设计模式

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

决定对象的粒度

指定对象接口

描述对象的实现

运用复用机制

关联运行时刻和编译时刻的结构

设计应支持变化

浏览模式的意图部分

研究模式怎样互相关联

研究目的相似的模式

检查重新设计的原因

考虑你的设计中哪些是可变的

怎样使用设计模式

大致浏览一遍模式

回头研究结构部分、参与者部分和协作部分

看代码示例部分，看看这个模式代码形式的具体例子

选择模式参与者的名字，使它们在应用上下文中有意义

定义类

定义模式中专用于应用的操作名称

实现执行模式中责任和协作的操作

创 建 型

类

Factory Method

意 图

定义一个用于创建对象的接口，让子类决定实例化哪一个类。 FactoryMethod使一个类的实例化延迟到其子类

别 名

虚构造器（Virtual Constructor）

动 机

框架使用抽象类定义和维护对象之间的关系。这些对象的创建通常也由框架负责

适用性

当一个类不知道它所必须创建的对象的类的时候。

当一个类希望由它的子类来指定它所创建的对象的时候

当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候

结 构

参与者

P r o d u c t

定义工厂方法所创建的对象的接口

C o n c r e t e P r o d u c t

实现P r o d u c t接口

C r e a t o r

声明工厂方法，该方法返回一个 P r o d u c t类型的对象。C r e a t o r也可以定义一个工厂方法的缺省实现，它返回一个缺省的 C o n c r e t e P r o d u c t对象

可以调用工厂方法以创建一个P r o d u c t对象

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

重定义工厂方法以返回一个C o n c r e t e P r o d u c t实例

协 作

Creator依赖于它的子类来定义工厂方法，所以它返回一个适当的 C o n c r e t e P r o d u c t实例

效 果

工厂方法的一个潜在缺点在于客户可能仅仅为了创建一个特定的 C o n c r e t e P r o d u c t对象，就不得不创建C r e a t o r的子类。当C r e a t o r子类不必需时，客户现在必然要处理类演化的其他方面；但是当客户无论如何必须创建 C r e a t o r的子类时，创建子类也是可行的。

为子类提供挂钩（h o o k）

连接平行的类层次

实 现

主要有两种不同的情况

C r e a t o r类是一个抽象类并且不提供它所声明的工厂方法的实现。

C r e a t o r是一个具体的类而且为工厂方法提供一个缺省的实现。

用一个独立的操作创建对象，这样子类才能重定义它们的创建方式

参数化工厂方法

特定语言的变化和问题

使用模板以避免创建子类

命名约定

代码示例

相关模式

Abstract Factory（3 . 1）经常用工厂方法来实现。

工厂方法通常在Template Methods（5 . 1 0）中被调用。

P r o t o t y p e s（3 . 4）不需要创建C r e a t o r的子类。但是，它们通常要求一个针对 P r o d u c t类的I n i t i a l i z e操作。C r e a t o r使用I n i t i a l i z e来初始化对象。而Factory Method不需要这样的操作。

对象

Abstract Factory

意 图

别 名

动 机

适用性

一个系统要独立于它的产品的创建、组合和表示时

一个系统要由多个产品系列中的一个来配置时

当你要强调一系列相关的产品对象的设计以便进行联合使用时

当你提供一个产品类库，而只想显示它们的接口而不是实现时

结 构

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

C o n c r e t e P r o d u c t

定义一个将被相应的具体工厂创建的产品对象

实现A b s t r a c t P r o d u c t接口

C l i e n t

仅使用由A b s t r a c t F a c t o r y和A b s t r a c t P r o d u c t类声明的接口

协 作

效 果

它分离了具体的类

它使得易于交换产品系列

它有利于产品的一致性

难以支持新种类的产品

实 现

将工厂作为单件

创建产品

定义可扩展的工厂

代码示例

B u i l d e r

意图

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。

动机

适用性

当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时

当构造过程必须允许被构造的对象有不同的表示时

结 构

参与者

B u i l d e r

为创建一个P r o d u c t对象的各个部件指定抽象接口

C o n c r e t e B u i l d e r

实现B u i l d e r的接口以构造和装配该产品的各个部件

定义并明确它所创建的表示

提供一个检索产品的接口

Director

构造一个使用B u i l d e r接口的对象

P r o d u c t

表示被构造的复杂对象

包含定义组成部件的类，包括将这些部件装配成最终产品的接口

协作

客户创建D i r e c t o r对象，并用它所想要的B u i l d e r对象进行配置

一旦产品部件被生成，导向器就会通知生成器

生成器处理导向器的请求，并将部件添加到该产品中

客户从生成器中检索产品

效果

它使你可以改变一个产品的内部表示

它将构造代码和表示代码分开

它使你可对构造过程进行更精细的控制

实现

装配和构造接口

为什么产品没有抽象类

在B u i l d e r中却省的方法为空

代码示例

相关模式

P r o t o t y p e

意 图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象

别 名

动 机

适用性

当一个系统应该独立于它的产品创建、构成和表示时

当要实例化的类是在运行时刻指定时

为了避免创建一个与产品类层次平行的工厂类层次时

当一个类的实例只能有几个不同状态组合中的一种时

结 构

参与者

P r o t o t y p e

声明一个克隆自身的接口

C o n c r e t e P r o t o t y p e

实现一个克隆自身的操作

C l i e n t

让一个原型克隆自身从而创建一个新的对象

协 作

客户请求一个原型克隆自身

效 果

P r o t o t y p e有许多和Abstract Factory（3 . 1）和B u i l d e r（3 . 2）一样的效果：它对客户隐藏了具体的产品类，因此减少了客户知道的名字的数目。此外，这些模式使客户无需改变即可使用与特定应用相关的类。

运行时刻增加和删除产品

改变值以指定新对象

改变结构以指定新对象

减少子类的构造

用类动态配置应用

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

实 现

使用一个原型管理器

实现克隆操作

初始化克隆对象

代码示例

相关模式

大量使用C o m p o s i t e（4 . 3）和D e c o r a t o r（4 . 4）模式的设计通常也可从 P r o t o t y p e模式处获益。

Subtopic

P r o t o t y p e和Abstract Factory（3 . 1）模式在某种方面是相互竞争的。但是它们也可以一起使用。 Abstract Factory可以存储一个被克隆的原型的集合，并且返回产品对象。

