**AID产品线全国面向对象项目峰会**

**本次答辩文档为本人亲自整理，结合各方面资料汇总而成**

**版权所有，转载请注明出处，谢谢合作！**

****安伟超

达内时代科技集团有限公司

天津天大中心教学部

[anwc@tedu.cn](mailto:anwc@tedu.cn)

17600945626

最后修改：2019.5.20

目录

[写在前面 6](#_Toc14164930)

[一、 抽象数据类型 7](#_Toc14164931)

[浅析 7](#_Toc14164932)

[定义 8](#_Toc14164933)

[抽象数据类型的基本实现思路 9](#_Toc14164934)

[基本思想 9](#_Toc14164935)

[基本操作 9](#_Toc14164936)

[引申：可变与不可变类型 9](#_Toc14164937)

[思考：数据结构是抽象数据类型的物理实现 9](#_Toc14164938)

[举例：抽象数据类型——复数 10](#_Toc14164939)

[抽象数据类型在Python中的实际应用——面向对象编程(Object Oriented Programming,简称OOP) 11](#_Toc14164940)

[二、 面向对象基本概念 12](#_Toc14164941)

[引言 12](#_Toc14164942)

[面向过程 13](#_Toc14164943)

[面向对象 13](#_Toc14164944)

[定义 13](#_Toc14164945)

[理解 13](#_Toc14164946)

[背景 14](#_Toc14164947)

[基本概念 16](#_Toc14164948)

[类的实现语法 16](#_Toc14164949)

[类的成员 17](#_Toc14164950)

[三、 面向对象三大特性 21](#_Toc14164951)

[封装 21](#_Toc14164952)

[定义 21](#_Toc14164953)

[私有成员 22](#_Toc14164954)

[继承 23](#_Toc14164955)

[定义 23](#_Toc14164956)

[语法 23](#_Toc14164957)

[单继承 23](#_Toc14164958)

[多继承 23](#_Toc14164959)

[MRO 23](#_Toc14164960)

[多态 24](#_Toc14164961)

[定义 24](#_Toc14164962)

[举个栗子 24](#_Toc14164963)

[四、 Python面向对象发展史——经典类、新式类、MRO算法演变及多继承 25](#_Toc14164964)

[相关概念介绍 25](#_Toc14164965)

[什么是MRO 25](#_Toc14164966)

[二义性 25](#_Toc14164967)

[深度遍历优先 26](#_Toc14164968)

[广度遍历优先 26](#_Toc14164969)

[拓扑排序 26](#_Toc14164970)

[Python版本介绍 27](#_Toc14164971)

[经典类 27](#_Toc14164972)

[新式类 27](#_Toc14164973)

[线性化过程中的单调性原则 27](#_Toc14164974)

[C3算法演变史 27](#_Toc14164975)

[问题由来 27](#_Toc14164976)

[Python2.2之前的经典类 30](#_Toc14164977)

[Python2.2版本中出现的新式类 31](#_Toc14164978)

[Python2.3-Python2.7，经典类和新式类并存，C3算法产生 31](#_Toc14164979)

[五、 Python中的元类 32](#_Toc14164980)

[类和对象的基本概念 32](#_Toc14164981)

[对象 32](#_Toc14164982)

[类 33](#_Toc14164983)

[类的特殊特性 33](#_Toc14164984)

[对象的特征 33](#_Toc14164985)

[type 33](#_Toc14164986)

[自定义元类 37](#_Toc14164987)

[说明 38](#_Toc14164988)

[六、 UML统一建模语言之——类图(组件及其实现) 38](#_Toc14164989)

[定义 38](#_Toc14164990)

[作用 38](#_Toc14164991)

[使用场景 38](#_Toc14164992)

[类图组件介绍 39](#_Toc14164993)

[类 39](#_Toc14164994)

[包 39](#_Toc14164995)

[接口 39](#_Toc14164996)

[关系 39](#_Toc14164997)

[七、 面向对象底层挖掘——类、实例的创建及初始化 41](#_Toc14164998)

[类的生命周期 41](#_Toc14164999)

[走进new方法，迎接init方法 42](#_Toc14165000)

[八、 面向对象六大设计原则(面向对象程序设计：OOD) 43](#_Toc14165001)

[封闭开放原则(目标，总的指导思想) 43](#_Toc14165002)

[定义 43](#_Toc14165003)

[理解 43](#_Toc14165004)

[封闭开放原则设计实例：计算器功能设计 45](#_Toc14165005)

[类的单一职责(一个类的定义) 46](#_Toc14165006)

[定义 46](#_Toc14165007)

[理解 46](#_Toc14165008)

[依赖倒置原则(依赖抽象) 47](#_Toc14165009)

[定义 47](#_Toc14165010)

[理解 47](#_Toc14165011)

[接口隔离原则 52](#_Toc14165012)

[定义 52](#_Toc14165013)

[理解 52](#_Toc14165014)

[组合复用原则(复用的最佳实践) 54](#_Toc14165015)

[定义 54](#_Toc14165016)

[理解 54](#_Toc14165017)

[里氏替换(继承后的重写，指导继承的设计) 55](#_Toc14165018)

[定义 55](#_Toc14165019)

[理解 55](#_Toc14165020)

[迪米特法则(类与类交互的原则) 55](#_Toc14165021)

[定义 55](#_Toc14165022)

[理解 55](#_Toc14165023)

[九、23+1种设计模式 56](#_Toc14165024)

[基础概念介绍 56](#_Toc14165025)

[设计模式 56](#_Toc14165026)

[抽象 56](#_Toc14165027)

[设计模式分类 57](#_Toc14165028)

[创建型模式 58](#_Toc14165029)

[工厂模式 58](#_Toc14165030)

[抽象工厂模式 63](#_Toc14165031)

[建造者模式 67](#_Toc14165032)

[原型模式 69](#_Toc14165033)

[单例模式 72](#_Toc14165034)

[结构型模式 81](#_Toc14165035)

[适配器模式 81](#_Toc14165036)

[桥接模式 82](#_Toc14165037)

[组合模式 85](#_Toc14165038)

[装饰模式 88](#_Toc14165039)

[外观模式 90](#_Toc14165040)

[享元模式 92](#_Toc14165041)

[行为型模式 94](#_Toc14165042)

[解释器模式 94](#_Toc14165043)

[模板方法模式 96](#_Toc14165044)

[责任链模式 98](#_Toc14165045)

[命令模式 99](#_Toc14165046)

[迭代器模式 101](#_Toc14165047)

[中介者模式 105](#_Toc14165048)

[备忘录模式 107](#_Toc14165049)

[观察者模式 109](#_Toc14165050)

[状态模式 111](#_Toc14165051)

[策略模式 114](#_Toc14165052)

[访问者模式 116](#_Toc14165053)

# 写在前面

程序猿，一个高大上中不失苦逼，牛x PLUS中不失煎熬的职业，一个让众多旁人羡煞不已的职业。我记得很早以前看过这样一个段子：要嫁就嫁程序员，为啥嘞？人傻钱多死的早。当然了，肯定是句玩笑话，但我们已经能够从中看出，社会对于程序员这个职位寄予的期望还是很高的。

程序员（英文Programmer）是从事程序开发、程序维护的专业人员。一般将程序员分为程序设计人员和程序编码人员，但两者的界限并不非常清楚，特别是在中国。软件从业人员分为[初级程序员](https://baike.baidu.com/item/%E5%88%9D%E7%BA%A7%E7%A8%8B%E5%BA%8F%E5%91%98/441400" \t "_blank)、中级程序员、高级程序员（现为软件设计师）、[系统分析员](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E5%88%86%E6%9E%90%E5%91%98/2296869" \t "_blank)，[系统架构师](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E6%9E%B6%E6%9E%84%E5%B8%88/3473607)，[测试工程师](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E5%B7%A5%E7%A8%8B%E5%B8%88/1907788)六大类(援引自百度百科，关键字：程序员)。

我个人觉得，程序员这个词，是一类专业从事软件相关行业的职位的习惯统称。其实说的再专业，对于一个程序猿，尤其是初级猿来说，其基本工作就是敲代码。从业这么多年，我见过形形色色的程序猿，有初入行的大学毕业生，有转行成功的小年轻，也有在某个开发领域摸爬滚打两三年，三四年的中年猿，也有已经小有成就的资深猿。除了最后这一类资深猿已步入成功人士行列，其余我见过的猿，他们早早晚晚都会遇到所谓的职业发展瓶颈：工作已两三年，三四年，还是个初级猿，想晋升，走技术路线，发现升上去很难；可是内心早已厌倦初级猿的那些活以及初级猿那可怜的行业最低标准薪水。为什么会难？初级猿，中级猿，走技术路线的晋升，无非就是向软件设计方向，即架构方向发展，可是架构设计，需要的不仅仅是扎实的编码功底，更重要的，是思想。国家软考，各位应该多少做过了解，国家软考分初，中，高级，其中中级软件设计师的难度是很高的，高级系统架构师就更别说了，这些级别的考试，其内容更加偏重的是编程理论与架构理论，诸如数据结构，编译原理，设计模式，面向对象程序设计等是重要的考试内容。而其中，面向对象程序设计是考察编程思想，编程理论的绝对重点。

编程思想，是一个程序猿最基本的生存技能。会写代码不重要，无非就是记记语法的事情，这很简单；会写“好”代码，这就很难了。老话说得好，思想高度决定上层建筑，优秀的编程思想，是高质量代码的有力保障。面向对象，作为当下90%的主流编程语言采用的编程思想，是所有程序员必须彻底熟练的技能。可以说，会写代码的猿不一定是个好猿(不一定完全理解面向对象)，但是(完全理解面向对象)一个好猿不仅会写代码，而且能写出高质量的代码。

基于此，特撰写这篇文章，跟大家一起聊聊理论，聊聊思想，聊聊面向对象。本文会从编程理论之一的数据结构出发，结合数据结构的基本理论，引入我们今天的主题。本文中，你会先对数据结构的基本理论做一个简单的了解，然后从数据结构基本理论我们引入面向对象的基本概念，了解完相关的基本概念之后，我们会重点介绍面向对象的三大特性，同时会针对Python面向对象相关的一些底层理论给各位做一个详细的介绍。紧接着我们会用大的篇幅讲解面向对象的六大设计原则，以及23+1种设计模式的实现原理及思想。本文，基本囊括面向对象所涉及到的所有知识点，篇幅较长，还请各位耐心阅读。好了废话不多说，我们先聊聊抽象数据类型。

# 抽象数据类型

## 浅析

在实际开发中，无论是设计所谓的xx管理系统，亦或是开发一个xx网站、APP，其实无非就是在跟数据打交道。而为了实现这些项目的开发，我们需要用到各种各样的编程语言，诸如C++，Java，Python，PHP……可是，在这些高级编程语言中，只有一些最最基本的数据类型，比如Python，有数字类型，字符型，序列，容器等，可是，仅凭这些基本数据类型，是无法满足我们的开发需求的，道理也很简单，难道只用这几种数据类型的数据，我就能开发出来一个网站？一个APP？当然不可能。这就是有关数据，产生的第一个问题：数据是否够用？

再比如，我独立开发出来了一个软件，哇很牛逼，非常受欢迎，那我肯定得指着他挣钱，但是为了让他挣钱就必须让他商业化，商业化意味着该软件必须投入使用，那么无论这个软件是C/S还是B/S架构，任何人都可以看得到你的源代码，甚至可以随意修改。显然，这是不安全的。这就是有关数据，产生的第二个问题：数据是否安全？何为安全，即：数据是否可以随意访问，数据内部是否可以随意修改？

再举例，比如我想开发一个电子控制系统，需要硬件，软件，硬件需要CPU，单片机，电源等等，软件需要采用c进行嵌入式开发；或者我想开发一个开关控制系统，也是需要硬件，软件，硬件同样也需要单片机，软件同样要用到c，那么问题来了：难道我每想做一个具体的产品，我都得一点一点的从头开始设计吗？这些软件之间是否有共性的问题呢？换句话表达：数据之间是否有共性呢？如何以数据为中心，如何围绕数据进行思考和设计？这就是有关数据，产生的第三个问题。

基于以上三个问题，我们现在引入抽象数据类型的概念。

## 定义

**全称：Abstract Data Type**

**中文名称：抽象数据类型**

**英文简称：ADT**

**定义：**

**1.百度百科定义：**抽象数据类型(ADT)是指一个**数学模型以及定义在此数学模型上的一组操作。**抽象数据类型需要通过固有数据类型（高级编程语言中已实现的数据类型）来实现。抽象数据类型是与表示无关的数据类型，是一个数据模型及定义在该模型上的一组运算。对一个抽象数据类型进行定义时，必须给出它的名字及各运算的运算符名，即函数名，并且规定这些函数的参数性质。一旦定义了一个抽象数据类型及其具体实现，程序设计中就可以像使用基本数据类型那样，十分方便地使用抽象数据类型

**2.通俗解释**：最开始的计算机语言，关注的都是如何更加有效率地计算，可以说其目的是计算层面的抽象。然而随着这个行业的不断发展，计算机不仅仅用于计算，开发也不仅只关注计算过程了，**数据层面**的抽象也变得同样重要。虽然计算机语言一开始就有对数据的抽象，但是那些都只是对一些最基本的数据类型而不包括我们想要用的，多种多样的数据。

程序处理的数据，通常是不同的类型的。只有事先约定好不同类型的数据的存储方式，计算机才能正确理解逻辑上不同的数据类型。所有编程语言都会有一组内置的基本数据类型。另外在实际工作过程中，或早或晚总会碰到一些没法用现有数据类型解决的问题，这时就需要自定义一些数据类型来解决。像Python这样比较高级的语言，在基本类型的基础上还添加了一些额外的数据结构如tuple,list,dict（这些如果从广义上来说也算是Python的数据类型）。

以上基本数据类型都是比较simple的结构，但是他们具有一个共通的问题，就是都会把数据暴露在外。这点类似于一个人如果有了对某个变量的权限的话他就可以看到这个变量代表的数据结构中的所有数据。而如果我们不希望被看到，那么这个问题可能就比较严重了。为了解决这个问题，就必须要有一种数据类型，它可以让使用者只需要考虑如何去使用，而不需要去关注类型内部的具体实现方式以及数据的表示等等。这样的类型从概念上来说就是抽象数据类型了。

**抽象数据类型 = 逻辑结构+抽象运算。**逻辑结构，即我们所说的数据之间的逻辑关系，抽象运算，即我们赋予抽象数据类型能够提供的自定义“运算”（操作||功能）

抽象数据类型，实质上，就是在描述问题本身，而跟计算机，是没有任何的关系的，也就是我们前面所说到的，脱离计算机，以数据为中心，围绕数据展开思考与设计。

抽象数据类型存在的目标是什么？很简单，即：在不涉及具体的，和计算机系统相关的细节情况下，优先考虑，理解，思考问题本身，在此基础之上，实现用计算机求解问题的过程。

## 抽象数据类型的基本实现思路

### 基本思想

实现一个抽象数据类型，其基本思想均是把数据定义为抽象的数据对象集合，只为他们定义可用的合法操作，而不暴露具体的内部细节。注意：不论是操作细节还是数据存储细节均不暴露

### 基本操作

基于以上的基本实现思想，一般而言，抽象数据类型应该至少具备以下三种操作：

1. 构造操作：即如何通过抽象数据类型，创建抽象数据对象
2. 解析操作(也可以理解为获取操作)：一定会有一系列getxxx()方法去获取该抽象数据类型内部的任意数据及操作
3. 修改操作：一定会有一系列的setxxx()方法去给当前抽象数据类型修改任意数据及操作

### 引申：可变与不可变类型

基于以上说到的三种基本操作，可以判断一个抽象数据类型的可变性

1. 如果一个抽象数据类型同时具备以上三种操作，那么这个抽象数据类型就是可变的
2. 如果一个抽象数据类型只具备一二两种操作，不具备第三种操作，那么这个抽象数据类型就是不可变的

### 思考：数据结构是抽象数据类型的物理实现

基于以上抽象数据类型的所有内容，我们对这句话应该就不难理解了。抽象数据类型，是我们自定义的，区别于各大编程语言中的那些基本的数据类型，它高度自定义，可以存储任意数据，解决了开发过程中数据不够使用的问题，同时，抽象数据类型只提供功能，不提供类型内部的数据及操作的访问权限，解决了数据使用过程中的安全性问题，再者，抽象数据类型将开发过程中对计算机运算的关注转移到对数据的抽象，实现了以数据为中心的思考与设计。但，其终归只是一种类型。而在数据结构的概念中，一切的概念都是**围绕一个个具体的数据对象展开的**，这些数据对象，就是**通过一个个具体的抽象数据类型来创建的，由类型得到对象，由抽象的概念实例化出一个具体存在的数据，这个过程，就叫做物理实现。**所以，Clifford A.Shaffer在《数据结构与算法分析》一书中，对数据结构的定义是：“**数据结构是ADT（抽象数据类型Abstract Data Type） 的物理实现。**”

## 举例：抽象数据类型——复数

存储结构：

**class** Complex(object):  
 *'''  
 创建一个复数抽象数据类型  
 定义成员：  
 数据：实部，虚部  
 操作：  
 1.构造操作  
 2.删除操作  
 3.返回复数z实部值操作  
 4.返回复数z虚部值操作  
 ...  
 '''* **def** \_\_init\_\_(self):  
 self.real;#定义实部  
 self.imag;#定义虚部  
   
 **def** AssignComplex(self,z,v1,v2):  
 *'''构造一个复数z'''* **pass** ...  
   
 **def** DestroyComplex(self,z):  
 *'''删除(销毁)一个复数z'''* **pass** ...  
   
 **def** getReal(self,z):  
 *'''返回复数z的实部值'''* **pass** ...  
   
 **def** getImag(self,z):  
 *'''返回复数z的虚部值'''* **pass** ...  
 ...

逻辑结构（采用二元组表示法）：

ADT Complex{

数据对象：D={real,imag|real,imag均为实数}

数据关系：R={<real，imag>|real是实部，imag是虚部}

基本操作:

AssignComplex(z,v1,v2):构造复数z

DestroyComplex(z):删除(销毁)一个复数

GetReal(z):返回复数z的实部

GetImag(z):返回复数z的虚部

…

}

## 抽象数据类型在Python中的实际应用——面向对象编程(Object Oriented Programming,简称OOP)

Python中，‘一切皆对象’。是对象，就有属性及方法。是对象，就有封装、继承、多态三大特性。这些都是套话了，理解起来其实也不难。但是，如果你不了解抽象数据类型，那么对于这些概念的理解你仍然只是理解了表象。

数据结构中，常见的逻辑结构，比如线性表，比如树形结构，比如图形结构，其本质，都是对象，都是一种抽象数据类型。我们所说的这些名词仅仅只是逻辑层面对这类对象的一个描述。而根据我们对于抽象数据类型的了解，其实现是需要基于现有的一些基本数据类型，比如整型，浮点型，字符串……同时，抽象数据类型实现了基本数据类型实现不了的数据对象的封装功能，其只对外提供功能接口但不会轻易提供访问抽象数据类型内部的方法及权限(至少一般用户是不具备这些方法及权限的)，并且抽象数据类型高度自定义，根据实际问题出发，基于数据解决问题，将传统的围绕计算为中心的设计转化为围绕问题本质、围绕数据为中心的设计，这些种种，其实就是面向对象的思想。

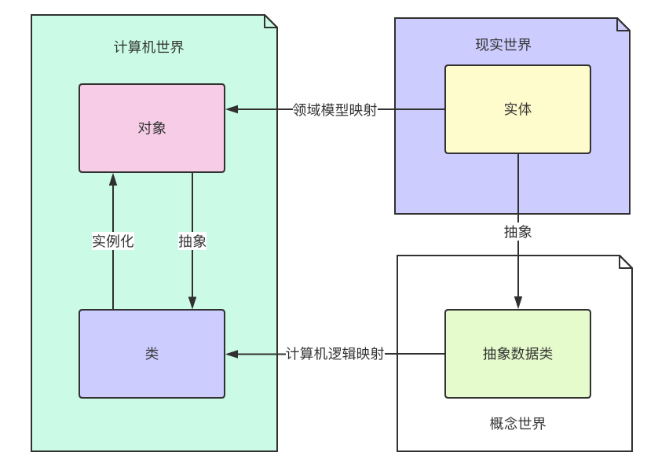
抽象数据类型与面向对象是有关系的，而且关系还很大。面向对象中类的概念主要来自于抽象数据类型理论，是基于抽象数据类型发展而来的。最基本的，抽象数据类型是一种以数据为中心的自定义类型，可提供相关属性与方法供直接使用。而面向对象中的类，是对现实对象进行的一个抽象化，将现实对象抽象成数据理论，从实物中提炼出对象的属性和行为，并在程序中用字段（属性）和方法（函数）描述出来。这一点，与抽象数据类型的理论恰恰是相辅相成的。同样的，抽象数据类型理论是为了解决数据的安全性问题，即解决基本数据类型存在的将数据本身暴露在外的问题。而面向对象中的类，封装是其三大特性之一，其就是为了解决类内部的成员，谁可以访问，谁不可以访问的问题，这一点上，又是不谋而合。再比如，抽象数据类型解决了基本数据类型解决不了的问题，即如何围绕问题本质、围绕数据进行程序设计。面向对象，其本身就是一种编程思想，其注重的就是功能，以功能为中心，注重解决问题的方法，围绕具体事务展开抽象，提炼属性与方法，并使用属性与函数进行描述，这一点，又不谋而合。

综上，抽象数据类型的基本理论奠定了面向对象的基本思想。接下来，我们从抽象数据类型开始出发，一步步开始面向对象的学习。后文所有内容的主线思路，都是围绕抽象数据类型展开。

