面向对象编程

从面向过程到面向对象

面向过程的程序设计的核心是过程（流水线式思维），过程即解决问题的步骤，面向过程的设计就好比精心设计好一条流水线，考虑周全什么时候处理什么东西。

优点：极大的降低了程序的复杂度

缺点：一套流水线或者流程就是用来解决一个问题，生产汽水的流水线无法生产汽车，即便是能，也得是大改，改一个组件，牵一发而动全身。

应用场景： 一旦完成基本很少改变的场景，著名的例子有Linux内核，git，以及Apache HTTP Server等。

面向对象的程序设计的核心是对象，Python中 一切皆对象，上帝思维，在上帝眼中这世间的一切都是对象，没有的东西也能创造出来。

优点： 解决了程序的扩展性。对某一个对象单独修改，会立刻反映到整个体系中，如对游戏中一个人物参数的特征和技能修改都很容易。

缺点： 可控性差，无法向面向过程的程序设计流水线式的可以很精准的预测问题的处理流程与结果，面向对象的程序一旦开始就由对象之间的交互解决问题，即便是上帝也无法预测最终结果。于是我们经常看到一个游戏人某一参数的修改极有可能导致某个游戏的角色出现数据异常的技能，比如一刀砍死3个人，这样就会很影响游戏的平衡。

应用场景： 需求经常变化的软件，一般需求的变化都集中在用户层，互联网应用，企业内部软件，游戏等都是面向对象的程序设计大显身手的好地方。

面向对象发展历程

在这里把面向对象方法的发展分为三个阶段：雏形阶段、完善阶段和繁荣阶段。

雏形阶段

20世纪60年代挪威计算中心开发的Simula 67，首先引入了类的概念和继承机制，它是面向对象语言的先驱。该语言的诞生是面向对象发展史上的第一个里程碑。随后20世纪70年代的CLU、并发Pascal、Ada和Modula 2等语言对抽象数据类型理论的发展起到了重要作用，它们支持数据与操作的封装。犹他大学的博士生Alan Kay设计出了一个实验性的语言Flex，该语言从Simula 67中借鉴了许多概念，如

类、对象和继承等。1972年Palo Alno研究中心（PARC）发布了Smalltalk 72，其中正式使用了“面向对象”这个术语。Smalltalk的问世标志着面向对象程序设计方法的正式形成，但是这个时期的Smalltalk语言还不够完善。

完善阶段

PARC先后发布了Smalltalk 72、76和78等版本，直至1981年推出该语言完善的版本Smalltalk 80。Smalltalk 80的问世被认为是面向对象语言发展史上最重要的里程碑。迄今绝大部分面向对象的基本概念及其支持机制在Smalltalk 80中都已具备。它是第一个完善的、能够实际应用的面向对象语言。但是随后的Smalltalk的应用尚不够广泛，其原因是：

- 追求纯OO的宗旨使得许多软件开发人员感到不便。
- 一种新的软件开发方法被广泛地接受需要一定的时间。
- 针对该语言的商品化软件开发工作到1987年才开始进行。

繁荣阶段

从20世纪80年代中期到90年代，是面向对象语言走向繁荣的阶段。其主要表现是大批比较实用的面向对象编程语言的涌现，例如 C++、Objective□C、Object Pascal、CLOS（Common Lisp Object System）、Eiffel和Actor等。这些面向对象的编程语言分为纯OO型语言和混合型OO语言。混合型语言是在传统的过程式语言基础上增加了OO语言成分形成的，在实用性方面具有更大的优势。此时的纯OO型语言也比较重视实用性。现在，在面向对象编程方面，普遍采用语言、类库和可视化编程环境相结合的方式，如Visual C++、JBuilder和Delphi等。面向对象方法也从编程发展到设计、分析，进而发展到整个软件生命周期。 到20世纪90年代，面向对象的分析与设计方法已多达数十种，这些方法都各有所长。目前，统一建模语言（Unified Modeling Language，UML）已经成为世界性的建模语言，适用于多种开发方法。把UML作为面向对象的建模语言，不但在软件产业界获得了普遍支持，在学术界影响也很大。在面向对象的过程指导方面，目前还没有国际规范发布。当前较为流行的用于面向对象软件开发的过程指导有“统一软件开发过程”（也有人称为RUP）和国内的青鸟面向对象软件开发过程指导等。 当前，面向对象方法几乎覆盖了计算机软件领域的所有分支。例如，已经出现了面向对象的编程语言、面向对象的分析、面向对象的设计、面向对象的测试、面向对象的维护、面向对象的图形用户界面、面向对象的数据库、面向对象的数据结构、面向对象的智能程序设计、面向对象的软件开发环境和面向对象的体系结构等。此外，许多新领域都以面向对象理论为基础或作为主要技术，如面向对象的软件体系结构、领域工程、智能代理（Agent）、基于构件的软件工程和面向服务的软件开发等。

Object-Oriented Analysis:面向对象分析法

面向对象分析法指的是在一个系统的开发过程中进行了系统业务调查以后，按照面向对象的思想来分析问题。OOA与结构化分析有较大的区别。OOA所强调的是在系统调查资料的基础上，针对OO方法所需要的素材进行的归类花分析和整理，而不是对管理业务现状和方法的分析。

OOA在定义属性的同时，要识别实例连接。实例连接是一个示例与另一个实例的映射关系。 OOA在定义服务的同时要识别消息连接。当一个对象需要向另一个对象发送消息时，它们之间就存在消息连

接。 OOA中的5个层次和5个活动继续贯穿在OOD（面向对象设计）过程中。OOD模型由4各部分组成。它们分别是设计问题域部分、设计人机交互部分、设计任务管理部分、和设计数据管理部分。

OOA的主要原则

(1)抽象： 从许多实物中舍弃个别的、非本质的特征，抽取共同的、本质性的特征，就叫做抽象。抽象是形成概念的必须手段。 抽象原则有两个方面的意义： 第一，尽管问题域中的事物是很复杂的，但是分析员并不需要了解和描述它们的一切，只需要分析其中与系统目标有关的事物及其本质性特征。 第二，通过舍弃个体事物在细节上的差异，抽取其共同特性而得到一批事物的抽象概念。

抽象是面向对象方法中使用最为广泛的原则。抽象原则包括过程抽象和数据抽象两个方面。

过程抽象是指，任何一个完成确定功能的操作序列，其使用者都可以把它看做一个单一的实体，尽管实际上它可能是由一系列更低级的操作完成的。

数据抽象是根据施加于数据之上的操作来定义数据类型，并限定数据的值只能由这些操作来修改和观察。数据抽象是OOA的核心原则。它强调把数据（属性）和操作（服务）结合为一个不可分的系统单位（即对象），对象的外部只需要知道它做什么，而不必知道它如何做。

(2)封装: 就是把对象的属性和服务结合为一个不可分的系统单位，并尽可能隐蔽对象的内部细节。

(3)继承: 特殊类的对象拥有的其一般类的全部属性与服务，称作特殊类对一般类的继承。

在OOA中运用继承原则，就是在每个由一般类和特殊类形成的一半----特殊结构总，把一般类的对象实例和所有特殊类的对象实例都共同具有的属性和服务，一次性的在一般类中进行显式的定义。在特殊类中不在重复的定义一般类中已定义的东西，但是在语义上，特殊类却自动的、隐含地拥有它的一般类（以及所有更上层的一般类）中定义的全部属性和服务。继承原则的好处是：是系统模型比较简练也比较清晰。

(4)分类: 就是把具有相同属性和服务的对象划分为一类，用类作为这些对象的抽象描述。分类原则实际上是抽象原则运用于对象描述时的一种表现形式。

(5)聚合: 又称组装，其原则是：把一个复杂的事物看成若干比较简单的事物组装体，从而简化对复杂事物的描述。

(6)关联: 是人类思考问题时经常运用的思想方法：通过一个事物联想到另外的事物。能使人发生联想的原因是事物之间确实存在着某些联系。

(7)消息通信: 这一原则要求对象之间只能通过消息进行通信，而不允许在对象之外直接地存取对象内部的属性。通过消息进行通信是由于封装原则而引起的。在OOA中要求消息连接表示出对象之间的动态联系。

(8)粒度控制: 一般来讲，人在面对一个复杂的问题域时，不可能在同一时刻既能纵观全局，又能洞察秋毫。因此需要控制自己的视野：考虑全局时，注意其大的组成部分，暂时不详查每一部分的具体的细节：考虑某部分的细节时则暂时撇开其余的部分。着就是粒度控制原则

(9)行为分析: 显示世界中事物的行为是复杂的。由大量的事物所构成的问题域中各种行为旺旺相互依赖交织

面向对象分析产生三种模型

1.对象模型：对用例模型进行分析，把系统分解成互相协作的分析类，通过类图\对象图描述对象\对象的属性\对象间的关系，是系统的静态模型 2，动态模型：描述系统的动态行为，通过时序图/协作图/描述对象的交互，以揭示对象间如何协作来完成每个具体的用例。单个对象的状态变化/动态行为可以通过状态图来表示、 3.功能模型（即用例模型à作为输入）

OOA方法的具体步骤

1，确定对象和类。这里所说的对象是对数据及其处理方式的抽象，它反映了系统保存和处理现实世界总某些事物的信息能力。。类是多个对象的共同属性和方法集合的描述，它包括如何在一个类中建立一个新对象的描述。

2，确定结构（structure）。结构是指问题域的复杂性和连接关系。类成员结构反映了泛华—特化关系，整体-部分结构反映整体和局部之间的关系

3，确定主题（subject）。主题是指事物的总体概貌和总体分析模型

4，确定属性（attribute）。属性就是数据元素，可用来描述对象或分类结构的实例，可在图中给出，并在对象的存储中指定。

5，确定方法（method）。方法是在收到消息后必须进行的一些处理方法：方法要在图中定义，并在对象的存储中指定。对于每个对象和结构来说，那些用来增加、修改、删除和选择一个方法本身都是隐含的（虽然它们是要在对象的存储中定义的，但并不在图上给出），而有些则是显示的。

Object-oriented Design:面向对象设计

面向对象设计（Object-oriented Design,OOD）方法是oo方法中一个中间过渡环节。其主要作用是对OOA分析的结构作进一步的规范化整理，以便能够被oop直接接受。

OOD的 目标是管理程序内部各部分的相互依赖。为了达到这个目标，OOD要求将程序分成块，每个块的规模应该小到可以管理的程度，然后分别将各个块隐藏在借口（interface）的后面，让它们只通过接口相互交流。比如说，如果用OOD的方法类设计一个服务器-客户端（client-server）应用，那么服务器和客户端之间不应该有直接地依赖，而是应该让服务器的接口和客户端的接口相互依赖。

OOD是一种解决软件问题的设计范式（paradigm），一种抽象的范式。使用OOD这种设计范式，我们可以用对象（object）来表现问题领域（problem domain）的实体，每个对象都有相应的状态和行为。我们刚才说到：OOD是一种抽象的范式。抽象可以分成很多层次，从非常概括的到非常特殊的都有，而对象可能处于任何一个抽象层次上。另外，彼此不同但又相互关联的对象可以共同构成抽象：只要这些对象之间有相似性，就可以把它们当成同一类的对象类处理。

