们各自的优点有所

了解。

Adapter与Bridge

A d a p t e r（4 . 1）模式和 B r i d g e（4 . 2）模式具有一些共同的特征。它们都给另一对象提供

了一定程度上的间接性，因而有利于系统的灵活性。它们都涉及到从自身以外的一个接口向

这个对象转发请求。

这些模式的不同之处主要在于它们各自的用途。 A d a p t e r模式主要是为了解决两个已有接

口之间不匹配的问题。它不考虑这些接口是怎样实现的，也不考虑它们各自可能会如何演化。

这种方式不需要对两个独立设计的类中的任一个进行重新设计，就能够使它们协同工作。另一方面，B r i d g e模式则对抽象接口与它的（可能是多个）实现部分进行桥接。虽然这一模式

允许你修改实现它的类，它仍然为用户提供了一个稳定的接口。 B r i d g e模式也会在系统演化

时适应新的实现。

由于这些不同点， A d a p t e r和B r i d g e模式通常被用于软件生命周期的不同阶段。当你发现

两个不兼容的类必须同时工作时，就有必要使用 A d a p t e r模式，其目的一般是为了避免代码重

复。此处耦合不可预见。相反， B r i d g e的使用者必须事先知道：一个抽象将有多个实现部分，

并且抽象和实现两者是独立演化的。 A d a p t e r模式在类已经设计好后实施；而B r i d g e模式在设

计类之前实施。这并不意味着A d a p t e r模式不如B r i d g e模式，只是因为它们针对了不同的问题。

你可能认为f a c a d e (参见F a c a d e ( 4 . 5 ) )是另外一组对象的适配器。但这种解释忽视了一个事

实：即 F a c a d e定义一个新的接口，而A d a p t e r则复用一个原有的接口。记住，适配器使两个已

有的接口协同工作，而不是定义一个全新的接口。

Composite、Decorator与Proxy

C o m p o s i t e ( 4 . 3 )模式和D e c o r a t o r ( 4 . 4 )模式具有类似的结构图，这说明它们都基于递归组合

来组织可变数目的对象。这一共同点可能会使你认为， d e c o r a t o r对象是一个退化的c o m p o s i t e，

但这一观点没有领会 D e c o r a t o r模式要点。相似点仅止于递归组合，同样，这是因为这两个模

式的目的不同。

Decorator 旨在使你能够不需要生成子类即可给对象添加职责。这就避免了静态实现所有

功能组合，从而导致子类急剧增加。 C o m p o s i t e则有不同的目的，它旨在构造类，使多个相关

的对象能够以统一的方式处理，而多重对象可以被当作一个对象来处理。它重点不在于修饰，

而在于表示。

尽管它们的目的截然不同，但却具有互补性。因此 Composite 和 D e c o r a t o r模式通常协同

使用。在使用这两种模式进行设计时，我们无需定义新的类，仅需将一些对象插接在一起即

可构建应用。这时系统中将会有一个抽象类，它有一些 c o m p o s i t e子类和d e c o r a t o r子类，还有

一些实现系统的基本构建模块。此时， composites 和d e c o r a t o r将拥有共同的接口。从

D e c o r a t o r模式的角度看，c o m p o s i t e是一个C o n c r e t e C o m p o n e n t。而从c o m p o s i t e模式的角度看，

d e c o r a t o r则是一个L e a f。当然，他们不一定要同时使用，正如我们所见，它们的目的有很大

的差别。

另一种与D e c o r a t o r模式结构相似的模式是 P r o x y ( 4 . 7 )。这两种模式都描述了怎样为对象提

供一定程度上的间接引用，proxy 和d e c o r a t o r对象的实现部分都保留了指向另一个对象的指针，

它们向这个对象发送请求。然而同样，它们具有不同的设计目的。

像D e c o r a t o r模式一样， Proxy 模式构成一个对象并为用户提供一致的接口。但与

D e c o r a t o r模式不同的是，Proxy 模式不能动态地添加或分离性质，它也不是为递归组合而设

计的。它的目的是，当直接访问一个实体不方便或不符合需要时，为这个实体提供一个替代

者，例如，实体在远程设备上，访问受到限制或者实体是持久存储的。

在P r o x y模式中，实体定义了关键功能，而 P r o x y提供（或拒绝）对它的访问。在

D e c o r a t o r模式中，组件仅提供了部分功能，而一个或多个 D e c o r a t o r负责完成其他功能。

D e c o r a t o r模式适用于编译时不能（至少不方便）确定对象的全部功能的情况。这种开放性使递归组合成为 D e c o r a t o r模式中一个必不可少的部分。而在 P r o x y模式中则不是这样，因为

P r o x y模式强调一种关系（P r o x y与它的实体之间的关系），这种关系可以静态的表达。

模式间的这些差异非常重要，因为它们针对了面向对象设计过程中一些特定的经常发生

问题的解决方法。但这并不意味着这些模式不能结合使用。可以设想有一个 p r o x y - d e c o r a t o r，

它可以给p r o x y添加功能，或是一个 d e c o r a t o r- p r o x y用来修饰一个远程对象。尽管这种混合可

能有用（我们手边还没有现成的例子），但它们可以分割成一些有用的模式。

行 为 型

行为模式涉及到算法和对象间职责的分配。行为模式不仅描述对象或类的模式，还描述

它们之间的通信模式。这些模式刻划了在运行时难以跟踪的复杂的控制流。它们将你的注意

力从控制流转移到对象间的联系方式上来。

行为类模式使用继承机制在类间分派行为。本章包括两个这样的模式。其中 Te m p l a t e

M e t h o d（5 . 1 0）较为简单和常用。模板方法是一个算法的抽象定义，它逐步地定义该算法，

每一步调用一个抽象操作或一个原语操作，子类定义抽象操作以具体实现该算法。另一种行

为类模式是I n t e r p r e t e r（5 . 3）。它将一个文法表示为一个类层次，并实现一个解释器作为这些

类的实例上的一个操作。

行为对象模式使用对象复合而不是继承。一些行为对象模式描述了一组对等的对象怎样

相互协作以完成其中任一个对象都无法单独完成的任务。这里一个重要的问题是对等的对象

如何互相了解对方。对等对象可以保持显式的对对方的引用，但那会增加它们的耦合度。在

极端情况下，每一个对象都要了解所有其他的对象。 M e d i a t o r（5 . 5）在对等对象间引入一个

m e d i a t o r对象以避免这种情况的出现。m e d i a t o r提供了松耦合所需的间接性。

Chain of Responsibility(5.1)提供更松的耦合。它让你通过一条候选对象链隐式的向一个对

象发送请求。根据运行时刻情况任一候选者都可以响应相应的请求。候选者的数目是任意的，

你可以在运行时刻决定哪些候选者参与到链中。

O b s e r v e r ( 5 . 7 )模式定义并保持对象间的依赖关系。典型的 O b s e r v e r的例子是Smalltalk 中的

模型/视图/控制器，其中一旦模型的状态发生变化，模型的所有视图都会得到通知。

其他的行为对象模式常将行为封装在一个对象中并将请求指派给它。 S t r a t e g y ( 5 . 9 )模式将

算法封装在对象中，这样可以方便地指定和改变一个对象所使用的算法。 C o m m a n d ( 5 . 2 )模式

将请求封装在对象中，这样它就可作为参数来传递，也可以被存储在历史列表里，或者以其

他方式使用。S t a t e ( 5 . 8 )模式封装一个对象的状态，使得当这个对象的状态对象变化时，该对

象可改变它的行为。Vi s i t o r ( 5 . 11 )封装分布于多个类之间的行为，而 I t e r a t o r ( 5 . 4 )则抽象了访问

和遍历一个集合中的对象的方式。

类

Interpreter

效 果

复杂的文法难以维护解释器模式为文法中的每一条规则至少定义了一个类(使用BNF定义的文法规则需要更多的类)。因此包含许多规则的文法可能难以管理和维护。

可应用其他的设计模式来缓解这一问题。但当文法非常复杂时,其他的技术如语法分析程序或编译器生成器更为合适。

4)增加了新的解释表达式的方式解释器模式使得实现新表达式“计算”变得容易。例如,你可以在表达式类上定义一个新的操作以支持优美打印或表达式的类型检查。如果你经常创建新的解释表达式的方式,那么可以考虑使用Visitor(5.11)模式以避免修改这些代表文法的类。