# 面向对象基本概念

## 引言

我们举个例子：老安最近想买车了，同事这边有开丰田的，推荐卡罗拉双擎，油电混，省钱环保性价比高；有开哈弗的，推荐H5，买的人多各方面也不错；公司大领导独爱德系车型，推荐迈腾；家里老爹喜欢SUV，推荐逍客；老安自己觉得现在天津机动车牌照难摇不想等，想直接买新能源……车，是一个总称；丰田卡罗拉双擎，哈弗H5等等，是某一品牌的某一款具体车型；老安买到手的“新能源双座敞篷”爱玛大电摩，电动牌照号：5759717是一辆具体的车（电动咋的了- -、）。也就是说，车，是一类交通工具的总称，而具体的每一辆机动车，是车这个交通工具的某几个具体存在的实物。讲到这里，我们先看下面这个图：



在现实世界中，机动车，非机动车，自行车，电动车吧啦吧啦各种车，其实都是个车，即车，是这些具体车辆的总称，是一类交通工具，那么我可以在概念世界中抽象出来一个类型，叫车。车这个类型，有哪些具体的实体？汽车，自行车……，每一个具体的实体，是车这个类的某一个具体的实例。

好，讲到这里先解释一下上面出现的“抽象”这个词，什么叫抽象？从具体的事物抽出、概括出它们共同的特性，属性与关系等，而将个别的、非本质的方面舍弃，这种思维过程，就叫做抽象。

我们继续理解这个抽象的过程。

笔是某一类产品，只要能用来写字的都叫笔，那笔有哪些种？钢笔、中性笔、毛笔、签字笔……，这些笔都有一个共性，都能写字。那我们抽象一下，拿出他们的共性：能写字这个特点，然后抽象出一个概念：能写字的叫笔。

手机是某一类产品，只要能拿来打电话发短信的都叫手机，手机都有哪些种？三星NOTE 9、iPhone XS、华为P30 PRO……，这些产品都有这几个共性：都能打电话，发短信，上网，打游戏。那我们抽象一下，拿出他们的共性：打电话，发短信，上网，打游戏这些特点，然后抽象出一个概念：能打电话，发短信，上网，打游戏的都叫手机。

现实世界，存在着很多独立的实体，但是，总有若干个实体，具备很多相同或相似的，共性的特征，我们从现实世界中这些若干个具有共性特征的实体中，就可以抽象出一个概念：抽象数据类。这些抽象数据类，映射到计算机中，就是一个个具体的类，而这些若干个实体，映射到计算机中，就是一个个对象。由上图可以看到，类可以实例化一个或若干个对象，若干个对象也可以抽象为一个类。

为了帮助大家理解面向对象的基本概念，我们先了解什么叫面向过程。

## 面向过程

“面向过程”(Procedure Oriented)是一种以过程为中心的编程思想。“面向过程”也可称之为“面向记录”编程思想，他们不支持丰富的“[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1)”特性（比如继承、[多态](https://baike.baidu.com/item/%E5%A4%9A%E6%80%81)），并且它们不允许混合持久化状态和域逻辑。

面向过程以事件为中心，分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。

面向过程的程序设计思路：程序 = 数据结构 + 算法。即：做一件事，肯定伴随着数据的获取及调用以及每一步要做的具体的事是什么。计算机层面，即：设计一个程序，需要考虑两方面，第一，数据的存取操作；第二，算法的执行操作。

面向过程程序设计其实也是有优点的，比如对于复杂问题的流程化，进而实现对问题的简单化，再比如所有的细节自己掌控等等。但是，面向过程有其最致命的缺点：安全性低，可拓展性差。我们都知道，C是几乎绝大多数工科类专业大一的必修课，没错，C语言就是面向过程的。其特点：简洁紧凑，灵活方便，允许直接访问物理地址，可移植性好，执行效率高……,但，作为一门高级编程语言，其也有不可忽略且同样致命的缺点：数据的封装。C在数据的安全性上有非常大的缺陷，做一个假设：如果全程使用C做一个微信……乖乖，用户等死吧……

## 面向对象

### 定义

面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)(Object Oriented,OO)是[软件开发方法](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E6%96%B9%E6%B3%95/971447" \t "_blank)。面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的概念和应用已超越了[程序设计](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1/223952)和软件开发，扩展到如[数据库系统](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B3%BB%E7%BB%9F/215176)、交互式界面、应用结构、应用平台、[分布式系统](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E7%B3%BB%E7%BB%9F/4905336)、[网络管理](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E7%AE%A1%E7%90%86/5903609)结构、CAD技术、[人工智能](https://baike.baidu.com/item/%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD/9180" \t "_blank)等领域。面向对象是一种对现实世界理解和抽象的方法，是计算机编程技术发展到一定阶段后的产物。

### 理解

面向对象与面向过程，是两种不同的编程思想。早期的计算机编程是基于面向过程的方法，例如实现算术运算1+1+2 = 4，通过设计一个算法就可以解决当时的问题。随着计算机技术的不断提高，计算机被用于解决越来越复杂的问题。一切事物皆对象，通过面向对象的方式，将现实世界的事物抽象成对象，现实世界中的关系抽象成类、[继承](https://baike.baidu.com/item/%E7%BB%A7%E6%89%BF)，帮助人们实现对现实世界的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)与数据建模。通过面向对象的方法，更利于用人理解的方式对复杂系统进行分析、设计与编程。同时，面向对象能有效提高[编程](https://baike.baidu.com/item/%E7%BC%96%E7%A8%8B)的效率，通过封装技术，消息机制可以像搭积木的一样快速开发出一个全新的系统。面向对象是指一种[程序设计范型](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E8%8C%83%E5%9E%8B)，同时也是一种程序开发的方法。对象指的是[类](https://baike.baidu.com/item/%E7%B1%BB)的集合。它将[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)作为[程序](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F)的基本单元，将程序和[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE)[封装](https://baike.baidu.com/item/%E5%B0%81%E8%A3%85)其中，以提高软件的重用性、灵活性和扩展性。

起初，“面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)”是专指在[程序设计](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1" \t "_blank)中采用[封装](https://baike.baidu.com/item/%E5%B0%81%E8%A3%85)、[继承](https://baike.baidu.com/item/%E7%BB%A7%E6%89%BF)、[多态](https://baike.baidu.com/item/%E5%A4%9A%E6%80%81)等设计方法。

面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)的思想已经涉及到[软件开发](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91)的各个方面。如，面向对象的分析（[OOA](https://baike.baidu.com/item/OOA)，Object Oriented Analysis），面向对象的设计（[OOD](https://baike.baidu.com/item/OOD" \t "_blank)，Object Oriented Design）、以及我们经常说的面向对象的编程实现（[OOP](https://baike.baidu.com/item/OOP" \t "_blank)，Object Oriented Programming）。

面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的分析根据[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)关键的问题域来分解系统。面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的设计是一种提供符号设计系统的面向对象的实现过程，它用非常接近实际领域术语的方法把系统构造成“现实世界”的对象。面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列[函数](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0)的集合，或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。

### 背景

面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)是在[结构化设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)出现很多问题的情况下应运而生的。[结构化设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)求解问题的基本策略是从功能的角度审视问题域。它将应用程序看成实现某些特定任务的[功能模块](https://baike.baidu.com/item/%E5%8A%9F%E8%83%BD%E6%A8%A1%E5%9D%97)，其中子过程是实现某项具体操作的底层功能模块。在每个功能模块中，用[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)描述待处理数据的组织形式，用[算法](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95)描述具体的操作过程。面对日趋复杂的应用系统，这种开发思路在下面几个方面逐渐暴露了一些弱点。

1.审视问题域的视角

在现实世界中存在的[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)是问题域中的主角，所谓客体是指客观存在的[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)实体和主观[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)的概念，他是人类观察问题和解决问题的主要目标。例如，对于一个学校[学生管理系统](https://baike.baidu.com/item/%E5%AD%A6%E7%94%9F%E7%AE%A1%E7%90%86%E7%B3%BB%E7%BB%9F)来说，无论是简单还是复杂，始终是围绕学生和老师这两个[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)实施。在自然界，每个[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)都具有一些属性和行为，例如学生有学号、姓名、性别等属性，以及上课、考试、做实验等行为。因此，每个个体都可以用属性和行为来描述。

通常人类观察问题的视角是这些[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)，客体的属性反应客体在某一时刻的状态，客体的行为反映客体能从事的操作。这些操作附在[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)之上并能用来设置、改变和获取客体的状态。任何问题域都有一系列的[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)，因此解决问题的基本方式是让这些客体之间相互驱动、相互作用，最终使每个客体按照设计者的意愿改变其属性状态。

[结构化设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)所采用的设计思路不是将[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)作为一个整体，而是将依附于客体之上的行为抽取出来，以功能为目标来设计构造应用系统。这种做法导致在进行[程序设计](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)的时候，不得不将[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)所构成的现实世界映射到由[功能模块](https://baike.baidu.com/item/%E5%8A%9F%E8%83%BD%E6%A8%A1%E5%9D%97)组成的解空间中，这种变换过程，不仅增加了程序设计的复杂程度，而且背离了人们观察问题和解决问题的基本思路。另外，再仔细思考会发现，在任何一个问题域中，[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)是稳定的，而行为是不稳定的。例如，不管是国家图书馆，还是学校图书馆，还是国际图书馆，都会含有图书这个[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)，但管理图书的方法可能是截然不同的。[结构化设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)将审视问题的视角定位于不稳定的操作之上，并将描述[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)的属性和行为分开，使得应用程序的日后维护和扩展相当困难，甚至一个微小的变动，都会波及到整个系统。面对问题规模的日趋扩大、环境的日趋复杂、需求变化的日趋加快，将利用计算机解决问题的基本方法统一到人类解决问题的习惯方法之上，彻底改变[软件设计](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E8%AE%BE%E8%AE%A1)方法与人类解决问题的常规方式扭曲的现象迫在眉睫，这是提出面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的首要原因。

2.[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)级别

[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)是人类解决问题的基本法宝。良好的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)策略可以控制问题的复杂程度，增强系统的通用性和[可扩展性](https://baike.baidu.com/item/%E5%8F%AF%E6%89%A9%E5%B1%95%E6%80%A7)。[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)主要包括过程抽象和数据抽象。[结构化设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)应用的是过程[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)。所谓过程[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)是将问题域中具有明确功能定义的操作抽取出来，并将其作为一个实体看待。这种[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)级别对于软件系统结构的设计显得有些武断，并且稳定性差，导致很难准确无误地设计出系统的每一个操作环节。一旦某个[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)属性的表示方式发生了变化，就有可能牵扯到已有系统的很多部分。而数据[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)是较过程抽象更高级别的抽象方式，将描述[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)的属性和行为绑定在一起，实现统一的抽象，从而达到对现实世界客体的真正[模拟](https://baike.baidu.com/item/%E6%A8%A1%E6%8B%9F)。

3.[封装](https://baike.baidu.com/item/%E5%B0%81%E8%A3%85)体

[封装](https://baike.baidu.com/item/%E5%B0%81%E8%A3%85)是指将现实世界中存在的某个[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)的属性与行为绑定在一起，并放置在一个逻辑单元内。该逻辑单元负责将所描述的属性隐藏起来，外界对[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)内部属性的所有访问只能通过提供的[用户接口](https://baike.baidu.com/item/%E7%94%A8%E6%88%B7%E6%8E%A5%E5%8F%A3)实现。这样做既可以实现对[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)属性的保护作用，又可以提高[软件系统](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E7%B3%BB%E7%BB%9F)的可维护性。只要用户接口不改变，任何[封装](https://baike.baidu.com/item/%E5%B0%81%E8%A3%85)体内部的改变都不会对[软件系统](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E7%B3%BB%E7%BB%9F)的其他部分造成影响。[结构化设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)没有做到[客体](https://baike.baidu.com/item/%E5%AE%A2%E4%BD%93)的整体[封装](https://baike.baidu.com/item/%E5%B0%81%E8%A3%85)，只是封装了各个[功能模块](https://baike.baidu.com/item/%E5%8A%9F%E8%83%BD%E6%A8%A1%E5%9D%97)，而每个功能模块可以随意地对没有保护能力客体属性实施操作，并且由于描述属性的数据与行为被分割开来，所以一旦某个客体属性的表达方式发生了变化，或某个行为效果发生了改变，就有可能对整个系统产生影响。

4.可重用性

可[重用](https://baike.baidu.com/item/%E9%87%8D%E7%94%A8)性标识着软件产品的可复用能力，是衡量一个软件产品成功与否的重要标志。当今的软件开发行业，人们越来越追求开发更多的、更有通用性的可重用[构件](https://baike.baidu.com/item/%E6%9E%84%E4%BB%B6)，从而使软件开发过程彻底改善，即从过去的语句级编写发展到现在的构件组装，从而提高软件开发效率，推动应用领域迅速扩展。然而，[结构化程序设计方法](https://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E5%8C%96%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E6%96%B9%E6%B3%95)的基本单位是模块，每个模块只是实现特定功能的过程描述，因此，它的可重用单位只能是模块。例如，在[C语言](https://baike.baidu.com/item/C%E8%AF%AD%E8%A8%80)编写程序时使用大量的[标准函数](https://baike.baidu.com/item/%E6%A0%87%E5%87%86%E5%87%BD%E6%95%B0)。但对于今天的软件开发来说，这样的重用力度显得微不足道，而且当参与操作的某些[数据类型](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B)发生变化时，就不能够再使用那些函数了。因此，渴望更大力度的可重用[构件](https://baike.baidu.com/item/%E6%9E%84%E4%BB%B6)是如今应用领域对[软件开发](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91)提出的新需求。

上述弱点驱使人们寻求一种新的程序设计方法，以适应现代社会对[软件开发](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91)的更高要求，面向[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)由此产生。

### 基本概念

#### 对象

对象是人们要进行研究的任何事物，小到一颗沙粒，大到整个银河系，所有的一切，皆对象。对象不仅能够表示具体的事物，还能表示抽象的规划、计划或事件。

对象具有状态，一个对象用数据值来描述它的状态

对象还有行为，即操作，用于改变对象的状态，对象及其操作就是对象的行为

对象实现了数据和操作的结合，使数据和操作封装于对象这个统一体中

对象，是类的具体实例，即归属于某个类别的“个体”

#### 类

具有相同特性（[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0" \t "_blank)）和行为（功能）的[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)就是类。因此，[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)是类，类的具体化就是对象，也可以说类的实例是对象，类实际上就是一种[数据类型](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B)。

类具有属性，它是[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)的状态的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)，用[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)来描述类的属性。

类具有操作，它是[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的行为的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)，用操作名和实现该操作的方法来描述。

类具有结构。在客观世界中有若干类，这些类之间有一定的结构关系。通常有两种主要的结构关系，即一般--具体结构关系，整体--部分结构关系。

①一般--具体结构称为分类结构，也可以说是“或”关系，或者是“is a”关系。

②整体--部分结构称为组装结构，它们之间的关系是一种“与”关系，或者是“has a”关系。

通常，类与类之间的行为不同，对象与对象之间的数据不同。

类是创建对象的模板。在面向对象的思想中，所有的对象（实例）都是由类创建的。

#### 成员

数据成员：名词类型的状态。（属性）

方法成员：动词类型的行为。（方法）

### 类的实现语法

定义类：

1. 代码

class 类名:

“””文档字符串，用于当前类的文档说明”””

def \_init\_(self,参数列表):

self.实例变量 = 参数

方法成员

1. 说明

-- 类名所有单词首字母大写.

-- \_init\_ 也叫构造函数，创建对象时被调用，也可以省略。

-- self 变量绑定的是被创建的对象（获得的实例），名称可以随意。

创建对象：

变量 = 类名 (参数列表)

### 类的成员

#### 属性

类的公有属性：

在类的内部与外部，皆可以通过类及类的实例（对象）来进行访问；但是只能通过类来进行修改。

类的私有属性：

在类的内部，皆可以通过类及类的实例（对象）来进行访问；

在类的外部，无法通过类及类的实例（对象）来进行访问。

实例的公有属性：

在类的内部，无法通过类访问，但可以通过实例访问

在类的外部，无法通过类访问，但可以通过实例访问

实例的私有属性：

在类的内部，无法通过类访问，但可以通过实例访问

在类的外部，无法通过类访问，也无法通过实例访问

附：测试代码：

**class** A(object):  
 classVariable = **'这是类属性，且是公有属性'** \_\_classVariable = **'这是类属性，且是私有属性'  
 def** \_\_init\_\_(self):  
 self.objectVariable = **'这是实例属性，且是公有属性'** self.\_\_objectVatiable = **'这是实例属性，且是私有属性'  
 def** test\_in\_all\_variable(self):  
 # 以下测试为：在类的内部部，对于类的公有属性、私有属性，实例的公有属性、私有属性调用情况  
 # #######################  
 # print(A.classVariable)  
 # print(self.classVariable)  
 # 测试类的公有属性在类的内部的调用情况  
 # 结论：在类的内部，类的公有属性可以直接通过类或实例进行访问  
 # ########################  
 # print(A.\_\_classVariable)  
 # print(self.\_\_classVariable)  
 # 测试类的私有属性在类的内部的调用情况  
 # 结论：在类的内部，类的私有属性可以直接通过类或实例进行访问  
 #########################  
 # print(A.objectVariable)  
 # print(self.objectVariable)  
 # 测试实例的公有属性在类的内部的调用情况  
 # 结论：在类的内部，实例的公有属性无法直接通过类进行访问  
 # 实例的公有属性可直接通过实例进行访问  
 #########################  
 # print(A.\_\_objectVatiable)  
 # print(a.\_\_objectVatiable)  
 # 测试实例的私有属性在类的内部的调用情况  
 # 结论：在类的内部，实例的私有属性无法直接通过类进行访问  
 # 实例的私有属性可直接通过实例进行访问  
 #########################  
  
a = A()  
a.test\_in\_all\_variable()  
  
# 以下测试为：在类的外部，对于类的公有属性、私有属性，实例的公有属性、私有属性调用情况  
####################################  
# print(A.classVariable)  
# print(a.classVariable)  
# 测试类的公有属性在类的外部的调用情况  
# 结论：在类的外部，类的公有属性可直接通过类及类的实例访问  
####################################  
# print(A.\_\_classVariable)  
# print(a.\_\_classVariable)  
# 测试类的私有属性在类的外部的调用情况  
# 结论:在类的外部,类的私有属性无法通过类及类的实例访问  
####################################  
# print(A.objectVariable)  
# print(a.objectVariable)  
# 测试实例的公有属性在类的外部的调用情况  
# 结论:在类的外部,实例的公有属性不能通过类访问,可以通过实例访问  
####################################  
# print(A.\_\_objectVariable)  
# print(a.\_\_objectVariable)  
# 测试实例的私有属性在类的外部的调用情况  
# 结论:在类的外部,实例的私有属性不能通过类访问,也不能通过实例访问  
####################################

现：将上述测试内容辅以表格进行总结展示，方便归纳整理：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 区域/调用者 | 类的公有属性 | | 类的私有属性 | | 实例的公有属性 | | 实例的私有属性 | |
| 类 | 实例 | 类 | 实例 | 类 | 实例 | 类 | 实例 |
| 类的内部 | √ | √ | √ | √ | × | √ | × | √ |
| 类的外部 | √ | √ | × | × | × | √ | × | × |

内置属性：

\_\_dict\_\_ : 类的属性（包含一个字典，由类的数据属性组成）

\_\_doc\_\_ :类的文档字符串

\_\_name\_\_: 类名

\_\_module\_\_: 类定义所在的模块（类的全名是'\_\_main\_\_.className'，如果类位于一个导入模块mymod中，那么className.\_\_module\_\_ 等于 mymod）

\_\_bases\_\_ : 类的所有父类构成元素（包含了以个由所有父类组成的元组）

#### 方法

类的公有方法：

归属于类对象，类本身及类的实例皆可在类的内外部调用

类的私有方法：

归属于类对象，类本身及类的实例在类的内部可调用

实例的公有方法：

归属于实例，在类的内外部皆可由实例调用

实例的私有方法：

归属于实例，在类的内部可由实例调用

初始化方法：

\_\_init\_\_:初始化当前类所创建的实例

析构方法：

\_\_del\_\_:删除当前实例对象

魔术方法：

一系列的\_\_开头\_\_结尾的内置方法，无需手动调用，对类进行一系列操作时自动调用

\_\_new\_\_:真正的构造方法，用于创建当前类的实例对象

\_\_str\_\_、\_\_repr\_\_:两者的目的都是为了显式的显示对象的一些信息，方便查看和调试

\_\_call\_\_:将类对象变为可调用的对象，因为函数本身就是个对象，每一个函数都具备\_\_call\_\_方法。

\_\_iter\_\_、\_\_next\_\_：迭代器函数重写时使用，目的在于实现自定类的序列化

\_\_getitem\_\_、\_\_setitem\_\_、\_\_delitem\_\_：重写此三个方法可以将自定义类模拟成字典。

\_\_getattr\_\_、\_\_setattr\_\_、\_\_delattr\_\_：获取当前类的相关属性、设置当前类的相关属性、删除当前类的相关属性

…

抽象方法：

没有方法体的方法，常出现于抽象类中

代码举例：

**from** abc **import** ABCMeta,abstractmethod  
**class** TestUseCase(object,metaclass=ABCMeta):  
 '''测试类，用于展示类中可能出现的部分方法'''  
 **def** \_\_new\_\_(cls, *\*args*, *\*\*kwargs*):  
 '''\_\_new\_\_方法，用于构造(创建)实例对象'''  
 **return** object.\_\_new\_\_(cls)  
  
 **def** \_\_init\_\_(self):  
 '''初始化方法，初始化当前实例'''  
 **pass  
  
 def** \_\_call\_\_(self, *\*args*, *\*\*kwargs*):  
 '''\_\_call\_\_方法，用于实现类的可调用'''  
 **pass  
  
 def** \_\_del\_\_(self):  
 '''析构方法，用于回收实例对象'''  
 **pass  
  
 def** \_\_str\_\_(self):  
 '''显式显示对象信息'''  
 **pass  
  
 def** \_\_repr\_\_(self):  
 '''显式显示对象信息'''  
 **pass** ###################  
 **def** \_\_iter\_\_(self):  
 **pass** # 迭代器函数重写  
  