面向对象设计（Object-Oriented Design，OOD）方法是面向对象程序设计方法中一个环节。其主要作用是对分析模型进行整理，生成设计模型提供给OOP作为开发依据。OOD包括：架构设计、用例设计、子系统设计、类设计等。架构设计的侧重点在于系统的体系框架的合理性，保证系统架构在系统的各个非功能性需求中保持一种平衡；子系统设计一般是采用纵向切割，关注的是系统的功能划分；类设计是根据通过一组对象、序列图展示系统的逻辑实现。

OOD介绍 (what and why)

有很多人都认为：OOD是对结构化设计（Structured Design，SD）的扩展，其实这是不对的。OOD的软件设计观念和SD完全不同。SD注重的是数据结构和处理数据结构的过程。而在OOD中，过程和数据结构都被对象隐藏起来，两者几乎是互不相关的。不过，追根溯源，OOD和SD有着非常深的渊源。

1967年前后，OOD和SD 的概念几乎同时诞生，它们分别以不同的方式来表现数据结构和算法。当时，围绕着这两个概念，很多科学家写了大量的论文。其中，由Dijkstra和 Hoare两人所写的一些论文讲到了“恰当的程序控制结构”这个话题，声称goto语句是有害的，应该用顺序、循环、分支这三种控制结构来构成整个程序流程。这些概念发展构成了结构化程序设计方法；而由Ole-Johan Dahl所写的另一些论文则主要讨论编程语言中的单位划分，其中的一种程序单位就是类，它已经拥有了面向对象程序设计的主要特征。

这两种概念立刻就分道扬镳了。在结构化这边的历史大家都很熟悉：NATO会议采纳了Dijkstra的思想，整个软件产业都同意goto语句的确是有害的，结构化方法、瀑布模型从70年代开始大行其道。同时，无数的科学家和软件工程师也帮助结构化方法不断发展完善，其中有很多今天足以使我们振聋发聩的名字，例如Constantine、Yourdon、DeMarco和Dijkstra。有很长一段时间，整个世界都相信：结构化方法就是拯救软件工业的 “银弹”。当然，时间最后证明了一切。

而此时，面向对象则在研究和教育领域缓慢发展。结构化程序设计几乎可以应用于任何编程语言之上，而面向对象程序设计则需要语言的支持[1]，这也妨碍了面向对象技术的发展。实际上，在60年代后期，支持面向对象特性的语言只有Simula-67这一种。到70年代，施乐帕洛阿尔托研究中心（PARC）的 Alan Key等人又发明了另一种基于面向对象方法的语言，那就是大名鼎鼎的Smalltalk。但是，直到80年代中期，Smalltalk和另外几种面向对象语言仍然只停留在实验室里。 到90年代，OOD突然就风靡了整个软件行业，这绝对是软件开发史上的一次革命。不过，登高才能望远，新事物总是站在旧事物的基础之上的。70年代和80年代的设计方法揭示出许多有价值的概念，谁都不能也不敢忽视它们，OOD也一样。

OOD设计原则(how)

五个基本原则（SOLID）：

S－单一职责原则

```
一个类有且仅有一个职责，只有一个引起它变化的原因。
```

简单来说一个类只做好一件事就行，不去管跟自己不相干的，狗拿耗子多管闲事，其核心就是解耦以及高内聚。这个原则看着很简单，我们在写代码的时候即便不知道这个原则也会往这个方向靠拢，写出功能相对单一的类，不过这个原则很容易违背，因为可能由于某种原因，原来功能单一的类需要被细化成颗粒更小的职责1跟职责2，所以在每次迭代过程中可能需要重新梳理重构之前编写的代码，将不同的职责封装到不同的类或者模块中。

O－开放关闭原则

开闭原则的定义是说一个软件实体如类，模块和函数应该对扩展开放，而对修改关闭，具体来说就是你应该通过扩展来实现变化，而不是通过修改原有的代码来实现变化，该原则是面相对象设计最基本的原则。在项目中每当需求需改的时候经常需要对代码有很大的改动，很大程度上就是因为我们对这个原则理解的不够透彻。开闭原则的关键在于抽象，我们需要抽象出那些不会变化或者基本不变的东西，这部分东西相对稳定，这也就是对修改关闭的地方（这并不意味着不可以再修改），而对于那些容易变化的部分我们也对其封装，但是这部分是可以动态修改的，这也就是对扩展开发的地方。

实现开闭原则的指导思想就是：

抽象出相对稳定的接口，封装变化的接口

不过在软件开发过程中，要一开始就完全按照开闭原则来可能比较困难，更多的情况是在不断的迭代重构过程中去改进，在可预见的变化范围内去做设计

L－里氏替换原则

该原则的定义：所有引用基类的地方必须能透明地使用其子类的对象。简单来说，所有使用基类代码的地方，如果换成子类对象的时候还能够正常运行，则满足这个原则，否则就是继承关系有问题，应该废除两者的继承关系，这个原则可以用来判断我们的对象继承关系是否合理。

I－接口隔离原则

接口隔离原则（Interface Segregation Principle，ISP）定义可以解释为：

```
1 | 不应该强迫客户依赖于它们不用的方法。
```

D－依赖倒置原则

依赖倒置原则（Dependence Inversion Principle）是程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

面向过程的开发，上层调用下层，上层依赖于下层，当下层剧烈变动时上层也要跟着变动，这就会导致模块的复用性降低而且大大提高了开发的成本。

面向对象的开发很好的解决了这个问题，一般情况下抽象的变化概率很小，让用户程序依赖于抽象，

实现的细节也依赖于抽象。即使实现细节不断变动，只要抽象不变，客户程序就不需要变化。这大大降低了客户程序与实现细节的耦合度。

OOP:面向对象编程

面向过程以过程为中心，完成功能时先分析步骤，用函数将各步骤实现，使用时按步骤依次调用；面向对象则以事物为中心，先从问题中抽象出一个或多个事物，每个事物都具有自己的属性和行为，通过事物之间的消息传递共同完成功能。

Python

```
def get_up():
    print('get up....')

def have_breakfast():
    print('have breakfast...')

def open_door():
    print('open the door...')

def lock_door():
    print('lock the door...')

def drive_car():
    print('driving...')

def working():
    print('work hard...')

# 依次调用完成功能
get_up()
have_breakfast()
open_door()
lock_door()
drive_car()
working()
```

面向对象设计

```python
class Person():
    def get_up(self):
        print('get up....')

    def have_breakfast(self):
        print('have breakfast...')

    def open_door(self, door):
        door.open()

    def lock_door(self, door):
        door.lock()

    def drive_car(self, car):
        car.drive()

    def working(self):
        print('work hard...')

class Door():
    def open(self):
        print('open...')

    def lock(self):
        print('lock...')

class Car():
    def drive(self):
        print('drive...')

# 通过各对象之间的消息传递共同完成功能
p = Person()
car = Car()
door = Door()
p.get_up()
p.have_breakfast()
p.open_door(door)
p.lock_door(door)
p.drive_car(car)
p.working()
```

设计模式概述

二十年前，软件设计领域的四位大师（GoF，“四人帮”，又称Gang of Four，即Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides）通过论著《设计模式：可复用面向对象软件的基础》阐述了

设计模式领域的开创性成果。

当时软件设计仍以C++与Smalltalk为主流，同时利用时间设计理念（例如强调继承）进行编写，但抛开历史的局限、他们已经为业界带来一场规模可观的浪潮。然而历史告诉我们，任何言过其实且无法在实践层面履行承诺的“浪潮”都必然令人失望，并最终遭到淘汰。如今二十年弹指一挥，我们应当重新审视当初提出的23种模式（其中一些亦可被称为变种），考虑它们与二十年来愈发成熟的编程语言还能否契合或者说并发出新的火花。

面向对象、组件技术、分布式组件技术、设计模式、敏捷开发……短短的十几年前IT界犹如时尚界一样不断涌现各种新潮的东西——所有的这一切的一切都为了“复用”。软件开发是一件非常复杂的事情，其复杂性的体现足够用一本书来说明——《人月神话》。“焦油坑”现在通常用来比喻陷入麻烦的项目；“没有银弹”的诅咒了软件行业整整10年，真的没有出现任何一种软件开发方法、任何一种工具能把软件开发复杂度在十年体提升十倍；那个经典的论断——人才是制约软件开发的唯一因素，除此之外别无其他直批软件开发最本质的的一面。面对软件的复杂性，人们想了很多办法。比如有人提出了，能不能用抽象的手法把系统的各个模块定义出来，于是就有了面向对象；还有人说，“对象”抽象的粒度太细，我们要像硬件工程师学习，于是就有了组件；还有人从软件开发过程着手于是就有了软件工程；所有的这一切都是试图通过各种“复用”的方式降低软件开发复杂度。

设计模式也不是一个例外，它也是一种复用手段。它的提出者—— Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides（统称为GOF、Gang of Four ~_~!!），1995年这四位博士合作写了一本200多页的书阐述了一种新的“复用”方式，这个方法为“设计模式”，这个本书叫《设计模式——可复用面向对象软件的基础》

GOF认为在世界上中存在一种可以复用“模式”，比如写剧本的都会有一定的格式、用词、甚至雷同的地方（就像TVB的经典台词 ），比如搞建筑的当设计门、窗、房屋结构的时候也会按照一定的“模式”。于是GOF就试图去总结软件开发也存在的“模式”，他们找到了23个然后记录在他们的书里。GOF欣喜若狂，他们认为发现了一种非常非常了不起的软件开发方法，用这种方法可以有效降低软件开发复杂度。在书中他们阐述了设计模式的意义

语言的进步: 人类之所以能从猿进化成“人”是由于人类有丰富的语言甚至为之派生类一个新的学科——文学；语言的简练、专业是一个种族、一门学科进步的最大表现。（有兴趣的可以去看一下康德为什么在哲学史上如此重要，TAOCP和高德纳为什么会在计算机科学发展史上那么不可动摇）GOF认为设计模式提炼了一些专业名词，比如：“这个数据库连接用‘工厂模式’就可以解决兼容其他数据库的问题了”人们不必去解释什么是工厂模式，这些专业词汇是成为了这个行业的通用语言。

经验的复用: 因为设计模式是一种经过千锤百炼的通用解决方式（正如TVB里安慰别人必须要有的“做人呢，最要紧就是开心。”）所以它可以被直接套用，而且它可以成为一种目标，把“坏的代码”修改成这种模式就变成“好的代码”了；菜鸟还可以用它“照葫芦画瓢”做出更加符合面向对象的设计。

好了关于设计模式的介绍就到这里，下面进入正题说一下如何正确学习和使用设计模式。关于设计模式我的所有知识都来自那本《设计模式——可复用面向对象软件的基础》，我也读过市面上其他的书就我个人感觉这些书要么引入“玄而又玄”的东西不写个1000页不罢休；要么就是像23个设计模式一样编写一堆新的设计模式条目。要学懂设计模式我觉的应该先理解设计模式的真实意图——可复用面向对象软件的基础，这是那本经典名著的副标题，正如我们开头说的，GOF是也是为了软件的复用而才

有的设计模式。

既然为了复用我们不妨放开思路——只要能够达到软件复用的目的我们可以不必有那么多顾虑、可以不折手段，不管是设计模式、面向对象还是TDD，敏捷开发，只要能让软件复用，能降低复杂度我们就可以利用。如此一来我们便有了“主动权”，我们不必循规蹈矩的去看设计模式，也不必循规蹈矩的去画UML、写用例、写单元测试，我们需要弄清楚所有的这一切它们达到“复用的方法”是什么？这个方法便是以后我们要对付软件复杂度的有力武器。设计模式，它之所以能实现复用是因为它总结了我们在面向对象设计中“更加面向对象”的设计方案，这些方案被用一种方式记录下来。既然如此我觉得学习设计模式的思路也应该是——从面向对象出发，不考虑我们要用什么模式而考虑我们要如何面向对象，以此为基础最终我们的设计会和设计模式保持一致（即便不保持一致，最后我们的设计方案也是要优于设计模式更加贴近现实）。只要这样利用设计模式我们才不会为了设计模式而设计模式（有句话说：当你手里拿着锤子的时候，全世界都是钉子）才不会误用设计模式。