实 现

与Flyweight模式共享终结符在一些文法中,一个句子可能多次出现同一个终结符。

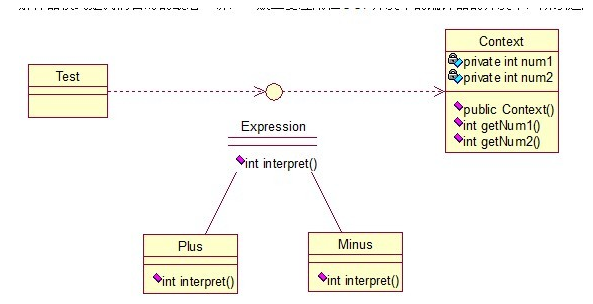
此时最好共享那个符号的单个拷贝。计算机程序的文法是很好的例子—每个程序变量在整个代码中将会出现多次。在动机一节的例子中,一个句子中终结符dog(由LiteralExpression类描述)也可出现多次。

　　终结节点通常不存储关于它们在抽象语法树中位置的信息。在解释过程中，任何它们所需要的上下文信息都由父节点传递给它们。因此在共享的(内部的)状态和传入的(外部的)状态区分得很明确,这就用到了Flyweight（4.6）模式。

　　例如，dogLiteralExpression的每一实例接收一个包含目前已匹配子串信息的上下文。且每一个这样的LiteralExpression在它的解释操作中做同样一件事（它检查输入的下一部分是否包含一个dog）无论该实例出现在语法树的哪个位置。

代码示例

，一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。



Context类是一个上下文环境类，Plus和Minus分别是用来计算的实现，代码如下：

public interface Expression {

public int interpret(Context context);

}

public class Plus implements Expression {

@Override

public int interpret(Context context) {

return context.getNum1()+context.getNum2();

}

}

public class Minus implements Expression {

@Override

public int interpret(Context context) {

return context.getNum1()-context.getNum2();

}

}

public class Context {

private int num1;

private int num2;

public Context(int num1, int num2) {

this.num1 = num1;

this.num2 = num2;

}

public int getNum1() {

return num1;

}

public void setNum1(int num1) {

this.num1 = num1;

}

public int getNum2() {

return num2;

}

public void setNum2(int num2) {

this.num2 = num2;

}

}

public class Test {

public static void main(String[] args) {

// 计算9+2-8的值

int result = new Minus().interpret((new Context(new Plus()

.interpret(new Context(9, 2)), 8)));

System.out.println(result);

}

}

最后输出正确的结果：3。

基本就这样，解释器模式用来做各种各样的解释器，如正则表达式等的解释器等等！

对象

ChainofResponsibility

效 果

降低耦合度

　　该模式使得一个对象无需知道是其他哪一个对象处理其请求。对象仅需知道该请求会被“正确”地处理。接收者和发送者都没有对方的明确的信息，且链中的对象不需知道链的结构。

　　结果是，职责链可简化对象的相互连接。它们仅需保持一个指向其后继者的引用，而不需保持它所有的候选接受者的引用。

增强了给对象指派职责( R e s p o n s i b i l i t y )的灵活性

　　当在对象中分派职责时，职责链给你更多的灵活性。你可以通过在运行时刻对该链进行动态的增加或修改来增加或改变处理一个请求的那些职责。你可以将这种机制与静态的特例化处理对象的继承机制结合起来使用。

不保证被接受

　　既然一个请求没有明确的接收者，那么就不能保证它一定会被处理 —该请求可能一直到链的末端都得不到处理。一个请求也可能因该链没有被正确配置而得不到处理。

实 现

连接后继者

　　如果没有已有的引用可定义一个链，那么你必须自己引入它们。这种情况下Handler不仅定义该请求的接口，通常也维护后继链接。这样Handler就提供了HandleRequest的缺省实现:HandleRequest向后继者(如果有的话)转发请求。如果ConcreteHandler子类对该请求不感兴趣，它不需重定义转发操作，因为它的缺省实现进行无条件的转发。

在S m a l l t a l k中自动转发

　　你可以使用Smalltalk中的doesNotUnderstand机制转发请求。

　　没有相应方法的消息被doseNotUnderstand的实现捕捉（trapin），此实现可被重定义，从而可向一个对象的后继者转发该消息。这样就不需要手工实现转发；类仅处理它感兴趣的请求，而依赖doesNotUnderstand转发所有其他的请求。

表示请求

　　可以有不同的方法表示请求。最简单的形式，比如在HandleHelp的例子中，请求是一个硬编码的(hard-coded)操作调用。这种形式方便而且安全，但你只能转发Handler类定义的固定的一组请求。

　　另一选择是使用一个处理函数，这个函数以一个请求码(如一个整型常数或一个字符串)为参数。这种方法支持请求数目不限。唯一的要求是发送方和接受方在请求如何编码问题上应达成一致。

　　这种方法更为灵活，但它需要用条件语句来区分请求代码以分派请求。另外，无法用类型安全的方法来传递请求参数，因此它们必须被手工打包和解包。显然，相对于直接调用一个操作来说它不太安全。

　　为解决参数传递问题，我们可使用独立的请求对象来封装请求参数。Request类可明确地描述请求，而新类型的请求可用它的子类来定义。这些子类可定义不同的请求参数。处理者必须知道请求的类型(即它们正使用哪一个Request子类)以访问这些参数。

　　为标识请求，Request可定义一个访问器(accessor)函数以返回该类的标识符。或者，如果实现语言支持的话，接受者可使用运行时的类型信息。

代码示例

接下来我们将要谈谈责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传 递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的 调整。先看看关系图：

Abstracthandler类提供了get和set方法，方便MyHandle类设置和修改引用对象，MyHandle类是核心，实例化后生成一系列相互持有的对象，构成一条链。

public interface Handler {

public void operator();

}

public abstract class AbstractHandler {

private Handler handler;

public Handler getHandler() {

return handler;

}

public void setHandler(Handler handler) {

this.handler = handler;

}

}

public class MyHandler extends AbstractHandler implements Handler {

private String name;

public MyHandler(String name) {

this.name = name;

}

@Override

public void operator() {

System.out.println(name+"deal!");

if(getHandler()!=null){

getHandler().operator();

}

}

}

public class Test {

public static void main(String[] args) {

MyHandler h1 = new MyHandler("h1");

MyHandler h2 = new MyHandler("h2");

MyHandler h3 = new MyHandler("h3");

h1.setHandler(h2);

h2.setHandler(h3);

h1.operator();

}

}

输出：

h1deal!

h2deal!

h3deal!