 **def** \_\_next\_\_(self):  
 **pass** ###################  
  
 @classmethod  
 **def** classMethodPublic(cls):  
 '''类的公有方法'''  
 **pass** @classmethod  
 **def** \_\_classMethodPrivate(cls):  
 '''类的私有方法'''  
 **pass  
  
 def** objectMethodPublic(self):  
 '''实例的公有方法'''  
 **pass  
  
 def** \_\_objectMethodPrivate(self):  
 '''实例的私有方法'''  
 **pass** @staticmethod  
 **def** staticMethod():  
 '''静态方法'''  
 **pass** @abstractmethod  
 **def** abstractMethod(self):  
 '''抽象方法'''  
 **pass**

# 面向对象三大特性

## 封装

### 定义

封装，即隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别；将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81/3814213)进行有机的结合，形成“类”，其中数据和函数都是类的成员。

在面向对象程序设计中(OOD)中，基本大致的思路是将属性及方法封装在当前对象中，由对象提供“接口”，其他附加到该接口的对象不需要关心其实现的方法，只需要使用就可以了。这个概念大致的意思可以理解为：“你不用管我怎么做的，你用就完了，别问，别说话，更别尝试查看我内部，你看不到的”。对象的接口包含了公共的属性和方法

从数据的角度来看，封装，是将基本数据类型复合成抽象数据类型

从行为角度来看，封装，向类外提供必要的功能，同时隐藏内部的实现细节

从代码角度来看，封装，其实是对象对外提供公共接口(属性及方法)，整体的功能实现依靠公共接口对内部方法的实现，外部只需要调用这些公共接口就可以完成功能而没有必要知道具体的内部细节。

从设计角度来看，大致有以下几种思路：

1.分而治之：将一个大的用例拆分为若干个小用例，拆分的依据是逻辑层面上的划分，将每个小用例封装为一个类，那么这个类的职责就很明确：单独实现一个具体的功能，这样做的好处在于：降低编码难度，提高代码的可读性，可复用性，可扩展性

2.对于经常变化的地方，进行单独封装，避免影响其他功能或其他类

3.高内聚，低耦合：什么叫高内聚，即类内的各个方法职责尽量单一，每一个类的职责尽量单一(单一职责的类)；什么叫低耦合，即类与类之间的关系越少越好，关联性越小越好，依赖性越小越好，让一个类的改变尽量小的去影响其他的类。

### 私有成员

当类中的某些成员，我们不想让外界访问或使用，我们就可以将其封装为私有成员。私有成员，即当前类私有，当前实例私有，只有在类的内部或实例的内部才能调用这些私有成员，一旦访问范围超出当前类或当前实例，则无法访问。

私有成员的实现非常简单，在声明的成员变量之前添加’\_\_’(双下划线)即可。

私有成员可通过\_\_dict\_\_属性及dir函数(自省函数)访问

代码举例：

**class** TestUseCase(object):  
 '''测试类，用于测试私有成员的封装'''  
 \_\_classVariable = **'我是私有的类变量'** classVariable = **'我是公有的类变量'  
  
 def** \_\_init\_\_(self):  
 self.objectVariable = **'我是公有的实例变量'** self.\_\_objectVariable = **'我是私有的实例变量'  
  
 def** \_\_objectFunction(self):  
 **return '我是私有的实例方法'  
  
 def** objectFunction(self):  
 **return '我是公有的实例方法'  
  
 def** testFunction(self):  
 '''测试方法(公有的实例方法)，测试在类的内部对于私有成员访问'''  
 print(TestUseCase.\_\_classVariable)  
 print(self.\_\_objectVariable)  
 **return** self.\_\_objectFunction()  
  
usecase1 = TestUseCase()  
res = usecase1.testFunction()#通过公有成员接口访问内部私有成员  
print(res)  
#######################  
# 测试在类的外部访问私有成员  
TestUseCase.\_\_classVariable()#报错：当前测试类没有该属性，说明在外部不能访问私有成员

### \_\_slots\_\_

#### 作用

限定一个类创建的实例只能有固定的实例变量，不能再额外添加。

#### 语法

在类中定义:\_\_slots\_\_ = (“变量名1”,”变量名2”…..)

#### 说明

含有\_\_slots\_\_属性的类所创建的对象没有\_\_dict\_\_属性, 即此实例不用字典来存储对象的实例属性。

#### 优点

访止用户因错写属性的名称而发生程序错误。

#### 缺点

丧失了动态语言可以在运行时为对象添加变量的灵活性。

### @property

在绑定属性时，如果我们直接把属性赋值给对象，比如：

p = Person()  
p.name= **'Mary'**

会出现什么问题？先看下面这个例子，注意双下划线的私有属性：

class Person(object):  
 def \_\_init\_\_(self, name, age):  
 self.\_\_name = name  
 self.\_\_age = age  
  
 def get\_age\_fun(self):  
 return self.\_\_age  
  
 def set\_age\_fun(self, value):  
 if not isinstance(value, int):  
 raise ValueError(**'年龄必须是数字!'**)  
 if value < 0 or value > 100:  
 raise ValueError(**'年龄必须是0-100'**)  
 self.\_\_age = value  
  
 def print\_info(self):  
 print(**'%s: %s'** % (self.\_\_name, self.\_\_age))  
  
  
p = Person(**'balala'**,20)  
p.\_\_age = 17  
print(p.\_\_age) *# 17*print(p.get\_age\_fun()) *# 20 表面上看，上面代码“成功”地设置了\_\_age变量 17,但实际上这个\_\_age变量和class内部的\_\_age变量不是一个变量！  
# 内部的\_\_age变量已经被Python解释器自动改成了\_Person\_age，而外部代码给p新增了一个\_\_age变量。 所以调用 get\_age\_fun输出的是初始值*p.set\_age\_fun(35)  
print(p.get\_age\_fun()) *# 35*print(p.print\_info()) *# balala: 35*

输出结果：

17  
20  
35  
balala: 35

表面上看，外部代码“成功”地设置了\_\_age变量 17，但实际上这个\_age变量和class内部的\_age变量不是一个变量！  
内部的\_age变量已经被Python解释器自动改成了\_Person\_age，而外部代码给p新增了一个\_age变量。 所以调用 get\_age\_fun输出的是初始值 20  
而set\_age\_fun 通过class内部改变了age变量值,所以最终输出 balala: 35

我们再稍微调整下:(注意只改变了一个变量名: 原来的私有属性 \_\_age 单下划线为: \_age,也可以定义为:age. )

解释:以一个下划线开头的实例变量名，比如\_age，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当看到这样的变量时，意思是，"虽然可以被访问，但是，请视为私有变量，不要随意访问。"

class Person(object):  
 def \_\_init\_\_(self, name, age):  
 self.\_\_name = name  
 self.\_age = age  
  
 def get\_age\_fun(self):  
 return self.\_age  
  
 def set\_age\_fun(self, value):  
 if not isinstance(value, int):  
 raise ValueError(**'年龄必须是数字!'**)  
 if value < 0 or value > 100:  
 raise ValueError(**'年龄必须是0-100'**)  
 self.\_age = value  
  
 def print\_info(self):  
 print(**'%s: %s'** % (self.\_\_name, self.\_age))  
  
  
p = Person(**'balala'**,20)  
p.\_age = 17  
print(p.\_age) *# 17*print(p.get\_age\_fun()) *# 这里是17 不再是 20,因为此时\_age是全局变量,外部直接影响到类内部的更新值*p.set\_age\_fun(35)  
print(p.get\_age\_fun()) *# 35*print(p.print\_info()) *# balala: 35*

输出结果：

17  
17  
35  
balala: 35

看的出私有和全局的设置

但是，上面的调用方法是不是略显复杂，没有直接用属性这么直接简单。

有没有可以用类似属性这样简单的方式来访问类的变量呢？

我们进入正题：看看@property的妙用之处：

class Person(object):  
 def \_\_init\_\_(self, name, age):  
 self.\_\_name = name  
 self.\_\_age = age  
  
 @property  
 def get\_age\_fun(self):  
 return self.\_\_age  
  
 @get\_age\_fun.setter *# get\_age\_fun是上面声明的方法* def set\_age\_fun(self, value):  
 if not isinstance(value, int):  
 raise ValueError(**'年龄必须是数字!'**)  
 if value < 0 or value > 100:  
 raise ValueError(**'年龄必须是0-100'**)  
 self.\_\_age = value  
  
 def print\_info(self):  
 print(**'%s: %s'** % (self.\_\_name, self.\_\_age))  
  
  
p = Person(**'balala'**,20)  
p.\_\_age = 17  
print(p.\_\_age) *# 17*print(p.get\_age\_fun) *# 20 注意这里不带()  
  
#p.set\_age\_fun(35) 注意不能这样调用赋值了*p.set\_age\_fun = 35 *# 这里set\_age\_fun 就是 声明的函数不带()*print(p.get\_age\_fun) *# 35*print(p.print\_info()) *# balala: 35*

输出结果：

17  
20  
35  
balala: 35

## 继承

面向对象编程，其最大的优势就是代码的高复用，而继承，就是实现代码复用的方法之一。

### 定义

类与类之间的类型和子类型的关系。

通俗点理解。比如，你爸爸妈妈穷极一生攒的家底子，早早晚晚都是你的。钱是他们挣的，但是因为你是他们的儿子，这钱其实也就是你的(我们只是在这举例，就问题讨论例子，不要掺杂所谓孝心所谓懂事bulabulabula……杠精稍稍)。再比如：对于车的性能，要求要有安全气囊，要有ABS防抱死系统，奥迪生产了一批车，这批车就具备安全气囊和ABS防抱死。车是一个大类，奥迪车是一个小类，其继承了车这个类的性能。

### 语法

class 子类名称(父类名称):  
 def \_\_init\_\_(self):  
 super(子类名称,self).父类方法()*#调用父类方法* 父类名称.父类方法()*#调用父类方法*

### 单继承

即继承列表中只有一个父类

class 子类名称(父类名称1):  
 def \_\_init\_\_(self):  
 super(子类名称,self).父类方法()*#调用父类方法* 父类名称.父类方法()*#调用父类方法*

### 多继承

即继承列表中有多个父类，同一子类同时继承多个父类。

class 子类名称(父类名称1，父类名称2，父类名称3,…父类名称n):  
 def \_\_init\_\_(self):  
 super(子类名称,self).父类方法()*#调用父类方法* 父类名称.父类方法()*#调用父类方法*

### isinstance

isinstance(obj,class or tuple):返回这个对象obj是否是某个类的对象，或者某些类中的一个类的对象。

### MRO

Method Resolution Order,即方法解析顺序，是用来处理Python中多继承的二义性问题的算法

现版本Python3中，已将MRO封装在自定义类的\_\_mro\_\_属性中，在一个多级类的继承关系中，无论是单继承还是多继承，可以直接通过查看任何一个类的\_\_mro\_\_属性获取由当前类开始查找的mro算法的查找结果。

## 多态

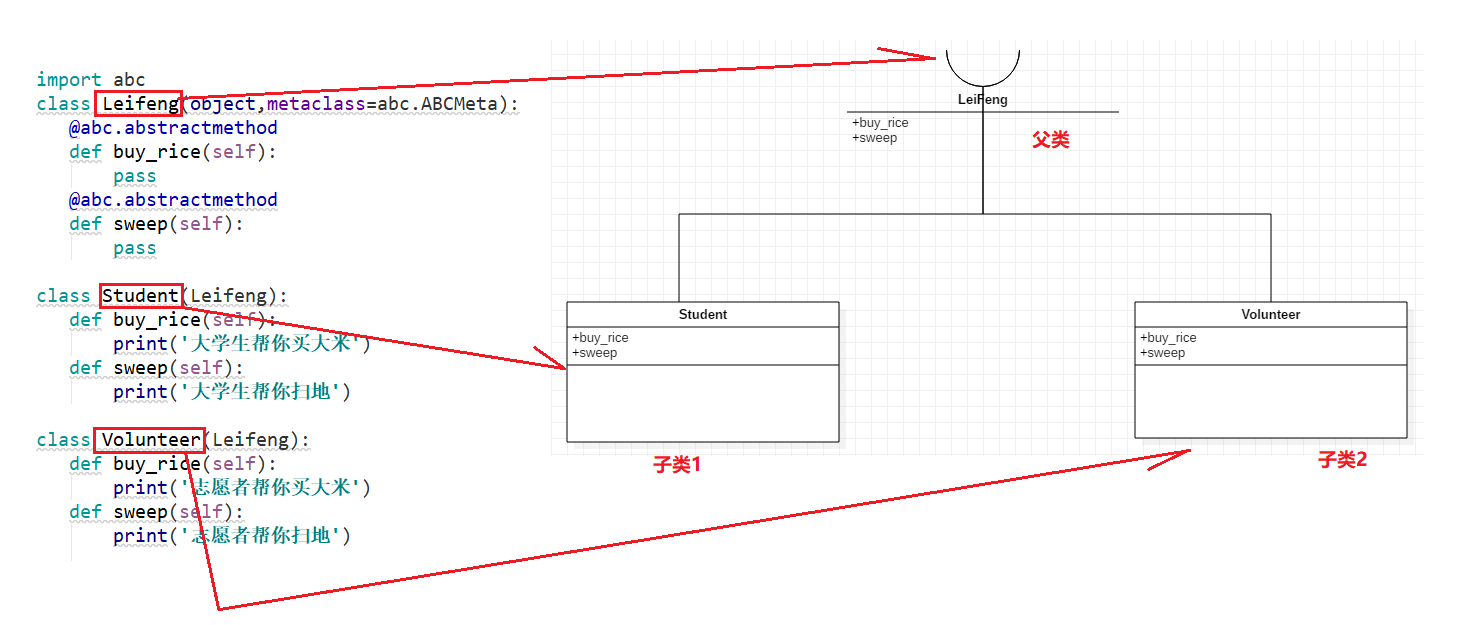
### 定义

一种事物，可以有多种形态。类也是一样。子类继承了父类，父类的方法可以重写，也可以不重写。多态的最基本的实现就是基于继承。

### 举个栗子

代码举例：

import abc  
class Leifeng(object,metaclass=abc.ABCMeta):  
 @abc.abstractmethod  
 def but\_rice(self):  
 pass  
 @abc.abstractmethod  
 def sweep(self):  
 pass  
  
class Student(Leifeng):  
 def buy\_rice(self):  
 print(**'大学生帮你买大米'**)  
 def sweep(self):  
 print(**'大学生帮你扫地'**)  
  
class Volunteer(Leifeng):  
 def buy\_rice(self):  
 print(**'志愿者帮你买大米'**)  
 def sweep(self):  
 print(**'志愿者帮你扫地'**)

图解：

# 专题讲解：Python面向对象发展史——经典类、新式类、MRO算法演变及多继承

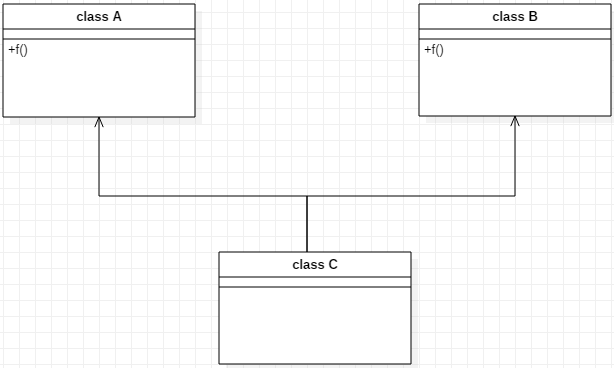
## 相关概念介绍

### 什么是MRO

Method Resolution Order,即方法解析顺序，是用来处理Python中多继承的二义性问题的算法

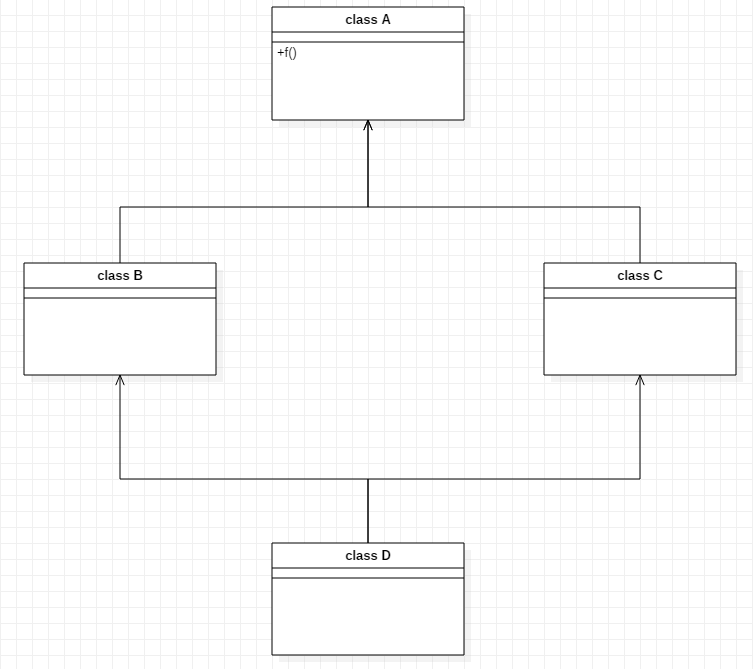
### 二义性

Python因为支持多继承，而多继承的编程语言往往存在二义性问题。二义性问题分为两种：

1. 有两个基类A和B，A和B中都定义了方法f(),C类继承了A类和B类，那么调用C类的f()时出现不确定情况。详细看图解：

这样的多继承称为“魔鬼三角继承”

1. 有一个基类A，定义了方法f()，B类和C类继承了A类（的f()方法），D类继承了B和C类，那么出现一个问题，D不知道应该继承B的f()方法还是C的f()方法。具体看图解：



这样的多继承称为“恐怖菱形继承”

### 深度遍历优先

指代一种搜索算法，其算法核心是尽可能先对纵深方向进行搜索，当一条分支搜索到叶子结点后，终止，从左往右找到下一条分支，继续开始纵深方向的搜索，直至搜索到最后一条分支的叶子节点，停止搜索。

### 广度遍历优先

指代一种搜索算法，其算法核心是针对树形结构的每一层，从根结点出发，按照一层一层的顺序，每层从左至右的顺序搜索，当搜索到某一层最后一个非叶子结点，结束该层搜索，进入下一层，继续搜索，直至搜索到最后一个叶子节点，停止搜索。

### 拓扑排序

#### 有向无环图

如果一个有向图无法从某个顶点出发经过若干条边回到该点，则这个图是一个有向无环图（DAG图）

#### 拓扑序列

对一个有向无环图(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边(u,v)∈E(G)，且u在线性序列中出现在v之前。通常，这样的线性序列称为满足拓扑排序(TopologicalOrder)的序列，简称拓扑序列。

#### 拓扑排序算法

循环执行以下两步，直到不存在入度为0的顶点为止

1.选择一个入度为0的顶点并输出之；

2.从网中删除此顶点及所有出边。

### Python版本介绍

如果按照Python面向对象中多继承的MRO算法演变史为依据，可大致将Python的版本演变划分为四个阶段：

1.Python2.2之前

2.Python2.2

3.Python2.2-Python2.7

### 经典类

一种没有继承的类，所有的类型都是type类型，如果经典类作为父类，子类调用父类构造函数会报错。

### 新式类

每一个类都继承自一个基类，默认继承自object，子类可以调用基类的构造函数，所有类都有一个公共的祖先类object。

### 线性化过程中的单调性原则

遍历时应该先从唯一基类开始查找，即：子类不能改变基类的方法搜索顺序。

## C3算法演变史

### 问题由来

问题：Python中调用父类构造方法的两种方式

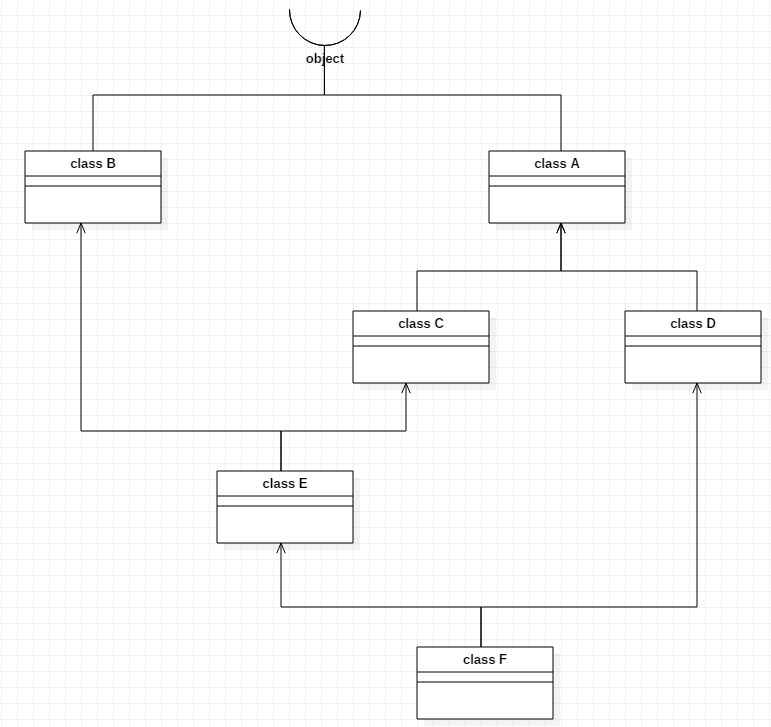
示例代码：单继承

class A(object):  
 def \_\_init\_\_(self):  
 self.name = **'A:name'** print(**'A:\_\_init\_\_'**)  
 def fun(self):  
 print(**"A:fun"**)  
class B(A):  
 def \_\_init\_\_(self):  
 print(**"B:\_\_init\_\_"**)  
 A.\_\_init\_\_(self) *# 使用类名直接调用* super(B, self).\_\_init\_\_() *# 使用super关键字调用* def fun(self):  
 print(**"B:fun"**)  
 A.fun(self)  
 super(B, self).fun()  
 print(self.name)  
b = B()

