**一、面向对象的程序设计的由来**

**1.第一阶段：面向机器，1940年以前**

最早的程序设计都是采用机器语言来编写的,直接使用二进制码来表示机器能够识别和执行的指令和数据。

简单来说,就是直接编写 0 和 1 的序列来代表程序语言。例如:使用 0000 代表 加载(LOAD),0001 代表 存储(STORE)等。

优点：机器语言由机器直接执行,速度快；

缺点：写比较困难，修改也麻烦，这样直接导致程序编写效率十分低下,编写程序花费的时间往往是实际运行时间的几十倍或几百倍。

由于机器语言实在是太难编写了,于是就发展出了汇编语言。

汇编语言亦称符号语言,用助记符代替机器 指令的操作码,用地址符号(Symbol)或标号(Label)代替指令或操作数的地址。

汇编语言由于是采用了助记符号来编写程序,比用机器语言的二进制代码编程要方便些,在一定程度上简化了编程过程。例如 使用 LOAD 来代替 0000,使用 STORE 来代替 0001。

即使汇编语言相比机器语言提升了可读性,但其本质上还是一种面向机器的语言,编写同样困难,也很容易出错。

**2.第二阶段：面向过程**

面向机器的语言通常情况下被认为是一种“低级语言”,为了解决面向机器的语言存在的问题,计算机科学的前辈们又创建了面向过程的语言。

面向过程的语言被认为是一种“高级语言”,相比面向机器的语言来说,面向过程的语言已经不再关注机器本身的操作指令、存储等方面,而是关注如何一步一步的解决具体的问题,即:解决问题的过程,这应该也是面向过程说法的来由。

相比面向机器的思想来说,面向过程是一次思想上的飞跃,将程序员从复杂的机器操作和运行的细节中解放出来,转而关注具体需要解决的问题;

面向过程的语言也不再需要和具体的机器绑定,从而具备了移植性和通用性;

面向过程的语言本身也更加容易编写和维护。

这些因素叠加起来,大大减轻了程序员的负担, 提升了程序员的工作效率,从而促进了软件行业的快速发展。

典型的面向过程的语言有:COBOL、FORTRAN、BASIC、C 语言等。

**3.第三阶段：结构化程序设计**

根本原因就是一些面向过程语言中的goto语句导致的面条式代码，极大的限制了程序的规模。

**结构化程序设计**（英语：Structured programming），一种编程范型。它采用子程序（函数就是一种子程序）、代码区块、for循环以及while循环等结构，来替换传统的goto。希望借此来改善计算机程序的明晰性、质量以及开发时间，并且避免写出面条式代码。

随着计算机硬件的飞速发展,以及应用复杂度越来越高,软件规模越来越大,原有的程序开发方式已经越来越不能满足需求了。1960 年代中期开始爆发了第一次软件危机,典型表现有软件质量低下、项目无法如期完成、项目严重超支等,因为软件而导致的重大事故时有发生。例如 1963 年美国 (http://en.wikipedia.org/wiki/Mariner\_1) 的水手一号火箭发射失败事故,就是因为一行 FORTRAN 代码 错误导致的。

软件危机最典型的例子莫过于 IBM 的 System/360 的操作系统开发。佛瑞德·布鲁克斯(Frederick P. Brooks, Jr.)作为项目主管,率领 2000 多个程序员夜以继日的工作,共计花费了 5000 人一年的工作量,写出将近 100 万行的源码,总共投入 5 亿美元,是美国的“曼哈顿”原子弹计划投入的 1/4。尽管投入如此巨大, 但项目进度却一再延迟,软件质量也得不到保障。布鲁克斯后来基于这个项目经验而总结的《人月神话》 一书,成了史上最畅销的软件工程书籍。

为了解决问题,在 1968、1969 年连续召开两次著名的 NATO 会议,会议正式创造了“软件危机”一词, 并提出了针对性的解决方法“软件工程”。虽然“软件工程”提出之后也曾被视为软件领域的银弹,但后来事实证明,软件工程同样无法解决软件危机。

差不多同一时间,“结构化程序设计”作为另外一种解决软件危机的方案被提出来了。 Edsger Dijkstra 于 1968 发表了著名的《GOTO 有害论》的论文,引起了长达数年的论战,并由此产生了结构化程序设计方 法。同时,第一个结构化的程序语言 Pascal 也在此时诞生,并迅速流行起来。