此处强调一点就是，链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。

Command

适用性

在不同的时刻指定、排列和执行请求。

　　一个Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

支持取消操作。

　　Command的Excute操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command接口必须添加一个Unexecute操作，该操作取消上一次Execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用Unexecute和Execute来实现重数不限的“取消”和“重做”。

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。

　　在Command接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用Execute操作重新执行它们。

用构建在原语操作上的高层操作构造一个系统。

　　这样一种结构在支持事务(transaction)

　　的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

协 作

某Invoker对象存储该ConcreteCommand对象。

该Invoker通过调用Command对象的Execute操作来提交一个请求。若该命令是可撤消的，ConcreteCommand就在执行Excute操作之前存储当前状态以用于取消该命令。

实 现

一个命令对象应达到何种智能程度命令对象的能力可大可小。

一个极端是它仅确定一个接收者和执行该请求的动作。另一极端是它自己实现所有功能，根本不需要额外的接收者对象。当需要定义与已有的类无关的命令，当没有合适的接收者，或当一个命令隐式地知道它的接收者时，可以使用后一极端方式。例如，创建另一个应用窗口的命令对象本身可能和任何其他的对象一样有能力创建该窗口。在这两个极端间的情况是命令对象有足够的信息可以动态的找到它们的接收者。

支持取消（undo）和重做（redo）如果Command提供方法逆转(reverse)它们操作的执行(例如Unexecute或Undo操作)，就可支持取消和重做功能。

为达到这个目的，ConcreteCommand类可能需要存储额外的状态信息。这个状态包括：•接收者对象，它真正执行处理该请求的各操作。

　　•接收者上执行操作的参数。

　　•如果处理请求的操作会改变接收者对象中的某些值，那么这些值也必须先存储起来。接收者还必须提供一些操作，以使该命令可将接收者恢复到它先前的状态。

　　若应用只支持一次取消操作，那么只需存储最近一次被执行的命令。而若要支持多级的取消和重做，就需要有一个已被执行命令的历史表列(historylist)，该表列的最大长度决定了取消和重做的级数。历史表列存储了已被执行的命令序列。向后遍历该表列并逆向执行(reverse-executing)命令是取消它们的结果；向前遍历并执行命令是重执行它们。

　　有时可能不得不将一个可撤消的命令在它可以被放入历史列表中之前先拷贝下来。这是因为执行原来的请求的命令对象将在稍后执行其他的请求。如果命令的状态在各次调用之间会发生变化，那就必须进行拷贝以区分相同命令的不同调用。

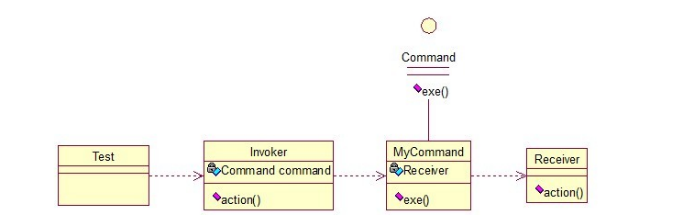
　　例如，一个删除选定对象的删除命令(DeleteCommand)在它每次被执行时，必须存储不同的对象集合。因此该删除命令对象在执行后必须被拷贝，并且将该拷贝放入历史表列中。如果该命令的状态在执行时从不改变，则不需要拷贝，而仅需将一个对该命令的引用放入历史表列中。在放入历史表列中之前必须被拷贝的那些Command起着原型（参见Prototype模式（3.4））的作用。

避免取消操作过程中的错误积累在实现一个可靠的、能保持原先语义的取消/重做机制时，可能会遇到滞后影响问题。

由于命令重复的执行、取消执行，和重执行的过程可能会积累错误，以至一个应用的状态最终偏离初始值。这就有必要在Command中存入更多的信息以保证这些对象可被精确地复原成它们的初始状态。这里可使用Memento模式（5.6）来让该Command访问这些信息而不暴露其他对象的内部信息。

代码示例

命令模式很好理解，举个例子，司令员下令让士兵去干件事情，从整个事情的角度来考虑，司令员的作用是，发出口令，口令经过传递，传到了士兵耳朵里，士兵去 执行。这个过程好在，三者相互解耦，任何一方都不用去依赖其他人，只需要做好自己的事儿就行，司令员要的是结果，不会去关注到底士兵是怎么实现的。我们看 看关系图：



Invoker是调用者（司令员），Receiver是被调用者（士兵），MyCommand是命令，实现了Command接口，持有接收对象，看实现代码：

[java] view plaincopy

public interface Command {

public void exe();

}

[java] view plaincopy

public class MyCommand implements Command {

private Receiver receiver;

public MyCommand(Receiver receiver) {

this.receiver = receiver;

}

@Override

public void exe() {

receiver.action();

}

}

[java] view plaincopy

public class Receiver {

public void action(){

System.out.println("command received!");

}

}

[java] view plaincopy

public class Invoker {

private Command command;

public Invoker(Command command) {

this.command = command;

}

public void action(){

command.exe();

}

}

[java] view plaincopy

public class Test {

public static void main(String[] args) {

Receiver receiver = new Receiver();

Command cmd = new MyCommand(receiver);

Invoker invoker = new Invoker(cmd);

invoker.action();

}

}

输出：command received!

这个很哈理解，命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

Iterator

效 果

它支持以不同的方式遍历一个聚合复杂的聚合可用多种方式进行遍历。

例如,代码生成和语义检查要遍历语法分析树。代码生成可以按中序或者按前序来遍历语法分析树。迭代器模式使得改变遍历算法变得很容易:仅需用一个不同的迭代器的实例代替原先的实例即可。

　　你也可以自己定义迭代器的子类以支持新的遍历。

实 现

谁控制该迭代一个基本的问题是决定由哪一方来控制该迭代,是迭代器还是使用该迭代器的客户。

当由客户来控制迭代时,该迭代器称为一个外部迭代器(externaliterator)，而当由迭代器控制迭代时,该迭代器称为一个内部迭代器(internaliterator)。使用外部迭代器的客户必须主动推进遍历的步伐，显式地向迭代器请求下一个元素。相反地,若使用内部迭代器，客户只需向其提交一个待执行的操作，而迭代器将对聚合中的每一个元素实施该操作。

　　外部迭代器比内部迭代器更灵活。例如,若要比较两个集合是否相等，这个功能很容易用外部迭代器实现，而几乎无法用内部迭代器实现。在象C++这样不提供匿名函数、闭包,或象Smalltalk和CLOS这样不提供连续(continuation)的语言中，内部迭代器的弱点更为明显。但另一方面,内部迭代器的使用较为容易,因为它们已经定义好了迭代逻辑。

谁定义遍历算法迭代器不是唯一可定义遍历算法的地方。

聚合本身也可以定义遍历算法，并在遍历过程中用迭代器来存储当前迭代的状态。我们称这种迭代器为一个游标(cursor),因为它仅用来指示当前位置。客户会以这个游标为一个参数调用该聚合的Next操作,而Next操作将改变这个指示器的状态。

　　如果迭代器负责遍历算法,那么将易于在相同的聚合上使用不同的迭代算法,同时也易于在不同的聚合上重用相同的算法。从另一方面说,遍历算法可能需要访问聚合的私有变量。如果这样，将遍历算法放入迭代器中会破坏聚合的封装性。