总结：两种调用父类方法的方式其实都可行，区别在于，直接使用父类名称进行调用，父类方法只会被调用一次，使用super函数调用就是当前子类的基类的方法，可能调用不止一次。

示例代码：多继承

class A(object):  
 def \_\_init\_\_(self):  
 print(**'进入a'**)  
 print(**'离开a'**)  
  
class B(object):  
 def \_\_init\_\_(self):  
 print(**'进入b'**)  
 print(**'离开b'**)  
class C(A):  
 def \_\_init\_\_(self):  
 print(**'进入c'**)  
 super(C,self).\_\_init\_\_()  
 print(**'离开c'**)  
class D(A):  
 def \_\_init\_\_(self):  
 print(**'进入d'**)  
 super(D,self).\_\_init\_\_()  
 print(**'离开d'**)  
class E(B,C):  
 def \_\_init\_\_(self):  
 print(**'进入e'**)  
 B.\_\_init\_\_(self)  
 C.\_\_init\_\_(self)  
 print(**'离开e'**)  
  
class F(E,D):  
 def \_\_init\_\_(self):  
 print(**'进入f'**)  
 E.\_\_init\_\_(self)  
 D.\_\_init\_\_(self)  
 print(**'离开f'**)  
  
f = F()

 继承关系如下图来表示：

运行结果如下：

我们的本意是希望调用构造函数的时候，对于基类的构造方法也进行调用，但是实际结果是：

1. A和D的构造函数被调用了2次；
2. 当调用super(C, self).\_\_init\_\_()的时候，竟然进入D的构造函数

也就是说，D的构造函数被调用了两次，一次是F调用的，一次是C调用的，可是从继承关系上看，C的基类应该是A才对啊……

什么鬼，不明白，懵圈了……

Ok,一起来看下Python中的C3算法

### Python2.2之前的经典类

在python 2.2之前，python中使用经典类（classicclass），经典类是一种没有继承的类，所有类型都是type类型，如果经典类作为父类，子类调用父类构造函数会报错。当时用作MRO的算法是DFS（深度优先），下面的例子是当时使用DFS算法的示例（向右是基类方向）：

1.正常的继承方式:

A-->B-->D

A-->C-->E

DFS的遍历顺序为：A-->B-->D-->C-->E

这种情况下，不会产生问题

2.'恐怖'菱形继承方式

A-->B-->D

A-->C-->D  
DFS遍历顺序为：A-->B-->D-->C

此时，如果公共父类D中也定义了f()，C中重写了方法f()，那么C中的f()方法永远也访问不到，因为按照遍历的顺序始终先发现D中的f()方法，导致子类无法重写基类方法。

### Python2.2版本中出现的新式类

在python2.2开始，为了使类的内置类型更加统一，引入了新式类（new-style class），新式类每个类都继承自一个基类，默认继承自object，子类可以调用基类的构造函数。由于所有类都有一个公共的祖先类object，所以新式类不能使用DFS作为MRO。在当时有两种MRO并存：

1.如果是经典类，MRO使用DFS

2.如果是新式类，MRO使用BFS

针对新式类的BFS示例如下（向右是基类方向）：

1.正常的继承方式:

A-->B-->D

A-->C-->E

BFS的遍历顺序为：A-->B-->C-->D-->E

D是B的唯一基类，但是遍历时因为是广度遍历优先，先遍历节点C，这种情况下应该先从唯一基类进行查找，这个原则称为单调性。而在这里，明显，违背了单调性原则

2.'恐怖'菱形继承方式

A-->B-->D

A-->C-->D

BFS遍历顺序为：A-->B-->C-->D

BFS解决了前面提到的子类无法重写基类方法的问题

### Python2.3-Python2.7，经典类和新式类并存，C3算法产生

由于DFS和BFS针对经典类和新式类都有缺陷，从python2.3开始，引入了C3算法。针对前面两个例子，C3算法的遍历顺序如下：

1.正常的继承方式:

A-->B-->D

A-->C-->E

C3的遍历顺序为：A-->B-->D-->C-->E

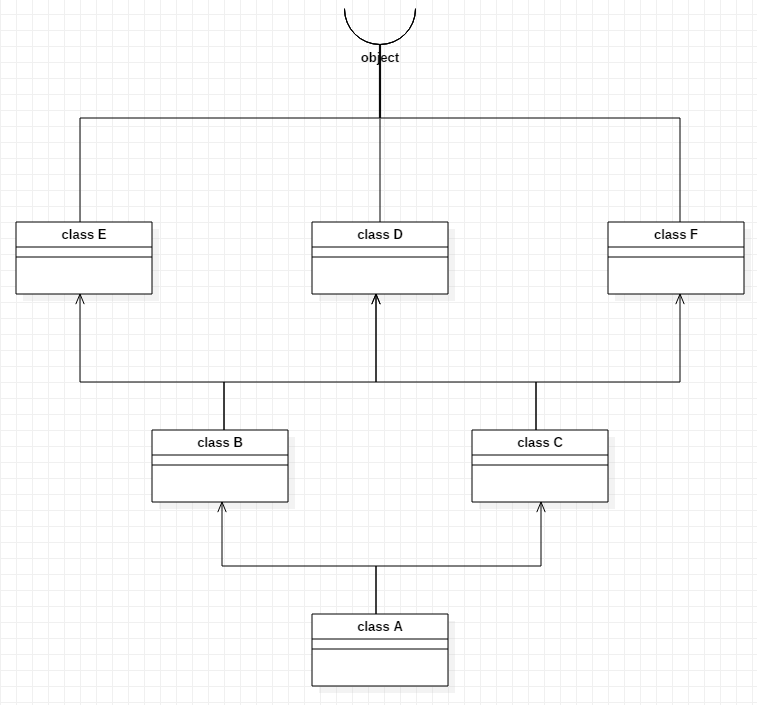
2.'恐怖'菱形继承方式

A-->B-->D

A-->C-->D

C3遍历顺序为：A-->B-->C-->D

看起来是DFS和BFS的综合，但是并非如此，下面的例子说明了C3算法的具体实现，如图：



首先找入度为0的点，只有A，把A取出，把A相关的边去掉，再找下一个入度为0的点，B和C满足条件，从左侧开始取，取出B，这时顺序是AB，然后去掉B相关的边，这时候入度为0的点有E和C，依然取左边的E，这时候顺序为ABE，接着去掉E相关的边，这时只有一个点入度为0，那就是C，取C，顺序为ABEC。去掉C的边得到两个入度为0的点D和F，取出D，顺序为ABECD，然后去掉D相关的边，那么下一个入度为0的就是F，然后是object。所以最后的排序就为ABECDFobject。

# 专题讲解：Python中的元类

## 类和对象的基本概念

### 对象

对象是人们要进行研究的任何事物，小到一颗沙粒，大到整个银河系，所有的一切，皆对象。对象不仅能够表示具体的事物，还能表示抽象的规划、计划或事件。

对象具有状态，一个对象用数据值来描述它的状态

对象还有行为，即操作，用于改变对象的状态，对象及其操作就是对象的行为

对象实现了数据和操作的结合，使数据和操作封装于对象这个统一体中

对象，是类的具体实例，即归属于某个类别的“个体”

### 类

具有相同特性（[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0" \t "_blank)）和行为（功能）的[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)就是类。因此，[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1)的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)是类，类的具体化就是对象，也可以说类的实例是对象，类实际上就是一种[数据类型](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B)。

类具有属性，它是[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)的状态的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)，用[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)来描述类的属性。

类具有操作，它是[对象](https://baike.baidu.com/item/%E5%AF%B9%E8%B1%A1" \t "_blank)的行为的[抽象](https://baike.baidu.com/item/%E6%8A%BD%E8%B1%A1)，用操作名和实现该操作的方法来描述。

类具有结构。在客观世界中有若干类，这些类之间有一定的结构关系。通常有两种主要的结构关系，即一般--具体结构关系，整体--部分结构关系。

①一般--具体结构称为分类结构，也可以说是“或”关系，或者是“is a”关系。

②整体--部分结构称为组装结构，它们之间的关系是一种“与”关系，或者是“has a”关系。

通常，类与类之间的行为不同，对象与对象之间的数据不同。

类是创建对象的模板。在面向对象的思想中，所有的对象（实例）都是由类创建的。

### 类的特殊特性

类本身，也是一个对象。其实在你定义一个类的时候，就会在内存中创建一个名为类名的对象。

### 对象的特征

1. 用变量进行存储
2. 复制对象
3. 添加属性
4. 当作参数传递给函数

经过如上概念，对类，应该有了一个最最基本的了解。但是，问题继续来了：对象不可能凭空产生，它一定是由某些事物生成的。既然类也是对象，那不例外，类，同样遵循这个规律。

当你在使用class关键字的时候，Python就会自动创建这个对象。但是，虽然为自动创建，同样可以手动来进行模拟实现。

## type

type，我们常见，如果需要返回对象的类型，就可以使用type函数。

先看如下测试结果：

anwc@anweichao-pc:~$ ipython3  
Python 3.6.5 (default, Apr 1 2018, 05:46:30)  
Type **'copyright'**, **'credits'** or **'license'** for more information  
IPython 6.5.0 -- An enhanced Interactive Python. Type **'?'** for help.  
  
In [1]: type(1)  
Out[1]: int  
  
In [2]: type(int)  
Out[2]: type  
  
In [3]: type(**'1'**)  
Out[3]: str  
  
In [4]: type(str)  
Out[4]: type  
  
In [5]: type(1.11)  
Out[5]: float  
  
In [6]: type(float)  
Out[6]: type  
  
In [7]: type(True)  
Out[7]: bool  
  
In [8]: type(bool)  
Out[8]: type  
  
In [9]: type([1,2,3])  
Out[9]: list  
  
In [10]: type(list)  
Out[10]: type  
  
In [11]: type((1,2,3))  
Out[11]: tuple  
  
In [12]: type(tuple)  
Out[12]: type  
  
In [13]: type({**'1'**:2})  
Out[13]: dict  
  
In [14]: type(dict)  
Out[14]: type

我们查看整型int的类型发现其实type，查看字符型str的类型发现也是type，查看列表list的类型发现也是type……好像这些我们见过的基本的数据类型其类型通通都是type……而第一章我们说到，抽象数据类型其实就是类，那我们日常使用的这些数据对象不就是基于这些数据类型实例化得到的么(小整数对象池除外)，那说白了这些数据类型其实不也是个类么……有点懵，不着急，我们先往下看。

以上我们看到了type的最常用的方式，没错，只传递一个参数，作用很简单，返回对象的类型。而上面我们看到了，int，str，float，list，tuple等等的type返回值都是type！聪明的兄弟已经恍然大悟，没错，这个被返回的type，就是Python的元类！

1. type函数传递一个参数的用法：当只传递一个参数的时候，其功能单一，仅仅只是返回对象的类型；
2. type函数传递三个参数的时候：此时，用来创建类对象！

语法：

类名 = type(String,tuple,dict)

参数解释：

String：字符型参数，类的名称

Tuple：元组，继承关系

Dict：字典，定义属性及方法

继续，我们在终端下看看如何手动创建类：

anwc@anweichao-pc:~$ ipython3  
Python 3.6.5 (default, Apr 1 2018, 05:46:30)  
Type **'copyright'**, **'credits'** or **'license'** for more information  
IPython 6.5.0 -- An enhanced Interactive Python. Type **'?'** for help.  
  
In [1]: Myancestor = type(**'Myancestor'**,(),{})  
  
In [2]: Myancestor  
Out[2]: \_\_main\_\_.Myancestor  
  
In [3]: print(Myancestor)  
<class **'\_\_main\_\_.Myancestor'**>  
  
此时你手动创建了一个类Myancestor，我们来看看它的元类是谁：  
  
In [4]: type(Myancestor)  
Out[4]: type

看到了吗？type函数手动创建的类，其元类仍是type！  
继续玩，我们手动创建类的时候还能给其设定继承关系和成员属性及方法：  
In [5]: MyOld = type(**'MyOld'**,(Myancestor,),{**'name'**:**'子代'**})  
  
In [6]: MyOld.\_\_mro\_\_  
Out[6]: (\_\_main\_\_.MyOld, \_\_main\_\_.Myancestor, object)  
  
添加方法？很简单，先定义好一个函数然后传进字典中即可：  
In [7]: def Myfun():  
 ...: pass  
 ...:  
 ...:  
  
In [8]:  
  
In [8]: MyBaba = type(**'MyBaba'**,(MyOld,),{**'name'**:**'子代1'**,**'Myfun'**:Myfun})

以上我们创建的类，其元类都是type，因为：你是使用type来创建的类，其元类必然是type！

## 自定义元类

一个类没有声明自己的元类，则默认其元类是type，除了使用内置元类type，我们也可以通过继承type来自定义元类，然后使用metaclass关键字参数为一个类自定义指定元类。

代码举例：

*#自定义元类*class Meta\_Type(type):  
 def \_\_call\_\_(self,\*args,\*\*kwargs):  
  
 *#self是Meta\_Type的实例，也就是Foo  
 #因为每个类都默认继承object，所以可以调用\_\_new\_\_(Foo)方法，来实例化一个对象f1 == object.\_\_new\_\_(Foo) ==> f1* obj = object.\_\_new\_\_(self)  
  
 *#调用Foo的构造函数实例化对象 == Foo.\_\_init\_\_(f1,\*args,\*\*kwargs)* self.\_\_init\_\_(obj,\*args,\*\*kwargs)  
  
 *#最后要返回f1这个对象* return obj  
  
*#metaclass = Meta\_Type ====> 会触发Foo = Meta\_Type(Foo,'Foo',(),{}) ===>会触发Meta\_Type的 \_\_init\_\_*class Foo(metaclass=Meta\_Type):  
 def \_\_init\_\_(self,name):  
 self.name = name  
  
*#当Foo()时，表示执行了Foo，则会调用元类的\_\_call\_\_,\_\_call\_\_中会生成对象f1，并且调用Foo下的\_\_init\_\_方法，最终完成实例化*f1 = Foo(**'Menawey'**)  
  
*#>>>我是元类*print(f1.\_\_dict\_\_)

## 说明

元类，是一种99%的人都不需要关心的深度魔法。当你好奇你是否需要懂它，了解它时，恰恰说明你并不需要它，而明确需要它的人不需要找任何原因解释为什么需要。元类，是你注定得不到搞不明白的东西！很简单，元类，大量应用于框架，函数库的开发中。

# 专题讲解：UML统一建模语言之——类图(组件及其实现)

## 定义

类图(Class diagram)是显示了模型的静态结构，特别是模型中存在的类、类的内部结构以及它们与其他类的关系等。类图不显示暂时性的信息。类图是面向对象建模的主要组成部分。它既用于应用程序的系统分类的一般概念建模，也用于详细建模，将模型转换成编程代码。类图也可用于数据建模。

## 作用

类图的作用：

（1）：在[软件工程](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%B7%A5%E7%A8%8B/25279" \t "_blank)中，类图是一种静态的结构图，描述了系统的类的集合，类的属性和类之间的关系，可以简化了人们对系统的理解；  
　　（2）：类图是系统分析和设计阶段的重要产物，是系统编码和测试的重要模型。

## 使用场景

主要在三种场合中使用类图。

（1）为系统词汇建模型：为系统的词汇建模实际上是从词汇表中发现类，发现它的责任。

（2）模型化简单的协作：协作是指一些类、接口和其他的元素一起工作提供一些合作的行为，这些行为不是简单地将元素相加能得到的。例如：当你为一个分布式的系统中的事务处理过程建模型时，你不可能只通过一个类来明白事务是怎样进行的，事实上这个过程的执行涉及到一系列的类的协同工作。使用类图来可视化这些类和他们的关系。

（3）模型化一个逻辑数据库模式：想象模式是概念上设计数据库的蓝图。在很多领域，你将想保存持久性数据到[关系数据库](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%95%B0%E6%8D%AE%E5%BA%93)或[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1)的数据库。你可以用类图为这些数据库模式建立模型。

## 类图组件介绍

### 类

类（Class）一般包含3个组成部分。第一个是类名；第二个是属性（attributes）；第三个是该类提供的方法（ 类的性质可以放在第四部分；如果类中含有内部类，则会出现第五个组成部分）。类名部分是不能省略的，其他组成部分可以省略。类名书写规范：正体字说明类是可被实例化的，斜体字说明类为抽象类。

属性和方法书写规范：修饰符 [描述信息] 属性、方法名称 [参数] [：返回类型|类型]。

属性和方法之前可附加的可见性修饰符：

加号（+）表示public；减号（-）表示private；井号(#)表示protected；省略这些修饰符表示具有package（包）级别的可见性。

如果属性或方法具有下划线，则说明它是静态的。描述信息使用 << 开头，使用 >> 结尾。类的性质是由一个属性、一个[赋值](https://baike.baidu.com/item/%E8%B5%8B%E5%80%BC)方法和一个取值方法组成。书写方式和方法类似。

### 包

包（Package）是一种常规用途的组合机制。UML中的一个包直接对应于[Java](https://baike.baidu.com/item/Java/85979" \t "_blank)中的一个包。在Java中，一个包可能含有其他包、类或者同时含有这两者。进行建模时，通常使用逻辑性的包，用于对模型进行组织；使用物理性的包，用于转换成系统中的Java包。每个包的名称对这个包进行了惟一性的标识。

### 接口

接口（Interface）是一系列操作的集合，它指定了一个类所提供的服务。它直接对应于Java中的一个接口类型。Python的开发中，抽象类就是接口。

### 关系

常见的关系有：[继承](https://baike.baidu.com/item/%E7%BB%A7%E6%89%BF/2261239" \t "_blank)（Inheritance），关联关系（Association），聚合关系（Aggregation），复合关系（Composition），依赖关系（Dependency）,实现关系(Realization/Implementation)。

其中，聚合关系（Aggregation），复合关系（Composition）属于关联关系（Association）。

一般关系表现为继承或实现关系(is a)，也称作泛化关系。

关联关系表现为变量(has a )。

依赖关系表现为函数中的参数(use a)。

#### 一般关系

表示为类与类之间的继承关系，接口与接口之间的继承，类对接口的实现关系。

表示方法： 用一个空心箭头+实线，箭头指向父类。或空心箭头+虚线，如果父类是接口(抽象类)。

#### 关联关系

类与类之间的联接，它使一个类知道另一个类的属性和方法。即：A类中有B类的成员，但是A类与B类是平级关系。

表示方法：用 实线+箭头， 箭头指向被使用的类。

#### 聚合关系

是关联关系的一种，是强的关联关系。聚合关系是整体和个体的关系。关联关系的两个类处于同一层次上，而聚合关系两个类处于不同的层次，一个是整体，一个是部分。即：A类中有B类的成员，A类是整体，B类是部分，B类脱离A类仍然可以单独存在。

表示方法：空心菱形+实线+箭头，箭头指向个体。

#### 组合关系

是关联关系的一种，是比聚合关系强的关系。它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期，组合关系不能共享。即：A类中有B类的成员，A类是整体，B类是部分，且B类脱离A类便不复存在。

表示方法：实心菱形+实线+箭头，

#### 依赖关系

是类与类之间的连接，表示一个类依赖于另一个类的定义。例如如果A依赖于B，则B体现为[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F" \t "_blank)，方法的参数、或静态方法的调用。

表示方法：虚线+箭头 箭头指向被依赖的一方，也就是指向局部变量。

上文中的部分图示即采用类图进行的表示。

# 专题讲解：面向对象底层挖掘——类、实例的创建及初始化

## 类的生命周期

顾名思义，生命周期描述的是一个物体从诞生到消亡的过程。那么类的生命周期，描述的就是一个类及其实例从无到有的一个过程。

当我们创建一个对象时，会自动分配一个内存地址给这个对象使用，这就是生命周期的开始；当我们不使用这个对象时，就会回收这个对象，并释放对象所占用的内存地址，这就是对象的消亡

如何监控对象的生命周期？主要通过以下三个方法监控：

1.\_\_new\_\_方法：创建实例

2.\_\_init\_\_方法：初始化实例

3.\_\_del\_\_方法：回收实例

举个栗子：

class Myobj(object):  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 print(**'我被创建啦！'**)  
 return object.\_\_new\_\_(cls)  
  
 def \_\_init\_\_(self):  
 print(**'我被初始化啦！'**)  
  
 def \_\_del\_\_(self):  
 print(**'我被回收啦！'**)  
  
obj = Myobj()  
  
*# 运行结果：  
# D:\project\venv\Scripts\python.exe E:/老安课程串讲资料/老安串讲资料12——对象的生命周期.py  
# 我被创建啦！  
# 我被初始化啦！  
# 我被回收啦！  
#  
# Process finished with exit code 0*

根据以上结果我们可以发现，无论我们是先定义new方法还是后定义new方法，最终执行都是先执行new方法。为什么呢？接着往下看。

## 走进new方法，迎接init方法

\_\_new\_\_方法是在新式类中新出现的方法，它作用在构造方法(初始化方法)初始化实例之前。

可以这么理解：在Python中存在于类里面的构造方法(初始化方法)\_\_init\_\_负责实例的初始化，而在\_\_init\_\_方法调用之前，\_\_new\_\_方法决定是否要使用这个\_\_init\_\_方法进行初始化！因为\_\_new\_\_可以调用其他类的构造方法！或者直接返回别的对象作为本类的实例！