结构化程序设计的主要特点是抛弃 goto 语句,采取“自顶向下、逐步细化、模块化”的指导思想。

结构化程序设计本质上还是一种面向过程的设计思想,但通过“自顶向下、逐步细化、模块化”的方法,将软件的复杂度控制在一定范围内,从而从整体上降低了软件开发的复杂度。

结构化程序方法成为了 1970 年 代软件开发的潮流。

科学研究证明,人脑存在人类短期记忆一般一次只能记住 5-9 个事物,这就是著名的 7+- 2 原理。结构化程序设计是面向过程设计思想的一个改进,使得软件开发更加符合人类思维的 7+-2 特点。

**4.第四阶段：面向对象程序设计**

结构化编程的风靡在一定程度上缓解了软件危机,然而好景不长,随着硬件的快速发展,业务需求越来越复杂,以及编程应用领域越来越广泛,第二次软件危机很快就到来了。

第二次软件危机的根本原因还是在于软件生产力远远跟不上硬件和业务的发展,相比第一次软件危机主要体现在“复杂性”,第二次软件危机主要体现在“可扩展性”、“可维护性”上面。

传统的面向过程(包括 结构化程序设计)方法已经越来越不能适应快速多变的业务需求了,软件领域迫切希望找到新的银弹来解 决软件危机,在这种背景下,面向对象的思想开始流行起来。

面向对象的思想并不是在第二次软件危机后才出现的,早在 1967 年的 Simula 语言中就开始提出来了,但第二次软件危机促进了面向对象的发展。

面向对象真正开始流行是在 1980s 年代,主要得益于 C++的功 劳,后来的 Java、C#把面向对象推向了新的高峰。到现在为止,面向对象已经成为了主流的开发思想。

虽然面向对象开始也被当做解决软件危机的银弹,但事实证明,和软件工程一样,面向对象也不是银弹, 而只是一种新的软件方法而已。

虽然面向对象并不是解决软件危机的银弹,但和面向过程相比,面向对象的思想更加贴近人类思维的特点, 更加脱离机器思维,是一次软件设计思想上的飞跃。

补充：

**编程语言发展史上的杰出人物**：

* 约翰·巴科斯，发明了Fortran。
* 阿兰·库珀，开发了Visual Basic。
* 艾兹格·迪杰斯特拉，开创了正确运用编程语言(proper programming)的框架。
* 詹姆斯·高斯林，开发了Oak，该语言为Java的先驱。
* 安德斯·海尔斯伯格，开发了Turbo Pascal、Delphi，以及C#。
* 葛丽丝·霍普，开发了Flow-Matic，该语言对COBOL造成了影响。
* 肯尼斯·艾佛森，开发了APL，并与Roger Hui合作开发了J。
* 比尔·乔伊，发明了vi，BSD Unix的前期作者，以及SunOS的发起人，该操作系统后来改名为Solaris。
* 艾伦·凯，开创了面向对象编程语言，以及Smalltalk的发起人。
* Brian Kernighan，与丹尼斯·里奇合著第一本C程序设计语言的书籍，同时也是AWK与AMPL程序设计语言的共同作者。
* 约翰·麦卡锡，发明了LISP。
* 约翰·冯·诺伊曼，操作系统概念的发起者。
* 丹尼斯·里奇，发明了C。
* 比雅尼·斯特劳斯特鲁普，开发了C++。
* 肯·汤普逊，发明了Unix。
* 尼克劳斯·维尔特，发明了Pascal与Modula。
* 拉里·沃尔，创造了Perl与Perl 6。
* 吉多·范罗苏姆，创造了Python。

**二、面向对象程序设计的意义**

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

优点是：极大的降低了程序的复杂度；

缺点是：可扩展性差，修改代码麻烦；

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

面向对象的程序设计的核心是对象（上帝式思维），要理解对象为何物，必须把自己当成上帝，上帝眼里世间存在的万物皆为对象，不存在的也可以创造出来。面向对象的程序设计好比如来设计西游记，如来要解决的问题是把经书传给东土大唐，如来想了想解决这个问题需要四个人：唐僧，沙和尚，猪八戒，孙悟空，每个人都有各自的特征和技能（这就是对象的概念，特征和技能分别对应对象的数据属性和方法属性），然而这并不好玩，于是如来又安排了一群妖魔鬼怪，为了防止师徒四人在取经路上被搞死，又安排了一群神仙保驾护航，这些都是对象。然后取经开始，师徒四人与妖魔鬼怪神仙交互着直到最后取得真经。如来根本不会管师徒四人按照什么流程去取。