迭代器健壮程度如何在遍历一个聚合的同时更改这个聚合可能是危险的。

如果在遍历聚合的时候增加或删除该聚合元素,可能会导致两次访问同一个元素或者遗漏掉某个元素。

　　一个简单的解决办法是拷贝该聚合，并对该拷贝实施遍历,但一般来说这样做代价太高。

　　一个健壮的迭代器(robustiterator)保证插入和删除操作不会干扰遍历,且不需拷贝该聚合。

　　有许多方法来实现健壮的迭代器。其中大多数需要向这个聚合注册该迭代器。当插入或删除元素时，该聚合要么调整迭代器的内部状态,要么在内部的维护额外的信息以保证正确的遍历。

　　Kofler在ET++[Kof93]中对如何实现健壮的迭代器做了很充分的讨论。Murray讨论了如何为USLStandardComponents列表类实现健壮的迭代器[Mur93]。

附加的迭代器操作迭代器的最小接口由First、Next、IsDone和CurrentItem操作组成。

其他一些操作可能也很有用。例如,对有序的聚合可用一个Previous操作将迭代器定位到前一个元素。SkipTo操作用于已排序并做了索引的聚合中，它将迭代器定位到符合指定条件的元素对象上。

在C++中使用多态的迭代器使用多态迭代器是有代价的。

它们要求用一个FactoryMethod动态的分配迭代器对象。因此仅当必须多态时才使用它们。否则使用在栈中分配内存的具体的迭代器。

　　多态迭代器有另一个缺点:客户必须负责删除它们。这容易导致错误,因为你容易忘记释放一个使用堆分配的迭代器对象，当一个操作有多个出口时尤其如此。而且其间如果有异常被触发的话，迭代器对象将永远不会被释放。

　　Proxy（4.4）模式提供了一个补救方法。我们可使用一个栈分配的Proxy作为实际迭代器的中间代理。该代理在其析构器中删除该迭代器。这样当该代理生命周期结束时，实际迭代器将同它一起被释放。即使是在发生异常时，该代理机制能保证正确地清除迭代器对象。这就是著名的C++“资源分配即初始化”技术[ES90]的一个应用。下面的代码示例给出了一个例子。

迭代器可有特权访问迭代器可被看为创建它的聚合的一个扩展。

迭代器和聚合紧密耦合。在C++中我们可让迭代器作为它的聚合的一个友元(friend)来表示这种紧密的关系。这样你就不需要在聚合类中定义一些仅为迭代器所使用的操作。

　　但是,这样的特权访问可能使定义新的遍历变得很难,因为它将要求改变该聚合的接口增加另一个友元。为避免这一问题,迭代器类可包含一些protected操作来访问聚合类的重要的非公共可见的成员。迭代器子类(且只有迭代器子类)可使用这些protected操作来得到对该聚合的特权访问。

用于复合对象的迭代器在Composite(4.3)模式中的那些递归聚合结构上,外部迭代器可能难以实现,因为在该结构中不同对象处于嵌套聚合的多个不同层次，因此一个外部迭代器为跟踪当前的对象必须存储一条纵贯该Composite的路径。

有时使用一个内部迭代器会更容易一些。它仅需递归地调用自己即可，这样就隐式地将路径存储在调用栈中，而无需显式地维护当前对象位置。

　　如果复合中的节点有一个接口可以从一个节点移到它的兄弟节点、父节点和子节点,那么基于游标的迭代器是个更好的选择。游标只需跟踪当前的节点;它可依赖这种节点接口来遍历该复合对象。

　　复合常常需要用多种方法遍历。前序,后序,中序以及广度优先遍历都是常用的。你可用不同的迭代器类来支持不同的遍历。

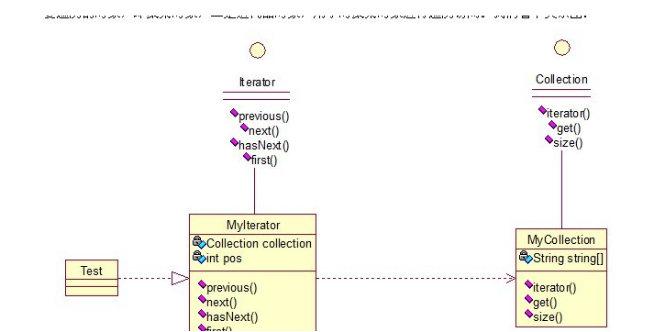
空迭代器一个空迭代器(NullIterator)是一个退化的迭代器,它有助于处理边界条件。

根据定义，一个NullIterator总是已经完成了遍历：即,它的IsDone操作总是返回true。

　　空迭代器使得更容易遍历树形结构的聚合(如复合对象)。在遍历过程中的每一节点,都可向当前的元素请求遍历其各个子结点的迭代器。该聚合元素将返回一个具体的迭代器。但叶节点元素返回NullIterator的一个实例。这就使我们可以用一种统一的方式实现在整个结构上的遍历。

代码示例

顾名思义，迭代器模式就是顺序访问聚集中的对象，一般来说，集合中非常常见，如果对集合类比较熟悉的话，理解本模式会十分轻松。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。我们看下关系图：



这个思路和我们常用的一模一样，MyCollection中定义了集合的一些操作，MyIterator中定义了一系列迭代操作，且持有Collection实例，我们来看看实现代码：

两个接口：

public interface Collection {

public Iterator iterator();

/\*取得集合元素\*/

public Object get(int i);

/\*取得集合大小\*/

public int size();

}

public interface Iterator {

//前移

public Object previous();

//后移

public Object next();

public boolean hasNext();

//取得第一个元素

public Object first();

}

两个实现：

public class MyCollection implements Collection {

public String string[] = {"A","B","C","D","E"};

@Override

public Iterator iterator() {

return new MyIterator(this);

}

@Override

public Object get(int i) {

return string[i];

}

@Override

public int size() {

return string.length;

}

}

public class MyIterator implements Iterator {

private Collection collection;

private int pos = -1;

public MyIterator(Collection collection){

this.collection = collection;

}

@Override

public Object previous() {

if(pos > 0){

pos--;

}

return collection.get(pos);

}

@Override

public Object next() {

if(pos<collection.size()-1){

pos++;

}

return collection.get(pos);

}

@Override

public boolean hasNext() {

if(pos<collection.size()-1){

return true;

}else{

return false;

}

}

@Override

public Object first() {

pos = 0;

return collection.get(pos);

}

}

测试类：

public class Test {

public static void main(String[] args) {

Collection collection = new MyCollection();

Iterator it = collection.iterator();

while(it.hasNext()){

System.out.println(it.next());

}

}

}

输出：A B C D E

此处我们貌似模拟了一个集合类的过程，感觉是不是很爽？其实JDK中各个类也都是这些基本的东西，加一些设计模式，再加一些优化放到一起的，只要我们把这些东西学会了，掌握好了，我们也可以写出自己的集合类，甚至框架！