\_\_new\_\_方法的特性：

1.\_\_new\_\_方法是在类准备将自身实例化时调用

2.\_\_new\_\_方法始终都是类的静态方法，即使没有被加上静态方法装饰器

看如下代码：

class MyObj(object):  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 ...  
 return object.\_\_new\_\_(cls)  
   
 def \_\_init\_\_(self):  
 pass

上述大段可以这样理解：new方法，其实才是真正的构造方法，即创建实例。只不过，对于new方法来说，他可以自己决定返回哪个类的对象，即：return 返回值后的object.\_\_new\_\_(cls)是默认值，即默认返回当前类的实例。

new方法返回实例之后，init方法的self参数负责接收当前实例，进而继续对实例进行初始化。

所以，明确两个方法的职责：一定是new先创建实例，再是init通过self参数接收实例进而初始化实例。

所以，针对以上，我们可以得出结论：如果new方法没有返回当前类的实例，则init方法就不会调用，因为其self参数并没有接收到实例这个参数值。

而new方法返回当前类的实例的方式，就是通过调用object的new方法将当前类作为参数传递，得到其返回值后返回其返回值，最终得到当前类的实例。

所以，可以明确，所有自定义类的实例全部都是通过object来创建的。这就是为什么Python3版本后所有的类都是新式类，而新式类都是默认继承自object了。

# 专题讲解：面向对象六大设计原则(面向对象程序设计：OOD)

## 封闭开放原则(目标，总的指导思想)

### 定义

软件实体(模块，类，函数等)应该可以扩展，但是不可以修改，也就是说软件要对扩展开放，对修改关闭。

注意：对修改关闭不是说软件设计不能做修改，只是尽量不要做不必要的修改。怎么才能做到呢？那就是有相应的扩展性。

### 理解

其实，软件有相应的扩展性是好处，但是不能说每个地方都有扩展。反而造成了代码的臃肿。所以这里的扩展与修改关闭是有限制的。

开闭原则，可以说是其他五大原则的实现，也是面向对象设计的终极目标。我们也可以说成开闭原则是其他原则的核心。

通俗理解：比如我们平常喝水用的一次性纸杯。平常人只是用来装水。喝完水就扔了。这就是这个纸杯的生命周期。纸杯这一生只完成了它的一个功能：装水。纸杯此时就很封闭了，没有什么扩展性。

此时，我看到身边有一支花苗，我想要拿回家种。但是没有容器呀？ 啊？旁边不是有一个纸杯吗，可以用此纸杯来种这朵花苗。

纸杯有了它的另外一个扩展性，就是种花苗。

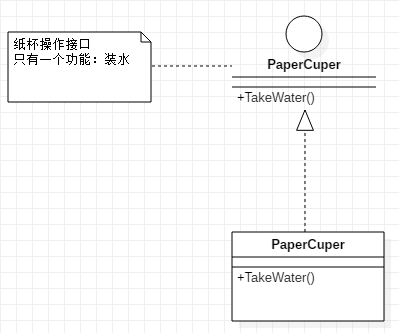
纸杯不仅有装水、种花苗的用途，以后还可以有装小垃圾、冲茶、回收等功能。对于以后这些功能，我们要想到他们的扩张性。

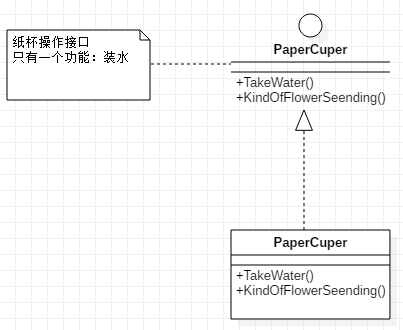
在纸杯只有一个装水的功能的时候，我们只写一个纸杯功能类，说纸杯能装水。但是以后有扩展呢？这一方面我们要预先判断。预先判断它以后可能会根据需求的变动而扩展。对于纸杯本来的装水功能，不能说不能修改，此功能只能在此函数、类中修改。这就是开闭原则的核心。

所以，纸杯在开闭原则所体现的是：尽量少修改，未来可能扩展的模块、类做好预算的判断。如果要修改，只能在此函数此类修改，不能牵涉到其他地方。

下面，我们用UML类图来直观地说明一下纸杯的设计吧。

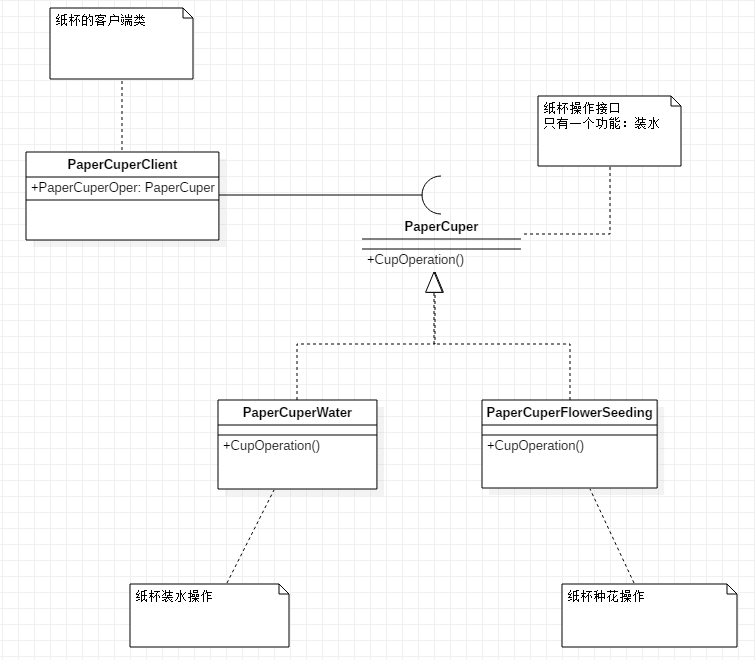
当纸杯只有一个功能，装水时。有一个纸杯操作接口，有一个纸杯操作实现类。



当我们要添加一个功能：种花苗时，我们不也是加一个方法么？如下：

当添加n个方法时，不也就是在纸杯的操作接口上面添加n个方法吗？

仔细思考，此时，已经背离了我们的开闭原则，因为每添加一个方法，都要在操作类上面做修改。所以，我们按照开闭原则，重做了以下的合理设计：



从上面可以看出，我们把纸杯的操作类，统一写成一个接口，每个不同的操作继承此接口来完成各自的操作。我们还开到多了一个类，叫客户端类，其实此类也不难理解，也就是要最终操作纸杯的类，即此类依赖各个操作类。

### 封闭开放原则设计实例：计算器功能设计

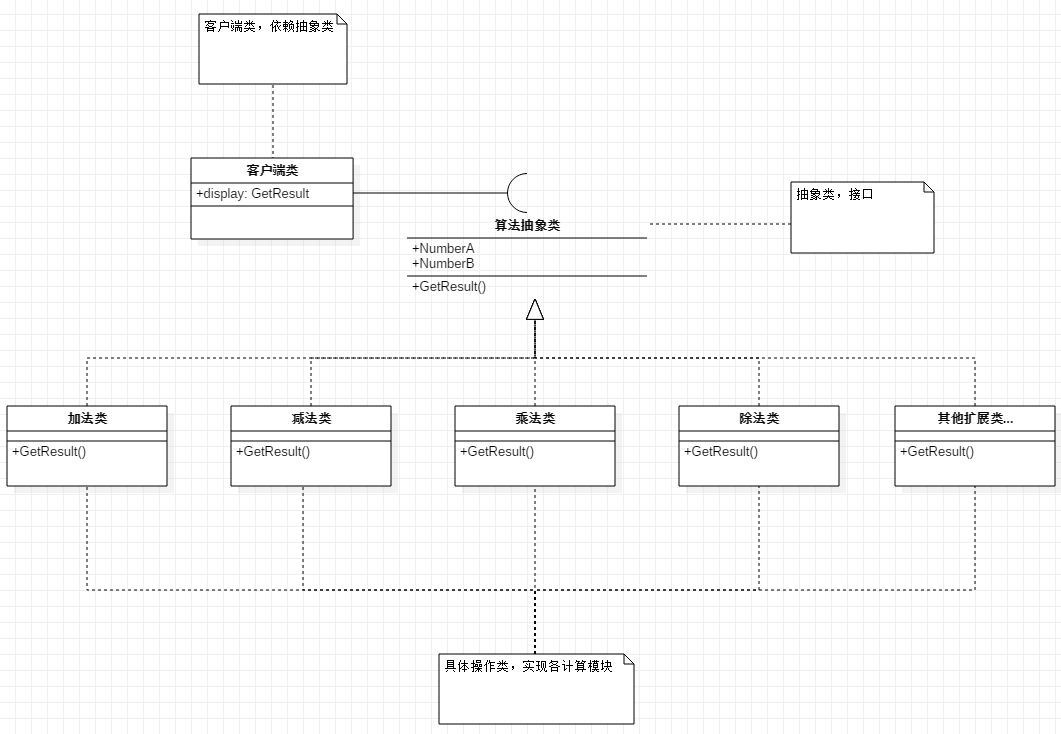
需求：实现基本加减乘除，后续可能继续添加取余，求幂，开根号等功能。

设计思路：假设我们现在只有一个需求：实现加法，所以我们可以直接通过一个类，在类中实现一个加法的方法即可。但，现在要求还要实现减法操作。如果直接在当前类中添加减法的方法，就会直接违背开闭原则。

所以，为此我们需要有一个预判性，即：考虑到功能的扩展。如果是这样，我们就不能使用如此简单的方法来实现。这个预判行直接导致我们项目以后的扩展性，如果需求发送变更，程序修改起来要尽可能简单高效且不对其他模块造成影响。

所以，我们要做一个算法法则的抽象类，加减乘除等等功能模块都继承此操作接口(抽象类)，再实现一个算法客户端类，该类依赖算法抽象类(即依赖了所有具体算法类)，进而操作各个算法模块。

UML类图实现如下：



## 类的单一职责(一个类的定义)

### 定义

单一职责原则是指应该有且只有一个原因引起类的变更。

### 理解

通俗点说，一个类应该只做一类事情，一个类应该只负责一个功能。

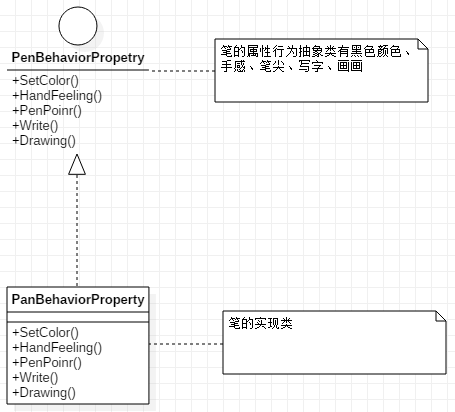
其实按照初学者的思路来理解，单一职责看上去很简单，不就是一个类一个方法么？的确，这属于终极目标。但此终极目标并不现实。因为，一个类一个方法的确是高内聚低耦合的最佳实现，但是显然这也使得代码很臃肿，维护起来更加不便，在中型甚至更大型的项目中尤为明显。如果你的项目足够简单，类足够少，那你可以这样做。所以，这里需要强调的一点是，单一职责原则是一个类处理一类事情，也只有一个类事情能影响到这个类，而并不是一个类处理一个方法。

举个栗子。比如我面前现在有只晨光牌的水性笔。笔有属性，有行为，且其行为是被动的。属性有：黑色，手感好，笔尖0.5。被动的行为有：写字，画画，扎人……

按照一般的做法，我们会把笔的属性和行为都放在一个类来做。单一职责现在就起作用了，我们要分开成属性和行为。

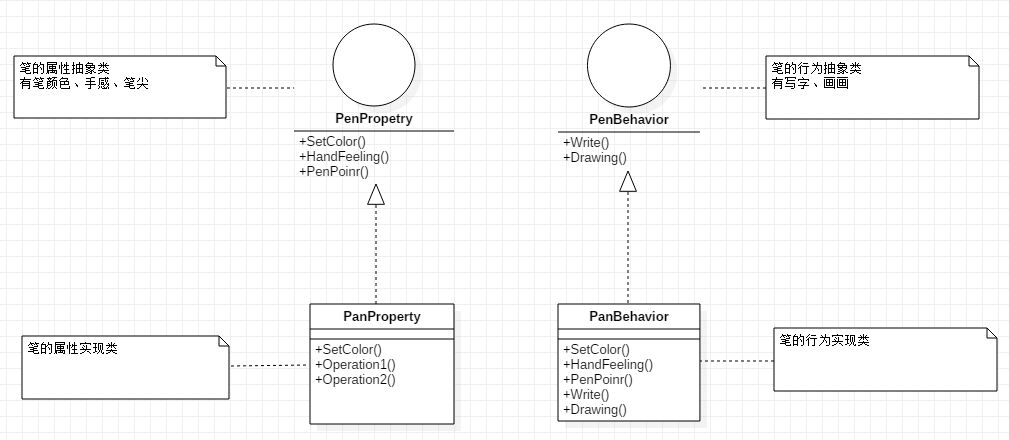
下面我们使用UML图来进行理解。

先看一般的设计：



一般一个接口、一个类来处理笔的各种事情，上面的图充分解析了这种说法，也是平常的简单设计思路。

现在，我们按照单一职责原则，来设计笔的属性，行为隔离开，如下：



## 依赖倒置原则(依赖抽象)

### 定义

1. 高层次的模块不应该依赖于低层次的模块，应该依赖于抽象。
2. 抽象不应该依赖具体实现，具体实现应该依赖抽象

以上两点是依赖倒置原则的概念，也是核心。主要是说模块之间不要依赖具体实现，依赖接口或抽象。其实依赖倒置原则的核心思想是面向接口编程。

### 理解

还是看一个非常简单的栗子吧。

比如说，学生看语文书。那其实就两类呗，一个学生类，一个语文书类。先思考：学生类和语文书类是什么关系？依赖？关联？如果是关联，是组合？还是聚合？

哈哈没那么复杂。关联关系指的是A类中有B类的成员，而不是使用。而依赖是指A类中用到了B类的成员。所以在这里只是简单的依赖关系~

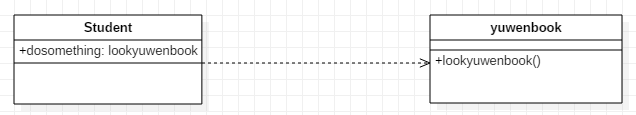
先看代码：

#定义学生类

class Student(object):  
 def \_\_init\_\_(self):  
 pass  
#定义操作：看书。看啥书？语文书。所以接下来语文类要有个操作，负责实现看语文书，所以这里方法内部使用语文书实例的方法，所以是依赖关系。  
 def dosometing(self,book):  
 book.lookYuWenBook()  
  
#定义语文类

class YuWenBook(object):  
 def \_\_init\_\_(self):  
 pass  
  
 def lookYuWenBook(self):  
 print(**'我在看语文书'**)  
  
#测试用例

def test():  
 Student().dosometing(YuWenBook())  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 test()

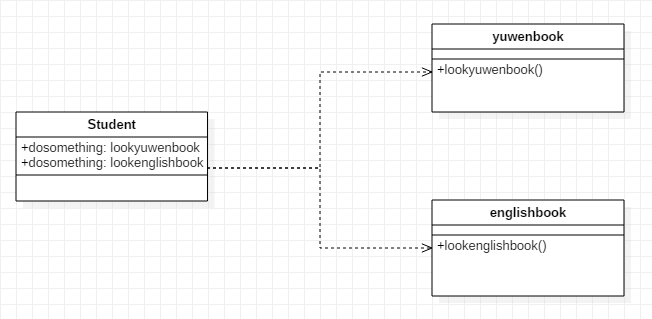
 对应的UML类图如下：

以上程序都没有错，创建一个语文对象，然后将其传入学生对象的方法内，调用语文对象的方法，实现了看书的功能。

但是，这是不好扩展的，背离了面向对象的高内聚低耦合的思想。

因为，如果，我们要添加学生看英语书呢？怎么添加？首先，要在学生类中添加第二个dosomething方法，接收英语书的实例，然后再写个英语书类，其中实现一个方法看英语书，再把英语书类的实例当作参数传入学生类的第二个dosomething方法，调用其方法实现看英语书……妈耶好乱好复杂啊！看代码吧：

class Student(object):  
 def \_\_init\_\_(self):  
 pass  
  
 def dosometing(self,yuwenbook):  
 yuwenbook.lookYuWenBook()  
  
 def dosomething(self,englishbook):  
 englishbook.lookenglishbook()  
  
class YuWenBook(object):  
 def \_\_init\_\_(self):  
 pass  
  
 def lookYuWenBook(self):  
 print(**'我在看语文书'**)  
  
class EnglishBook(object):  
 def \_\_init\_\_(self):  
 pass  
  
 def lookenglishbook(self):  
 print(**'我在看英语书'**)  
  
def test():  
 Student().dosometing(YuWenBook())  
 Student().dosomething((EnglishBook()))  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 test()

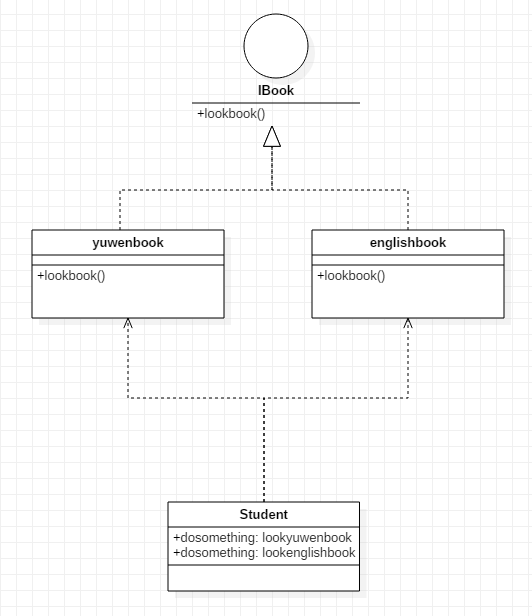
 对应的UML类图改变如下：

显然，这样的设计是不合格的，因为已经修改了Student的代码。

怎么进行重新设计呢？先思考开闭原则：对修改关闭，多扩展开放。所以，我们尽量不要修改Student类中的代码。看如下修改后的代码：

from abc import ABCMeta,abstractmethod  
class IBook(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def lookbook(self):  
 pass  
  
class Student(object):  
 def \_\_init\_\_(self):  
 pass  
  
 def dosomething(self,book):  
 book.lookbook()  
  
class YuWenBook(IBook):  
  
 def lookbook(self):  
 print(**'我在看语文书'**)  
  
class EnglishBook(IBook):  
  
 def lookbook(self):  
 print(**'我在看英语书'**)  
  
def test():  
 Student().dosomething(YuWenBook())  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 test()

我们创建了个抽象类IBook，让语文类和英语类各自继承它并重写抽象方法。这样的话，如果我们还想新增个数学类，历史类，物理类等等，我们只需要新增对应的功能类，让其继承抽象类即可实现功能的扩展。

 对应的UML类图如下：

## 接口隔离原则

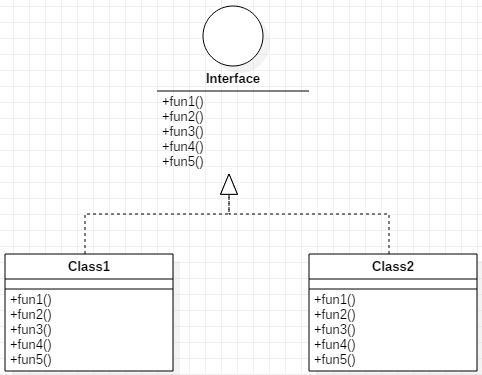
### 定义

客户端不应该依赖它不需要的接口。一个类对另一个类的依赖应该建立在最小的接口上。

### 理解

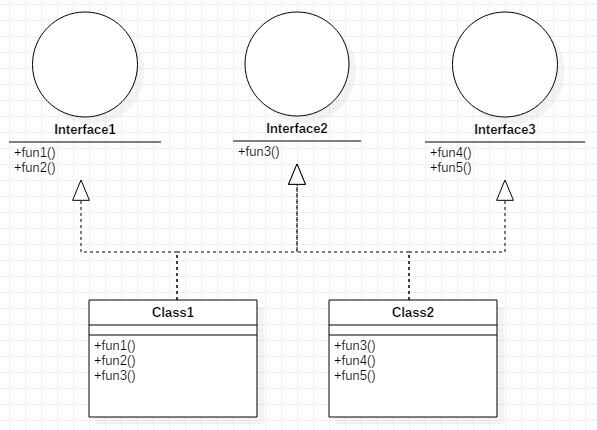
通俗一点说，就是接口要尽量细分，把不需要的方法尽量写在多个不同的接口上。 假如我有一个抽象类，里面有5个抽象方法，现有两个具体类class1和class2，class1要实现抽象类中的1-3的方法，class2要实现抽象类中的3-5的方法，假如只有一个抽象类，则我们的代码长这样：

from abc import ABCMeta,abstractmethod  
class InterFace(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def fun1(self):  
 pass  
  
 @abstractmethod  
 def fun2(self):  
 pass  
  
 @abstractmethod  
 def fun3(self):  
 pass  
  
 @abstractmethod  
 def fun4(self):  
 pass  
  
 @abstractmethod  
 def fun5(self):  
 pass  
  
class C1(InterFace):  
  
 def fun1(self):  
 print(**'使用了fun1'**)  
  
 def fun2(self):  
 print(**'使用了fun2'**)  
  
 def fun3(self):  
 print(**'使用了fun3'**)  
  
 def fun4(self):  
 print(**'对不起不需要fun4'**)  
  
 def fun5(self):  
 print(**'对不起不需要fun5'**)  
  
class C2(InterFace):  
  
 def fun1(self):  
 print(**'对不起不需要fun1'**)  
  
 def fun2(self):  
 print(**'对不起不需要fun2'**)  
  
 def fun3(self):  
 print(**'使用了fun3'**)  
  
 def fun4(self):  
 print(**'使用了fun4'**)  
  
 def fun5(self):  
 print(**'使用了fun5'**)