面向对象的程序设计的

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

缺点：可控性差，无法向面向过程的程序设计流水线式的可以很精准的预测问题的处理流程与结果，面向对象的程序一旦开始就由对象之间的交互解决问题，无法预测最终结果。于是我们经常看到一个游戏人某一参数的修改极有可能导致bug的技能出现，一刀砍死3个人，这个游戏就失去平衡。

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

面向对象的程序设计并不是全部。对于一个软件质量来说，面向对象的程序设计只是用来解决扩展性。

**三、类和对象**

**1.定义**

python中一切皆为对象，且python3统一了类与类型的概念，类型就是类。

基于面向对象设计一个款游戏：英雄联盟，每个玩家选一个英雄，每个英雄都有自己的特征和和技能，特征即数据属性，技能即方法属性，特征与技能的结合体就一个对象。

从一组对象中提取相似的部分就是类，类所有对象都具有的特征和技能的结合体。

**在python中，用变量表示特征，用函数表示技能，因而类是变量与函数的结合体，对象是变量与方法（指向类的函数）的结合体。**

**2.类的声明**

在python中声明类与声明函数很相似

* 声明类

[复制代码](javascript:void(0);)

'''

class 类名:

'类的文档字符串'

类体

'''

#我们创建一个类

class Data:

pass

[复制代码](javascript:void(0);)

* 分类：新式类和经典类

注意：  
1.只有在python2中才分新式类和经典类，python3中统一都是新式类  
2.新式类和经典类声明的最大不同在于,所有新式类必须继承至少一个父类  
3.所有类甭管是否显式声明父类，都有一个默认继承object父类（讲继承时会讲，先记住）

[复制代码](javascript:void(0);)

在python2中的区分

经典类：

class 类名:

　　pass

新式类：

class 类名(父类):

　　pass

在python3中，上述两种定义方式全都是新式类

[复制代码](javascript:void(0);)

例

class Garen: #定义英雄盖伦的类，不同的玩家可以用它实例出自己英雄;

camp='Demacia' #所有玩家的英雄(盖伦)的阵营都是Demacia;

def attack(self,enemy): #普通攻击技能，enemy是敌人;

enemy.life\_value-=self.aggressivity #根据自己的攻击力，攻击敌人就减掉敌人的生命值。

**3.类的两种用法**

* 类的属性引用

[复制代码](javascript:void(0);)

class Garen: #定义英雄盖伦的类，不同的玩家可以用它实例出自己英雄;

camp='Demacia' #所有玩家的英雄(盖伦)的阵营都是Demacia;

def attack(self,enemy): #普通攻击技能，enemy是敌人;

enemy.life\_value-=self.aggressivity #根据自己的攻击力，攻击敌人就减掉敌人的生命值。

print(Garen.camp) #引用类的数据属性，该属性与所有对象/实例共享

print(Garen.attack) #引用类的函数属性，该属性也共享

print(Garen.\_\_dict\_\_) #查看类的属性字典，或者说名称空间

# Garen.name='Garen')#增加属性

# del Garen.name #删除属性

[复制代码](javascript:void(0);)

输出

Demacia

<function Garen.attack at 0x106acd400>

{'\_\_doc\_\_': None, 'attack': <function Garen.attack at 0x106acd400>, '\_\_module\_\_': '\_\_main\_\_', '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Garen' objects>, 'camp': 'Demacia', '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Garen' objects>}