“设计模式为设计师们提供一种共通的词汇储备，帮助其沟通、编写文档并探索设计方案。设计模式允许我们立足于高级抽象层面进行探讨，而非设计标注或者编程语言，这就大大降低了系统复杂性。设计模式提升了我们设计及与同事进行设计探讨时的切入点层级。”（第389页）

“理解本书中的设计模式能够帮助大家更轻松地掌握现有系统。……人们在学习面向对象编程时，往往会对需要面对的系统抱怨不休，包括其要求以令人费解的方式实现继承且难以追踪并控制工作流。之所以出现这些问题，很大程度是因为他们并不了解系统的设计模式。了解这些设计模式能够帮助各位更好地掌控现有面向对象系统。”（第389页）

“设计模式提供了一种既能解释决策结果，又能解释设计“理由“的方式。设计模式的适用性、结果性与实现部分能够引导大家做出更为明智的决策。”

“在开发可利用软件时，一大问题在于其往往必须进行重构或者重组[OJ90]。设计模式能帮助大家了解如何对设计进行重组，并降低日后需要面对的重构工作量。”

最重要的是，他们还准确预测出了模式弊端所引发的问题：

```
1 | “人们很容易将模式视为一种解决方案，或者说一种能够进行实际采纳及复用的技术。相比之下，人们
```

因此，设计模式的目的是让代码易维护、易扩展，不能为了模式而模式，因此一个简单的工具脚本是不需要用到任何模式的。

创建型模式(Creational Pattern)

创建型模式(Creational Pattern)对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离。为了使软件的结构更加清晰，外界对于这些对象只需要知道它们共同的接口，而不清楚其具体的实现细节，使整个系统的设计更加符合单一职责原则。

创建型模式在创建什么(What)，由谁创建(Who)，何时创建(When)等方面都为软件设计者提供了尽可能大的灵活性。创建型模式隐藏了类的实例的创建细节，通过隐藏对象如何被创建和组合在一起达到

使整个系统独立的目的。

常见模式详解

工厂方法（factory_method）

- 模式定义 (what)

在工厂方法模式中，我们执行单个函数，传入一个参数(提供信息表明我们想要什么)，但并不要求知道任何关于对象如何实现以及对象来自哪里的细节。

在工厂设计模式中，用户可以请求一个对象，而无需知道这个对象来自哪里；也就是说，无需知道使用哪个类来生成这个对象。工厂背后的思想是简化对象的创建。与客户端自己基于类实例化直接创建对象相比，基于一个中心化函数来实现，也更易于追踪创建了哪些对象。通过将创建对象的代码和使用对象的代码解耦，工厂能够降低应用维护的复杂度。

若需要将对象的创建和使用解耦，工厂方法也能派上用场。创建对象时，我们并没有与某个 12 特定类耦合/绑定到一起，而只是通过调用某个函数来提供关于我们想要什么的部分信息。这意味着修改这个函数比较容易，不需要同时修改使用这个函数的代码。

- 模式动机 (why)

1.隐藏具体类名，很多类隐藏得很深的，而且可能会在后续版本换掉 2.避免你辛苦的准备构造方法的参数 3.这个工厂类可以被配置成其它类 4.这个工厂对象可以被传递

避免硬编码, new Class() 其实也是一种硬编码！！！

- python 代码示例

```python

import json
import xml.etree.ElementTree as etree

class JSONConnector(object):
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf8') as f:
            self.data = json.load(f)

    @property
    def parsed_data(self):
        return self.data


class XMLConnector(object):
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
    def parsed_data(self):
        return self.tree


def connection_factory(filepath):
    """ 工厂方法 """
    if filepath.endswith('json'):
        connector = JSONConnector
    elif filepath.endswith('xml'):
        connector = XMLConnector
    else:
        raise ValueError('Cannot connect to {}'.format(filepath))
    return connector(filepath)
```

抽象工厂模式（abstract_factory）

模式定义

抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，属于对象创建型模式。

模式动机

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候我

们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

为了更清晰地理解工厂方法模式，需要先引入两个概念：

- **产品等级结构**: 产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。

- **产品族** 在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。

当系统所提供的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。

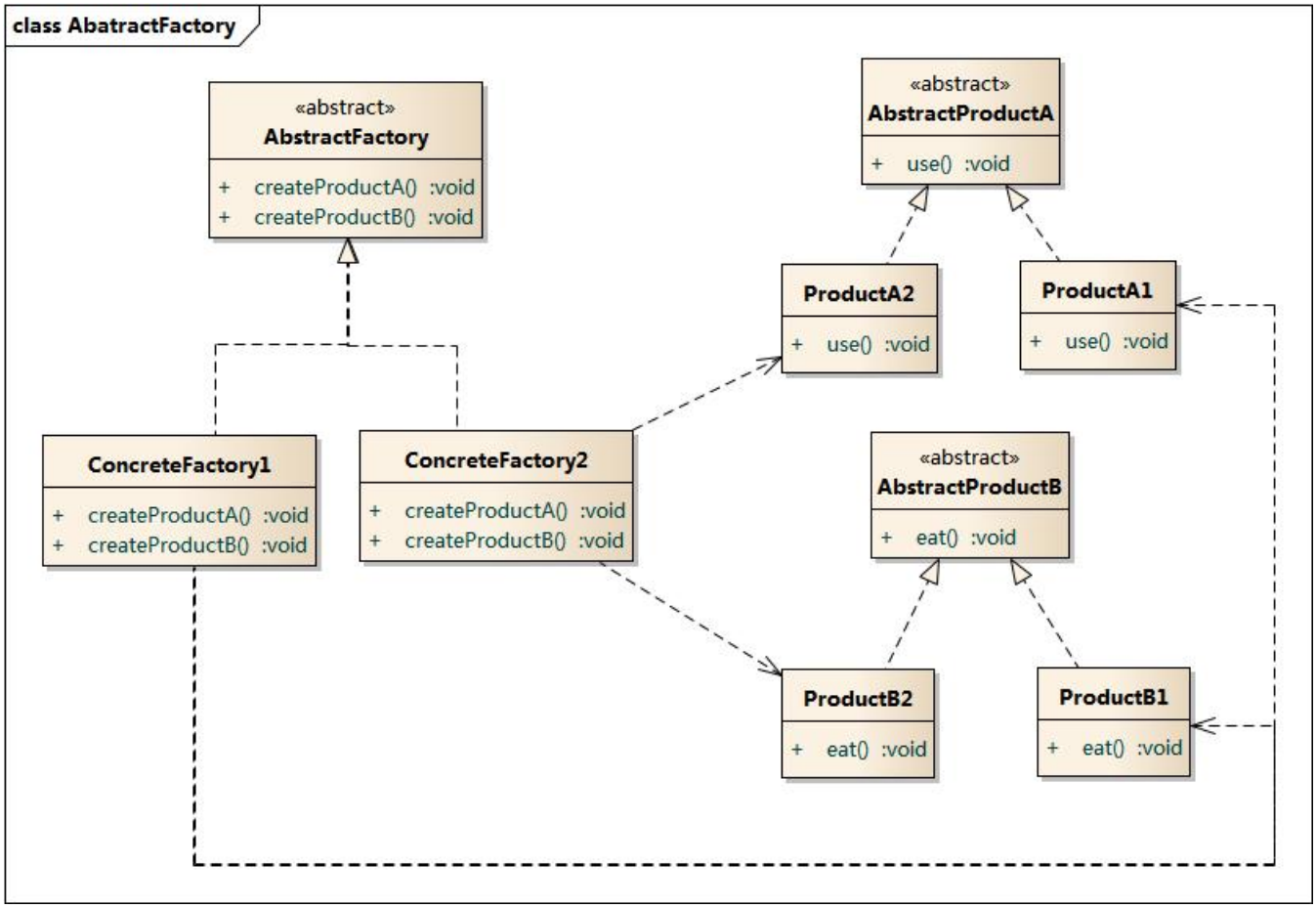
抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。

抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构，一个工厂等级结构可以负责多个不同产品等级结构中的产品对象的创建 。当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率。

模式结构

抽象工厂模式包含如下角色：

- AbstractFactory：抽象工厂
- ConcreteFactory：具体工厂
- AbstractProduct：抽象产品
- Product：具体产品



模式优缺点

1. 优点

- 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。另外，应用抽象工厂模式可以实现高内聚低耦合的设计目的，因此抽象工厂模式得到了广泛的应用。
- 当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。这对一些需要根据当前环境来决定其行为的软件系统来说，是一种非常实用的设计模式。
- 增加新的具体工厂和产品族很方便，无须修改已有系统，符合“开闭原则”。

1. 缺点

- 在产品族中扩展新的产品是很困难的，它需要修改抽象工厂的接口

Python 代码描述

描述一个这样的过程：有一个 GUIFactory，GUI 有 Mac 和 Windows 两种风格，现在需要创建两种不同风格的 Button，显示在 Application 中。

Python

```python
import six
import abc
import random


class PetShop(object):

    """A pet shop"""

    def __init__(self, animal_factory=None):
        """pet_factory is our abstract factory.  We can set it at will."""

        self.pet_factory = animal_factory

    def show_pet(self):
        """Creates and shows a pet using the abstract factory"""

        pet = self.pet_factory.get_pet()
        print("We have a lovely {}".format(pet))
        print("It says {}".format(pet.speak()))
        print("We also have {}".format(self.pet_factory.get_food()))


# Stuff that our factory makes

class Dog(object):

    def speak(self):
        return "woof"

    def __str__(self):
        return "Dog"


class Cat(object):

    def speak(self):
        return "meow"

    def __str__(self):
        return "Cat"


# Factory classes

class DogFactory(object):

    def get_pet(self):
```

```
        return Dog()

    def get_food(self):
        return "dog food"


class CatFactory(object):

    def get_pet(self):
        return Cat()

    def get_food(self):
        return "cat food"


# Create the proper family
def get_factory():
    """Let's be dynamic!"""
    return random.choice([DogFactory, CatFactory])()


# Implementation 2 of an abstract factory
@six.add_metaclass(abc.ABCMeta)
class Pet(object):

    @classmethod
    def from_name(cls, name):
        for sub_cls in cls.__subclasses__():
            if name == sub_cls.__name__.lower():
                return sub_cls()

    @abc.abstractmethod
    def speak(self):
        """"""


class Kitty(Pet):
    def speak(self):
        return "Miao"


class Duck(Pet):
    def speak(self):
        return "Quak"


# Show pets with various factories
if __name__ == "__main__":
```

```
    for i in range(3):
        shop = PetShop(get_factory())
        shop.show_pet()
        print("=" * 20)

    for name0 in ["kitty", "duck"]:
        pet = Pet.from_name(name0)
        print("{}: {}".format(name0, pet.speak()))

### OUTPUT ###
# We have a lovely Cat
# It says meow
# We also have cat food
# ====================
# We have a lovely Dog
# It says woof
# We also have dog food
# ====================
# We have a lovely Cat
# It says meow
# We also have cat food
# ====================
# kitty: Miao
# duck: Quak