 对应的UML类图如下：

明显，这个单一接口比较臃肿。有什么办法解决呢？

接下来运用接口隔离原则，对这个单一接口做分离，看规律：

 Class1需要fun1-3，class2需要fun3-5，则可以将单一接口分离为3个接口，接口1存放fun1-2，接口2存放fun3，接口3存放fun4-5即可，如图：

## 组合复用原则(复用的最佳实践)

### 定义

优先使用组合，使系统更灵活，其次才考虑继承，达到复用的目的。

### 理解

在我们想复用代码时，我们会优先的想到，继承！可是，具有继承关系的两个类耦合度是最高的！因为父类改动，子类很有可能受影响，反之亦然。

如果父类的功能比较稳定，建议使用继承来实现代码的复用，因为继承是静态的，在运行时无法动态调用。

而组合与聚合，都是整体与部分的关系，组合：部分离不开整体；聚合：部分可以离开整体。组合与聚合是通过获得其他类的引用，在运行时刻动态定义的，也就是在一个对象中保存其他对象的属性，这种方式要求对象有良好定义的接口，并且这个接口也不经常发生改变，而且对象只能通过接口来访问，这样我们并不破坏封装性，所以只要类型一致，运行时还可以通过一个对象替换另一个对象。

## 里氏替换(继承后的重写，指导继承的设计)

### 定义

子类能够替换其基类对象被使用。

### 理解

听上去概念好像很简单，不就是父类实现的方法就能被子类实现，父类在外部的调用，替换成子类也可以嘛。

其实不然。里氏替换真正的理解：

1. 子类可以实现父类的抽象方法，但是不能覆盖父类的非抽象方法
2. 子类可以增加自己持有的方法
3. 子类覆盖或实现父类的方法时，方法的前置条件(参数)要比父类方法的参数更宽松
4. 当子类的方法实现父类的抽象方法时，犯法的后置条件(返回值)要比父类更严格。

从上面可以看出，当我们继承父类时，不要覆盖父类已经实现好的方法，只能覆盖未实现好的方法。

为什么？因为如果覆盖了已经实现的方法后，子类父类耦合度会高。

所以，虽然继承是面向对象的三大特性之一，但是如果使用不当，反而会让程序更复杂，代码的耦合度更高。

## 迪米特法则(类与类交互的原则)

### 定义

又名最少知道原则。类与类之间应该是最少知道的，应当多加一个类来辅助类与类之间的关系。

### 理解

通俗点说，就是一个软件实体应当尽可能少的与其他实体发送相互作用，每一个软件单位对其他的单位都只有最少的知识。

比如，如果类A依赖类B，如果类A发送改变，那么最后的结果是类B可能要看一看自身的代码是否要发生改变，甚至类B也要发生改变。所以，按照迪米特法则来说，建议一个中间类MAB，来处理类A和类B之间的关系。这就降低了类A与类B的耦合度。

# 九、专题讲解：23+1种设计模式及其Python实现

## 基础概念介绍

### 设计模式

设计模式是经过总结、优化的，对我们经常会碰到的一些编程问题的可重用解决方案。一个设计模式并不像一个类或一个库那样能够直接作用于我们的代码。反之，设计模式更为高级，它是一种必须在特定情形下实现的一种方法模板。

设计模式不会绑定具体的编程语言。一个好的设计模式应该能够用大部分编程语言实现(如果做不到全部的话，具体取决于语言特性)。最为重要的是，设计模式也是一把双刃剑，如果设计模式被用在不恰当的情形下将会造成灾难，进而带来无穷的麻烦。然而如果设计模式在正确的时间被用在正确地地方，它将是你的救星。起初，你会认为“模式”就是为了解决一类特定问题而特别想出来的明智之举。

说的没错，看起来的确是通过很多人一起工作，从不同的角度看待问题进而形成的一个最通用、最灵活的解决方案。也许这些问题你曾经见过或是曾经解决过，但是你的解决方案很可能没有模式这么完备。虽然被称为“设计模式”，但是它们同“设计“领域并非紧密联系。设计模式同传统意义上的分析、设计与实现不同，事实上设计模式将一个完整的理念根植于程序中，所以它可能出现在分析阶段或是更高层的设计阶段。

很有趣的是因为设计模式的具体体现是程序代码，因此可能会让你认为它不会在具体实现阶段之前出现(事实上在进入具体实现阶段之前你都没有意识到正在使用具体的设计模式)。可以通过程序设计的基本概念来理解模式：增加一个抽象层。抽象一个事物就是隔离任何具体细节，这么做的目的是为了将那些不变的核心部分从其他细节中分离出来。当你发现你程序中的某些部分经常因为某些原因改动，而你不想让这些改动的部分引发其他部分的改动，这时候你就需要思考那些不会变动的设计方法了。这么做不仅会使代码可维护性更高，而且会让代码更易于理解，从而降低开发成本。

### 抽象

#### 抽象方法

不包含任何可实现代码的方法就叫做抽象方法，即一个方法中没有任何一个方法体

#### 定义

抽象类是包含抽象方法的类，只能在其子类中实现抽象方法的代码

#### 抽象方法的定义

1. 导入abc模块

import abc

1. 如果需要单个导入的话，一般我们只需要导入ABCMeta和abstractmethod就行

from abc import ABCMeta,abstractmethod

1. 将元类设置为ABCMeta

\_\_metaclass\_\_ = ABCMeta

1. 抽象方法前加装饰器abstractmethod

@abstractmethod

#### 抽象类的特点

1.要定义但是并不完整的实现所有方法

2.基本的大概意思其实就是父类

3.父类需要明确表示出哪些方法的特征，

#### 需要使用抽象类的地方

1.用作父类

2.用作检验实例类型

3.用作抛出异常说明

### 设计模式分类

设计模式分为基本的三种类型：

1.创建模式，提供实例化的方法，为适合的状况提供相应的对象创建方法。

2.结构化模式，通常用来处理实体之间的关系，使得这些实体能够更好地协同工作。

3.行为模式，用于在不同的实体建进行通信，为实体之间的通信提供更容易，更灵活的通信方法。

具体包含以下分类：

创建型

1. Factory Method（工厂方法）

2. Abstract Factory（抽象工厂）

3. Builder（建造者）

4. Prototype（原型）

5. Singleton（单例）

结构型

6. Adapter Class/Object（适配器）

7. Bridge（桥接）

8. Composite（组合）

9. Decorator（装饰）

10. Facade（外观）

11. Flyweight（享元）

12. Proxy（代理）

行为型

13. Interpreter（解释器）

14. Template Method（模板方法）

15. Chain of Responsibility（责任链）

16. Command（命令）

17. Iterator（迭代器）

18. Mediator（中介者）

19. Memento（备忘录）

20. Observer（观察者）

21. State（状态）

22. Strategy（策略）

23. Visitor（访问者）

针对Python中常用的创建型模式，下文将针对每一种创建型模式进行精讲。

## 创建型模式

### 工厂模式

#### 定义

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。

#### 适用情况

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。

#### 分类

工厂模式在实际应用中大致分为以下两类：

1.简单工厂模式

2.工厂方法模式

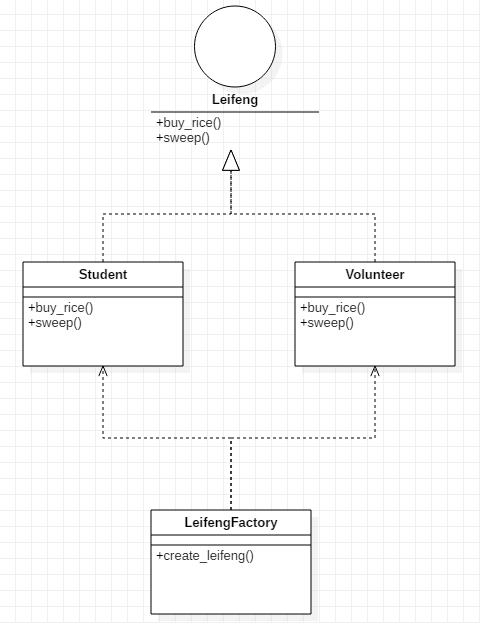
#### 举例

有一个学雷锋活动，有买米和扫地两个内容，参与的人有大学生和社区志愿者，他们各自的方法不一样。

#### 简单工厂模式实现

import abc  
class Leifeng(object,metaclass=abc.ABCMeta):  
 *'''定义雷锋抽象类，后续会有学生类和志愿者类继承该抽象类'''* @abc.abstractmethod  
 def \_\_init\_\_(self):  
 pass  
 @abc.abstractmethod  
 def but\_rice(self):  
 pass  
 @abc.abstractmethod  
 def sweep(self):  
 pass  
  
class Student(Leifeng):  
 *'''学生类，继承雷锋抽象类，重写两个抽象方法'''* def buy\_rice(self):  
 print(**'大学生帮你买大米'**)  
 def sweep(self):  
 print(**'大学生帮你扫地'**)  
  
class Volunteer(Leifeng):  
 *'''志愿者类，继承雷锋类，重写两个抽象方法'''* def buy\_rice(self):  
 print(**'志愿者帮你买大米'**)  
 def sweep(self):  
 print(**'志愿者帮你扫地'**)  
  
class LeifengFactory:  
 *'''创建雷锋工厂类，根据传入的参数类型返回相应对象'''* def create\_leifeng(self,type):  
 map\_ = {  
 **'大学生'**:Student(),  
 **'志愿者'**:Volunteer()  
 }  
 return map\_[type]  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 student1 = LeifengFactory().create\_leifeng(**'大学生'**)  
 *# student2 = LeifengFactory().create\_leifeng('大学生')* vol1 = LeifengFactory().create\_leifeng(**'志愿者'**)  
 *# vol2 = LeifengFactory().create\_leifeng('志愿者')* student1.buy\_rice()  
 student1.sweep()  
 vol1.buy\_rice()  
 vol1.sweep()

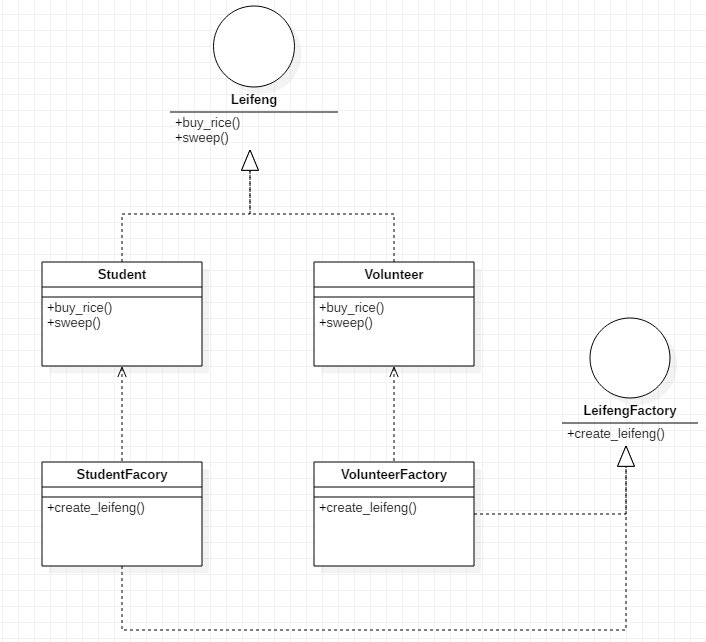
#### 简单工厂模式类图



#### 工厂方法模式实现

import abc  
class Leifeng(object,metaclass=abc.ABCMeta):  
 *'''定义一个雷锋基类，后续大学生类和志愿者类都要继承该类'''* @abc.abstractmethod  
 def but\_rice(self):  
 pass  
 @abc.abstractmethod  
 def sweep(self):  
 pass  
  
class Student(Leifeng):  
 *'''学生类，继承雷锋类，重写两个成员方法'''* def buy\_rice(self):  
 print(**'大学生帮你买大米'**)  
 def sweep(self):  
 print(**'大学生帮你扫地'**)  
  
class Volunteer(Leifeng):  
 *'''志愿者类，继承雷锋类，重写两个成员方法'''* def buy\_rice(self):  
 print(**'志愿者帮你买大米'**)  
 def sweep(self):  
 print(**'志愿者帮你扫地'**)  
*#以上为工厂类  
#以下为客户端(工厂方法类)*class LeifengFactory(object,metaclass=abc.ABCMeta):  
 *'''定义雷锋工厂类，定义一个雷锋工厂方法'''* @abc.abstractmethod  
 def create\_leifeng(self):  
 pass  
  
class StudentFactory(LeifengFactory):  
 *'''定义学生工厂类，继承雷锋工厂并重写雷锋工厂方法'''* def create\_leifeng(self):  
 return Student()  
  
class VolunteerFactory(LeifengFactory):  
 *'''定义志愿者工厂类，继承雷锋工厂并重写雷锋工厂方法'''* def create\_leifeng(self):  
 return Volunteer()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 studentFactory = StudentFactory()*#创建学生工厂对象* student = studentFactory.create\_leifeng()*#通过学生工厂创建学生对象* student.buy\_rice()  
 student.sweep()  
  
 volunteerFactory = VolunteerFactory()*#创建志愿者工厂对象* vol = volunteerFactory.create\_leifeng()*#通过志愿者工厂创建志愿者对象* vol.buy\_rice()  
 vol.sweep()

#### 工厂方法模式类图



#### 工厂方法模式相对于简单工厂模式的优点

在简单工厂中，如果需要新增类，例如加一个中学生类（MiddleStudent），就需要新写一个类，同时要修改工厂类的map\_，加入'中学生':MiddleStudent()，这样就违背了封闭开放原则中的一个'类写好后，尽量不要修改里面的内容'，这个原则。而在工厂方法中，需要增加一个中学生类和一个中学生工厂类（MiddleStudentFactory），虽然比较繁琐，但是符合封闭开放原则。在工厂方法中，将判断输入的类型，返回相应的类这个过程从工厂类中移到了客户端中实现，所以当需要新增类时，也是要修改代码的，但此时，只需修改客户端的代码就行，工厂类的代码我们不需要修改。

### 抽象工厂模式

#### 定义

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。

#### 适用情况

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。

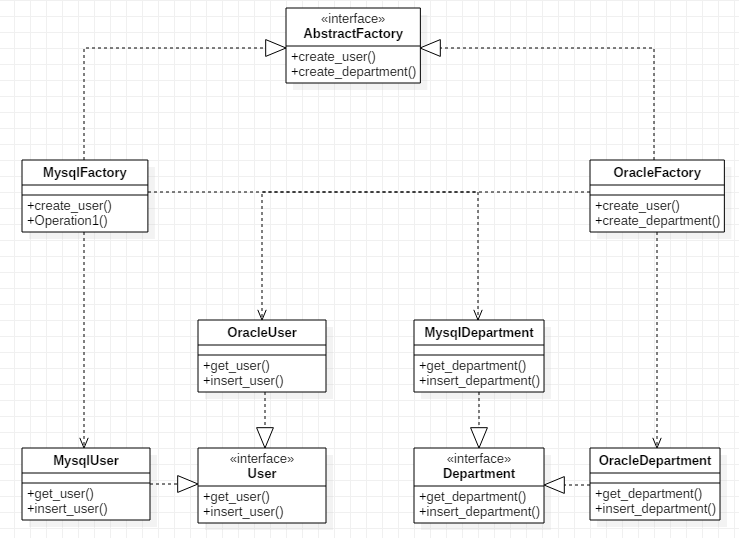
#### 举例

模拟向数据库用户表和部门表获取及插入数据，并且我们可能采用mysql和oracle数据库。

#### 抽象工厂模式实现

import sys  
from abc import ABCMeta,abstractmethod  
  
class User(object,metaclass=ABCMeta):  
 *'''定义抽象user类，并定义抽象获取user数据方法，抽象插入user数据方法'''* @abstractmethod  
 def get\_user(self):  
 pass  
 @abstractmethod  
 def insert\_user(self):  
 pass  
  
class Department(object,metaclass=ABCMeta):  
 *'''定义抽象department类，并定义抽象获取department数据方法，抽象插入department数据方法'''* @abstractmethod  
 def get\_department(self):  
 pass  
 @abstractmethod  
 def insert\_department(self):  
 pass  
*#以上为抽象类  
#以下为具体操作数据库类*class MysqlUser(User):  
 *'''定义操作具体user数据库的类，继承抽象user类，采用mysql数据库实现'''* def get\_user(self):  
 print(**'Mysqluser get user'**)  
 def insert\_user(self):  
 print(**'Mysqluser insert user'**)  
  
class MysqlDepartment(Department):  
 *'''定义操作具体department数据库的类，继承抽象department类，采用mysql数据库实现'''* def get\_department(self):  
 print(**'Mysqldepartment get department'**)  
 def insert\_department(self):  
 print(**'Mysqldepartment insert department'**)  
  
class OracleUser(User):  
 *'''定义操作具体user数据库的类，继承抽象user类，采用Oracle数据库实现'''* def get\_user(self):  
 print(**'Oracleuser get user'**)  
 def insert\_user(self):  
 print(**'Oracleuser insert user'**)  
  
class OracleDepartment(Department):  
 *'''定义操作具体department数据库的类，继承抽象department类，采用Oracle数据库实现'''* def get\_department(self):  
 print(**'Oracledepartment get department'**)  
 def insert\_department(self):  
 print(**'Oracledepartment insert department'**)  
  
*#以下为抽象工厂类*class AbstractFactory(object,metaclass=ABCMeta):  
 *'''定义抽象工厂类及抽象方法，后续mysql工厂类级oracle工厂类会继承该类'''* @abstractmethod  
 def create\_user(self):  
 pass  
 @abstractmethod  
 def create\_department(self):  
 pass  
  
class MysqlFactory(AbstractFactory):  
 *'''定义mysql工厂类，继承抽象工厂类'''* def create\_user(self):  
 return MysqlUser()  
  
 def create\_department(self):  
 return MysqlDepartment()  
  
  
class OracleFactory(AbstractFactory):  
 *'''定义Oracle工厂类，继承抽象工厂类'''* def create\_user(self):  
 return OracleUser()  
  
 def create\_department(self):  
 return OracleDepartment()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 db = input(**'请输入数据库类型：'**)  
 myfactory = **''** if db == **'mysql'**:  
 myfactory = MysqlFactory()  
 elif db == **'oracle'**:  
 myfactory = OracleFactory()  
 else:  
 print(**'不支持的数据库类型'**)  
 sys.exit()  
  
 user = myfactory.create\_user()  
 department = myfactory.create\_department()  
 user.insert\_user()  
 user.get\_user()  
 department.insert\_department()  
 department.get\_department()

#### 抽象工厂模式类图



### 建造者模式

#### 定义

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。

#### 适用情况

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

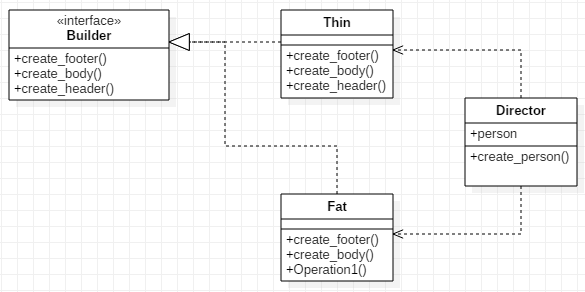
#### 举例

建筑工程师(指挥者)指挥工人(建造者)建造

#### 建造者模式实现

from abc import ABCMeta,abstractmethod  
  
class Builder(object,metaclass=ABCMeta):  
 *'''定义建造者抽象类，定义抽象方法用于子类继承'''* @abstractmethod  
 def create\_footer(self):  
 pass  
  
 @abstractmethod  
 def create\_body(self):  
 pass  
  
 @abstractmethod  
 def create\_header(self):  
 pass  
  
class Thin(Builder):  
 *'''定义瘦子类，继承抽象方法并重写'''* def create\_footer(self):  
 print(**'瘦子的脚创建了'**)  
  
 def create\_body(self):  
 print(**'瘦子的身体创建了'**)  
  
 def create\_header(self):  
 print(**'瘦子的头创建了'**)  
  
class Fat(Builder):  
 *'''定义胖子类，继承抽象方法并重写'''* def create\_footer(self):  
 print(**'胖子的脚创建了'**)  
  
 def create\_body(self):  
 print(**'胖子的身体创建了'**)  
  
 def create\_header(self):  
 print(**'胖子的头创建了'**)  
  
class Director(object):  
 *'''定义指挥者类，根据传入参数决定调用哪个子类的方法'''* def \_\_init\_\_(self,person):  
 self.person = person  
 def create\_person(self):  
 self.person.create\_footer()  
 self.person.create\_body()  
 self.person.create\_header()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 thin = Thin()  
 fat = Fat()  
 direct\_thin = Director(thin)  
 direct\_fat = Director(fat)  
 direct\_thin.create\_person()  
 direct\_fat.create\_person()

#### 建造者模式类图



### 原型模式

#### 定义

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。

#### 适用情况

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，或者当一个类的实例只能有几个不同状态组合中的一种时。

2.需要大量的基于某个基础原型进行微量修改而得到新原型时使用

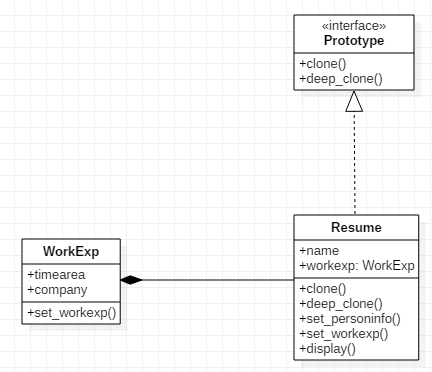
#### 举例

投递简历，需要引用你的过往工作经历。

#### 原型模式实现

from copy import copy,deepcopy  
from abc import ABCMeta,abstractmethod  
  
class Prototype(object,metaclass=ABCMeta):  
 *'''创建原型抽象类，用于子类继承'''* @abstractmethod  
 def clone(self):  
 pass  
  