* 类的实例化

类名加括号就是实例化，会自动触发\_\_init\_\_函数的运行，可以用它来为每个实例定制自己的特征

[复制代码](javascript:void(0);)

class Garen: #定义英雄盖伦的类，不同的玩家可以用它实例出自己英雄;

camp='Demacia' #所有玩家的英雄(盖伦)的阵营都是Demacia;

def \_\_init\_\_(self,nickname,aggressivity=58,life\_value=455): #英雄的初始攻击力58...;

self.nickname=nickname #为自己的盖伦起个别名;

self.aggressivity=aggressivity #英雄都有自己的攻击力;

self.life\_value=life\_value #英雄都有自己的生命值;

def attack(self,enemy): #普通攻击技能，enemy是敌人;

enemy.life\_value-=self.aggressivity #根据自己的攻击力，攻击敌人就减掉敌人的生命值。

g1=Garen('草丛伦') #就是在执行Garen.\_\_init\_\_(g1,'草丛伦')，然后执行\_\_init\_\_内的代码g1.nickname=‘草丛伦’等

print(g1)

print(g1.nickname)

[复制代码](javascript:void(0);)

输出

<\_\_main\_\_.Garen object at 0x10ee26e10>

草丛伦

注： self的作用是在实例化时自动将对象/实例本身传给\_\_init\_\_的第一个参数，self可以是任意名字

补充：

[复制代码](javascript:void(0);)

我们定义的类的属性到底存到哪里了？有两种方式查看

dir(类名)：查出的是一个名字列表

类名.\_\_dict\_\_:查出的是一个字典，key为属性名，value为属性值

二：特殊的类属性

类名.\_\_name\_\_# 类的名字(字符串)

类名.\_\_doc\_\_# 类的文档字符串

类名.\_\_base\_\_# 类的第一个父类(在讲继承时会讲)

类名.\_\_bases\_\_# 类所有父类构成的元组(在讲继承时会讲)

类名.\_\_dict\_\_# 类的字典属性

类名.\_\_module\_\_# 类定义所在的模块

类名.\_\_class\_\_# 实例对应的类(仅新式类中)

[复制代码](javascript:void(0);)

**3.对象的产生和引用**

* 对象是关于类而实际存在的一个例子，即类的实例化。

g1=Garen('草丛伦') #实例化类就是对象

print(g1)

print(g1.nickname)

print(type(g1)) #查看g1的类型就是类Garen

print(isinstance(g1,Garen)) #g1就是Garen的实例

输出

<\_\_main\_\_.Garen object at 0x108d55dd8>

草丛伦

<class '\_\_main\_\_.Garen'>

True

* 对象的引用只有一种：属性引用

对象/实例本身其实只有数据属性

[复制代码](javascript:void(0);)

class Garen: #定义英雄盖伦的类，不同的玩家可以用它实例出自己英雄;

camp='Demacia' #所有玩家的英雄(盖伦)的阵营都是Demacia;

def \_\_init\_\_(self,nickname,aggressivity=58,life\_value=455): #英雄的初始攻击力58...;

self.nickname=nickname #为自己的盖伦起个别名;

self.aggressivity=aggressivity #英雄都有自己的攻击力;

self.life\_value=life\_value #英雄都有自己的生命值;

def attack(self,enemy): #普通攻击技能，enemy是敌人;

enemy.life\_value-=self.aggressivity #根据自己的攻击力，攻击敌人就减掉敌人的生命值。

g1=Garen('草丛伦') #类的实例化就是对象

print(g1.nickname)　　#对象的属性引用

print(g1.aggressivity)

print(g1.life\_value)

[复制代码](javascript:void(0);)

输出

草丛伦

58

455

补充：

对象/实例本身只有数据属性，但是python的class机制会将类的函数绑定到对象上，称为对象的方法，或者叫绑定方法，绑定方法唯一绑定一个对象，同一个类的方法绑定到不同的对象上，属于不同的方法，内存地址都不会一样。

print(g1.attack)　　#对象的绑定方法

print(Garen.attack)　　#对象的绑定方法attack本质就是调用类的函数attack的功能，二者是一种绑定关系

输出

<bound method Garen.attack of <\_\_main\_\_.Garen object at 0x10809ee10>>

<function Garen.attack at 0x1081e3488>

对象的绑定方法的特别之处在于：obj.func()会把obj传给func的第一个参数。

**4.对象之间的交互**

仿照garen类再创建一个Riven类，用瑞雯攻击盖伦，完成对象交互

[复制代码](javascript:void(0);)

class Garen: #定义英雄盖伦的类，不同的玩家可以用它实例出自己英雄;