```

适用性

在以下情况可以考虑使用抽象工厂模式：

- 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。
- 需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。
- 提供一个产品类库，而只想显示它们的接口而不是实现时。

原型模式（prototype）

当需要在原有对象的基础上创建一个该对象的副本时，我们就可以使用原型模式。 复制对象方法已经是python的一部分了，而且python同时提供了浅复制copy.copy和深复制copy.deepcopy方法，我们在实践中就不需要设计这种模式了。 **浅拷贝:** 构建一个新的对象然后插入到原来的引用上。只拷贝父对象，不会拷贝对象的内部的子对象。 **深拷贝:** 构造一个新的对象以递归的形式，然后插入复制到它原来的对象上。拷贝对象及其子对象

原型模式的实现可以参考下述python代码实现：

Python

```
class Prototype(object):

```

```
    value = 'default'

    def clone(self, **attrs):
        """Clone a prototype and update inner attributes dictionary"""
        # Python in Practice, Mark Summerfield
        obj = self.__class__()
        obj.__dict__.update(attrs)
        return obj


class PrototypeDispatcher(object):

    def __init__(self):
        self._objects = {}

    def get_objects(self):
        """Get all objects"""
        return self._objects

    def register_object(self, name, obj):
        """Register an object"""
        self._objects[name] = obj

    def unregister_object(self, name):
        """Unregister an object"""
        del self._objects[name]


def main():
    dispatcher = PrototypeDispatcher()
    prototype = Prototype()

    d = prototype.clone()
    a = prototype.clone(value='a-value', category='a')
    b = prototype.clone(value='b-value', is_checked=True)
    dispatcher.register_object('objecta', a)
    dispatcher.register_object('objectb', b)
    dispatcher.register_object('default', d)
    print([{n: p.value} for n, p in dispatcher.get_objects().items()])


if __name__ == '__main__':
    main()

### OUTPUT ###
# [{'objectb': 'b-value'}, {'default': 'default'}, {'objecta': 'a-value'}]
```

单例模式（singleton）

模式定义

单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。当你希望在整个系统中，某个类只能出现一个实例时，单例对象就能派上用场。

比如，某个服务器程序的配置信息存放在一个文件中，客户端通过一个 AppConfig 的类来读取配置文件的信息。如果在程序运行期间，有很多地方都需要使用配置文件的内容，也就是说，很多地方都需要创建 AppConfig 对象的实例，这就导致系统中存在多个 AppConfig 的实例对象，而这样会严重浪费内存资源，尤其是在配置文件内容很多的情况下。事实上，类似 AppConfig 这样的类，我们希望在程序运行期间只存在一个实例对象。

模式动机

比如，某个服务器程序的配置信息存放在一个文件中，客户端通过一个 AppConfig 的类来读取配置文件的信息。如果在程序运行期间，有很多地方都需要使用配置文件的内容，也就是说，很多地方都需要创建 AppConfig 对象的实例，这就导致系统中存在多个 AppConfig 的实例对象，而这样会严重浪费内存资源，尤其是在配置文件内容很多的情况下。事实上，类似 AppConfig 这样的类，我们希望在程序运行期间只存在一个实例对象。

模式实现方式

在 Python 中，我们可以用多种方法来实现单例模式：

- 使用模块
- 使用 **new**
- 使用装饰器（decorator）
- 使用元类（metaclass）

使用模块

其实，Python 的模块就是天然的单例模式，因为模块在第一次导入时，会生成 .pyc 文件，当第二次导入时，就会直接加载 .pyc 文件，而不会再次执行模块代码。因此，我们只需把相关的函数和数据定义在一个模块中，就可以获得一个单例对象了。如果我们真的想要一个单例类，可以考虑这样做：

```python
# mysingleton.py
class My_Singleton(object):
    def foo(self):
        pass

my_singleton = My_Singleton()
```

将上面的代码保存在文件 mysingleton.py 中，然后这样使用：

```python
from mysingleton import my_singleton

my_singleton.foo()
```

使用 new

为了使类只能出现一个实例，我们可以使用 **new** 来控制实例的创建过程，代码如下：

```python
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kw):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kw)
        return cls._instance

class MyClass(Singleton):
    a = 1
```

在上面的代码中，我们将类的实例和一个类变量 *instance 关联起来，如果 cls*.instance 为 None 则创建实例，否则直接返回 cls._instance。

```python
>>> one = MyClass()
>>> two = MyClass()
>>> one == two
True
>>> one is two
True
>>> id(one), id(two)
(4303862608, 4303862608)
```

使用装饰器

装饰器（decorator）可以动态地修改一个类或函数的功能。这里，我们也可以使用装饰器来装饰某个类，使其只能生成一个实例，代码如下：

```python
from functools import wraps

def singleton(cls):
    instances = {}
    @wraps(cls)
    def getinstance(*args, **kw):
        if cls not in instances:
            instances[cls] = cls(*args, **kw)
        return instances[cls]
    return getinstance

@singleton
class MyClass(object):
    a = 1
```

在上面，我们定义了一个装饰器 singleton，它返回了一个内部函数 getinstance，该函数会判断某个类是否在字典 instances 中，如果不存在，则会将 cls 作为 key，cls(*args, **kw) 作为 value 存到 instances 中，否则，直接返回 instances[cls]。

使用 metaclass

元类（metaclass）可以控制类的创建过程，它主要做三件事：

- 拦截类的创建
- 修改类的定义
- 返回修改后的类

使用元类实现单例模式的代码如下：

```python
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class MyClass(metaclass=Singleton):
   pass
```

Borg模式

所谓的单例模式，就是确保这个类只有一个对象。虽然单例听起来感觉好像很专业的样子，这可能并非是个好主意。有时候我们可能确实想要很多对象，但是这些对象有共享的状态。我们在乎的是共享的状态，毕竟谁会在乎无限的可能

对于python的一个类而言，self.**dict**是可以重新绑定的。如果我们在**init**方法中重新绑定一个新的字典，那么所有的对象就拥有”we are one”的性质。

Python

```python
class Borg(object):
    __shared_state = {}

    def __init__(self):
        self.__dict__ = self.__shared_state
        self.state = 'Init'

    def __str__(self):
        return self.state

if __name__ == '__main__':
    rm1 = Borg()
    rm2 = Borg()

    rm1.state = 'Idle'
    rm2.state = 'Running'

    print('rm1: {0}'.format(rm1))
    print('rm2: {0}'.format(rm2))

    rm2.state = 'Zombie'

    print('rm1: {0}'.format(rm1))
    print('rm2: {0}'.format(rm2))

    print('rm1 id: {0}'.format(id(rm1)))
    print('rm2 id: {0}'.format(id(rm2)))

    rm3 = YourBorg()

    print('rm1: {0}'.format(rm1))
    print('rm2: {0}'.format(rm2))
    print('rm3: {0}'.format(rm3))

### OUTPUT ###
# rm1: Running
# rm2: Running
# rm1: Zombie
# rm2: Zombie
# rm1 id: 140732837899224
# rm2 id: 140732837899296
# rm1: Init
# rm2: Init
# rm3: Init
```

单例模式常用的场景

1. Windows的Task Manager（任务管理器）就是很典型的单例模式
2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。
3. 网站的计数器，一般也是采用单例模式实现，否则难以同步。
4. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
5. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。
6. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。
7. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。
8. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。
9. HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.

总结以上，不难看出单例模式应用的场景一般发现在以下条件下：

- 资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。
- 控制资源的情况下，方便资源之间的互相通信。如线程池等。

结构型模式（Structural Patterns）

结构型模式描述类和对象之间如何进行有效的组织，形成良好的软件体系结构，主要的方法是使用继承关系来组织各个类。

结构型模式(Structural Pattern)描述如何将类或者对象结合在一起形成更大的结构，就像搭积木，可以通过简单积木的组合形成复杂的、功能更为强大的结构。

结构型模式可以分为类结构型模式和对象结构型模式：

- 类结构型模式关心类的组合，由多个类可以组合成一个更大的系统，在类结构型模式中一般只存在继承关系和实现关系。
- 对象结构型模式关心类与对象的组合，通过关联关系使得在一 个类中定义另一个类的实例对象，然后通过该对象调用其方法。 根据“合成复用原则”，在系统中尽量使用关联关系来替代继 承关系，因此大部分结构型模式都是对象结构型模式。

常见模式详解

适配器模式（adapter）

模式动机

- 在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。
- 通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。
- 在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。
- 在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是适配器(Adapter)，它所包装的对象就是适配者(Adaptee)，即被适配的类。
- 适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

模式定义

适配器模式(Adapter pattern)是一种结构型设计模式，帮助我们实现两个不兼容接口之间的兼容。首先，解释一下不兼容接口的真正含义。如果我们希望把一个老组件用于一个新系统中，或者把一个新组件用于一个老系统中，不对代码进行任何修改两者就能够通信的情况很少见。但又并非总是能修改代码，或因为我们无法访问这些代码(例如，组件以外部库的方式提供)，或因为修改代码本身就不切实际。在这些情况下，我们 可以编写一个额外的代码层，该代码层包含让两个接口之间能够通信需要进行的所有修改。这个代码层就叫适配器。

Python

```python
class Tea(object):
    def __init__(self):
        self.__name = 'Tea'
        self.__price = 2

    def get_price(self):
        return self.__price

    def get_name(self):
        return self.__name

    def add_something(self, add):
        self.__name += '_with_' + add.add_name()
        self.__price += add.add_price()


class Coffee(object):
    def __init__(self):
        self.__name = 'coffee'
        self.__price = 3
```

```
    def get_price(self):
        return self.__price

    def get_name(self):
        return self.__name


class Ice(object):

    __name = 'ice'
    __price = 0.3

    def add_ice_price(self):
        return self.__price

    def add_ice_name(self):
        return self.__name


class Suger(object):

    __name = 'suger'
    __price = 0.5

    def add_suger_price(self):
        return self.__price

    def add_suger_name(self):
        return self.__name

class Adapter(object):

    def __init__(self, obj, adapted_methods):
        """We set the adapted methods in the object's dict"""
        self.obj = obj
        self.__dict__.update(adapted_methods)

    def __getattr__(self, attr):
        """All non-adapted calls are passed to the object"""
        return getattr(self.obj, attr)


if __name__ == '__main__':
    tea = Tea()
    ice = Ice()
    tea.add_something(Adapter(ice, dict(add_name=ice.add_ice_name, add_price=ice.
    print(tea.get_name())
    print(tea.get_price())
```

Python框架Grok和第三方包Traits各自都使用了适配器模式来获得API一致性和接口兼容性。

适配器模式的注意事项

适配器模式应用于： 当系统的数据和行为都正确，但接口不符时，我们应该考虑用适配器，目的是使控制范围这外的一个原有对象与某个接口匹配。适配器模式主要应用于希望复用一些现在的类，但是接口又与复用环境要求不一致的情况。

适配器模式最好在详细设计不要考虑它，它不是为了解决还处在开发阶段的问题，而是解决正在服役的项目问题。

桥接模式（bridge）

模式定义

桥接模式(Bridge Pattern)：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。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

