GRASP 设计原则

作者：valarchie

原文链接：https://ld246.com/article/1566743555361

来源网站：链滴

GRASP设计原则

GRASP(General Responsibility Assignment Software Pattern)是通用职责分配软件设计模式。

它由《UML和模式应用》(Applying UML and Patterns)一书作者Craig Larman提出。在面向对象计的过程中一般的通用方式是构思对象的职责、角色和协作。通常来说，我们在编码过程中先分析问域，从中抽象出对象解决问题。简单的面向对象和优良的面向对象设计的区别在于将如何更合理的划对象的角色，给对象赋予合理的职责以及对象之间的交互关系。

在我所理解的GRASP九大原则、SOLID七大原则、GOF23种设计模式这三类设计原则模式中。GRAS处于最上层，SOLID基于它再进一步细化阐述，GOF再根据这些原则进一步的归纳出更具体的模式。

GoF模式是针对特定问题而提出的解决方案，而GRASP站在一个更高的角度来看待面向对象软件的设，它是GoF设计模式的基础。GRASP是对象职责分配的基本原则，其核心思想是职责分配，用职责设对象。

信息专家模式 (Information Expert)

定义：

如果某个类拥有执行某个职责所必需的的信息的话，那么将这个职责分配给这个类。

理解：

当我们不确定某个职责该分配给类A还是类B的时候，我们可以遵循这个原则。这个设计原则和单一设原则不同，单一职责原则考虑的是单个类中的职责是否都属于一类职责。而信息专家模式考虑则是该该同一类职责放进类A还是类B中。假设我们有一个长方形Rectangle类（类中有width和height属性和一个Measure类，我们应该把getArea()方法放进Rectangle中去，还是将width和height参数传给easure类，在Measure中实现getArea()呢？依照该准则，既然Rectangle方法已经有了实现getArea(所必须的属性的话，那么就把该把getArea()方法放进Retangle类中。同理如果有一个计算属性呢？设是长宽高比例widthHeightRatio的话，也遵循该原则。==这个原则和DDD设计思想当中的充血模是一致的，大家可以了解一下。==

创造者(Creator)

定义：

如果符合下面的一个或者多个条件，则可将创建类A实例的职责分配给类B

- B包含A；
- B聚合A；
- B拥有初始化A的数据并在创建类A的实例时将数据传递给类A；
- B记录A的实例；
- B频繁使用A。

理解：

在面向对象的设计当中，无法避免去创建对象。假设对象B创建对象A，那么对象B就产生了与对象A

耦合。而这种耦合是无法消除的，即使你将创建对象A的职责分配给对象C，这种耦合还是存在的，是从对象B转移到对象C上，系统内还是依然存在这个耦合，无法避免。那么当我们无法消除耦合的时，我们应该考虑的是如何降低这个耦合的耦合度。这个原则给出了指导方针。以上的几个条件潜在的明了，其实B已经对A有了耦合，既然B已经存在了对A的耦合，那么我们不妨再将创建A的职责分配给。这样分配的话，系统内仅存在一个A与B的耦合。如果将创建A的职责分配给C的话，那么系统内就存在B与A(B包含A、B频繁使用A等条件)和C与A这两个耦合。在JDK的Map中Entry类的创建职责就配给了持有它的Map。还有一个比较经典的例子就是Order创建SKU。

低耦合(Low coupling)

定义：

耦合是评价一个系统中各个元素（类、模块、子系统）之间依赖的程度，而良好的设计应该尽可能降这个程度。

以下是一些耦合关系的体现：

- A具有一个B类型的属性；
- A调用B的方法；
- A的方法包含对B的引用，如方法参数类型为B或返回类型为B；
- A是B的直接或者间接子类；
- B是一个接口，A实现了该接口。

理解：

在以上的这些耦合条件中，出现得越多代表耦合程度越高。这些条件简单笼统的来说就是A对B的“感”。这种感知体现在对象属性、方法参数、方法返回值以及接口上面。高耦合的类过多地依赖其他类这种设计将会导致：一个类的修改导致其他类产生较大影响，系统难以维护和理解。在重用一个高耦的类时不得不重用它所依赖的其他类，系统重用性差。如何降低耦合的程度有以下一些方法：尽量减对其他类的引用，提高方法和属性的访问权限，尽量使用组合/聚合原则来替代继承。

其实面向对象编程中的多态就是一种降低类型耦合的方法，如果没有多态的话，我们的方法需要知道有子类类型，而多态的话只需要知道父类即可。降低了类型耦合。

高内聚(High cohesion)