S i n g l e t o n

意 图

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。

别 名

动 机

对一些类来说，只有一个实例是很重要的。

适用性

当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

结 构

参与者

S i n g l e t o n

定义一个 I n s t a n c e操作，允许客户访问它的唯一实例。 I n s t a n c e是一个类操作（即S m a l l t a l k中的一个类方法和C + +中的一个静态成员函数）。

可能负责创建它自己的唯一实例。

协 作

客户只能通过S i n g l e t o n的I n s t a n c e操作访问一个S i n g l e t o n的实例。

效 果

对唯一实例的受控访问

缩小名空间

允许对操作和表示的精化

允许可变数目的实例

比类操作更灵活

实 现

保证一个唯一的实例

创建S i n g l e t o n类的子类

代码示例

相关模式

很多模式可以使用 S i n g l e t o n模式实现。参见 Abstract Factory（3 . 1）、B u i l d e r（3 . 2），和P r o t o t y p e（3 . 4）。

创建型模式的讨论

结 构 型

类

A d a p t e r (类)

意 图

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。

Subtopic

别 名

包装器 Wr a p p e r。

动 机

有时，为复用而设计的工具箱类不能够被复用的原因仅仅是因为它的接口与专业应用领域所需要的接口不匹配。

适用性

你想使用一个已经存在的类，而它的接口不符合你的需求。

你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作

（仅适用于对象A d a p t e r）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

结 构

类适配器

对象适配器

参与者

Ta r g e t

定义C l i e n t使用的与特定领域相关的接口。

C l i e n t

与符合Ta rg e t接口的对象协同

A d a p t e e

定义一个已经存在的接口，这个接口需要适配。

A d a p t e r

对A d a p t e e的接口与Ta rg e t接口进行适配

协 作

Client在A d a p t e r实例上调用一些操作。接着适配器调用 A d a p t e e的操作实现这个请求。

效 果

对象适配器

允许一个A d a p t e r与多个A d a p t e e — 即A d a p t e e本身以及它的所有子类（如果有子类的话）—同时工作。A d a p t e r也可以一次给所有的A d a p t e e添加功能。

使得重定义A d a p t e e的行为比较困难。这就需要生成 A d a p t e e的子类并且使得A d a p t e r引用这个子类而不是引用A d a p t e e本身

类适配器

用一个具体的A d a p t e r类对A d a p t e e和Ta rg e t进行匹配。结果是当我们想要匹配一个类以及所有它的子类时，类A d a p t e r将不能胜任工作。

使得A d a p t e r可以重定义A d a p t e e的部分行为，因为A d a p t e r是A d a p t e e的一个子类。

仅仅引入了一个对象，并不需要额外的指针以间接得到 a d a p t e e。

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

可插入的Adapter

使用双向适配器提供透明操作

实 现

使用C + +实现适配器类

可插入的适配器

使用抽象操作

使用代理对象

参数化的适配器

代码示例

类

相关模式

B r i d g e目的是将接口部分和实现部分分离，从而对它们可以较为容易也相对独立的加以改变。而 A d a p t e r则意味着改变一个已有对象的接口。

D e c o r a t o r ( 4 . 4 )模式增强了其他对象的功能而同时又不改变它的接口。因此 d e c o r a t o r对应用程序的透明性比适配器要好。结果是 d e c o r a t o r支持递归组合，而纯粹使用适配器是不可能实现这一点的。

模式P r o x y ( 4 . 7 )在不改变它的接口的条件下，为另一个对象定义了一个代理。

对象

A d a p t e r (对象)

意 图

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。

Subtopic

别 名

包装器 Wr a p p e r。

动 机

有时，为复用而设计的工具箱类不能够被复用的原因仅仅是因为它的接口与专业应用领域所需要的接口不匹配。

适用性

你想使用一个已经存在的类，而它的接口不符合你的需求。

你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作

（仅适用于对象A d a p t e r）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

结 构

类适配器

对象适配器

参与者

Ta r g e t

定义C l i e n t使用的与特定领域相关的接口。

C l i e n t

与符合Ta rg e t接口的对象协同

A d a p t e e

定义一个已经存在的接口，这个接口需要适配。

A d a p t e r

对A d a p t e e的接口与Ta rg e t接口进行适配

协 作

Client在A d a p t e r实例上调用一些操作。接着适配器调用 A d a p t e e的操作实现这个请求。

效 果

对象适配器

允许一个A d a p t e r与多个A d a p t e e — 即A d a p t e e本身以及它的所有子类（如果有子类的话）—同时工作。A d a p t e r也可以一次给所有的A d a p t e e添加功能。

使得重定义A d a p t e e的行为比较困难。这就需要生成 A d a p t e e的子类并且使得A d a p t e r引用这个子类而不是引用A d a p t e e本身

类适配器

用一个具体的A d a p t e r类对A d a p t e e和Ta rg e t进行匹配。结果是当我们想要匹配一个类以及所有它的子类时，类A d a p t e r将不能胜任工作。

使得A d a p t e r可以重定义A d a p t e e的部分行为，因为A d a p t e r是A d a p t e e的一个子类。

仅仅引入了一个对象，并不需要额外的指针以间接得到 a d a p t e e。

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

可插入的Adapter

使用双向适配器提供透明操作

实 现

使用C + +实现适配器类

可插入的适配器

使用抽象操作

使用代理对象

参数化的适配器

代码示例

类

相关模式

B r i d g e目的是将接口部分和实现部分分离，从而对它们可以较为容易也相对独立的加以改变。而 A d a p t e r则意味着改变一个已有对象的接口。

D e c o r a t o r ( 4 . 4 )模式增强了其他对象的功能而同时又不改变它的接口。因此 d e c o r a t o r对应用程序的透明性比适配器要好。结果是 d e c o r a t o r支持递归组合，而纯粹使用适配器是不可能实现这一点的。

模式P r o x y ( 4 . 7 )在不改变它的接口的条件下，为另一个对象定义了一个代理。

B r i d g e

意 图

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。

别 名

H a n d l e / B o d y

动 机

当一个抽象可能有多个实现时，通常用继承来协调它们。抽象类定义对该抽象的接口，而具体的子类则用不同方式加以实现。但是此方法有时不够灵活。继承机制将抽象部分与它的实现部分固定在一起，使得难以对抽象部分和实现部分独立地进行修改、扩充和重用。