Mediator

实 现

)忽略抽象的Mediator类当各Colleague仅与一个Mediator一起工作时,没有必要定义一个抽象的Mediator类。

Mediator类提供的抽象耦合已经使各Colleague可与不同的Mediator子类一起工作,反之亦然。

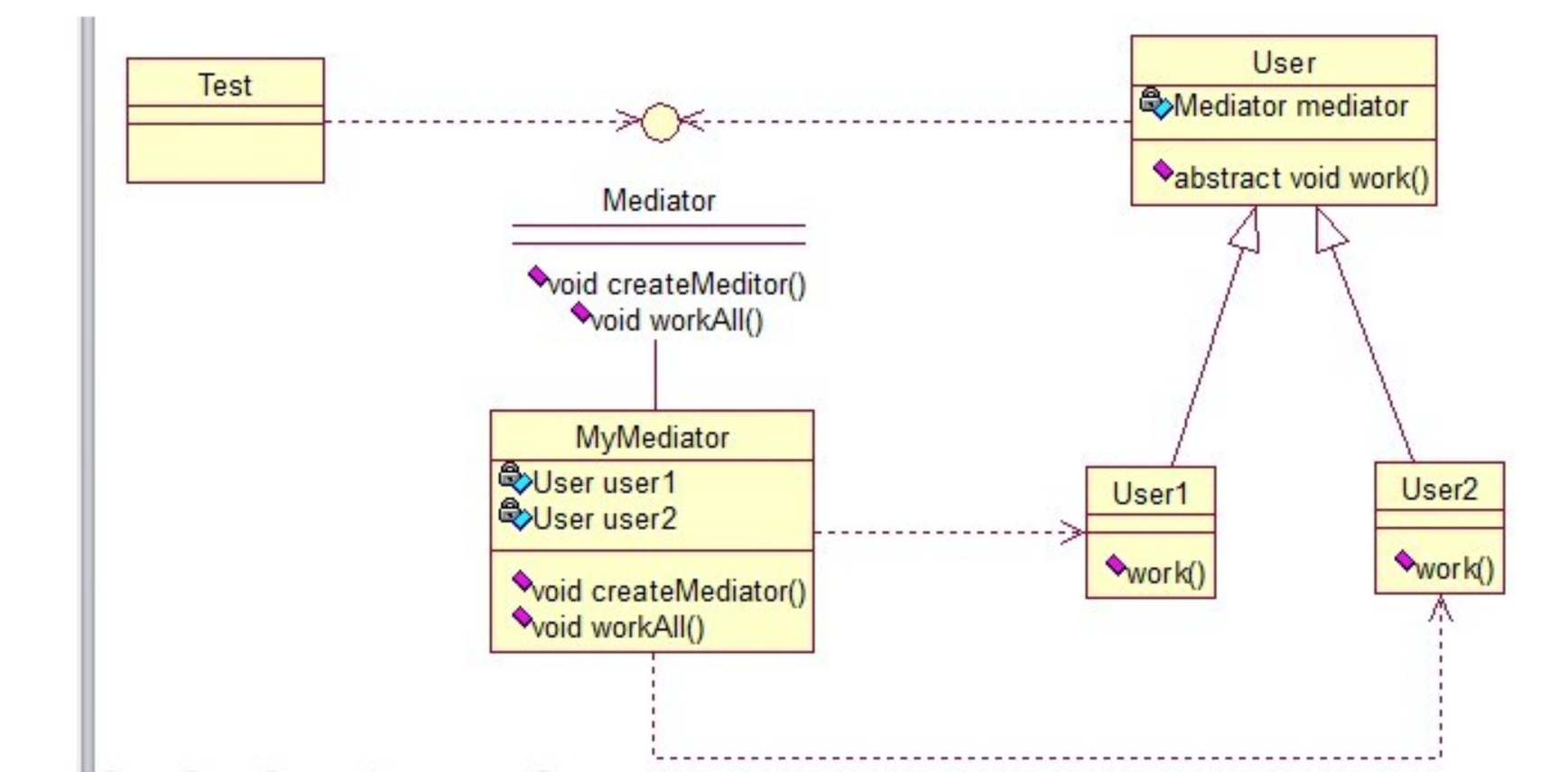
Colleague—Mediator通信当一个感兴趣的事件发生时,Colleague必须与其Mediator通信。

一种实现方法是使用Observer(5.7)模式，将Mediator实现为一个Observer，各Colleague作为Subject，一旦其状态改变就发送通知给Mediator。Mediator作出的响应是将状态改变的结果传播给其他的Colleague。

　　另一个方法是在Mediator中定义一个特殊的通知接口,各Colleague在通信时直接调用该接口。Windows下的Smalltalk/V使用某种形式的代理机制:当与Mediator通信时,Colleague将自身作为一个参数传递给Mediator,使其可以识别发送者。代码示例一节使用这种方法。而Smalltalk/V的实现方法将稍后在已知应用一节中讨论。

代码示例

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修 改。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用。先 看看图：



User类统一接口，User1和User2分别是不同的对象，二者之间有关联，如果不采用中介者模式，则需要二者相互持有引用，这样二者的耦合度很高， 为了解耦，引入了Mediator类，提供统一接口，MyMediator为其实现类，里面持有User1和User2的实例，用来实现对User1和 User2的控制。这样User1和User2两个对象相互独立，他们只需要保持好和Mediator之间的关系就行，剩下的全由MyMediator类 来维护！基本实现：

public interface Mediator {

public void createMediator();

public void workAll();

}

public class MyMediator implements Mediator {

private User user1;

private User user2;

public User getUser1() {

return user1;

}

public User getUser2() {

return user2;

}

@Override

public void createMediator() {

user1 = new User1(this);

user2 = new User2(this);

}

@Override

public void workAll() {

user1.work();

user2.work();

}

}

public abstract class User {

private Mediator mediator;

public Mediator getMediator(){

return mediator;

}

public User(Mediator mediator) {

this.mediator = mediator;

}

public abstract void work();

}

public class User1 extends User {

public User1(Mediator mediator){

super(mediator);

}

@Override

public void work() {

System.out.println("user1 exe!");

}

}

public class User2 extends User {

public User2(Mediator mediator){

super(mediator);

}

@Override

public void work() {

System.out.println("user2 exe!");

}

}

测试类：

[java] view plaincopy

public class Test {

public static void main(String[] args) {

Mediator mediator = new MyMediator();

mediator.createMediator();

mediator.workAll();

}

}

输出：

user1 exe!

user2 exe!

相关模式

Facade

与中介者的不同之处在于它是对一个对象子系统进行抽象，从而提供了一个更为方便的接口。它的协议是单向的，即Facade对象对这个子系统类提出请求，但反之则不行。相反，Mediator提供了各Colleague对象不支持或不能支持的协作行为，而且协议是多向的。