定义：

即功能性紧密相关的职责应该放在一个类里，并共同完成有限的功能。这点与SOLID原则当中的单一责和接口隔离是一致的。

理解：

很直观的例子就是，如果类的功能都是高内聚并职责单一的，类的复杂性就降低了，复杂性降低导致护的成本也就降低了。在传统的Dao设计模式当中，我们应该尽量拆分细粒度职责单一的Dao供Servic进行调用。在Service当中，哪一类的数据操作调用哪一个Dao就显而易见，并且单个Dao不会太过膨导致维护性变差。高内聚也代表了高隔离，高隔离就意味着，在修改某一个方法的时候，不至于影响太多其他类。

控制器 (Controller)

定义：

把接收或者处理系统事件消息的职责分配给一个类。这个类可以代表：整个系统、设备或者子系统；统事件发生时对应的用例场景，在相同的用例场景中使用相同的控制器来处理所有的系统事件。

理解：

一个控制器是负责接收或者处理事件的组件对象。MVC模式中的C就是控制器模式。而一个控制器应处理一类事件。例如我们项目中经常会有的UserController就承担添加用户，删除用户的事件。一个系统需要定义多个控制器，分别对应不同的事件处理。一般来说，控制器应当把要完成的功能委托给Srvice或者其他业务处理对象，它只负责协调和控制业务流程，尽量不要包含太多业务逻辑。

多态(Polymorphism)

定义：

当相关选择或行为随类型（类）变化而变化时，用多态操作为行为变化的类型分配职责。

理解：

在面向对象的设计当中经常要根据对象的类型来进行对应的操作。假设我们有一个画图Draw类，有个图形类Rectangle、Circle、Square。如果要按照不同图形类进行绘制的话，就需要在Draw类的方中使用if-else的程序结构，依次判断类型进行绘制。如果新增一个图形类的话，就又需要对这段代码行更改。这就违反了开闭原则。而采用多态的形式，将绘制的具体步骤交给图形类的子类实现。就不使用if-else的程序结构，在新增图形类的时候也不需要修改Draw类。通过引入多态，子类对象可以盖父类对象的行为，更好地适应变化。策略模式、工厂方法模式就是关于多态比较好的例子。

纯虚构(Pure Fabrication)

定义：

将一组高内聚的职责分配给一个虚构的或处理方便的“行为”类，它并不是问题域中的概念，而是虚的事物，以达到支持高内聚、低耦合和复用。

理解：

OO设计中的领域模型是对领域内的概念或现实世界中的对象的模型化表示。创建领域模型的关键思是减小软件人员的思维与软件模式之间的表示差异。因此，在OO设计时，系统内的大多数类都是来于现实世界中的真实类。然而，在给这些类分配职责时，有可能会遇到一些很难满足低耦合高内聚的计原则。纯虚构模式对这一问题给出的方案是：给人为制造的类分配一组高内聚的职责，该类并不代问题领域的概念，而代表虚构出来的事物。比较明显的一个例子就是适配器模式，通过虚构出适配器么一个概念来解耦两个对象之间的耦合。

许多项目都需要对数据库进行操作，将系统中的一些对象进行持久化。信息专家模式给出的建议是将久化的职责分配给具体的每一个模型类。但是这种建议已经被证明是不符合高内聚低耦合原则的。于，现在的做法往往会在项目中加入类似于DAO或者Repository这样的类。这些类在领域模型中是并存在的。

间接(Indirection)

定义:

分配职责给中间对象以协调组件或服务之间的操作，使得它们不直接耦合。中间对象就是在其他组件间建立的中介。

理解:

“中介”简单来说就是通过一个中间人来处理一件事。本来直接联系的两个对象可以通过另一个中间象进行交互，这样做便实现了隔离和解耦，一个对象的变动不会影响另一个对象，仅会影响到中间对。在设计模式当中的适配器模式，桥接模式都采用了一个中间对象来进行解耦。

受保护变化(Protected Variations)

定义:

找出预计有变化或不稳定的元素，为其创建稳定的“接口”而分配职责。

理解:

受保护变化模式简称PV，它是大多数编程和设计的基础，是模式的基本动机之一，它使系统能够适应隔离变化。它与面向对象设计原则中的开闭原则相对应，即在不修改原有元素（类、模块、子系统或统）的前提下扩展元素的功能。一个比较显而易见的例子就是策略模式，通过创建策略接口并定义其的抽象方法来应对可能出现的新策略。将来有新的策略的时候不需要改变原代码，而是新建实现策略口的类。