适用性

你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。

类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时 B r i d g e模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

（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。

正如在意图一节的第一个类图中所示的那样，有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。 R u m b a u g h称这种类层次结构为“嵌套的普化”（nested generalizations）

你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。一个简单的例子便是C o p l i e n的S t r i n g类[ C o p 9 2 ]，在这个类中多个对象可以共享同一个字符串表示（S t r i n g R e p）。

结 构

参与者

Abstraction

定义抽象类的接口。

维护一个指向I m p l e m e n t o r类型对象的指针

RefinedAbstraction

扩充由A b s t r a c t i o n定义的接口

Implementor

定义实现类的接口，该接口不一定要与 A b s t r a c t i o n的接口完全一致；事实上这两个接口可以完全不同。一般来讲， I m p l e m e n t o r接口仅提供基本操作，而 A b s t r a c t i o n则定义了基于这些基本操作的较高层次的操作。

ConcreteImplementor

实现I m p l e m e n t o r接口并定义它的具体实现

协 作

Abstraction将c l i e n t的请求转发给它的I m p l e m e n t o r对象

效 果

分离接口及其实现部分

提高可扩充性

实现细节对客户透明

实 现

仅有一个Implementor

创建正确的I m p l e m e n t o r对象

共享I m p l e m e n t o r对象

采用多重继承机制

代码示例

相关模式

Abstract Factory(3.1) 模式可以用来创建和配置一个特定的 B r i d g e模式。

Adapter(4.1) 模式用来帮助无关的类协同工作，它通常在系统设计完成后才会被使用然 而，B r i d g e模式则是在系统开始时就被使用，它使得抽象接口和实现部分可以独立进行改变。

C o m p o s i t e

意 图

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使得用户对单个对象和组合对象的使用具有一致性。

别 名

动 机

适用性

你想表示对象的部分-整体层次结构。

你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象

结 构

子主题 1

子主题 2

参与者

Component

为组合中的对象声明接口。

在适当的情况下，实现所有类共有接口的缺省行为。

声明一个接口用于访问和管理C o m p o n e n t的子组件。

(可选)在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它。

Leaf

在组合中表示叶节点对象，叶节点没有子节点。

在组合中定义图元对象的行为。

Composite

定义有子部件的那些部件的行为。

存储子部件。

在C o m p o n e n t接口中实现与子部件有关的操作。

Client

通过C o m p o n e n t接口操纵组合部件的对象。

协 作

用户使用C o m p o n e n t类接口与组合结构中的对象进行交互。如果接收者是一个叶节点 ,则直接处理请求。如果接收者是 Composite, 它通常将请求发送给它的子部件，在转发请求之前与/或之后可能执行一些辅助操作。

效 果

定义了包含基本对象和组合对象的类层次结构

简化客户代码

使得更容易增加新类型的组件

使你的设计变得更加一般化

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

共享组件

最大化C o m p o n e n t接口

声明管理子部件的操作

Component是否应该实现一个C o m p o n e n t列表

子部件排序

使用高速缓冲存贮改善性能

应该由谁删除Component

存贮组件最好用哪一种数据结构

代码示例

相关模式

通常部件-父部件连接用于Responsibility of Chain(5.1)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。

F l y w e i g h t ( 4 . 6 )让你共享组件，但不再能引用他们的父部件。

I t e r t o r ( 5 . 4 )可用来遍历C o m p o s i t e。

Vi s i t o r ( 5 . 11 )将本来应该分布在C o m p o s i t e和L e a f类中的操作和行为局部化。

D e c o r a t o r

意 图

动态地给一个对象添加一些额外的职责。就增加功能来说， Decorator模式相比生成子类更为灵活。

别 名

包装器Wr a p p e r

动 机

有时我们希望给某个对象而不是整个类添加一些功能。

适用性

在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

处理那些可以撤消的职责。

当不能采用生成子类的方法进行扩充时。

可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。

是因为类定义被隐藏，或类定义不能用于生成子类。

结 构

参与者

Component

定义一个对象接口，可以给这些对象动态地添加职责

C o n c r e t e C o m p o n e n t

定义一个对象，可以给这个对象添加一些职责

D e c o r a t o r

维持一个指向C o m p o n e n t对象的指针，并定义一个与C o m p o n e n t接口一致的接口。

C o n c r e t e D e c o r a t o r

向组件添加职责

协 作

D e c o r a t o r将请求转发给它的C o m p o n e n t对象，并有可能在转发请求前后执行一些附加的动作

效 果

优点

比静态继承更灵活

避免在层次结构高层的类有太多的特征

Decorator与它的C o m p o n e n t不一样

缺点

有许多小对象

实 现

接口的一致性

省略抽象的D e c o r a t o r类

保持C o m p o n e n t类的简单性

改变对象外壳与改变对象内核

代码示例

相关模式

A d a p t e r ( 4 . 1 )模式：D e c o r a t o r模式不同于A d a p t e r模式，因为装饰仅改变对象的职责而不改变它的接口；而适配器将给对象一个全新的接口。

C o m p o s i t e ( 4 . 3 )模式：可以将装饰视为一个退化的、仅有一个组件的组合。然而，装饰仅给对象添加一些额外的职责 — 它的目的不在于对象聚集。

S t r a t e g y ( 5 . 9 )模式：用一个装饰你可以改变对象的外表；而 S t r a t e g y模式使得你可以改变对象的内核。这是改变对象的两种途径。

F a c a d e

意 图

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。

别 名

动 机

将一个系统划分成为若干个子系统有利于降低系统的复杂性。

适用性

当你要为一个复杂子系统提供一个简单接口时。

客户程序与抽象类的实现部分之间存在着很大的依赖性。

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。

结 构

参与者

F a c a d e

知道哪些子系统类负责处理请求。

将客户的请求代理给适当的子系统对象。

Subsystem classes

实现子系统的功能。

处理由F a c a d e对象指派的任务。

没有f a c a d e的任何相关信息；即没有指向f a c a d e的指针。

协 作

客户程序通过发送请求给 F a c a d e的方式与子系统通讯，F a c a d e将这些消息转发给适当的子系统对象。尽管是子系统中的有关对象在做实际工作，但 F a c a d e模式本身也必须将它的接口转换成子系统的接口。