Colleague

可使用Observer(5.7)模式与Mediator通信。

Memento

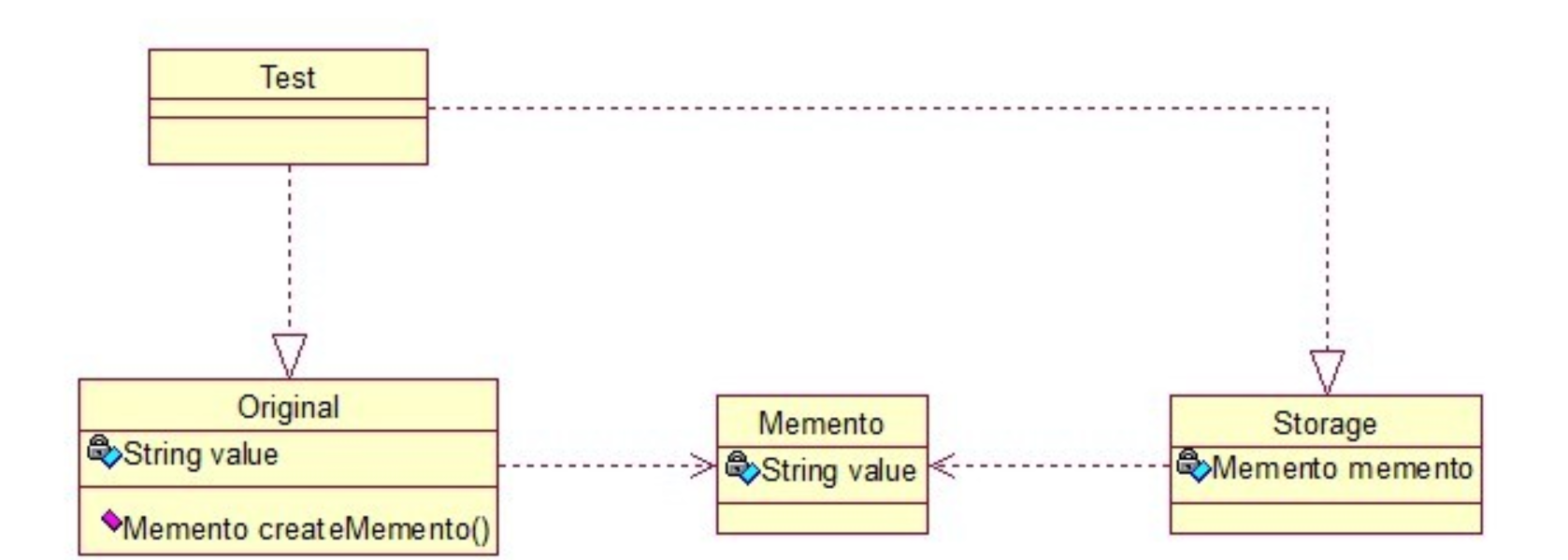
协 作

管理器向原发器请求一个备忘录,保留一段时间后,将其送回给原发器

有时管理者不会将备忘录返回给原发器,因为原发器可能根本不需要退到先前的状态。

代码示例

主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类A，A中有各种属性，A可以决定需 要备份的属性，备忘录类B是用来存储A的一些内部状态，类C呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。做个图来分析一下：



Original类是原始类，里面有需要保存的属性value及创建一个备忘录类，用来保存value值。Memento类是备忘录类，Storage类是存储备忘录的类，持有Memento类的实例，该模式很好理解。直接看源码：

public class Original {

private String value;

public String getValue() {

return value;

}

public void setValue(String value) {

this.value = value;

}

public Original(String value) {

this.value = value;

}

public Memento createMemento(){

return new Memento(value);

}

public void restoreMemento(Memento memento){

this.value = memento.getValue();

}

}

public class Memento {

private String value;

public Memento(String value) {

this.value = value;

}

public String getValue() {

return value;

}

public void setValue(String value) {

this.value = value;

}

}

public class Storage {

private Memento memento;

public Storage(Memento memento) {

this.memento = memento;

}

public Memento getMemento() {

return memento;

}

public void setMemento(Memento memento) {

this.memento = memento;

}

}

测试类：

public class Test {

public static void main(String[] args) {

// 创建原始类

Original origi = new Original("egg");

// 创建备忘录

Storage storage = new Storage(origi.createMemento());

// 修改原始类的状态

System.out.println("初始化状态为：" + origi.getValue());

origi.setValue("niu");

System.out.println("修改后的状态为：" + origi.getValue());

// 回复原始类的状态

origi.restoreMemento(storage.getMemento());

System.out.println("恢复后的状态为：" + origi.getValue());

}

}

输出：

初始化状态为：egg

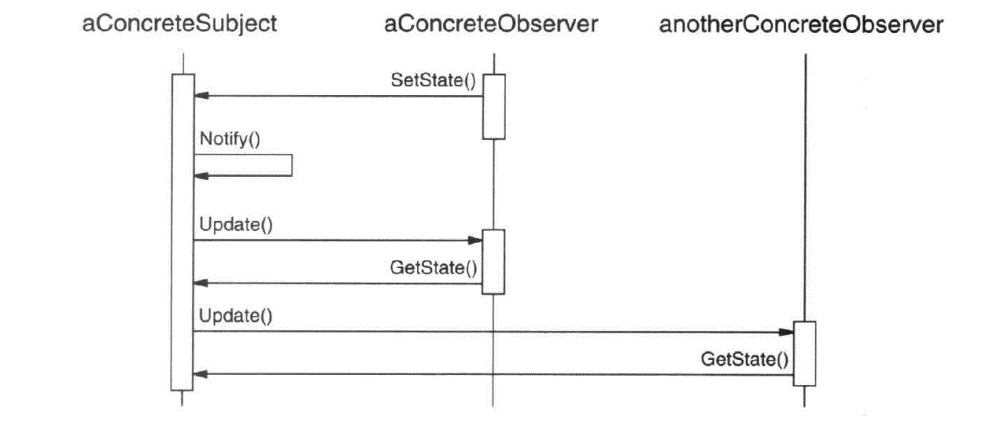
修改后的状态为：niu

恢复后的状态为：egg

简单描述下：新建原始类时，value被初始化为egg，后经过修改，将value的值置为niu，最后倒数第二行进行恢复状态，结果成功恢复了。其实我觉得这个模式叫“备份-恢复”模式最形象。

Observer

协 作



　　注意发出改变请求的Observer对象并不立即更新,而是将其推迟到它从目标得到一个通知之后。Notify不总是由目标对象调用。它也可被一个观察者或其它对象调用。实现一节将讨论一些常用的变化。

效 果

目标和观察者间的抽象耦合一个目标所知道的仅仅是它有一系列观察者,每个都符合抽象的Observer类的简单接口。目标不知道任何一个观察者属于哪一个具体的类。这样目标和观察者之间的耦合是抽象的和最小的。

因为目标和观察者不是紧密耦合的,它们可以属于一个系统中的不同抽象层次。一个处于较低层次的目标对象可与一个处于较高层次的观察者通信并通知它,这样就保持了系统层次的完整。如果目标和观察者混在一块,那么得到的对象要么横贯两个层次(违反了层次性),要么必须放在这两层的某一层中(这可能会损害层次抽象)。

意外的更新因为一个观察者并不知道其它观察者的存在,它可能对改变目标的最终代价一无所知。

在目标上一个看似无害的的操作可能会引起一系列对观察者以及依赖于这些观察者的那些对象的更新。此外,如果依赖准则的定义或维护不当，常常会引起错误的更新,这种错误通常很难捕捉。

　　简单的更新协议不提供具体细节说明目标中什么被改变了,这就使得上述问题更加严重。

　　如果没有其他协议帮助观察者发现什么发生了改变，它们可能会被迫尽力减少改变。

实 现

创建目标到其观察者之间的映射一个目标对象跟踪它应通知的观察者的最简单的方法是显式地在目标中保存对它们的引用。

然而,当目标很多而观察者较少时,这样存储可能代价太高。一个解决办法是用时间换空间,用一个关联查找机制(例如一个hash表)来维护目标到观察者的映射。这样一个没有观察者的目标就不产生存储开销。但另一方面,这一方法增加了访问观察者的开销。

观察多个目标在某些情况下,一个观察者依赖于多个目标可能是有意义的。

例如,一个表格对象可能依赖于多个数据源。在这种情况下,必须扩展Update接口以使观察者知道是哪一个目标送来的通知。目标对象可以简单地将自己作为Update操作的一个参数,让观察者知道应去检查哪一个目标。