 @abstractmethod  
 def deep\_clone(self):  
 pass  
  
class WorkExp(object):  
 *'''创建工作经验类，定义添加工作经验方法'''* def \_\_init\_\_(self):  
 self.timearea = **''** self.company = **''** def set\_workexp(self,timearea,company):  
 self.timearea = timearea  
 self.company = company  
  
class Resume(Prototype):  
 *'''创建简历类，继承原型抽象类，重写抽象方法的同时添加部分属性'''* def \_\_init\_\_(self,name):  
 self.name = name  
 self.workexp = WorkExp()  
  
 def set\_personinfo(self,sex,age):  
 self.sex = sex  
 self.age = age  
  
 def set\_workexp(self,timearea,company):  
 self.workexp.set\_workexp(timearea,company)  
  
 def display(self):  
 print(self.name)  
 print(self.sex,self.age)  
 print(**'工作经历：%s,%s'**%(self.workexp.timearea,self.workexp.company))  
  
 def clone(self):  
 return copy(self)  
  
 def deep\_clone(self):  
 return deepcopy(self)  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 obj1 = Resume(**'安伟超'**)  
 obj2 = obj1.clone()  
 obj3 = obj1.deep\_clone()  
  
 obj1.set\_personinfo(**'男'**,30)  
 obj1.set\_workexp(**'2016-2018'**,**'达内时代科技集团天津天大中心'**)  
  
 obj2.set\_personinfo(**'男'**,32)  
 obj2.set\_workexp(**'2018-2020'**,**'达内时代科技集团天津学府中心'**)  
  
 obj3.set\_personinfo(**'男'**,34)  
 obj3.set\_workexp(**'2020-2022'**,**'达内时代科技集团天津长虹中心'**)  
  
 obj1.display()  
 obj2.display()  
 obj3.display()

#### 原型模式类图



### 单例模式

#### 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。

#### 举例

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。

#### 单例模式问题由来

class A(object):  
 def \_\_init\_\_(self):  
 pass  
 def foo(self):  
 pass  
  
a = A()  
print(id(a))  
b = A()  
print(id(b))  
  
运行结果如下：  
anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
140534096496512  
140534096497128

很明显，通过打印实例的id可以发现，A类默认被创建了两个实例a和b

那么，如何让类只去实例化一个对象，而后再创建的实例是返回上一次的对象的引用呢？

#### 单例模式实现之——使用装饰器

def singleton(cls,\*args,\*\*kwargs):  
 *'''  
 使用装饰器的原理：  
 1.先创建外层函数，需要传入一个参数，此参数为类(对象)  
 2.创建一个空字典，用来保存单例  
 3.创建一个内层函数，用来获得单例  
 4.内层函数中进行判断：如果当前字典不存在单例，就创建单例，如果存在，直接返回该单例的引用  
 5.外层函数返回内层函数  
  
 '''* instances = {}  
 print(**'装饰器函数被调用！'**)  
 def get\_singleton(\*args,\*\*kwargs):  
 if cls not in instances:  
 instances[cls] = cls(\*args,\*\*kwargs)  
 print(**'装饰器内层函数也被调用！'**)  
 print(instances)  
 return instances[cls]  
 return get\_singleton  
  
@singleton  
*#此处，相当于：Student = singleton(Student)*class Student(object):  
 *'''  
 创建单例的原理：  
 1.先由类实例化对象：xiaoanan = Student(30,'小安安')并传参，此时，因为前面已添加装饰器@singleton，Student此时相当于绑定到了get\_singleton函数，这句实例化对象的语句其实就是如下格式：xiaoanan = get\_singleton(30,'小安安'),而实例化对象时，这句话相当于调用了装饰器的内部函数get\_singleton，因为是第一次创建对象xiaoanan ,当前字典中并没有单例xiaoanan,所以内层函数get\_singleton中会执行if语句真值表达式为True时的语句，即：instances[cls] = cls(\*args,\*\*kwargs)，而这里的cls接收到的参数是Student，所以此句代码相当于：instances[Student] = Student(30,'小安安'),即给字典键名为Student的键添加一个值，这个值是Student类实例化的对象  
 2.当又一次实例化对象xiaochaochao时，因为仍然还是通过Student类实例化该对象，而之前的单例已经存在于字典中了，所以不会再创建第二个单例，只会直接返回已有的单例并绑定当前的引用，进而真正实现了单例模式！  
  
 '''* def \_\_init\_\_(self,age,name):  
 self.age = age  
 self.name = name  
 print(**'\_\_init\_\_()方法被调用了'**)  
xiaoan = Student(30,**'小安安'**)  
xiaochaochao = Student(18,**'小超超'**)  
print(id(xiaoan))  
print(id(xiaochaochao))  
  
  
  
最终执行结果：  
anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
装饰器函数被调用！  
\_\_init\_\_()方法被调用了  
装饰器内层函数也被调用！  
{<class **'\_\_main\_\_.Student'**>: <\_\_main\_\_.Student object at 0x7fe08913f748>}  
装饰器内层函数也被调用！  
{<class **'\_\_main\_\_.Student'**>: <\_\_main\_\_.Student object at 0x7fe08913f748>}  
140602349188936  
140602349188936  
  
此方式即实现了单例模式。

#### 单例模式实现之——使用类

import threading  
import time  
  
class Singleton(object):  
 def \_\_init\_\_(self):  
 *# time.sleep(1)* pass  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
*# b = Singleton.instance()*for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
 程序暂时先写成这样，我们先运行一下，看看结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 看上去，没毛病，恩，实现了单例，但其实！有毛病！此时看上去没问题是因为执行速度过快，如果我们让\_\_init\_\_方法存在一些IO操作，就会发现问题。  
 下面，我们用睡眠代替可能的IO操作看看会出现什么效果  
  
import threading  
import time  
  
class Singleton(object):  
 def \_\_init\_\_(self):  
 time.sleep(1)  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
*# b = Singleton.instance()*for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
 此时，我们再看看程序运行的结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140107826004320  
 140107826107280  
 140107825398672  
 140107825883008  
 140107825392944  
 140107825882896  
 140107825398952  
 140107826107504  
 140107825835816  
 140107825881496  
  
 问题出现了！按照以上方式创建的单例，不支持多线程！  
 ok，如果非得采用这种方式来创建单例，解决办法是：加锁！  
 未加锁部分并发执行,加锁部分串行执行,速度降低,但是保证了数据安全  
 再看如下代码：  
  
import threading  
import time  
  
class Singleton(object):  
 \_instance\_lock = threading.Lock()  
 def \_\_init\_\_(self):  
 time.sleep(1)  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 with Singleton.\_instance\_lock:  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
*# b = Singleton.instance()*for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
time.sleep(5)  
a = Singleton.instance()  
print(id(a))  
  
  
 此时，我们再看看程序运行的结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
  
 到此，其实大致解决了我们的需求了，但是还是有一些小问题，就是当程序执行时，执行了time.sleep(20)后，下面实例化对象时，此时已经是单例模式了，但我们还是加了锁，这样不太好，再进行一些优化，把intance方法，改成下面的这样就行：  
  
  
import threading  
import time  
  
class Singleton(object):  
 \_instance\_lock = threading.Lock()  
 def \_\_init\_\_(self):  
 time.sleep(1)  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 with Singleton.\_instance\_lock:  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
*# b = Singleton.instance()*for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
time.sleep(5)  
a = Singleton.instance()  
print(id(a))  
  
 至此，支持多线程的单例实现结束，但这种方式还有弊端，不过不痛不痒了，很好发现：这种方式实现的单例，使用时会有限制，以后实例化对象必须通过调用类方法实现，即：  
 a = Singleton.instance()  
 而原本单单通过累的实例化，得到的就不是单例了。

#### 单例模式实现之——基于\_\_new\_\_方法

接上面的例子，已知，对于多线程情况下的单例，为了保证线程安全我们在内部加入了锁。

同时，我们知道，当我们实例化对象的时候，是先执行了\_\_new\_\_方法(如果不写，会默认调用Object.\_\_new\_\_,因为现版本中都是新式类，新式类默认继承自object)创建实例对象，再调用\_\_init\_\_方法初始化实例对象。基于以上，我们可以使用\_\_new\_\_方法实现单例模式。

import threading  
import time  
  
class Singleton(object):  
 \_instance\_lock = threading.Lock()  
 def \_\_init\_\_(self):  
 time.sleep(1)  
 def \_\_new\_\_(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 with Singleton.\_instance\_lock:  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = object.\_\_new\_\_(cls)  
 return Singleton.\_instance  
  
*# o1 = Singleton()  
# o2 = Singleton()  
# print(o1)  
# print(o2)*def test(args):  
 o = Singleton()  
 print(id(o))

#### 单例模式实现之——使用元类

class SingletonType(type):  
 def \_\_call\_\_(cls,\*args,\*\*kwargs):  
 if not hasattr(cls,**'instance'**):  
 cls.instance = super(SingletonType,cls).\_\_call\_\_(\*args,\*\*kwargs)  
 return cls.instance  
  
class Myobj(object,metaclass=SingletonType):  
 def \_\_init\_\_(self,name):  
 self.name = name  
  
for i in range(20):  
 obj = Myobj(**'安伟超'**)  
  
  
 print(id(obj))

## 结构型模式

### 适配器模式

#### 定义

将一个类的接口转换成为客户希望的另外一个接口

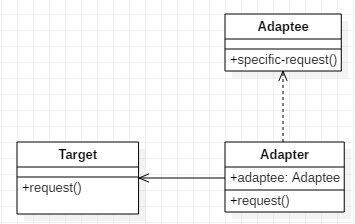
#### 适用情况

系统数据和行为都正确,但接口不符合时,目的是使控制范围之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存的类,但接口又与复用环境不一致的情况

#### 适配器模式实现

class Target(object):  
 def request(self):  
 print(**'普通请求'**)  
  
class Adaptee(object):  
 def specific\_request(self):  
 print(**'特殊请求'**)  
  
class Adapter(Target):  
 def \_\_init\_\_(self):  
 self.adaptee = Adaptee()  
 def request(self):  
 self.adaptee.specific\_request()  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 target = Adapter()  
 target.request()

#### 适配器模式类图



### 桥接模式

#### 定义

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.

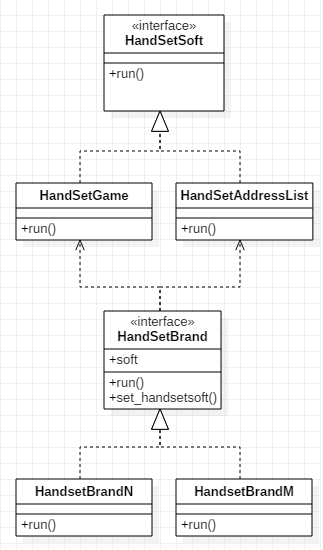
#### 举例

手机软件的分类 和 手机品牌的分类 耦合度低 两种分类中间用一个set\_handsetsoft桥接,各自的变化,不影响其他分类

#### 桥接模式实现

from abc import ABCMeta,abstractmethod  
*# 抽象手机软件类*class HandsetSoft(object,metaclass=ABCMeta):  
   
 @abstractmethod  
 def run(self):  
 pass  
  
*#具体游戏类,游戏是手机软件,继承抽象手机软件类*class HandsetGame(HandsetSoft):  
  
 def run(self):  
 print(**'运行手机游戏'**)  
  
*#手机通讯录*class HandsetAddressList(HandsetSoft):  
  
 def run(self):  
 print(**'运行手机通讯录'**)  
  
  
*#抽象手机品牌类*class HandsetBrand(object,metaclass=ABCMeta):  
  
   
 def \_\_init\_\_(self):  
 self.soft = **""** def set\_handsetsoft(self,soft):  
 self.soft = soft  
   
 @abstractmethod  
 def run(self):  
 pass  
  
*# 手机品牌N*class HandsetBrandN(HandsetBrand):  
  
 def run(self):  
 self.soft.run()  
  
*# 手机品牌M*class HandsetBrandM(HandsetBrand):  
  
 def run(self):  
 self.soft.run()  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 game = HandsetGame()  
 address = HandsetAddressList()  
  
 phoneN = HandsetBrandN()  
 phoneN.set\_handsetsoft(game)  
 phoneN.run()  
  
 phoneM = HandsetBrandM()  
 phoneM.set\_handsetsoft(address)  
 phoneM.run()

#### 桥接模式类图



### 组合模式

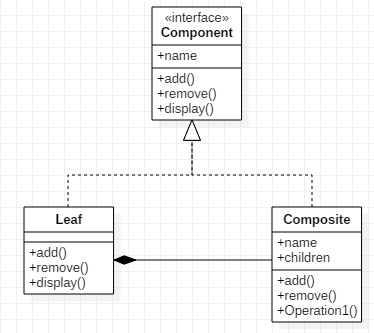
#### 定义

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

#### 组合模式实现

from abc import ABCMeta,abstractmethod  
  
*# 抽象一个组织类*class Component(object,metaclass=ABCMeta):  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 @abstractmethod  
 def add(self,comp):  
 pass  
  
 @abstractmethod  
 def remove(self,comp):  
 pass  
  
 @abstractmethod  
 def display(self, depth):  
 pass  
  
*# 叶子节点*class Leaf(Component):  
  
 def add(self,comp):  
 print(**'不能添加下级节点'**)  
  
 def remove(self,comp):  
 print(**'不能删除下级节点'**)  
  
 def display(self, depth):  
 strtemp = **''** for i in range(depth):  
 strtemp += strtemp+**'-'** print(strtemp+self.name)  
  
  
*# 枝节点*class Composite(Component):  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
 self.children = []  
  
 def add(self,comp):  
 self.children.append(comp)  
  
 def remove(self,comp):  
 self.children.remove(comp)  
  
 def display(self, depth):  
 strtemp = **''** for i in range(depth):  
 strtemp += strtemp+**'-'** print(strtemp+self.name)  
 for comp in self.children:  
 comp.display(depth+2)  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 *#生成树根* root = Composite(**"root"**)  
 *#根上长出2个叶子* root.add(Leaf(**'leaf A'**))  
 root.add(Leaf(**'leaf B'**))  
  
 *#根上长出树枝Composite X* comp = Composite(**"Composite X"**)  
 comp.add(Leaf(**'leaf XA'**))  
 comp.add(Leaf(**'leaf XB'**))  
 root.add(comp)  
  
 *#根上长出树枝Composite X* comp2 = Composite(**"Composite XY"**)  
 *#Composite X长出2个叶子* comp2.add(Leaf(**'leaf XYA'**))  
 comp2.add(Leaf(**'leaf XYB'**))  
 root.add(comp2)  
 *# 根上又长出2个叶子,C和D,D没张昊,掉了* root.add(Leaf(**'Leaf C'**))  
 leaf = Leaf(**"Leaf D"**)  
 root.add(leaf)  
 root.remove(leaf)  
 *#展示组织* root.display(1)

#### 组合模式类图



### 装饰模式

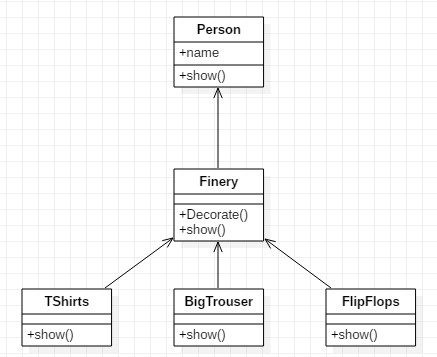
#### 定义

动态的给一个对象添加一些额外的职责,就增加功能来说,装饰模式比生成子类更为灵活.

#### 装饰模式实现

*# 定义对象接口*class Person(object):  
  
 def \_\_init\_\_(self,name):  
 self.name = name  
  
 def show(self):  
 print(**"装扮的%s"**%self.name)  
  
*#装饰类*class Finery(Person):  
  
 def \_\_init\_\_(self):  
 pass  
  
 def Decorate(self,componet):  
 self.componet = componet  
  
 def show(self):  
 if self.componet != None:  
 self.componet.show()  
  
*#装扮——T恤*class TShirts(Finery):  
  
 def \_\_init\_\_(self):  
 pass  
  
 def show(self):  
 print(**'T恤'**)  
 self.componet.show()  
  
*#装扮——大裤衩*class BigTrouser(Finery):  
  
 def \_\_init\_\_(self):  
 pass  
  
 def show(self):  
 print(**'大裤衩'**)  
 self.componet.show()  
  
*# 装扮——人字拖*class FlipFlops(Finery):  
  
 def \_\_init\_\_(self):  
 pass  
  
 def show(self):  
 print(**'人字拖'**)  
 self.componet.show()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 p = Person(**'Andy'**)  
 ff = FlipFlops()  
 bt = BigTrouser()  
 ts = TShirts()  
 ff.Decorate(p)  
 bt.Decorate(ff)  
 ts.Decorate(bt)  
 ts.show()

#### 装饰模式类图



### 外观模式

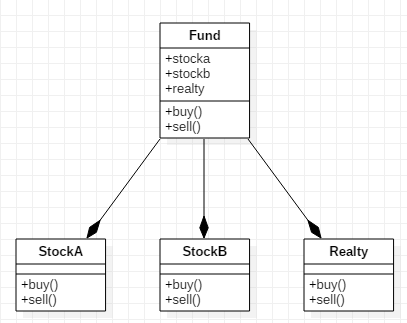
#### 定义

为子系统中的一组接口提供一个一致界面,此模式定义一个高层接口,使得子系统更加容易使用

#### 外观模式实现

*# 外观类*class Fund(object):  
  
 def \_\_init\_\_(self):  
 self.stocka = StockA()  
 self.stockb = StockB()  
 self.realty = Realty()  
  
 def buy(self):  
 self.stocka.buy()  
 self.stockb.buy()  
 self.realty.buy()  
  
 def sell(self):  
 self.stocka.sell()  
 self.stockb.sell()  
 self.realty.sell()  
  
*# 投资股票A类*class StockA(object):  
  
 def buy(self):  
 print(**'buy StockA'**)  
  
 def sell(self):  
 print(**'sell StockA'**)  
*# 投资股票B类*class StockB(object):  
  
 def buy(self):  
 print(**'buy StockB'**)  
  
 def sell(self):  
 print(**'sell StockB'**)  
  
*# 投资房地产*class Realty(object):  
  
 def buy(self):  
 print(**'buy Realty'**)  
  
 def sell(self):  
 print(**'sell Realty'**)  
  
  
  
if \_\_name\_\_==**"\_\_main\_\_"**:  
  
 fund = Fund()  
 fund.buy()  
 fund.sell()

#### 外观模式类图



### 享元模式

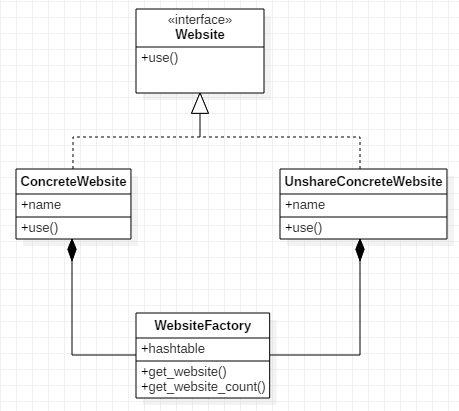
#### 定义

运用共享技术有效地支持大量细粒度的对象。对一个类进行的实例,只在第一次使用时建立,其他时候是用同一个实例,减少内存的开销。

#### 享元模式实现

from abc import ABCMeta,abstractmethod  
  
*# 抽象网站类*class Website(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def use(self):  
 pass  
  
*# 具体网站类*class ConcreteWebsite(Website):  
  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def use(self):  
 print(**"网站分类"**,self.name)  
*# 不共享的网站类*class UnshareConcreteWebsite(Website):  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def use(self):  
 print(**"不共享网站分类"**,self.name)  
  
*# 网站工厂*class WebsiteFactory(object):  
  
 def \_\_init\_\_(self):  
 self.hashtable = dict()  
  
 *# 获取网站类 如果存在直接返回,如果不存在建好之后返回* def get\_website(self, key):  
 if not key in self.hashtable:  
 self.hashtable[key] = ConcreteWebsite(key)  
 return self.hashtable[key]  
  
 *# 网站实例的个数* def get\_website\_count(self):  
 return len(self.hashtable.keys())  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 factory = WebsiteFactory()  
 f1 = factory.get\_website(**"blog"**)  
 f2 = factory.get\_website(**"blog"**)  
 f3 = factory.get\_website(**"blog"**)  
 f4 = factory.get\_website(**"website"**)  
 f5 = factory.get\_website(**"website"**)  
 f6 = factory.get\_website(**"website"**)  
 f7 = UnshareConcreteWebsite(**"test"**)  
 f1.use()  
 f2.use()  
 f3.use()  
 f4.use()  
 f5.use()  
 f6.use()  
 f7.use()

#### 享元模式类图



## 行为型模式

### 解释器模式

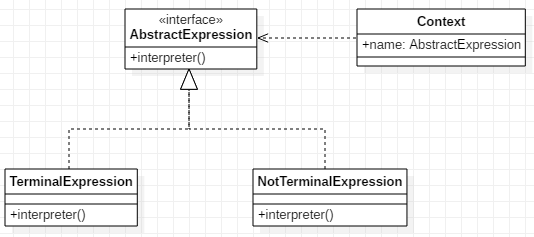
#### 定义

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.