使用F a c a d e的客户程序不需要直接访问子系统对象。

效 果

它对客户屏蔽子系统组件，因而减少了客户处理的对象的数目并使得子系统使用起来更加方便。

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。

如果应用需要，它并不限制它们使用子系统类。

实 现

降低客户-子系统之间的耦合度

公共子系统类与私有子系统类

代码示例

相关模式

Abstract Factory (3.1)模式可以与F a c a d e模式一起使用以提供一个接口，这一接口可用来以一种子系统独立的方式创建子系统对象。 Abstract Factory也可以代替F a c a d e模式隐藏那些与平台相关的类。

Mediator (5.5) 模式与F a c a d e模式的相似之处是，它抽象了一些已有的类的功能。然而，M e d i a t o r的目的是对同事之间的任意通讯进行抽象，通常集中不属于任何单个对象的功能。M e d i a t o r的同事对象知道中介者并与它通信，而不是直接与其他同类对象通信。相对而言，F a c a d e模式仅对子系统对象的接口进行抽象，从而使它们更容易使用；它并不定义新功能，子系统也不知道f a c a d e的存在。

通常来讲，仅需要一个F a c a d e对象，因此F a c a d e对象通常属于Singleton (3.5)模式。

P r o x y

意 图

为其他对象提供一种代理以控制对这个对象的访问。

别 名

Surrogate

动 机

对一个对象进行访问控制的一个原因是为了只有在我们确实需要这个对象时才对它进行创建和初始化。

适用性

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。

虚代理（Virtual Proxy）根据需要创建开销很大的对象。

保护代理（Protection Proxy）控制对原始对象的访问。

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。

结 构

参与者

P r o x y

保存一个引用使得代理可以访问实体。若 R e a l S u b j e c t和S u b j e c t的接口相同，P r o x y会引用S u b j e c t

提供一个与S u b j e c t的接口相同的接口，这样代理就可以用来替代实体。

控制对实体的存取，并可能负责创建和删除它。

其他功能依赖于代理的类型：

Remote Pro x y负责对请求及其参数进行编码，并向不同地址空间中的实体发送已编码的请求。

Vi rtual Pro x y可以缓存实体的附加信息，以便延迟对它的访问。例如，动机一节中提到的I m a g e P r o x y缓存了图像实体的尺寸。

P rotection Pro x y检查调用者是否具有实现一个请求所必需的访问权限。

S u b j e c t

定义RealSubject 和P r o x y的共用接口，这样就在任何使用R e a l S u b j e c t的地方都可以使用P r o x y。

R e a l S u b j e c t

定义P r o x y所代表的实体。

协 作

代理根据其种类，在适当的时候向 R e a l S u b j e c t转发请求。

效 果

Remote Proxy可以隐藏一个对象存在于不同地址空间的事实。

Virtual Proxy 可以进行最优化，例如根据要求创建对象。

Protection Proxies和Smart Reference都允许在访问一个对象时有一些附加的内务处理（Housekeeping task）。

实 现

代码示例

相关模式

A d a p t e r ( 4 . 1 )：适配器A d a p t e r为它所适配的对象提供了一个不同的接口。相反，代理提供了与它的实体相同的接口。然而，用于访问保护的代理可能会拒绝执行实体会执行的操作，因此，它的接口实际上可能只是实体接口的一个子集。

D e c o r a t o r ( 4 . 4 )：尽管 d e c o r a t o r的实现部分与代理相似，但 d e c o r a t o r的目的不一样。D e c o r a t o r为对象添加一个或多个功能，而代理则控制对对象的访问。

代理的实现与d e c o r a t o r的实现类似，但是在相似的程度上有所差别。 Protection Proxy的实现可能与d e c o r a t o r的实现差不多。另一方面， Remote Proxy不包含对实体的直接引用，而只是一个间接引用，如“主机 I D，主机上的局部地址。”Virtual Proxy开始的时候使用一个间接引用，例如一个文件名，但最终将获取并使用一个直接引用。

F l y w e i g h t

意 图

运用共享技术有效地支持大量细粒度的对象。

别 名

动 机

有些应用程序得益于在其整个设计过程中采用对象技术，但简单化的实现代价极大。

适用性

一个应用程序使用了大量的对象。

完全由于使用大量的对象，造成很大的存储开销。

对象的大多数状态都可变为外部状态。

如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

应用程序不依赖于对象标识。由于 F l y w e i g h t对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

结 构

参与者

F l y w e i g h t

描述一个接口，通过这个接口f l y w e i g h t可以接受并作用于外部状态。

C o n c r e t e F l y w e i g h t

实 现 F l y w e i g h t 接 口 ， 并 为 内 部 状 态 （ 如 果 有 的 话 ） 增 加 存 储 空 间 。C o n c r e t e F l y w e i g h t对象必须是可共享的。它所存储的状态必须是内部的；即，它必须独立于C o n c r e t e F l y w e i g h t对象的场景。

U n s h a r e d C o n c r e t e F l y w e i g h t

并非所有的F l y w e i g h t子类都需要被共享。 F l y w e i g h t接口使共享成为可能，但它并不强制共享。在F l y w e i g h t对象结构的某些层次， U n s h a r e d C o n c r e t e F l y w e i g h t对象通常将C o n c r e t e F l y w e i g h t对象作为子节点（R o w和C o l u m n就是这样）。

F l y w e i g h t F a c t o r y

创建并管理f l y w e i g h t对象。

确保合理地共享f l y w e i g h t。当用户请求一个f l y w e i g h t时，F l y w e i g h t F a c t o r y对象提供一个已创建的实例或者创建一个（如果不存在的话）。

Client

维持一个对f l y w e i g h t的引用。

计算或存储一个（多个）f l y w e i g h t的外部状态。

协 作

f l y w e i g h t执行时所需的状态必定是内部的或外部的。

内部状态存储于 C o n c r e t e F l y w e i g h t对象之中；

外部对象则由 C l i e n t对象存储或计算

当用户调用 f l y w e i g h t对象的操作时，将该状态传递给它。

用户不应直接对C o n c r e t e F l y w e i g h t类进行实例化，而只能从 F l y w e i g h t F a c t o r y对象得到C o n c r e t e F l y w e i g h t对象，这可以保证对它们适当地进行共享。