谁触发更新目标和它的观察者依赖于通知机制来保持一致。

但到底哪一个对象调用Notify来触发更新?此时有两个选择:a)由目标对象的状态设定操作在改变目标对象的状态后自动调用Notify。这种方法的优点是客户不需要记住要在目标对象上调用Notify，缺点是多个连续的操作会产生多次连续的更新,可能效率较低。b)让客户负责在适当的时候调用Notify。这样做的优点是客户可以在一系列的状态改变完成后再一次性地触发更新,避免了不必要的中间更新。缺点是给客户增加了触发更新的责任。由于客户可能会忘记调用Notify，这种方式较易出错。

对已删除目标的悬挂引用删除一个目标时应注意不要在其观察者中遗留对该目标的悬挂引用。

一种避免悬挂引用的方法是,当一个目标被删除时，让它通知它的观察者将对该目标的引用复位。一般来说,不能简单地删除观察者,因为其他的对象可能会引用它们,或者也可能它们还在观察其他的目标。

在发出通知前确保目标的状态自身是一致的在发出通知前确保状态自身一致这一点很重要,因为观察者在更新其状态的过程中需要查询目标的当前状态。

当Subject的子类调用继承的该项操作时,很容易无意中违反这条自身一致的准则。例如,下面的代码序列中,在目标尚处于一种不一致的状态时，通知就被触发了:你可以用抽象的Subject类中的模板方法(TemplateMethod(5.10))发送通知来避免这种错误。定义那些子类可以重定义的原语操作,并将Notify作为模板方法中的最后一个操作,这样当子类重定义了Subject的操作时，还可以保证该对象的状态是自身一致的。

　　顺便提一句，在文档中记录是哪一个Subject操作触发通知总是应该的6)避免特定于观察者的更新协议—推/拉模型观察者模式的实现经常需要让目标广播关于其改变的其他一些信息。目标将这些信息作为Update操作一个参数传递出去。这些信息的量可能很小，也可能很大。

　　一个极端情况是，目标向观察者发送关于改变的详细信息,而不管它们需要与否。我们称之为推模型(pushmodel)。另一个极端是拉模型(pullmodel);目标除最小通知外什么也不送出,而在此之后由观察者显式地向目标询问细节。

　　拉模型强调的是目标不知道它的观察者,而推模型假定目标知道一些观察者的需要的信息。推模型可能使得观察者相对难以复用，因为目标对观察者的假定可能并不总是正确的。

　　另一方面。拉模型可能效率较差,因为观察者对象需在没有目标对象帮助的情况下确定什么改变了。

显式地指定感兴趣的改变你可以扩展目标的注册接口,让各观察者注册为仅对特定事件感兴趣，以提高更新的效率。

当一个事件发生时,目标仅通知那些已注册为对该事件感兴趣的观察者。支持这种做法一种途径是，对使用目标对象的方面（aspects）的概念。可用如下代码将观察者对象注册为对目标对象的某特定事件感兴趣：voidSubject::Attach(Observer\*,Aspect&interest);此处此处interest指定感兴趣的事件。在通知的时刻,目标将这方面的改变作为Update操作的一个参数提供给它的观察者，例如:voidObserver::Update(Subject\*,Aspect&interest);8)封装复杂的更新语义当目标和观察者间的依赖关系特别复杂时,可能需要一个维护这些关系的对象。我们称这样的对象为更改管理器（ChangeManager）。它的目的是尽量减少观察者反映其目标的状态变化所需的工作量。例如,如果一个操作涉及到对几个相互依赖的目标进行改动,就必须保证仅在所有的目标都已更改完毕后，才一次性地通知它们的观察者,而不是每个目标都通知观察者。

　　ChangeManager有三个责任:a)它将一个目标映射到它的观察者并提供一个接口来维护这个映射。这就不需要由目标来维护对其观察者的引用,反之亦然。

　　b)它定义一个特定的更新策略。

　　c)根据一个目标的请求,它更新所有依赖于这个目标的观察者。

　　下下页的框图描述了一个简单的基于ChangeManager的Observer模式的实现。有两种特殊的ChangeManager。SimpleChangeManager总是更新每一个目标的所有观察者,比较简单。相反,DAGChangeManager处理目标及其观察者之间依赖关系构成的无环有向图。当一个观察者观察多个目标时,DAGChangeManager要比SimpleChangeManager更好一些。在这种情况下,两个或更多个目标中产生的改变可能会产生冗余的更新。DAGChangeManager保证观察者仅接收一个更新。当然，当不存在多重更新的问题时,SimpleChangeManager更好一些。

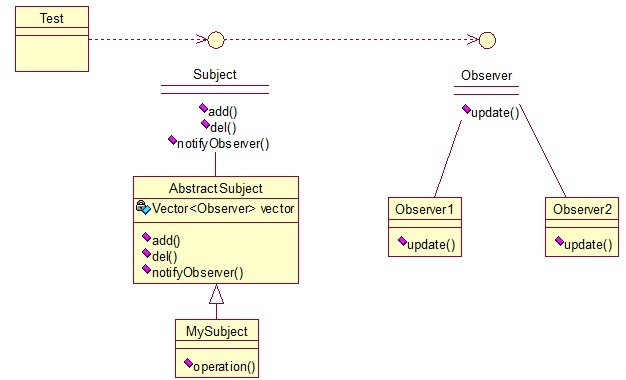
　　ChangeManager是一个Mediator(5.5)模式的实例。通常只有一个ChangeManager,并且它是全局可见的。这里Singleton(3.5)模式可能有用。

结合目标类和观察者类用不支持多重继承的语言(如Smalltalk)书写的类库通常不单独定义Subject和Observer类,而是将它们的接口结合到一个类中。

这就允许你定义一个既是一个目标又是一个观察者的对象，而不需要多重继承。例如在Smalltalk中,Subject和Observer接口定义于根类Object中，使得它们对所有的类都可用。

代码示例

包括这个模式在内的接下来的四个模式，都是类和类之间的关系，不涉及到继承，学的时候应该 记得归纳，记得本文最开始的那个图。观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意 思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！ 对象之间是一种一对多的关系。先来看看关系图：



我解释下这些类的作用：MySubject类就是我们的主对象，Observer1和Observer2是依赖于MySubject的对象，当 MySubject变化时，Observer1和Observer2必然变化。AbstractSubject类中定义着需要监控的对象列表，可以对其进 行修改：增加或删除被监控对象，且当MySubject变化时，负责通知在列表内存在的对象。我们看实现代码：

一个Observer接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Observer {
2. **public** **void** update();
3. }

两个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Observer1 **implements** Observer {
3. @Override
4. **public** **void** update() {
5. System.out.println("observer1 has received!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** Observer2 **implements** Observer {
3. @Override
4. **public** **void** update() {
5. System.out.println("observer2 has received!");
6. }
8. }

Subject接口及实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **interface** Subject {
3. /\*增加观察者\*/
4. **public** **void** add(Observer observer);
6. /\*删除观察者\*/
7. **public** **void** del(Observer observer);
9. /\*通知所有的观察者\*/
10. **public** **void** notifyObservers();
12. /\*自身的操作\*/
13. **public** **void** operation();
14. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **abstract** **class** AbstractSubject **implements** Subject {
3. **private** Vector<Observer> vector = **new** Vector<Observer>();
4. @Override
5. **public** **void** add(Observer observer) {
6. vector.add(observer);
7. }
9. @Override
10. **public** **void** del(Observer observer) {
11. vector.remove(observer);
12. }
14. @Override
15. **public** **void** notifyObservers() {
16. Enumeration<Observer> enumo = vector.elements();
17. **while**(enumo.hasMoreElements()){
18. enumo.nextElement().update();
19. }
20. }
21. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** MySubject **extends** AbstractSubject {
3. @Override
4. **public** **void** operation() {
5. System.out.println("update self!");
6. notifyObservers();
7. }
9. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8243942)

1. **public** **class** ObserverTest {
3. **public** **static** **void** main(String[] args) {
4. Subject sub = **new** MySubject();
5. sub.add(**new** Observer1());
6. sub.add(**new** Observer2());
8. sub.operation();
9. }
11. }

输出：

update self!

observer1 has received!

observer2 has received!