#### 解释器模式实现

from abc import ABCMeta,abstractmethod  
*#抽象一个解释器类*class AbstractExpression(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def interpreter(self, context):  
 pass  
  
*#具体解释器——终端 继承抽象解释器*class TerminalExpression(AbstractExpression):  
  
 def interpreter(self, context):  
 print(**"终端解释器"**,context)  
  
*#具体解释器——终端 继承抽象解释器*class NotTerminalExpression(AbstractExpression):  
  
 def interpreter(self, context):  
 print(**"非终端解释器"**,context)  
  
  
class Context(object):  
  
 def \_\_init\_\_(self):  
 self.name = **""**if \_\_name\_\_ == **"\_\_main\_\_"**:  
 context = Context()  
 context.name = **'Andy'** arr\_list = [NotTerminalExpression(),TerminalExpression(),TerminalExpression()]  
 for entry in arr\_list:  
 entry.interpreter(context)

#### 解释器模式类图



### 模板方法模式

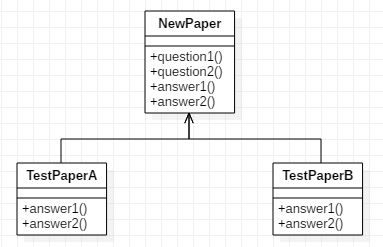
#### 定义

定义一个操作中的算法骨架，将一些步骤延迟至子类中.模板方法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。

#### 模板方法模式实现

class NewPaper(object):  
  
 def question1(self):  
 print(**"题目1"**)  
 print(self.answer1())  
  
 def question2(self):  
 print(**"题目2"**)  
 print(self.answer2())  
  
 def answer1(self):  
 return **''** def answer2(self):  
 return **''**class TestPaperA(NewPaper):  
  
 def answer1(self):  
 return **'答案A1'** def answer2(self):  
 return **'答案A2'**class TestPaperB(NewPaper):  
  
 def answer1(self):  
 return **'答案B1'** def answer2(self):  
 return **'答案B2'**if \_\_name\_\_ == **'\_\_main\_\_'**:  
 test1 = TestPaperA()  
 test2 = TestPaperB()  
 print(**"试卷A"**)  
 test1.question1()  
 test1.question2()  
 print(**"试卷B"**)  
 test2.question1()  
 test2.question2()

#### 模板方法模式类图



### 责任链模式

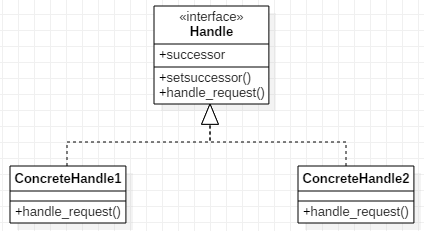
#### 定义

使多个对象都有机会处理请求，从而避免发送者和接收者的耦合关系。将对象连成链并沿着这条链传递请求直到被处理

#### 责任链模式实现

from abc import ABCMeta,abstractmethod  
*#抽象一个处理类*class Handle(object,metaclass=ABCMeta):  
  
 def \_\_init\_\_(self):  
 self.successor = **''** def setsuccessor(self, successor):  
 self.successor = successor  
  
 @abstractmethod  
 def handle\_request(self,request):  
 pass  
  
*# 具体处理者类1*class ConcreteHandle1(Handle):  
  
 def handle\_request(self,request):  
 if request>0 and request<=10:  
 print(**"ConcreteHandle1处理请求 "**,request)  
 else:  
 self.successor.handle\_request(request)  
  
*# 具体处理者类2*class ConcreteHandle2(Handle):  
  
 def handle\_request(self,request):  
 if request>10 and request<=20:  
 print(**"ConcreteHandle2处理请求 "**,request)  
 else:  
 self.successor.handle\_request(request)  
  
if \_\_name\_\_==**"\_\_main\_\_"**:  
 c1 = ConcreteHandle1()  
 c2 = ConcreteHandle2()  
 c1.setsuccessor(c2)  
  
 for i in range(6,15,2):  
 c1.handle\_request(i)

#### 责任链模式类图



### 命令模式

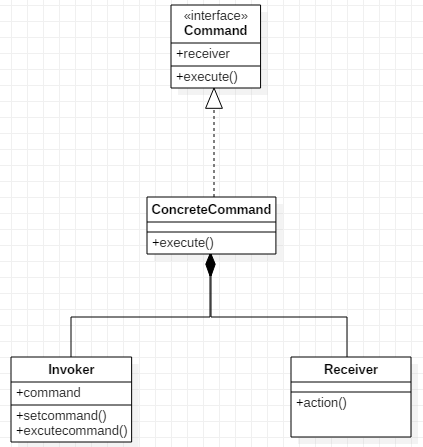
#### 定义

将请求封装成对象，从而使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.

#### 命令模式实现

from abc import ABCMeta,abstractmethod  
*# 命令类*class Command(object,metaclass=ABCMeta):  
  
 def \_\_init\_\_(self, receiver):  
 self.receiver = receiver  
  
 @abstractmethod  
 def execute(self):  
 pass  
  
*# 执行接收者的操作*class ConcreteCommand(Command):  
  
 def execute(self):  
 self.receiver.action()  
  
  
*# 接收命令*class Invoker(object):  
  
 def \_\_init\_\_(self):  
 self.command = **''** def setcommand(self,command):  
 self.command = command  
  
 def excutecommand(self):  
 self.command.execute()  
  
  
*# 具体执行类*class Receiver(object):  
  
 def action(self):  
 print(**"执行请求"**)  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
  
 r = Receiver() *#创建需要执行的命令* c = ConcreteCommand(r) *# 绑定要执行的命令* i = Invoker() *# 创建命令执行者* i.setcommand(c) *# 接受命令* i.excutecommand() *# 执行命令*

#### 命令模式类图



### 迭代器模式

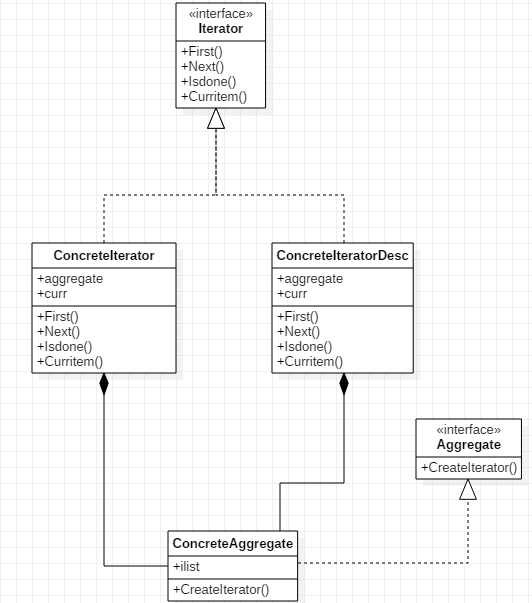
#### 定义

提供方法顺序访问一个聚合对象中各元素，而又不暴露该对象的内部表示.

#### 迭代器模式实现

from abc import ABCMeta,abstractmethod  
*#迭代器抽象类*class Iterator(object,metaclass=ABCMeta):  
   
 @abstractmethod  
 def First(self):  
 pass  
   
 @abstractmethod  
 def Next(self):  
 pass  
  
 @abstractmethod  
 def Isdone(self):  
 pass  
  
 @abstractmethod  
 def CurrItem(self):  
 pass  
  
*#聚集抽象类*class Aggregate(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def CreateIterator(self):  
 pass  
  
*#具体迭代器类*class ConcreteIterator(Iterator):  
  
 def \_\_init\_\_(self, aggregate):  
 self.aggregate = aggregate  
 self.curr = 0  
  
 def First(self):  
 return self.aggregate[0]  
  
 def Next(self):  
 ret = None  
 self.curr += 1  
 if self.curr < len(self.aggregate):  
 ret = self.aggregate[self.curr]  
 return ret  
  
 def Isdone(self):  
 return True if self.curr+1 >= len(self.aggregate) else False  
  
 def CurrItem(self):  
 return self.aggregate[self.curr]  
  
*#具体聚集类*class ConcreteAggregate(Aggregate):  
  
 def \_\_init\_\_(self):  
 self.ilist = []  
  
 def CreateIterator(self):  
 return ConcreteIterator(self)  
  
class ConcreteIteratorDesc(Iterator):  
 def \_\_init\_\_(self, aggregate):  
 self.aggregate = aggregate  
 self.curr = len(aggregate)-1  
  
 def First(self):  
 return self.aggregate[-1]  
  
 def Next(self):  
 ret = None  
 self.curr -= 1  
 if self.curr >= 0:  
 ret = self.aggregate[self.curr]  
 return ret  
  
 def Isdone(self):  
 return True if self.curr-1<0 else False  
  
 def CurrItem(self):  
 return self.aggregate[self.curr]  
  
if \_\_name\_\_==**"\_\_main\_\_"**:  
 ca = ConcreteAggregate()  
 ca.ilist.append(**"大鸟"**)  
 ca.ilist.append(**"小菜"**)  
 ca.ilist.append(**"老外"**)  
 ca.ilist.append(**"小偷"**)  
  
 itor = ConcreteIterator(ca.ilist)  
 print(itor.First())  
 while not itor.Isdone():  
 print(itor.Next())  
 print(**"————倒序————"**)  
 itordesc = ConcreteIteratorDesc(ca.ilist)  
 print(itordesc.First())  
 while not itordesc.Isdone():  
 print(itordesc.Next())

#### 迭代器模式类图



### 中介者模式

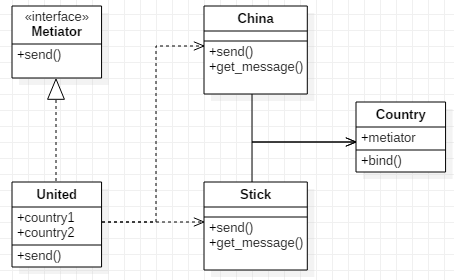
#### 定义

用一个对象来封装一系列的对象交互，中介者使各对象不需要显示地相互引用，从而使耦合松散，而且可以独立地改变它们之间的交互.

#### 中介者模式实现

from abc import ABCMeta,abstractmethod  
*# 抽象中介者类*class Metiator(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def send(self,message,concreteobj):  
 pass  
  
*# 具体中介者类——联合国*class United(Metiator):  
  
 def \_\_init\_\_(self):  
 self.country1 = **""** self.country2 = **""** def send(self,message,concreteobj):  
 if self.country1 == concreteobj:  
 self.country2.get\_message(message)  
 elif self.country2 == concreteobj:  
 self.country1.get\_message(message)  
 else:  
 print(**'没有对象'**)  
*# 抽象交互类*class Country(object):  
  
 def \_\_init\_\_(self):  
 self.metiator = **""** def bind(self,metiator):  
 self.metiator = metiator  
  
*# 具体交互类——中国*class China(Country):  
  
 def send(self, message):  
 self.metiator.send(message,self)  
  
 def get\_message(self,message):  
 print(**"中国获取对方消息:"**,message)  
*# 具体交互类——某国*class Stick(Country):  
  
 def send(self, message):  
 self.metiator.send(message,self)  
  
 def get\_message(self,message):  
 print(**"棒子获取对方消息:"**,message)  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 *# 创建中介者,具体的交互者* united = United()  
 china = China()  
 stick = Stick()  
 *# 交互者绑定交互的中介者* china.bind(united)  
 stick.bind(united)  
  
 united.country1 = china  
 united.country2 = stick  
 *#开始交互* china.send(**'棒子,别部署萨德,揍你哦'**)  
 stick.send(**'哥,我错了,不部署了'**)

#### 中介者模式类图



### 备忘录模式

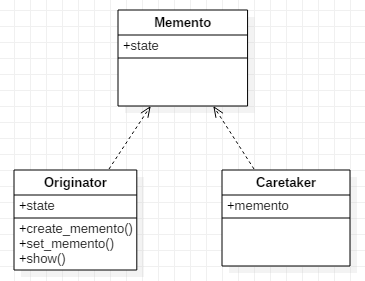
#### 定义

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,这样已经后就可将该对象恢复到原先保存的状态

#### 备忘录模式实现

*# 发起人类*class Originator(object):  
  
 def \_\_init\_\_(self, state):  
 self.state = state  
  
 def create\_memento(self):  
 return Memento(self.state)  
  
 def set\_memento(self, memento):  
 self.state = memento.state  
  
 def show(self):  
 print(**"当前状态 "**, self.state)  
  
*# 备忘录类*class Memento(object):  
  
 def \_\_init\_\_(self, state):  
 self.state = state  
  
*# 管理者类*class Caretaker(object):  
  
 def \_\_init\_\_(self,memento):  
 self.memento = memento  
  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 *# 初始状态* originator = Originator(state=**'On'**)  
 originator.show()  
 *# 备忘录* caretaker = Caretaker(originator.create\_memento())  
 *# 修改状态* originator.state = **'Off'** originator.show()  
 *# 复原状态* originator.set\_memento(caretaker.memento)  
 originator.show()

#### 备忘录模式类图



### 观察者模式

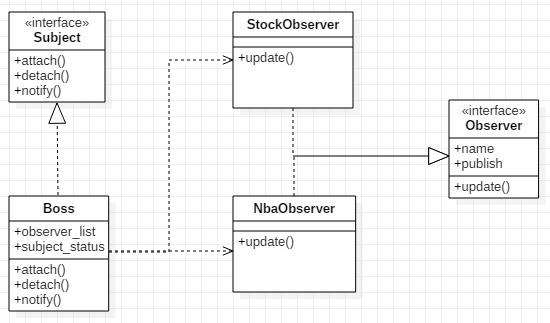
#### 定义

定义了一种一对多的关系，让多个观察对象同时监听一个主题对象，当主题对象状态发生变化时会通知所有观察者,是它们能够自动更新自己

#### 观察者模式实现

from abc import ABCMeta,abstractmethod  
*#抽象通知者类*class Subject(object):  
  
 @abstractmethod  
 def attach(self, observer):  
 pass  
  
 @abstractmethod  
 def detach(self,observer):  
 pass  
  
 @abstractmethod  
 def notify(self):  
 pass  
  
  
*#具体通知者类*class Boss(Subject):  
  
 def \_\_init\_\_(self):  
 self.observer\_list = []  
 self.subject\_status = **''** def attach(self, observer):  
 self.observer\_list.append(observer)  
  
 def detach(self,observer):  
 self.observer\_list.remove(observer)  
  
 def notify(self):  
 for item in self.observer\_list:  
 item.update()  
  
*#抽象观察者类*class Observer(object,metaclass=ABCMeta):  
  
 def \_\_init\_\_(self, name, publish):  
 self.name = name  
 self.publish = publish  
  
 @abstractmethod  
 def update(self):  
 pass  
  
*#具体观察者类-看股票的人*class StockObserver(Observer):  
  
 def update(self):  
 print(self.publish.subject\_status,self.name,**'关闭股票行情,继续工作'**)  
  
  
*#具体观察者类-看NBA的人*class NbaObserver(Observer):  
  
 def update(self):  
 print(self.publish.subject\_status,self.name,**'关闭NBA,继续工作'**)  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 publisher = Boss()  
 stocker = StockObserver(**'Andy'**,publisher)  
 nbaer = NbaObserver(**'Tracy'**,publisher)  
 publisher.attach(stocker)  
 publisher.attach(nbaer)  
 publisher.subject\_status = **'本老板回来了'** publisher.notify()

#### 观察者模式类图



### 状态模式

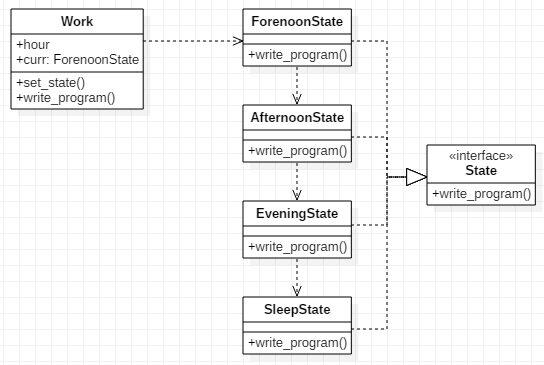
#### 定义

当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类

#### 状态模式实现

from abc import ABCMeta,abstractmethod  
class State(object,metaclass=ABCMeta):  
 def \_\_init\_\_(self):  
 pass  
  
 @abstractmethod  
 def write\_program(self, w):  
 pass  
  
class Work(object):  
  
 def \_\_init\_\_(self):  
 self.hour = 9  
 self.curr = ForenoonState()  
  
 def set\_state(self, s):  
 self.curr = s  
  
 def write\_program(self):  
 self.curr.write\_program(self)  
  
  
class ForenoonState(State):  
  
 def write\_program(self, w):  
 if w.hour < 12:  
 print(**"当前时间:%s点,"**%w.hour,**"精神百倍"**)  
 else:  
 w.set\_state(AfternoonState())  
 w.write\_program()  
  
class AfternoonState(State):  
  
 def write\_program(self, w):  
 if w.hour < 17:  
 print(**"当前时间:%s点,"**%w.hour,**"状态还行,继续努力"**)  
 else:  
 w.set\_state(EveningState())  
 w.write\_program()  
  
class EveningState(State):  
  
 def write\_program(self, w):  
 if w.hour < 21:  
 print(**"当前时间:%s点,"**%w.hour,**"加班呢,疲劳了"**)  
 else:  
 w.set\_state(SleepState())  
 w.write\_program()  
  
  
class SleepState(State):  
  
 def write\_program(self, w):  
 print(**"当前时间:%s点,"**%w.hour,**"不行了,睡着了"**)  
  
if \_\_name\_\_==**"\_\_main\_\_"**:  
 work = Work()  
 work.hour = 9  
 work.write\_program()  
 work.hour = 15  
 work.write\_program()  
 work.hour = 20  
 work.write\_program()  
 work.hour = 22  
 work.write\_program()

#### 状态模式类图



### 策略模式

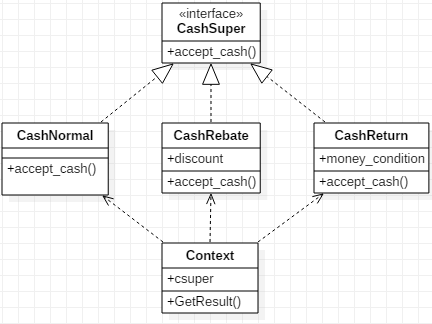
#### 定义

它定义了算法家族,分别封装起来,让他们之间可以相互替换,此模式让算法的变化,不会影响到使用算法的客户.

#### 策略模式实现

from abc import ABCMeta,abstractmethod  
*#现金收费抽象类*class CashSuper(object,metaclass=ABCMeta):  
  
 @abstractmethod  
 def accept\_cash(self,money):  
 pass  
*#正常收费子类*class CashNormal(CashSuper):  
  
 def accept\_cash(self,money):  
 return money  
  
*#打折收费子类*class CashRebate(CashSuper):  
  
 def \_\_init\_\_(self,discount=1):  
 self.discount = discount  
  
 def accept\_cash(self,money):  
 return money \* self.discount  
  
*#返利收费子类*class CashReturn(CashSuper):  
  
 def \_\_init\_\_(self,money\_condition=0,money\_return=0):  
 self.money\_condition = money\_condition  
 self.money\_return = money\_return  
  
 def accept\_cash(self,money):  
 if money>=self.money\_condition:  
 return money - (money / self.money\_condition) \* self.money\_return  
 return money  
  
*#具体策略类*class Context(object):  
  
 def \_\_init\_\_(self,csuper):  
 self.csuper = csuper  
  
 def GetResult(self,money):  
 return self.csuper.accept\_cash(money)  
  
  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 money = input(**"原价: "**)  
 strategy = {}  
 strategy[1] = Context(CashNormal())  
 strategy[2] = Context(CashRebate(0.8))  
 strategy[3] = Context(CashReturn(100,10))  
 mode = input(**"选择折扣方式: 1) 原价 2) 8折 3) 满100减10: "**)  
 if mode in strategy:  
 csuper = strategy[mode]  
 else:  
 print(**"不存在的折扣方式"**)  
 csuper = strategy[1]  
 print(**"需要支付: "**,csuper.GetResult(money))

#### 策略模式类图



### 访问者模式

#### 定义

不修改已有程序结构前提下，通过添加额外的访问者完成对代码功能的拓展

#### 访问者模式实现

*# 轮子,引擎, 车身这些定义好了都不需要变动*class Wheel(object):  
 def \_\_init\_\_(self, name):  
 self.name = name  
 def accept(self, visitor):  
 *# 每个visitor是同样的，但是其中的方法是不一样的，比如这里是visitWheel，  
 # 然后传入了self，想想？他其实想做什么就能做什么* visitor.visitWheel(self)  
  
class Engine(object):  
 def accept(self, visitor):  
 visitor.visitEngine(self)  
  
class Body(object):  
 def accept(self, visitor):  
 visitor.visitBody(self)  
  
*# 我们要组合成车*class Car(object):  
 def \_\_init\_\_(self):  
 self.engine = Engine()  
 self.body = Body()  
 self.wheels = [ Wheel(**"front left"**), Wheel(**"front right"**),  
 Wheel(**"back left"**) , Wheel(**"back right"**) ]  
  
 *# 这个也不需要在动，他只是上面部件的组合，只是做了属性的委托* def accept(self,visitor):  
 visitor.visitCar(self)  
 self.engine.accept(visitor)  
 self.body.accept(visitor)  
 for wheel in self.wheels:  
 wheel.accept(visitor)  
  
*# 这个才是我们的访问者，每次的修改都在这里面*class PrintVisitor(object):  
 def visitWheel(self, wheel):  
 print(**"Visiting "**+wheel.name+**" wheel"**)  
 def visitEngine(self, engine):  
 print(**"Visiting engine"**)  
 def visitBody(self, body):  
 print(**"Visiting body"**)  
 def visitCar(self, car):  
 print(**"Visiting car"**)  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 car = Car()  
 visitor = PrintVisitor()  
 car.accept(visitor)

#### 访问者模式类图