效 果

存储节约由以下几个因素决定

因为共享，实例总数减少的数目

对象内部状态的平均数目

外部状态是计算的还是存储的

实 现

删除外部状态

管理共享对象

代码示例

相关模式

F l y w e i g h t模式通常和C o m p o s i t e ( 4 . 3 )模式结合起来，用共享叶结点的有向无环图实现一个逻辑上的层次结构。

通常，最好用F l y w e i g h t实现S t a t e ( 5 . 8 )和S t r a t e g y ( 5 . 9 )对象。

结构型模式的讨论

Adapter与Bridge

Composite、Decorator与Proxy

行 为 型

类

Interpreter

意 图

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。

别 名

动 机

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。这样就可以构建一个解释器,该解释器通过解释这些句子来解决该问题。

适用性

当有一个语言需要解释执行,并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。

该文法简单对于复杂的文法,文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式,这样可以节省空间而且还可能节省时间。

效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的,而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下,转换器仍可用解释器模式实现,该模式仍是有用的。

结 构

参与者

AbstractExpression

声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享。

TerminalExpression

实现与文法中的终结符相关联的解释操作。

一个句子中的每个终结符需要该类的一个实例。

NonterminalExpression

对文法中的每一条规则R::=R1R2...Rn都需要一个NonterminalExpression类。

为从R1到Rn的每个符号都维护一个AbstractExpression类型的实例变量。

为文法中的非终结符实现解释(Interpret)操作。解释(Interpret)一般要递归地调用表示R1到Rn的那些对象的解释操作。

Context

包含解释器之外的一些全局信息。

Client

构建(或被给定)表示该文法定义的语言中一个特定的句子的抽象语法树。该抽象语法树由NonterminalExpression和TerminalExpression的实例装配而成。

调用解释操作。

协 作

Client构建(或被给定)一个句子,它是NonterminalExpression和TerminalExpression的实例的一个抽象语法树.然后初始化上下文并调用解释操作。

每一非终结符表达式节点定义相应子表达式的解释操作。而各终结符表达式的解释操作构成了递归的基础。

每一节点的解释操作用上下文来存储和访问解释器的状态。

效 果

1)易于改变和扩展文法因为该模式使用类来表示文法规则,你可使用继承来改变或扩展该文法。已有的表达式可被增量式地改变,而新的表达式可定义为旧表达式的变体。

也易于实现文法定义抽象语法树中各个节点的类的实现大体类似。这些类易于直接编写，通常它们也可用一个编译器或语法分析程序生成器自动生成。

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。

实 现

创建抽象语法树解释器模式并未解释如何创建一个抽象的语法树。换言之,它不涉及语法分析。抽象语法树可用一个表驱动的语法分析程序来生成，也可用手写的(通常为递归下降法)语法分析程序创建，或直接由Client提供。

定义解释操作并不一定要在表达式类中定义解释操作。如果经常要创建一种新的解释器,那么使用Visitor（5.11）模式将解释放入一个独立的“访问者”对象更好一些。例如,一个程序设计语言的会有许多在抽象语法树上的操作，比如类型检查、优化、代码生成，等等。恰当的做法是使用一个访问者以避免在每一个类上都定义这些操作。

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。

代码示例

相关模式

Composite模式（4.3）

抽象语法树是一个复合模式的实例。

Flyweight模式

说明了如何在抽象语法树中共享终结符。

Iterator

解释器可用一个迭代器遍历该结构。

Visitor

可用来在一个类中维护抽象语法树中的各节点的行为。

Template Method

意 图

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。TemplateMethod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。

别 名

动机考虑一个提供Application和Document类的应用框架。

动 机

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

适用性

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke和Johnson所描述过的“重分解以一般化”的一个很好的例子[OJ93]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook”操作（参见效果一节），这样就只允许在这些点进行扩展。

结 构

参与者

AbstractClass（抽象类，如Application）

定义抽象的原语操作（primitiveoperation），具体的子类将重定义它们以实现一个算法的各步骤。

实现一个模板方法,定义一个算法的骨架。该模板方法不仅调用原语操作，也调用定义在AbstractClass或其他对象中的操作。

ConcreteClass（具体类，如MyApplication）

实现原语操作以完成算法中与特定子类相关的步骤。

协 作

ConcreteClass靠AbstractClass来实现算法中不变的步骤。

效 果

模板方法是一种代码复用的基本技术。它们在类库中尤为重要，它们提取了类库中的公共行为。

模板方法导致一种反向的控制结构，这种结构有时被称为“好莱坞法则”，即“别找我们，我们找你”[Swe85]。这指的是一个父类调用一个子类的操作，而不是相反。

模板方法调用下列类型的操作

具体的操作（ConcreteClass或对客户类的操作）。

具体的AbstractClass的操作（即，通常对子类有用的操作）。

原语操作（即，抽象操作）。

FactoryMethod（参见FactoryMethod（3.5））。

钩子操作（hookoperations），它提供了缺省的行为，子类可以在必要时进行扩展。

实 现

使用C++访问控制在C++中，一个模板方法调用的原语操作可以被定义为保护成员。

尽量减少原语操作定义模板方法的一个重要目的是尽量减少一个子类具体实现该算法时必须重定义的那些原语操作的数目。

命名约定可以给应被重定义的那些操作的名字加上一个前缀以识别它们。

代码示例

相关模式

Strategy（5.9）：模板方法使用继承来改变算法的一部分。Strategy使用委托来改变整个算法。

对象

ChainofResponsibility

意 图

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。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为止。

别 名

考虑一个图形用户界面中的上下文有关的帮助机制。

动 机

适用性

有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。

你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

可处理一个请求的对象集合应被动态指定。

结 构

参与者

H a n d l e r

— 定义一个处理请求的接口。

— （可选） 实现后继链。

C o n c r e t e H a n d l e r

— 处理它所负责的请求。

— 可访问它的后继者。

— 如果可处理该请求，就处理之；否则将该请求转发给它的后继者。

C l i e n t

— 向链上的具体处理者( C o n c r e t e H a n d l e r )对象提交请求。

协 作

当客户提交一个请求时，请求沿链传递直至有一个 ConcreteHandler 对象负责处理它。

效 果

降低耦合度

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

不保证被接受

实 现

实现后继者链

定义新的链接(通常在H a n d l e r中定义，但也可由ConcreteHandlers 来定义)。

使用已有的链接

连接后继者

在S m a l l t a l k中自动转发

表示请求

代码示例

相关模式

职责链常与C o m p o s i t e（4 . 3）一起使用。

这种情况下，一个构件的父构件可作为它的后继。

Command

意 图

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

别 名

动作( A c t i o n )，事务( Tr a n s a c t i o n )