模式动机

假设这样一种情况： 我们有大中小型号的毛笔，有红蓝黑三种颜料。如果需要不同颜色，不同型号的毛笔有如下两种设计方法：

- 为每一种型号的毛笔都提供三种颜料的版本。
- 将毛笔和颜料分开，使用的时候自由组合。

针对第一种我们当下就需要红色颜料大号毛笔、蓝色颜料大号毛笔等九中型号，在之后的扩展中，每增加一个型号的毛笔就需要为其增加所有颜料的版本， 而每增加一种颜料也需要为所有的笔增添新的颜料类型。随着笔的类型和颜料种类的不断增长，其类数量的增加速度为O(n!) 我们需要维护庞大的种类集合。

而第二种我们将笔和颜料分开，就比如说准备大中小三种型号的毛笔，以及红蓝黑三种颜料，用的时候就拿笔区蘸一下颜料就可以，这样笔和颜料解耦，他们可以分开增加。其类的增加速度为O(n) 我们可以看出 O(n!)的增长速度远大于O(n)。

第一种方法在系统设计上我们采用的就是继承的方式。实现方式上有两种，一种就像我上面提到的只有一个抽象类，然后所有子类都是由这个抽象类里面派生出来了，还有一种代码编写方式就是提供毛笔抽象类和颜料抽象类 以及他们的派生子类，然后子类使用多继承分别继承他们，这种编写方式又一个问题就是在不支持多继承的语言中无法实现（例子请看我的 文章中的那一部分。）

第二种方法是采用组合关系的设计，其设计中也是将颜料和笔这两个变化分开，及分别由笔的抽象类派生出各种各样的笔和由颜料抽象类派生出的各种各样的颜料。但是将他们之间连接在一起的是关联关系方式（不懂的可以区我的这一篇文章看一下）。在结构型设计模式中大量使用这种关联关系来代

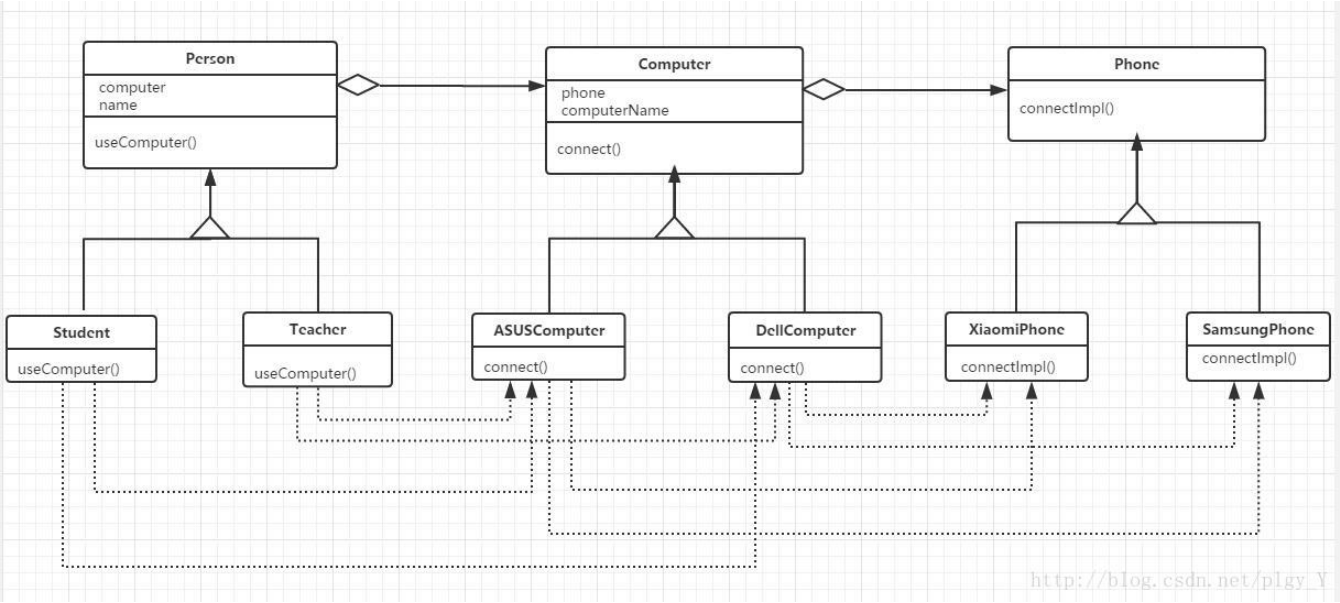
替继承关系以实现解耦，为什么关联关系可以解耦呢？我认为主要有以下几点：

- 继承关系中，父类的修改会传播到子类中，容易造成子类的不稳定。
- 继承关系在代码编写阶段已经固定，但是使用关联关系，我们可以将他们之间的关系确定推迟到程序运行时，更加方便用户控制。比如使用配置文件来确定关联关系中的两个类的类型。
- 使用关联关系也可以减少代码的编写量，尤其是在变化的种类很多以及后期增加种类的时候。
- 使用关联关系在后期增加变化种类，不需要更改已经存在的类。满足开闭原则
- 使用关联关系也将功能分割，符合单一职责原则

模式结构

桥接模式包含如下角色：

- Abstraction：抽象类
- RefinedAbstraction：扩充抽象类
- Implementor：实现类接口
- ConcreteImplementor：具体实现类



python 代码示例

Python

```python
#-*-encoding:utf-8-*-
# 桥接模式
# 抽象类：手机
class Phone(object):
     def __init__(self,name):
         self.name=name
     def connectImpl(self):
        pass
# 三星手机
class SamsungPhone(Phone):
```

```
     def __init__(self,name):
         # 调用父类构造方法的不同方式
         # Phone.__init__(self,name)
         super(SamsungPhone, self).__init__(name)
     def connectImpl(self):
         print '连接上了'+self.name
class XiaomiPhone(Phone):
    def __init__(self,name):
        # Phone.__init__(self, name)
        super(XiaomiPhone,self).__init__(name)
    def connectImpl(self):
        print '连接上了'+self.name
# 抽象类：电脑
class Computer(object):
    def __init__(self, phone,computerName):
        self.phone=phone
        self.computerName=computerName
    def connect(self):
        pass
# 华硕电脑
class ASUSComputer(Computer):
    def __init__(self,phone,computerName):
        Computer.__init__(self,phone,computerName)
    def connect(self):
        print self.computerName,
        self.phone.connectImpl()
# 戴尔电脑
class DellComputer(Computer):
    def __init__(self,phone,computerName):
        Computer.__init__(self,phone,computerName)
    def connect(self):
        print self.computerName,
        self.phone.connectImpl()
# 抽象类：人
class Person(object):
    def __init__(self,computer,name):
        self.computer=computer
        self.name=name
    def useComputer(self):
        pass
# 学生
class Student(Person):
    def __init__(self,computer,name):
        # Person.__init__(self,computer,name)
        super(Student,self).__init__(computer,name)
    def useComputer(self):
        print self.name+'使用',
        self.computer.connect()
```

```
# 教师
class Teacher(Person):
    def __init__(self,computer,name):
        # Person.__init__(self,computer,name)
        super(Teacher,self).__init__(computer,name)
    def useComputer(self):
        print self.name+'使用',
        self.computer.connect()
def main():
    # 华硕电脑连接上了小米手机A1
    ASUScomputer=ASUSComputer(XiaomiPhone('小米手机A1'),'华硕电脑A1')
    ASUScomputer.connect()
    xiaomiA2=XiaomiPhone('小米手机A2')
    print xiaomiA2.name
    # 华硕电脑连接上了三星手机B1
    ASUScomputer = ASUSComputer(SamsungPhone('三星手机B1'), '华硕电脑A1')
    ASUScomputer.connect()
    # 戴尔电脑连接上了三星手机B1
    dellcomputer=DellComputer(SamsungPhone('三星手机B1'), '戴尔电脑B1')
    dellcomputer.connect()
    # 戴尔电脑连接上了小米手机A1
    dellcomputer = DellComputer(XiaomiPhone('小米手机A1'), '戴尔电脑B1')
    dellcomputer.connect()
    # 学生A1使用华硕电脑A1连接上了小米手机A1
    student=Student(ASUSComputer(XiaomiPhone('小米手机A1'),'华硕电脑A1'),'学生A1')
    student.useComputer()
    # 老师B1使用戴尔电脑B1连接上了三星手机B1
    teacher=Teacher(DellComputer(SamsungPhone('三星手机B1'), '戴尔电脑B1'), '老师B1
    teacher.useComputer()
if __name__=='__main__':
    main()

# 输出
>>> 华硕电脑A1 连接上了小米手机A1
>>> 华硕电脑A1 连接上了三星手机B1
>>> 戴尔电脑B1 连接上了三星手机B1
>>> 戴尔电脑B1 连接上了小米手机A1
>>> 学生A1使用 华硕电脑A1 连接上了小米手机A1
>>> 老师B1使用 戴尔电脑B1 连接上了三星手机B1
```

如果不用桥接模式，那么通常的做法是，创建若干个基类用于表示各个抽象方式， 然后从每个基类中继承出两个或多个子类，用于表示对这种抽象方式的不同实现方法。 用了桥接模式之后，我们需要创建两套独立的"类体系"（class hierarchy）："抽象体系" 定义了我们所要执行的操作（比如接口或高层算法），而“实现体系”则包含具体实现方式， 抽象体系要调用实现体系以完成其操作。抽象体系中的类会把实现体系中的某个类实例 聚合进来，而这个实例将充当抽象接口与具体实现之间的桥梁(bridge)。

模式分析

理解桥接模式，重点需要理解如何将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化。

- **抽象化**：抽象化就是忽略一些信息，把不同的实体当作同样的实体对待。在面向对象中，将对象的共同性质抽取出来形成类的过程即为抽象化的过程。
- **实现化**：针对抽象化给出的具体实现，就是实现化，抽象化与实现化是一对互逆的概念，实现化产生的对象比抽象化更具体，是对抽象化事物的进一步具体化的产物。
- **脱耦**：脱耦就是将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联，将两个角色之间的继承关系改为关联关系。桥接模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用关联关系（组合或者聚合关系）而不是继承关系，从而使两者可以相对独立地变化，这就是桥接模式的用意。

模式优缺点

1. 优势

- 分离抽象接口及其实现部分。
- 桥接模式有时类似于多继承方案，但是多继承方案违背了类的单一职责原则（即一个类只有一个变化的原因），复用性比较差，而且多继承结构中类的个数非常庞大，桥接模式是比多继承方案更好的解决方法。
- 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。
- 实现细节对客户透明，可以对用户隐藏实现细节。

1. 缺点

- 桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。
- 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

使用场景

1. 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。列如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方法加以扩充。这时的桥接模式使你可以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. 想对客户完全隐藏抽象的实现部分。

5. 有许多类要生成，类的层次结构说明你必须得将一个对象分解成两个部分。
6. 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。

- composite

装饰模式（decorator）

模式定义

装饰模式：动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

模式动机

一般有两种方式可以实现给一个类或对象增加行为：

- **继承机制:** 使用继承机制是给现有类添加功能的一种有效途径，通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。