camp='Demacia' #所有玩家的英雄(盖伦)的阵营都是Demacia;

def \_\_init\_\_(self,nickname,aggressivity=58,life\_value=455): #英雄的初始攻击力58...;

self.nickname=nickname #为自己的盖伦起个别名;

self.aggressivity=aggressivity #英雄都有自己的攻击力;

self.life\_value=life\_value #英雄都有自己的生命值;

def attack(self,enemy): #普通攻击技能，enemy是敌人;

enemy.life\_value-=self.aggressivity #根据自己的攻击力，攻击敌人就用敌人的生命值减攻击力。

class Riven:

camp='Noxus' #所有玩家的英雄(锐雯)的阵营都是Noxus;

def \_\_init\_\_(self,nickname,aggressivity=54,life\_value=414): #英雄的初始攻击力54;

self.nickname=nickname #为自己的锐雯起个别名;

self.aggressivity=aggressivity #英雄都有自己的攻击力;

self.life\_value=life\_value #英雄都有自己的生命值;

def attack(self,enemy): #普通攻击技能，enemy是敌人;

enemy.life\_value-=self.aggressivity #根据自己的攻击力，攻击敌人就用敌人的生命值减攻击力。

g1=Garen('盖伦') #就是在执行Garen.\_\_init\_\_(g1,'草丛伦')，然后执行\_\_init\_\_内的代码g1.nickname=‘草丛伦’等

r1=Riven('瑞雯')

print(g1.life\_value)

print(r1.attack(g1)) #对象交互,瑞雯攻击盖伦

print(g1.life\_value)

[复制代码](javascript:void(0);)

输出

455

None

401

**5.类名称空间与对象/实例名称空间**

创建一个类就会创建一个类的名称空间，用来存储类中定义的所有名字，这些名字称为类的属性。

类有两种属性：数据属性和函数属性

其中类的数据属性是共享给所有对象的

定义在类内部的变量，是所有对象共有的，id全一样

print(id(r1.camp)) #本质就是在引用类的camp属性，二者id一样

print(id(Riven.camp))

4482776512

4482776512

而类的函数属性是绑定到所有对象的：

定义在类内部的函数，是绑定到所有对象的，是给对象来用，obj.func() 会把obj本身当做第一个参数出入

r1.attack就是在执行Riven.attack的功能，python的class机制会将Riven的函数属性attack绑定给r1，r1相当于拿到了一个指针，指向Riven类的attack功能,r1.attack()会将r1传给attack的第一个参数

print(id(r1.attack))

print(id(Riven.attack))

输出

4372850184

4374779288

注：

创建一个对象/实例就会创建一个对象/实例的名称空间，存放对象/实例的名字，称为对象/实例的属性

在obj.name会先从obj自己的名称空间里找name，找不到则去类中找，类也找不到就找父类...最后都找不到就抛出异常

print(g1.x) #先从g1.\_\_dict\_\_,找不到再找Garen.\_\_dict\_\_,找不到就会报错

**四.继承和派生**

**1.继承的定义**

继承是一种创建新类的方式，在python中，新建的类可以继承一个或多个父类，父类又可称为基类或超类，新建的类称为派生类或子类。

* 分类：单继承和多继承

[复制代码](javascript:void(0);)

class ParentClass1: #定义父类

pass

class ParentClass2: #定义父类

pass

class SubClass1(ParentClass1): #单继承，基类是ParentClass1，派生类是SubClass

pass

class SubClass2(ParentClass1,ParentClass2): #python支持多继承，用逗号分隔开多个继承的类

pass

[复制代码](javascript:void(0);)

* 查看继承

 \_\_base\_\_只查看从左到右继承的第一个子类，\_\_bases\_\_则是查看所有继承的父类

print(SubClass2.\_\_base\_\_)

print(SubClass2.\_\_bases\_\_)

输出

<class '\_\_main\_\_.ParentClass1'>

(<class '\_\_main\_\_.ParentClass1'>, <class '\_\_main\_\_.ParentClass2'>)

如果没有指定基类，python的类会默认继承object类，object是所有python类的基类，它提供了一些常见方法（如\_\_str\_\_）的实现。

print(ParentClass1.\_\_base\_\_)