动 机

有时必须向某对象提交请求，但并不知道关于被请求的操作或请求的接受者的任何信息。

适用性

抽象出待执行的动作以参数化某对象。

　　可用过程语言中的回调（callback）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。Command模式是回调机制的一个面向对象的替代品。

在不同的时刻指定、排列和执行请求。

支持取消操作。

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。

用构建在原语操作上的高层操作构造一个系统。

结 构

参与者

Command

-声明执行操作的接口。

ConcreteCommand(PasteCommand，OpenCommand)

—将一个接收者对象绑定于一个动作。

—调用接收者相应的操作，以实现Execute。

Client(Appliction)

—创建一个具体命令对象并设定它的接收者。

•Invoker(MenuItem)

—要求该命令执行这个请求。

Receiver(Document，Application)

—知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

协 作

Client创建一个ConcreteCommand对象并指定它的Receiver对象。

某Invoker对象存储该ConcreteCommand对象。

ConcreteCommand对象对调用它的Receiver的一些操作以执行该请求。

效 果

Command模式将调用操作的对象与知道如何实现该操作的对象解耦。

Command是头等的对象。它们可像其他的对象一样被操纵和扩展。

你可将多个命令装配成一个复合命令。例如是前面描述的MacroCommand类。一般说来，复合命令是Composite模式的一个实例。

增加新的Command很容易，因为这无需改变已有的类。

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。

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。

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。

使用C++模板对(1)不能被取消(2)不需要参数的命令，我们可使用C++模板来实现，这样可以避免为每一种动作和接收者都创建一个Command子类。我们将在代码示例一节说明这种做法。

代码示例

相关模式

Composite模式（4.3）可被用来实现宏命令。

Memento模式（5.6）可用来保持某个状态，命令用这一状态来取消它的效果。

在被放入历史表列前必须被拷贝的命令起到一种原型(3.4)的作用。

Iterator

意 图

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。

别 名

游标（Cursor）。

动 机

一个聚合对象,如列表(list),应该提供一种方法来让别人可以访问它的元素，而又不需暴露它的内部结构.此外，针对不同的需要，可能要以不同的方式遍历这个列表。但是即使可以预见到所需的那些遍历操作，你可能也不希望列表的接口中充斥着各种不同遍历的操作。有时还可能需要在同一个表列上同时进行多个遍历。

适用性

访问一个聚合对象的内容而无需暴露它的内部表示。

支持对聚合对象的多种遍历。

为遍历不同的聚合结构提供一个统一的接口(即,支持多态迭代)。

结 构

参与者

Iterator（迭代器）

迭代器定义访问和遍历元素的接口。

ConcreteIterator（具体迭代器）

具体迭代器实现迭代器接口。

对该聚合遍历时跟踪当前位置。

Aggregate（聚合）

聚合定义创建相应迭代器对象的接口。

ConcreteAggregate（具体聚合）

具体聚合实现创建相应迭代器的接口，该操作返回ConcreteIterator的一个适当的实例。

协 作

ConcreteIterator跟踪聚合中的当前对象，并能够计算出待遍历的后继对象。

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。

迭代器简化了聚合的接口有了迭代器的遍历接口，聚合本身就不再需要类似的遍历接口了。这样就简化了聚合的接口。

在同一个聚合上可以有多个遍历每个迭代器保持它自己的遍历状态。因此你可以同时进行多个遍历。

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。

谁定义遍历算法迭代器不是唯一可定义遍历算法的地方。

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。

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。

在C++中使用多态的迭代器使用多态迭代器是有代价的。

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。

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。

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。

代码示例

相关模式

Composite(4.3)

迭代器常被应用到象复合这样的递归结构上。

FactoryMethod

多态迭代器靠FactoryMethod来例化适当的迭代器子类。

Memento

常与迭代器模式一起使用。迭代器可使用一个memento来捕获一个迭代的状态。迭代器在其内部存储memento。

Mediator

意 图

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。

别 名

动 机

面向对象设计鼓励将行为分布到各个对象中。这种分布可能会导致对象间有许多连接。

在最坏的情况下,每一个对象都知道其他所有对象。

虽然将一个系统分割成许多对象通常可以增强可复用性 , 但是对象间相互连接的激增又会降低其可复用性。大量的相互连接使得一个对象似乎不太可能在没有其他对象的支持下工作— 系统表现为一个不可分割的整体。而且 ,对系统的行为进行任何较大的改动都十分困难，因为行为被分布在许多对象中。结果是 , 你可能不得不定义很多子类以定制系统的行为。

适用性

一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。

想定制一个分布在多个类中的行为，而又不想生成太多的子类。

结 构

参与者

Mediator(中介者

中介者定义一个接口用于与各同事（Colleague）对象通信。

ConcreteMediator(具体中介者，如FontDialogDirector)

具体中介者通过协调各同事对象实现协作行为。

了解并维护它的各个同事。

Colleagueclass(同事类，如ListBox,EntryField)

每一个同事类都知道它的中介者对象。

每一个同事对象在需与其他的同事通信的时候，与它的中介者通信。

协 作

同事向一个中介者对象发送和接收请求。中介者在各同事间适当地转发请求以实现协作行为。

效 果

减少了子类生成Mediator将原本分布于多个对象间的行为集中在一起。改变这些行为只需生成Meditator的子类即可。这样各个Colleague类可被重用。

它将各Colleague解耦Mediator有利于各Colleague间的松耦合.你可以独立的改变和复用各Colleague类和Mediator类。

它简化了对象协议用Mediator和各Colleague间的一对多的交互来代替多对多的交互一对多的关系更易于理解、维护和扩展。

它对对象如何协作进行了抽象将中介作为一个独立的概念并将其封装在一个对象中，使你将注意力从对象各自本身的行为转移到它们之间的交互上来。这有助于弄清楚一个系统中的对象是如何交互的。