这些东西，其实不难，只是有些抽象，不太容易整体理解，建议读者：**根据关系图，新建项目，自己写代码（或者参考我的代码）,按照总体思路走一遍，这样才能体会它的思想，理解起来容易！**

State

适用性

一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。

这个状态通常用一个或多个枚举常量表示。通常,有多个操作包含这一相同的条件结构。State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

效 果

它将与特定状态相关的行为局部化，并且将不同状态的行为分割开来State模式将所有与一个特定的状态相关的行为都放入一个对象中。

因为所有与状态相关的代码都存在于某一个State子类中,所以通过定义新的子类可以很容易的增加新的状态和转换。

　　另一个方法是使用数据值定义内部状态并且让Context操作来显式地检查这些数据。但这样将会使整个Context的实现中遍布看起来很相似的条件语句或case语句。增加一个新的状态可能需要改变若干个操作,这就使得维护变得复杂了。

　　State模式避免了这个问题,但可能会引入另一个问题,因为该模式将不同状态的行为分布在多个State子类中。这就增加了子类的数目，相对于单个类的实现来说不够紧凑。但是如果有许多状态时这样的分布实际上更好一些,否则需要使用巨大的条件语句。

　　正如很长的过程一样，巨大的条件语句是不受欢迎的。它们形成一大整块并且使得代码不够清晰，这又使得它们难以修改和扩展。State模式提供了一个更好的方法来组织与特定状态相关的代码。决定状态转移的逻辑不在单块的if或switch语句中,而是分布在State子类之间。

　　将每一个状态转换和动作封装到一个类中，就把着眼点从执行状态提高到整个对象的状态。

　　这将使代码结构化并使其意图更加清晰。

它使得状态转换显式化当一个对象仅以内部数据值来定义当前状态时,其状态仅表现为对一些变量的赋值，这不够明确。

为不同的状态引入独立的对象使得转换变得更加明确。

　　而且,State对象可保证Context不会发生内部状态不一致的情况，因为从Context的角度看，状态转换是原子的—只需重新绑定一个变量(即Context的State对象变量)，而无需为多个变量赋值[dCLF93]。

实 现

谁定义状态转换State模式不指定哪一个参与者定义状态转换准则。

如果该准则是固定的,那么它们可在Context中完全实现。然而若让State子类自身指定它们的后继状态以及何时进行转换,通常更灵活更合适。这需要Context增加一个接口,让State对象显式地设定Context的当前状态。

　　用这种方法分散转换逻辑可以很容易地定义新的State子类来修改和扩展该逻辑。这样做的一个缺点是，一个State子类至少拥有一个其他子类的信息,这就再各子类之间产生了实现依赖。

基于表的另一种方法在C++ProgrammingStyle[Car92]中,Cargil描述了另一种将结构加载在状态驱动的代码上的方法:他使用表将输入映射到状态转换。

对每一个状态,一张表将每一个可能的输入映射到一个后继状态。实际上,这种方法将条件代码(和State模式下的虚函数)映射为一个查找表。

　　表的主要好处是它们的规则性:你可以通过更改数据而不是更改程序代码来改变状态转换的准则。然而它也有一些缺点:•对表的查找通常不如(虚)函数调用效率高。

　　•用统一的、表格的形式表示转换逻辑使得转换准则变得不够明确而难以理解。

　　•通常难以加入伴随状态转换的一些动作。表驱动的方法描述了状态和它们之间的转换,但必须扩充这个机制以便在每一个转换上能够进行任意的计算。

　　表驱动的状态机和State模式的主要区别可以被总结如下:State模式对与状态相关的行为进行建模,而表驱动的方法着重于定义状态转换。

创建和销毁State对象一个常见的值得考虑的实现上的权衡是,究竟是(1)仅当需要State对象时才创建它们并随后销毁它们，还是(2)提前创建它们并且始终不销毁它们。

当将要进入的状态在运行时是不可知的,并且上下文不经常改变状态时,第一种选择较为可取。这种方法避免创建不会被用到的对象,如果State对象存储大量的信息时这一点很重要。

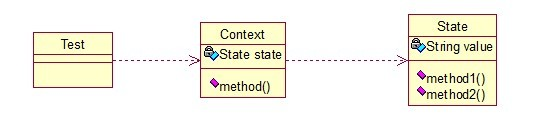
　　当状态改变很频繁时,第二种方法较好。在这种情况下最好避免销毁状态,因为可能很快再次需要用到它们。此时可以预先一次付清创建各个状态对象的开销,并且在运行过程中根本不存在销毁状态对象的开销。但是这种方法可能不太方便,因为Context必须保存对所有可能会进入的那些状态的引用。

使用动态继承改变一个响应特定请求的行为可以用在运行时刻改变这个对象的类的办法实现,但这在大多数面向对象程序设计语言中都是不可能的。

Self[US87]和其他一些基于委托的语言却是例外，它们提供这种机制,从而直接支持State模式。Self中的对象可将操作委托给其他对象以达到某种形式的动态继承。在运行时刻改变委托的目标有效地改变了继承的结构。这一机制允许对象改变它们的行为，也就是改变它们的类。

代码示例

核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！就拿QQ来说，有几种状态，在线、隐身、忙碌等，每个状态对应不同的操作，而且你的好友也 能看到你的状态，所以，状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。看图：



State类是个状态类，Context类可以实现切换，我们来看看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **package** com.xtfggef.dp.state;
3. /\*\*
4. \* 状态类的核心类
5. \* 2012-12-1
6. \* @author erqing
7. \*
8. \*/
9. **public** **class** State {
11. **private** String value;
13. **public** String getValue() {
14. **return** value;
15. }
17. **public** **void** setValue(String value) {
18. **this**.value = value;
19. }
21. **public** **void** method1(){
22. System.out.println("execute the first opt!");
23. }
25. **public** **void** method2(){
26. System.out.println("execute the second opt!");
27. }
28. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **package** com.xtfggef.dp.state;
3. /\*\*
4. \* 状态模式的切换类 2012-12-1
5. \* @author erqing
6. \*
7. \*/
8. **public** **class** Context {
10. **private** State state;
12. **public** Context(State state) {
13. **this**.state = state;
14. }
16. **public** State getState() {
17. **return** state;
18. }
20. **public** **void** setState(State state) {
21. **this**.state = state;
22. }
24. **public** **void** method() {
25. **if** (state.getValue().equals("state1")) {
26. state.method1();
27. } **else** **if** (state.getValue().equals("state2")) {
28. state.method2();
29. }
30. }
31. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8245537)

1. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
5. State state = **new** State();
6. Context context = **new** Context(state);
8. //设置第一种状态
9. state.setValue("state1");
10. context.method();
12. //设置第二种状态
13. state.setValue("state2");
14. context.method();
15. }
16. }

输出：

execute the first opt!  
execute the second opt!

根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。