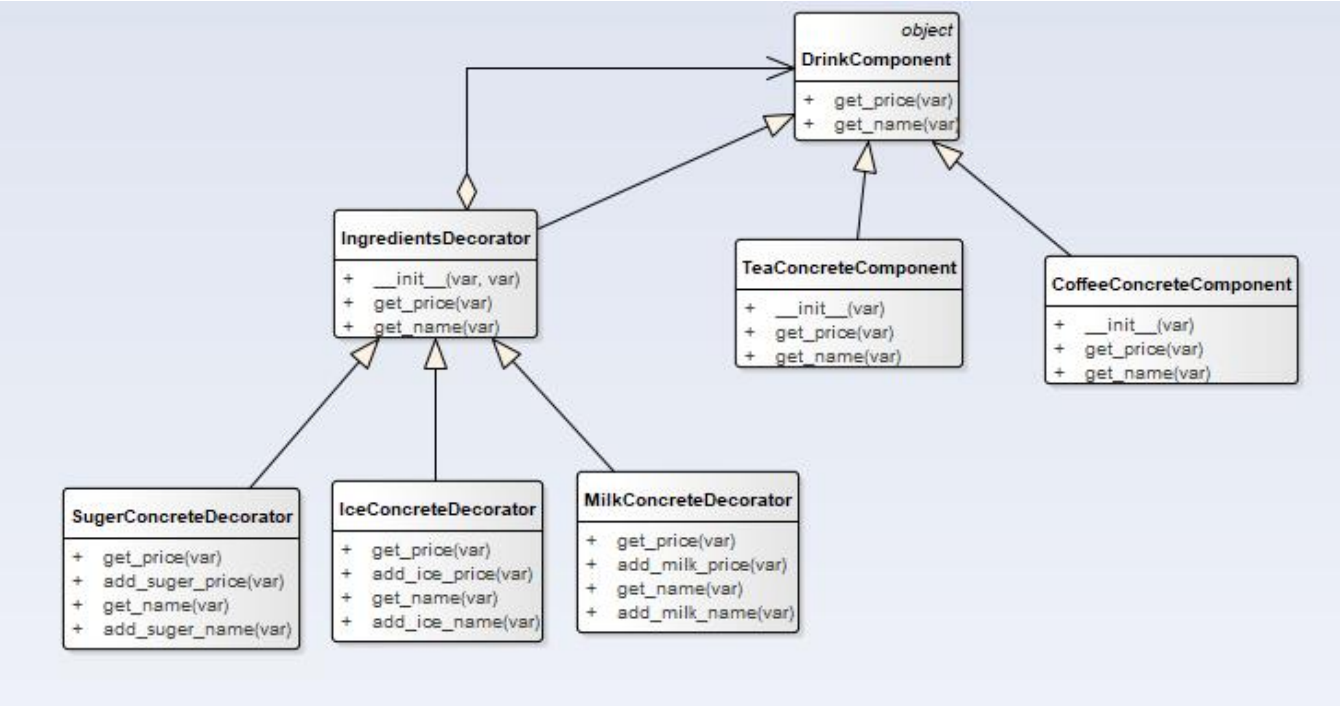
- **关联机制:** 即将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，我们称这个嵌入的对象为装饰器(Decorator)

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任，换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。这就是装饰模式的模式动机。

模式结构

装饰模式包含如下角色：

- Component: 抽象构件
- ConcreteComponent: 具体构件
- Decorator: 抽象装饰类
- ConcreteDecorator: 具体装饰类



python 代码示例

Python

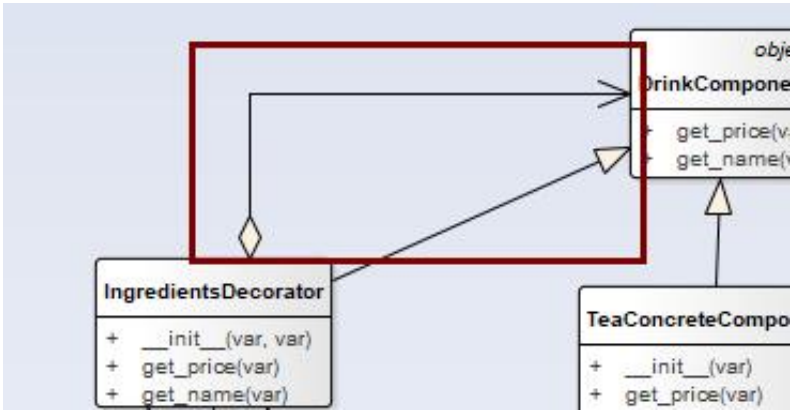
```python
class DrinkComponent(object):
    def get_price(self):
        pass

    def get_name(self):
        pass


class TeaConcreteComponent(DrinkComponent):
    def __init__(self):
        self.__name = 'Tea'
        self.__price = 2

    def get_price(self):
        return self.__price

    def get_name(self):
        return self.__name


class CoffeeConcreteComponent(DrinkComponent):
    def __init__(self):
        self.__name = 'coffee'
        self.__price = 3

    def get_price(self):
        return self.__price
```

```
28
29      def get_name(self):
30          return self.__name
31
32
33  class IngredientsDecorator(object):
34      def __init__(self, drink_component):
35          self.drink_component = drink_component
36
37      def get_price(self):
38          pass
39
40      def get_name(self):
41          pass
42
43
44  class IceConcreteDecorator(IngredientsDecorator):
45      def get_price(self):
46          return self.drink_component.get_price() + self.add_ice_price()
47
48      def add_ice_price(self):
49          return 0.3
50
51      def get_name(self):
52          return self.drink_component.get_name() + self.add_ice_name()
53
54      def add_ice_name(self):
55          return '+Ice'
56
57
58  class SugerConcreteDecorator(IngredientsDecorator):
59      def get_price(self):
60          return self.drink_component.get_price() + self.add_suger_price()
61
62      def add_suger_price(self):
63          return 0.5
64
65      def get_name(self):
66          return self.drink_component.get_name() + self.add_suger_name()
67
68      def add_suger_name(self):
69          return '+Suger'
70
71
72  class MilkConcreteDecorator(IngredientsDecorator):
73      def get_price(self):
74          return self.drink_component.get_price() + self.add_milk_price()
75
```

```
    def add_milk_price(self):
        return 1

    def get_name(self):
        return self.drink_component.get_name() + self.add_milk_name()

    def add_milk_name(self):
        return '+Milk'

if __name__ == '__main__':
    tea_milk = MilkConcreteDecorator(TeaConcreteComponent())
    print tea_milk.get_name()
    print tea_milk.get_price()

    tea_milk_ice = IceConcreteDecorator(MilkConcreteDecorator(TeaConcreteComponer
    print tea_milk_ice.get_name()
    print tea_milk_ice.get_price()
```

说明：

DrinkComponent 是抽象构件类，它是具体构建类（*ConcreteComponent）和抽象装饰器类（IngredientsDecorator）的父类，主要定义了具体构建类的业务方法。以及让我们在调用的时候可以统一的处理装饰前和装饰后的对象。方便我们使用装饰器类装饰一个已经被装饰的具体构建如加糖（加冰（咖啡））。在关联关系中我们主要说一下这个部分。



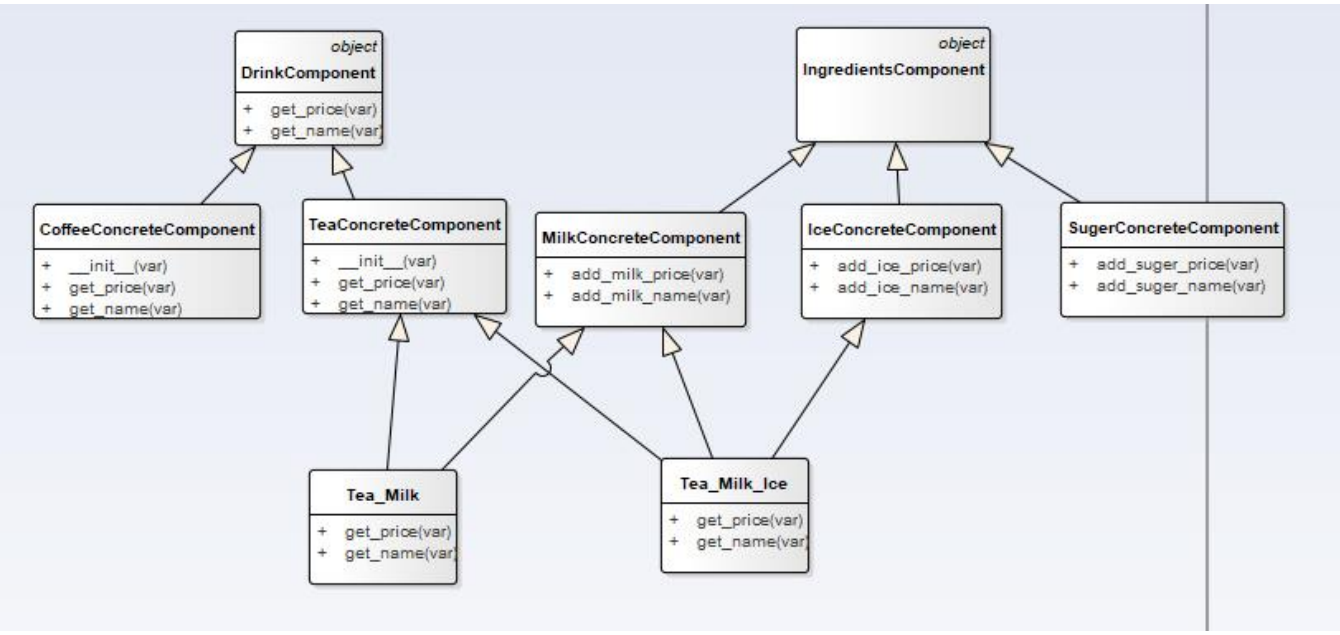
这是一个关联聚合关系。表示IngredientsDecorator是知道DrinkComponent类的存在的呢。这个大家可以这样理解。你在实现Concretecomponet的时候是不需要考虑IngredinetsDecorator的存在，因为你不会调用它的，也不继承它，也不知道你会被它调用。但是在设计实现ConcreteDecorator的时候你会在其属性中保持一个对DrinkComponet类型的类的引用。并且你会调用她的方法。这样你就要知道DrinkCompoent这个类里面都有什么方法及要知道DrinkComponent类的存在。在另一种设计模式桥接模式这种关系正好是相反的。

我们在代码中可以看到在IngredientsDecorator中也有get*price 和 get*name两种方法。这是为了保证在ConcreteComponent在被装饰器后还是可以像没有被装饰那样被调哟个。并且我们可以在调用的上面和下面添加功能以实现功能的增强。比如我们在代码中是这样写的

```python
def get_price(self):
    return self.drink_component.get_price() + self.add_milk_price()
```

```python
def get_price(self):
    print 'add Milk'
    price = self.drink_component.get_price()
    new_price = price + self.add_milk_price()
    return new_price
```

若不使用装饰模式，而使用继承方式，代码结构则变为




```python
class DrinkComponent(object):
    def get_price(self):
        pass

    def get_name(self):
        pass


class TeaConcreteComponent(DrinkComponent):
    def __init__(self):
        self.__name = 'Tea'
        self.__price = 2

    def get_price(self):
        return self.__price

    def get_name(self):
        return self.__name