它使控制集中化中介者模式将交互的复杂性变为中介者的复杂性。因为中介者封装了协议,它可能变得比任一个Colleague都复杂。这可能使得中介者自身成为一个难于维护的庞然大物。

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。

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

代码示例

相关模式

Facade

Colleague

Memento

意 图

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原先保存的状态。

别 名

Token

动 机

有时有必要记录一个对象的内部状态。

为了允许用户取消不确定的操作或从错误中恢复过来，需要实现检查点和取消机制,而要实现这些机制，你必须事先将状态信息保存在某处，这样才能将对象恢复到它们先前的状态。但是对象通常封装了其部分或所有的状态信息,使得其状态不能被其他对象访问，也就不可能在该对象之外保存其状态。而暴露其内部状态又将违反封装的原则，可能有损应用的可靠性和可扩展性。

适用性

必须保存一个对象在某一个时刻的(部分)状态,这样以后需要时它才能恢复到先前的状态。

如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

结 构

参与者

Memento(备忘录，如SolverState)

备忘录存储原发器对象的内部状态。原发器根据需要决定备忘录存储原发器的哪些内部状态。

防止原发器以外的其他对象访问备忘录。备忘录实际上有两个接口，管理者(caretaker)只能看到备忘录的窄接口—它只能将备忘录传递给其他对象。相反,原发器能够看到一个宽接口,允许它访问返回到先前状态所需的所有数据。理想的情况是只允许生成本备忘录的那个原发器访问本备忘录的内部状态。

Originator(原发器，如ConstraintSolver)

原发器创建一个备忘录,用以记录当前时刻它的内部状态。

使用备忘录恢复内部状态.。

Caretaker(负责人，如undomechanism)

负责保存好备忘录。

不能对备忘录的内容进行操作或检查。

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

备忘录是被动的。只有创建备忘录的原发器会对它的状态进行赋值和检索。

效 果

保持封装边界使用备忘录可以避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。该模式把可能很复杂的Originator内部信息对其他对象屏蔽起来,从而保持了封装边界。

它简化了原发器在其他的保持封装性的设计中,Originator负责保持客户请求过的内部状态版本。这就把所有存储管理的重任交给了Originator。让客户管理它们请求的状态将会简化Originator,并且使得客户工作结束时无需通知原发器。

使用备忘录可能代价很高如果原发器在生成备忘录时必须拷贝并存储大量的信息,或者客户非常频繁地创建备忘录和恢复原发器状态，可能会导致非常大的开销。除非封装和恢复Originator状态的开销不大,否则该模式可能并不合适。参见实现一节中关于增量式改变的讨论。

定义窄接口和宽接口在一些语言中可能难以保证只有原发器可访问备忘录的状态。

维护备忘录的潜在代价管理器负责删除它所维护的备忘录。然而,管理器不知道备忘录中有多少个状态。因此当存储备忘录时，一个本来很小的管理器，可能会产生大量的存储开销。

实 现

语言支持备忘录有两个接口:一个为原发器所使用的宽接口,一个为其他对象所使用的窄接口。理想的实现语言应可支持两级的静态保护。

存储增量式改变如果备忘录的创建及其返回（给它们的原发器）的顺序是可预测的，备忘录可以仅存储原发器内部状态的增量改变。

代码示例

相关模式

Command

命令可使用备忘录来为可撤消的操作维护状态。

Iterator

如前所述备忘录可用于迭代.

Observer

意 图

定义对象间的一种一对多的依赖关系,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被自动更新。

别 名

依赖(Dependents),发布-订阅(Publish-Subscribe)

动 机

将一个系统分割成一系列相互协作的类有一个常见的副作用：需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合，因为这样降低了它们的可重用性。

适用性

当一个抽象模型有两个方面,其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

当对一个对象的改变需要同时改变其它对象,而不知道具体有多少对象有待改变。

当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之,你不希望这些对象是紧密耦合的。

结 构

参与者

参与者•Subject（目标）

目标知道它的观察者。可以有任意多个观察者观察同一个目标。

提供注册和删除观察者对象的接口。

Observer（观察者）

为那些在目标发生改变时需获得通知的对象定义一个更新接口。

ConcreteSubject（具体目标）

将有关状态存入各ConcreteObserver对象。

当它的状态发生改变时,向它的各个观察者发出通知。

ConcreteObserver（具体观察者）

维护一个指向ConcreteSubject对象的引用。

存储有关状态，这些状态应与目标的状态保持一致。

实现Observer的更新接口以使自身状态与目标的状态保持一致。

协 作

当ConcreteSubject发生任何可能导致其观察者与其本身状态不一致的改变时，它将通知它的各个观察者。

在得到一个具体目标的改变通知后,ConcreteObserver对象可向目标对象查询信息ConcreteObserver使用这些信息以使它的状态与目标对象的状态一致。。

效 果

Observer模式允许你独立的改变目标和观察者。你可以单独复用目标对象而无需同时复用其观察者,反之亦然。它也使你可以在不改动目标和其他的观察者的前提下增加观察者。

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。

支持广播通信不像通常的请求,目标发送的通知不需指定它的接收者。通知被自动广播给所有已向该目标对象登记的有关对象。目标对象并不关心到底有多少对象对自己感兴趣;它唯一的责任就是通知它的各观察者。这给了你在任何时刻增加和删除观察者的自由。处理还是忽略一个通知取决于观察者。

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。

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。

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。

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。

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。

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。

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。

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。

代码示例

相关模式

Mediator

通过封装复杂的更新语义, ChangeManager充当目标和观察者之间的中介者。

Singleton

ChangeManager可使用Singleton模式来保证它是唯一的并且是可全局访问的。

State

意 图

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。

别 名

状态对象（ObjectsforStates）

动 机

适用性

一个对象的行为取决于它的状态,并且它必须在运行时刻根据状态改变它的行为。

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。

结 构

参与者

Context(环境，如TCPConnection)

定义客户感兴趣的接口。

State(状态，如TCPState)

定义一个接口以封装与Context的一个特定状态相关的行为。

ConcreteStatesubclasses(具体状态子类，如TCPEstablished,TCPListen,TCPClosed)