输出

<class 'object'>

**2.继承与抽象**

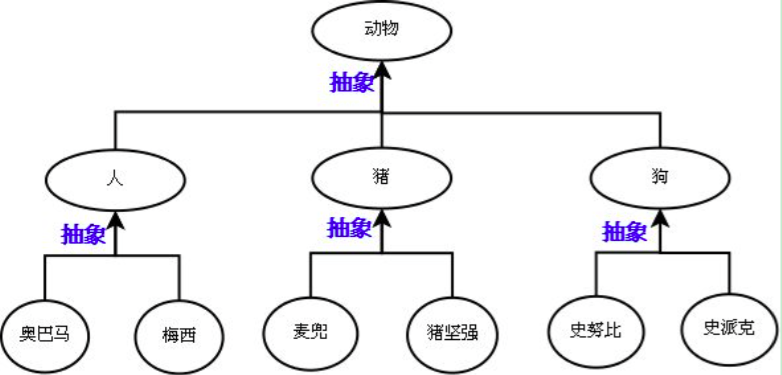
抽象即抽取类似或者说比较像的部分。

抽象分成两个层次：

1.将奥巴马和梅西这俩对象比较像的部分抽取成类；

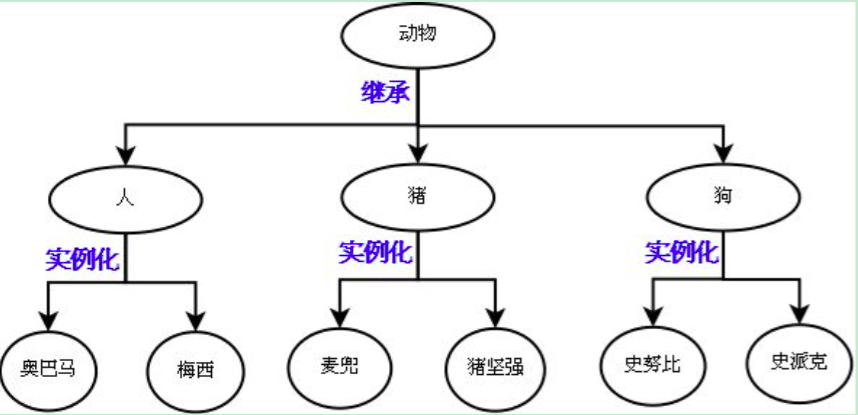
2.将人，猪，狗这三个类比较像的部分抽取成父类。

* 抽象：最主要的作用是划分类别（可以隔离关注点，降低复杂度）



* 继承：是基于抽象的结果，通过编程语言去实现它，肯定是先经历抽象这个过程，才能通过继承的方式去表达出抽象的结构。

抽象只是分析和设计的过程中，一个动作或者说一种技巧，通过抽象可以得到类。



**3.继承与重用性**

在开发程序的过程中，如果我们定义了一个类A，然后又想新建立另外一个类B，但是类B的大部分内容与类A的相同时

我们不可能从头开始写一个类B，这就用到了类的继承的概念。

通过继承的方式新建类B，让B继承A，B会‘遗传’A的所有属性(数据属性和函数属性)，实现代码重用

[复制代码](javascript:void(0);)

class Hero:

def \_\_init\_\_(self,nickname,aggressivity,life\_value):

self.nickname=nickname

self.aggressivity=aggressivity

self.life\_value=life\_value

def move\_forward(self):

print('%s move forward' %self.nickname)

def move\_backward(self):

print('%s move backward' %self.nickname)

def move\_left(self):

print('%s move forward' %self.nickname)

def move\_right(self):

print('%s move forward' %self.nickname)

def attack(self,enemy):

enemy.life\_value-=self.aggressivity

class Garen(Hero):

pass

class Riven(Hero):

pass

g1=Garen('草丛伦',100,300)

r1=Riven('锐雯雯',57,200)

print(g1.nickname)

print(r1.nickname)

print(g1.life\_value)

r1.attack(g1)

print(g1.life\_value)

[复制代码](javascript:void(0);)

输出

草丛伦

300

243

注意：

像g1.life\_value之类的属性引用，会先从实例中找life\_value然后去类中找，然后再去父类中找...直到最顶级的父类。

当然子类也可以添加自己新的属性或者在自己这里重新定义这些属性（不会影响到父类），需要注意的是，一旦重新定义了自己的属性且与父类重名，那么调用新增的属性时，就以自己为准了。