```

```python

class CoffeeConcreteComponent(DrinkComponent):
    def __init__(self):
        self.__name = 'coffee'
        self.__price = 3

    def get_price(self):
        return self.__price

    def get_name(self):
        return self.__name


class IceConcreteComponent(object):
    def add_ice_price(self):
        return 0.3

    def add_ice_name(self):
        return '+Ice'


class SugerConcreteComponent(object):
    def add_suger_price(self):
        return 0.5

    def add_suger_name(self):
        return '+Suger'


class MilkConcreteComponent(object):
    def add_milk_price(self):
        return 1

    def add_milk_name(self):
        return '+Milk'


class Tea_Milk(TeaConcreteComponent, MilkConcreteComponent):
    def get_price(self):
        return TeaConcreteComponent.get_price(self) + MilkConcreteComponent.add_m

    def get_name(self):
        return TeaConcreteComponent.get_name(self) + MilkConcreteComponent.add_mi


class Tea_Milk_Ice(TeaConcreteComponent, MilkConcreteComponent, IceConcreteCompor
    def get_price(self):
        return TeaConcreteComponent.get_price(self) + MilkConcreteComponent.add_m
```

```
68                 + IceConcreteComponent.add_ice_price(self)
69
70     def get_name(self):
71         return TeaConcreteComponent.get_name(self) + MilkConcreteComponent.add_mi
72                 + IceConcreteComponent.add_ice_name(self)
73
74 if __name__ == '__main__':
75     tea_milk = Tea_Milk()
76     print tea_milk.get_name()
77     print tea_milk.get_price()
78
79     tea_milk_ice = Tea_Milk_Ice()
80     print tea_milk_ice.get_name()
81     print tea_milk_ice.get_price()
```

从图和代码中看首先我们定义了DrinkComponent和IngredientsComponent即饮料和配料两个抽象类，并分别实现了具体的构建类。当我们要产生一个产品的时候。通过继承不同的具体构建类来实现。比如加冰加牛奶的茶。我们通过继承茶、牛奶和冰块三个类来实现。可以看出如果要实现所有的类那么我们需要14个子类来完成。支持多继承的语言才能这样实现如果是单继承的语言则需要通过多级继承来完成。不仅冗余度增加而且复杂的多级继承关系是后期维护的泪。

模式分析

桥接模式和装饰模式都是通过将继承关系转换为关联关系从而减少系统中类的数量，降低系统的耦合性。

桥接模式是解决一个系统有多个变化维度的一种设计模式。其难点是如何将其中的抽象化与实现化分离（抽象化是指将一组复杂物体的一个或几个特征抽取出来形成共用部分，在面向对象的程序设计中就是将对象共同的性质抽取出去而形成类的过程。实现化是指给出抽象化中的具体实现，他是对抽象化的建进一步具体化）。使用关联关系在运行时设置不同的具体类来让同一个抽象对象表现出不同的行为。在整个过程中抽象化的类其实是不稳定的，不稳定指我们通过传入不同的实现类改变了抽象类本身的运行结果。

装饰器模式是解决如何给一个类动态增加职责的一种设计模式。装饰器模式通过将一个类中需要动态执行的方法抽取出来形成装饰类，通过关联关系来将职责的添加推迟到了程序运行时。如果程序支持反射还可以通过配置文件来灵活更改对象的功能。装饰器模式支持多层装饰，通过不同的组合可以实现不同的行为。在整个过程中被装饰对象是稳定的。因为装饰模式是在被装饰对象的前后增加功能，而不是改变被装饰对象本身的功能。具体的装饰类和被装饰类可以独立变化，用户根据需要动态的增加装饰类和被装饰类，并在调用的时候进行组合，无需更改之前的代码，符合开闭原则。

从实现上看桥接模式中抽象类只包含一个实现类的引用，实现对实现类的调用。而装饰模式中抽象装饰类继承构件抽类类，并且还包含一个对抽象构建的引用。

模式优缺点

1. 优点

- 装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。
- 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的装饰器，从而实现不同的行为。
- 通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。
- 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”

1. 缺点

- 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，同时还将产生很多具体装饰类。这些装饰类和小对象的产生将增加系统的复杂度，加大学习与理解的难度。
- 这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

使用情景

- 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。
- 需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。
- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，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；第二类是因为类定义不能继承（如final类）.

Mixin模式

模式定义

Mixin 实质上是利用语言特性（比如 Ruby 的 include 语法、Python 的多重继承）来更简洁地实现组合模式。

Python 代码示例

Python

```python
#!/usr/bin/env python
# -*- coding:utf-8 -*-

class PythonMixin(object):
```

```
    def know_python(self):
        print("I code Python.")

    def other_python_develop(self):
        pass


class JSMixin(object):
    def know_js(self):
        print("I code JS.")

    def other_js_develop(self):
        pass

class DatabaseMixin(object):
    def know_dabase(self):
        print("I code sql.")


class Person(object):

    def __init__(self, name):
        self.name = name

    def others(self):
        pass


class PythonProgrammer(PythonMixin, Person):
    """
    混入两个Mixin类
    如果有Mixin类中有相同的方法，按从左到右的顺序获取优先混入的方法
    如果自己已经有的方法，则不会再混入
    """

    pass


class FullStackProgrammer(PythonMixin, JSMixin, DatabaseMixin, Person):
    """
    混入两个Mixin类
    如果有Mixin类中有相同的方法，按从左到右的顺序获取优先混入的方法
    如果自己已经有的方法，则不会再混入
    """

    pass
```

```python
if __name__ == '__main__':

    l = PythonProgrammer('l')
    print(l.name)
    l.know_python()

    m = FullStackProgrammer('m')
    print(m.name)
    m.know_python()
    m.know_dabase()
    m.know_js()
```

使用Mixin类实现多重继承要非常小心

- 首先它必须表示某一种功能，而不是某个物品，如同Java中的Runnable，Callable等
- 其次它必须责任单一，如果有多个功能，那就写多个Mixin类
- 然后，它不依赖于子类的实现
- 最后，子类即便没有继承这个Mixin类，也照样可以工作，就是缺少了某个功能。（比如飞机照样可以载客，就是不能飞了^_^）

Mix-in 类是具有以下特征的抽象类

- 不能单独生成实例，属于抽象类
- 不能继承普通的非Mixin类。

行为型模式(Behavioral Pattern)

模式介绍

行为型模式(Behavioral Pattern)是对在不同的对象之间划分责任和算法的抽象化。

行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用。

通过行为型模式，可以更加清晰地划分类与对象的职责，并研究系统在运行时实例对象 之间的交互。在系统运行时，对象并不是孤立的，它们可以通过相互通信与协作完成某些复杂功能，一个对象在运行时也将影响到其他对象的运行。

行为型模式分为类行为型模式和对象行为型模式两种：

- 类行为型模式：类的行为型模式使用继承关系在几个类之间分配行为，类行为型模式主要通过多态等方式来分配父类与子类的职责。
- 对象行为型模式：对象的行为型模式则使用对象的聚合关联关系来分配行为，对象行为型模式主要是通过对象关联等方式来分配两个或多个类的职责。根据“合成复用原则”，系统中要尽量使用关联关系来取代继承关系，因此大部分行为型设计模式都属于对象行为型设计模式。

常见模式详解

迭代器模式（iterator）

模式定义

可迭代对象：对象里面包含**iter()**方法的实现，对象的iter函数经调用之后会返回一个迭代器，里面包含具体数据获取的实现。

迭代器：包含有next方法的实现，在正确范围内返回期待的数据以及超出范围后能够抛出StopIteration的错误停止迭代。

迭代器模式：使用迭代器模式来提供对聚合对象的统一存取，即提供一个外部的迭代器来对聚合对象进行访问和遍历 , 而又不需暴露该对象的内部结构。又叫做游标（Cursor）模式 。

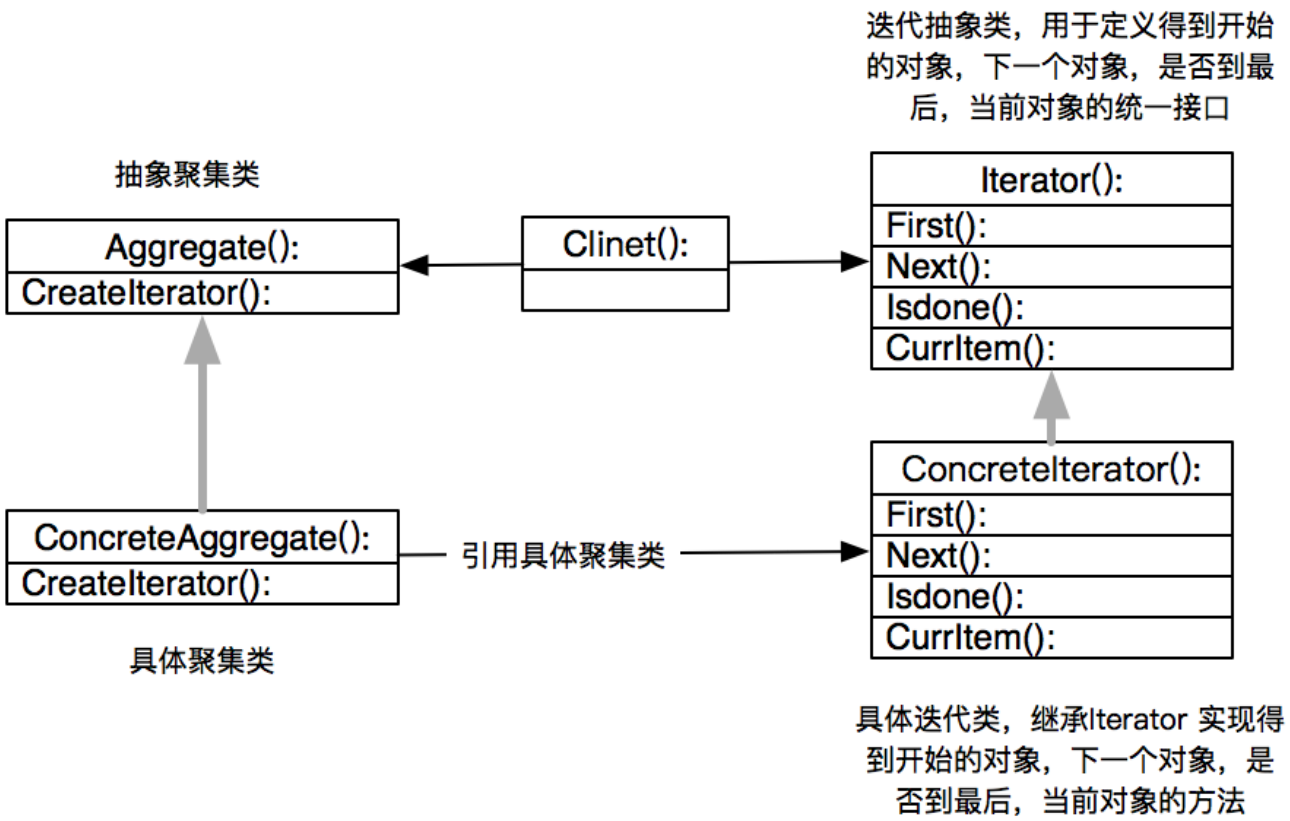
模式动机

在面向对象的软件设计中，我们经常会遇到一类集合对象，这类集合对象的内部结构可能有着各种各样的实现，但是归结起来，无非有两点是需要我们去关心的：一是集合内部的数据存储结构，二是遍历集合内部的数据。面向对象设计原则中有一条是类的单一职责原则，所以我们要尽可能的去分解这些职责，用不同的类去承担不同的职责。Iterator模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明的访问集合内部的数据。

模式结构

迭代器模式包含如下角色：

- 抽象迭代器(Iterator): 迭代器定义访问和遍历元素的接口。
- 具体迭代器(ConcreteIterator): 具体迭代器实现迭代器Iterator接口。对该聚合遍历时跟踪当前位置。
- 抽象聚合类(Aggregate): 聚合定义创建相应迭代器对象的接口。
- 具体聚合类(ConcreteAggregate): 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。



python 代码示例

python把迭代协议在语言层面就实现了，简而言之，实现了迭代器协议的对象，就是迭代器。什么事迭代器协议呢？再简而言之，满足下面两个条件即可：

- 实现了魔法方法 **iter**()，返回一个迭代对象，这个对象有一个next()方法，
- 实现 next() 方法，返回当前的元素，并指向下一个元素的位置，当前位置已经没有元素的时候，抛出StopIteration异常。