每一子类实现一个与Context的一个状态相关的行为。

协 作

Context将与状态相关的请求委托给当前的ConcreteState对象处理。

Context可将自身作为一个参数传递给处理该请求的状态对象。这使得状态对象在必要时可访问Context。

Context是客户使用的主要接口。客户可用状态对象来配置一个Context，一旦一个Context配置完毕,它的客户不再需要直接与状态对象打交道。

Context或ConcreteState子类都可决定哪个状态是另外哪一个的后继者，以及是在何种条件下进行状态转换。

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。

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。

State对象可被共享如果State对象没有实例变量—即它们表示的状态完全以它们的类型来编码—那么各Context对象可以共享一个State对象。当状态以这种方式被共享时,它们必然是没有内部状态,只有行为的轻量级对象(参见Flyweight（4.6）)。

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。

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。

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。

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。

代码示例

相关模式

Flyweight模式(4.6)解释了何时以及怎样共享状态对象。

状态对象通常是Singleton(3.5)。

Strategy

意 图

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。本模式使得算法可独立于使用它的客户而变化。

别 名

政策（Policy）

动 机

有许多算法可对一个正文流进行分行。

适用性

许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。

需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时[HO87],可以使用策略模式。

算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

一个类定义了多种行为,并且这些行为在这个类的操作中以多个条件语句的形式出现。

将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

结 构

参与者

Strategy(策略，如Compositor)

定义所有支持的算法的公共接口。Context使用这个接口来调用某ConcreteStrategy定义的算法。

ConcreteStrategy(具体策略，如SimpleCompositor,TeXCompositor,ArrayCompositor)

以Strategy接口实现某具体算法。

Context(上下文，如Composition)

用一个ConcreteStrategy对象来配置。

维护一个对Strategy对象的引用。

可定义一个接口来让Stategy访问它的数据。

协 作

Strategy和Context相互作用以实现选定的算法。

效 果

相关算法系列Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能。

一个替代继承的方法继承提供了另一种支持多种算法或行为的方法。

消除了一些条件语句Strategy模式提供了用条件语句选择所需的行为以外的另一种选择。

实现的选择Strategy模式可以提供相同行为的不同实现。客户可以根据不同时间/空间权衡取舍要求从不同策略中进行选择。

客户必须了解不同的Strategy本模式有一个潜在的缺点，就是一个客户要选择一个合适的Strategy就必须知道这些Strategy到底有何不同。

Strategy和Context之间的通信开销无论各个ConcreteStrategy实现的算法是简单还是复杂,它们都共享Strategy定义的接口。

增加了对象的数目Strategy增加了一个应用中的对象的数目。

实 现

定义Strategy和Context接口Strategy和Context接口必须使得ConcreteStrategy能够有效的访问它所需要的Context中的任何数据,反之亦然。

将Strategy作为模板参数在C++中，可利用模板机制用一个Strategy来配置一个类。

使Strategy对象成为可选的如果即使在不使用额外的Strategy对象的情况下，Context也还有意义的话，那么它还可以被简化。

代码示例

相关模式

Flyweight（4.6）：Strategy对象经常是很好的轻量级对象。

Visitor

意 图

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。

别 名

动 机

适用性

一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。

•需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Visitor使得你可以将相关的操作集中起来定义在一个类中。

定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

结 构

参与者

参与者•Visitor（访问者，如NodeVisitor）

为该对象结构中ConcreteElement的每一个类声明一个Visit操作。该操作的名字和特征标识了发送Visit请求给该访问者的那个类。这使得访问者可以确定正被访问元素的具体的类。这样访问者就可以通过该元素的特定接口直接访问它。

ConcreteVisitor（具体访问者，如TypeCheckingVisitor）

实现每个由Visitor声明的操作。每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVisitor为该算法提供了上下文并存储它的局部状态。

这一状态常常在遍历该结构的过程中累积结果。

Element（元素，如Node）

定义一个Accept操作，它以一个访问者为参数。

ConcreteElement（具体元素，如AssignmentNode，VariableRefNode）

实现Accept操作，该操作以一个访问者为参数。

ObjectStructure（对象结构，如Program）

能枚举它的元素。

可以提供一个高层的接口以允许该访问者访问它的元素。

可以是一个复合（参见Composite（4.3））或是一个集合，如一个列表或一个无序集合。

协 作

一个使用Visitor模式的客户必须创建一个ConcreteVisitor对象，然后遍历该对象结构，并用该访问者访问每一个元素。

当一个元素被访问时，它调用对应于它的类的Visitor操作。如果必要，该元素将自身作为这个操作的一个参数以便该访问者访问它的状态。

效 果

访问者模式使得易于增加新的操作访问者使得增加依赖于复杂对象结构的构件的操作变得容易了。仅需增加一个新的访问者即可在一个对象结构上定义一个新的操作。相反，如果每个功能都分散在多个类之上的话，定义新的操作时必须修改每一类。

访问者集中相关的操作而分离无关的操作相关的行为不是分布在定义该对象结构的各个类上，而是集中在一个访问者中。无关行为却被分别放在它们各自的访问者子类中。这就既简化了这些元素的类，也简化了在这些访问者中定义的算法。所有与它的算法相关的数据结构都可以被隐藏在访问者中。

增加新的ConcreteElement类很困难Visitor模式使得难以增加新的Element的子类。每添加一个新的ConcreteElement都要在Vistor中添加一个新的抽象操作，并在每一个ConcretVisitor类中实现相应的操作。有时可以在Visitor中提供一个缺省的实现，这一实现可以被大多数的ConcreteVisitor继承，但这与其说是一个规律还不如说是一种例外。

通过类层次进行访问一个迭代器（参见Iterator（5.4））可以通过调用节点对象的特定操作来遍历整个对象结构，同时访问这些对象。

累积状态当访问者访问对象结构中的每一个元素时，它可能会累积状态。如果没有访问者，这一状态将作为额外的参数传递给进行遍历的操作，或者定义为全局变量。

破坏封装访问者方法假定ConcreteElement接口的功能足够强，足以让访问者进行它们的工作。结果是，该模式常常迫使你提供访问元素内部状态的公共操作，这可能会破坏它的封装性。

实 现

代码示例

相关模式

Composite（4.3）：访问者可以用于对一个由Composite模式定义的对象结构进行操作。

Interpreter（5.3）：访问者可以用于解释。

讨论

封装变化

对象作为参数

通信应该被封装还是被分布

对发送者和接收者解耦

总结

总结