新版设计模式手册

一．创建型模式

1．单件模式

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。

适用性

1. 􀁺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。
2. 􀁺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。

2．抽象工厂

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。

适用性

1. 􀁺一个系统要独立于它的产品的创建、组合和表示时。
2. 􀁺一个系统要由多个产品系列中的一个来配置时。
3. 􀁺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。
4. 􀁺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。

3．建造者模式

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。

适用性

1. 􀁺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。
2. 􀁺当构造过程必须允许被构造的对象有不同的表示时。

4．工厂方法模式

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。Factory Method 使一个类的实例化延迟到其子类。

适用性

1. 􀁺当一个类不知道它所必须创建的对象的类的时候。
2. 􀁺当一个类希望由它的子类来指定它所创建的对象的时候。
3. 􀁺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。

5．原型模式

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。

适用性

1. 􀁺当要实例化的类是在运行时刻指定时，例如，通过动态装载；或者
2. 􀁺为了避免创建一个与产品类层次平行的工厂类层次时；或者
3. 􀁺当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

二．结构型模式

6．适配器模式

意图

将一个类的接口转换成客户希望的另外一个接口。Adapter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。

适用性

1. 􀁺你想使用一个已经存在的类，而它的接口不符合你的需求。
2. 􀁺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。
3. 􀁺（仅适用于对象Adapter）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。

7．桥接模式

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。

适用性

1. 􀁺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。
2. 􀁺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Bridge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。
3. 􀁺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。
4. 􀁺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。
5. 􀁺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。

8．组合模式

意图

将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

适用性

1. 􀁺你想表示对象的部分-整体层次结构。
2. 􀁺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。

9．装饰模式

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。

适用性

1. 􀁺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。
2. 􀁺处理那些可以撤消的职责。
3. 􀁺当不能采用生成子类的方法进行扩充时。一种情况是，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。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

10．外观模式

意图

为子系统中的一组接口提供一个一致的界面，Facade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。

适用性

1. 􀁺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。Facade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Facade层。
2. 􀁺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Facade 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。
3. 􀁺当你需要构建一个层次结构的子系统时，使用Facade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Facade 进行通讯，从而简化了它们之间的依赖关系。

11．享元模式

意图

运用共享技术有效地支持大量细粒度的对象。

适用性

1. 􀁺一个应用程序使用了大量的对象。
2. 􀁺完全由于使用大量的对象，造成很大的存储开销。
3. 􀁺对象的大多数状态都可变为外部状态。
4. 􀁺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。
5. 􀁺应用程序不依赖于对象标识。由于Flyweight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。

12．代理模式

意图

为其他对象提供一种代理以控制对这个对象的访问。

适用性

1. 􀁺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Proxy模式。下面是一 些可以使用Proxy模式常见情况：

1) 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NXProxy类实现了这一目的。Coplien[Cop92] 称这种代理为“大使”（Ambassador）。

2 )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ImageProxy就是这样一种代理的例子。

3) 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Choices操作系统[CIRM93]中KemelProxies为操作系统对象提供 了访问保护。

4 )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：

1. 􀂾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SmartPointers[Ede92]。
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。

三．行为型模式

13．职责链模式

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。

适用性

1. 􀁺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。
2. 􀁺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。
3. 􀁺可处理一个请求的对象集合应被动态指定。

14．命令模式

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。

适用性

1. 􀁺使用命令模式作为"CallBack"在面向对象系统中的替代。"CallBack"讲的便是先将一个函数登记上，然后在以后调用此函数。
2. 􀁺需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。
3. 这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在串形化之后传送到另外一台机器上去。
4. 􀁺系统需要支持命令的撤消(undo)。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供redo()方法，以供客户端在需要时，再重新实施命令效果。
5. 􀁺如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用Execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。
6. 􀁺一个系统需要支持交易(Transaction)。一个交易结构封装了一组数据更新命令。使用命令模式来实现交易结构可以使系统增加新的交易类型。

15．解释器模式

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。

适用性

1. 􀁺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。而当存在以下情况时该模式效果最好：
2. 􀁺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。
3. 􀁺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。

16．迭代器模式

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。

适用性

1. 􀁺访问一个聚合对象的内容而无需暴露它的内部表示。
2. 􀁺支持对聚合对象的多种遍历。
3. 􀁺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)。

17．中介者模式

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。

适用性

1. 􀁺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。
2. 􀁺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。
3. 􀁺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。

18．备忘录模式

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。

适用性

1. 􀁺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。
2. 􀁺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。

19．观察者模式

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。

适用性

1. 􀁺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。
2. 􀁺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。
3. 􀁺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。

20．状态模式

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。

适用性

1. 􀁺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。
2. 􀁺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。

21．策略模式

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。

适用性

1. 􀁺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。
2. 􀁺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[ H O 8 7 ] ,可以使用策略模式。
3. 􀁺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。
4. 􀁺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。

22．模版方法

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。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。

适用性

1. 􀁺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。
2. 􀁺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[ O J 9 3 ]。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。
3. 􀁺控制子类扩展。模板方法只在特定点调用hook操作，这样就只允许在这些点进行扩展。

23．访问者模式

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。

适用性

1. 􀁺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。
2. 􀁺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。当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作。
3. 􀁺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。