Python

```python

#!/usr/bin/env python

#============================================================================
class ReverseIterator(object):
    """
    Iterates the object given to it in reverse so it shows the difference.
    """

    def __init__(self, iterable_object):
        self.list = iterable_object
        # start at the end of the iterable_object
        self.index = len(iterable_object)

    def __iter__(self):
```

```
        # return an iterator
        return self

    def __next__(self):
        """ Return the list backwards so it's noticeably different."""
        if (self.index == 0):
            # the list is over, raise a stop index exception
            raise StopIteration
        self.index = self.index - 1
        return self.list[self.index]

#============================================================================
class Days(object):

    def __init__(self):
        self.days = [
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
        ]

    def reverse_iter(self):
        return ReverseIterator(self.days)

#============================================================================
if (__name__ == "__main__"):
    days = Days()
    for day in days.reverse_iter():
        print(day)
```

python内置实现了各种迭代协议，for循环就是一个很好的例子。for 循环的时候，首先对循环对象实现迭代器包装，返回一个迭代器对象，然后每循环一步，就调用哪个迭代器对象的next方法，循环结束的时候，自动处理了 StopIteration这个异常。for循环是对迭代器进行迭代的语法糖。无处不在的语法糖。

更复杂的遍历逻辑，都可以在 next 方法里构造。当然，看到了这里，也就大概知道了迭代器的协议，也已经是python的数据结构实现了的。

模式优势

迭代器结合了封装和多态的面向对象程序设计原理。使用迭代器，你可以对集合中的对象进行操作，而无需专门了解集合如何显现或者集合包含什么（对象的种类）。迭代器提供了不同固定迭代实现的

统一接口，它完全包含了如何操纵特定集合的详细信息，包括显示哪些项（过滤）及其显示顺序(排序)。

使用场景

- 访问一个聚合对象的内容而无需暴露它的内部表示。
- 需要为聚合对象提供多种遍历方式。
- 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)

mediator

memento

观察者模式（observer）

模式定义

观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

模式动机

建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应做出反应。在此，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展，这就是观察者模式的模式动机。

模式结构

观察者模式包含如下角色：

- Subject: 目标
- ConcreteSubject: 具体目标
- Observer: 观察者
- ConcreteObserver: 具体观察者

观察者通知方式

- 拉模型
    - 每当发生变化时，主题都会向所有已注册的观察者进行广播。

    - 出现变化时，观察者负责获取相应变化情况，或者从订户那里拉去数据。
    - 拉模型效率较低，因为它涉及两个步骤。第一步，主题通知观察者，第二部，观察者从主题那里提取所需的数据。

- 推模型

    - 变化由主题推送到观察者。
    - 在拉模型中，主题可以向观察者发送详细的信息。当主题发送大量观察者永不到的数据时，会使相应时间过长。
    - 由于只从主题发送所需的数据，所以能够提高性能。

python 代码示例

Python

```python
# -*- coding: UTF-8 -*-
from abc import ABCMeta, abstractmethod


# 订新闻的接口（主题）
class NewsPublisher(object):
    def __init__(self):
        self.__subscribers = []
        self.__latestNews = None

    # 提供注册
    def attch(self,subscriber):
        self.__subscribers.append(subscriber)

    # 提供注销
    def detach(self):
        return self.__subscribers.pop()

    # 返回已经订户列表
    def subscribers(self):
        return [type(x).__name__ for x in self.__subscribers]

    # 通知注册的所有用户
    def notifySubscribers(self):
        for sub in self.__subscribers:
            sub.update()

    # 创建新消息
    def addNews(self, news):
        self.__latestNews = news

    # 返回最新消息
    def getNews(self):
        return "Got News:",self.__latestNews
```

```python


# 用户（观察者）
class Subscriber(metaclass=ABCMeta):
    @abstractmethod
    def update(self):
        pass


#  SMS用户（实际观察者）
class SMSSubscriber(Subscriber):
    def __init__(self,publisher):
        self.publisher = publisher
        self.publisher.attch(self)

    def update(self):
        print(type(self).__name__, self.publisher.getNews())


# 邮件用户（实际观察者）
class EmailSubscriber(Subscriber):
    def __init__(self,publisher):
        self.publisher = publisher
        self.publisher.attch(self)

    def update(self):
        print(type(self).__name__, self.publisher.getNews())

# 其他用户（实际观察者）
class AnyOtherSubscriber(Subscriber):
    def __init__(self,publisher):
        self.publisher = publisher
        self.publisher.attch(self)

    def update(self):
        print(type(self).__name__, self.publisher.getNews())


if __name__ == "__main__":
    news_publisher = NewsPublisher()
    for Subscribers in [SMSSubscriber,EmailSubscriber,AnyOtherSubscriber]:
        Subscribers(news_publisher)     # 观察者，向主题订阅
    print("\nSubscribers:",news_publisher.subscribers())
    news_publisher.addNews("Hello World! ") # 增加消息
    news_publisher.notifySubscribers() # 发布消息
    print("\nDetached:",type(news_publisher.detach()).__name__) # 删除订阅者
    print("\nSubscribers:",news_publisher.subscribers()) # 查看订阅列表
    news_publisher.addNews("My second news") # 增加消息
```

```
83      news_publisher.notifySubscribers() # 发布消息
```

元编程

元编程概念

什么是元类

元类是类的一种。正如类定义了实例功能，元类也定义了类的功能。类是元类的实例。

在Python中你可以随意调用元类，实际上更有用的方法是使其本身成为一个真正的类。type是Python中常用的元类。你可能觉得奇怪，是的，type本身就是一个类，而且它就是type类型。你无法在Python中重新创造出完全像type这样的东西，但Python提供了一个小把戏。你只需要把type作为子类，就可以在Python中创建自己的元类。

元类是最常用的一类工厂。就像你通过调用类来创建类实例，Python也是通过调用元类来创建一个新类（当它执行 `class` 语句的时候），结合常规的 `__init__` 和 `__new__` 方法，元类可以允许你在创建类的时候做一些“额外的事情”，like registering the new class with some registry（暂时不知道这句话的含义，不知道怎么翻译，字面意思是：就像用某个注册表来注册新的类那样），甚至可以用别的东西完全代替已有的类。

当Python执行 `class` 语句时，它首先把整个的class语句作为一个正常的代码块来执行。由此产生的命名空间（一个字典）具有待定类的属性。元类取决于待定类的基类（元类是具有继承性的）、或待定类的 `mateclass` 。接下来，用类的名称、基类和属性调用元类，从而把元类实例化。

然而，元类实际上定义的一个类的类型，而不只是类工厂，所以你可以用元类来做更多的事情。例如，你可以定义元类的一般方法。这些元类方法和类方法有相似之处，因为它们可以被没有实例化的类调用。但这些元类方法和类方法也有不同之处，元类方法不能在类的实例中被调用。 `type.__subclasses__()` 是关于type的一个方法。你也可以定义常规的“魔法”函数，如 `__add__` ， `__iter__` 和 `__getattr__` ，以便实现或修改类的功能。

一切皆为对象

- 一切都有类型
- “class”和“type”之间本质上并无不同
- 类也是对象
- 它们的类型是 type

```Python
>>> class Something(object):
...     pass
...
>>> Something
<class '__main__.Something'>
>>> type(Something)
<type 'type'>
```

元类都要继承type，没错，元类创建类的能力来自type，type是Python的内建元类。所有严格地说，python所有的类都是type创建的。

学习元类一般都是为了自定义元类，主要目的是当创建类时自动改变类。

动态地创建类

- type

```Python
>>> def __init__(self):
...     self.message = 'Hello World'
...
>>> def say_hello(self):
...     print self.message
...
>>> attrs = {'__init__': __init__, 'say_hello': say_hello}
>>> bases = (object,)
>>> Hello = type('Hello', bases, attrs)
>>> Hello
<class '__main__.Hello'>
>>> h = Hello()
>>> h.say_hello()
Hello World
```

- **new**

这个方法我们一般很少定义，不过我们在一些开源框架中偶尔会遇到定义这个方法的类。实际上，这才是“真正的构造方法”，它会在对象实例化时第一个被调用，然后再调用**init**，它们的区别主要如下：

new的第一个参数是cls，而**init**的第一个参数是self **new**返回值是一个实例，而**init**没有任何返回值，只做初始化操作 **new**由于是返回一个实例对象，所以它可以给所有实例进行统一的初始化操作

```python
class Meta_upper(type):
    def __new__(mcs,cls_name,bases,attr_dict):
        upper_attrs = dict((attr.upper(),value) for attr,value in attr_dict.items
        return type.__new__(mcs,cls_name,bases,upper_attrs)

class Class(metaclass=Meta_upper) :
    def method_x():
        pass
"""
>>>a = Class()
>>>hasattr(a,'method_x')
False
>>>hasattr(a,'METHOD_X')
True
```

python3元类是由metaclass=Meta来在类中声明的,一般是使用元类的**new**,**init**和**call**来改变类的。

为了演示，假设你不想任何人创建这个类的实例：

```python
class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

# Example
class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')
```

这样的话，用户只能调用这个类的静态方法，而不能使用通常的方法来创建它的实例。例如：

```python
>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example1.py", line 7, in __call__
        raise TypeError("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>
```

现在，假如你想实现单例模式（只能创建唯一实例的类），实现起来也很简单：

```python
class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

# Example
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')

# output
>>> a = Spam()
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>
```

元编程使用场景

动态修改有什么意义？直接在类定义中写上需要的方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

```python
class User(Model):
    # 定义类的属性到列的映射:
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

# 创建一个实例:
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')
# 保存到数据库:
u.save()
```

model的元类

```python

class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        mappings = dict()
        for k, v in attrs.iteritems():
            if isinstance(v, Field):
                print('Found mapping: %s==>%s' % (k, v))
                mappings[k] = v
        for k in mappings.iterkeys():
            attrs.pop(k)
        attrs['__table__'] = name # 假设表名和类名一致
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        return type.__new__(cls, name, bases, attrs)

class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.iteritems():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fiel
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))
```

附录

`__init` `__call__` 和 `__new__` 调用顺序

Python

```python

class SpamMeta(type):
    def __init__(self, *args, **kwargs):
        print("NoInstances __init__")

    def __new__(cls, *args, **kwargs):
        print('NoInstances __new__')
        return super(NoInstances, cls).__new__(cls, *args, **kwargs)

    def __call__(self, *args, **kwargs):
        print('NoInstances __call__')
        return super(NoInstances, self).__call__(*args, **kwargs)


class Spam(metaclass=SpamMeta):
    def __init__(self, *args, **kwargs):
        print("Spam __init__")
        super(Spam, self).__init__(*args, **kwargs)

    def __new__(cls, *args, **kwargs):
        print('NoInstances __new__')
        return super(Spam, cls).__new__(cls, *args, **kwargs)

    def __call__(self, *args, **kwargs):
        print('Spam __call__')
        return super(Spam, self).__call__(*args, **kwargs)

    @staticmethod
    def grok(x):
        print('Spam.grok')


##########

NoInstances __new__
NoInstances __init__

###########
In [56]: a = Spam()
NoInstances __call__
NoInstances __new__
Spam __init__
```