在子类定义新的属性，覆盖掉父类的属性,称为派生。

[复制代码](javascript:void(0);)

class Hero:

def \_\_init\_\_(self,nickname,aggressivity,life\_value):

self.nickname=nickname

self.aggressivity=aggressivity

self.life\_value=life\_value

def move\_forward(self):

print('%s move forward' %self.nickname)

def move\_backward(self):

print('%s move backward' %self.nickname)

def move\_left(self):

print('%s move forward' %self.nickname)

def move\_right(self):

print('%s move forward' %self.nickname)

def attack(self,enemy):

enemy.life\_value-=self.aggressivity

class Garen(Hero):

pass

class Riven(Hero):

camp='Noxus'

def attack(self,enemy): #在自己这里定义新的attack,不再使用父类的attack,且不会影响父类

print('from riven')

def fly(self): #在自己这里定义新的

print('%s is flying' %self.nickname)

g1=Garen('草丛伦',100,300)

r1=Riven('锐雯雯',57,200)

print(g1.nickname)

print(r1.nickname)

print(g1.life\_value)

r1.attack(g1)

print(g1.life\_value)

[复制代码](javascript:void(0);)

输出

草丛伦

锐雯雯

300

from riven

300

**4.组合与重用性**

软件重用的重要方式除了继承之外还有另外一种方式，即：组合。组合指的是，在一个类中以另外一个类的对象作为数据属性，称为类的组合。

[复制代码](javascript:void(0);)

class Equip: #武器装备类

def fire(self):

print('release Fire skill')

class Riven: #英雄Riven的类,一个英雄需要有装备,因而需要组合Equip类

camp='Noxus'

def \_\_init\_\_(self,nickname):

self.nickname=nickname

self.equip=Equip() #用Equip类产生一个装备,赋值给实例的equip属性

r1=Riven('锐雯雯')

r1.equip.fire() #可以使用组合的类产生的对象所持有的方法

[复制代码](javascript:void(0);)

输出

release Fire skill

对比：

组合与继承都是有效地利用已有类的资源的重要方式。但是二者的概念和使用场景皆不同，

* 1.继承的方式

通过继承建立了派生类与基类之间的关系，它是一种'是'的关系，比如白马是马，人是动物。

当类之间有很多相同的功能，提取这些共同的功能做成基类，用继承比较好，比如教授是老师

[复制代码](javascript:void(0);)

class Teacher:

def \_\_init\_\_(self,name,gender):

self.name=name

self.gender=gender

def teach(self):

print('teaching')

class Professor(Teacher):

pass

p1=Professor('hexin','male')

p1.teach()

[复制代码](javascript:void(0);)

输出

teaching

* 2.组合的方式

用组合的方式建立了类与组合的类之间的关系，它是一种‘有’的关系,比如教授有生日，教授教python课程。

[复制代码](javascript:void(0);)

class BirthDate:

def \_\_init\_\_(self,year,month,day):

self.year=year

self.month=month

self.day=day

class Couse:

def \_\_init\_\_(self,name,price,period):

self.name=name

self.price=price

self.period=period

class Teacher:

def \_\_init\_\_(self,name,gender):

self.name=name

self.gender=gender

def teach(self):

print('teaching')

class Professor(Teacher):

def \_\_init\_\_(self,name,gender,birth,course):

Teacher.\_\_init\_\_(self,name,gender)

self.birth=birth

self.course=course

p1=Professor('hexin','male',

BirthDate('1995','1','27'),

Couse('python','28000','4 months'))

print(p1.birth.year,p1.birth.month,p1.birth.day)

print(p1.course.name,p1.course.price,p1.course.period)

[复制代码](javascript:void(0);)

输出

1995 1 27

python 28000 4 months

当类之间有显著不同，并且较小的类是较大的类所需要的组件时，用组合比较好

**5.接口与归一化设计**

接口定义：接口就是一些方法特征的集合------接口是对抽象的抽象。

继承有两种用途：

①继承基类的方法，并且做出自己的改变或者扩展（代码重用）；

②声明某个子类兼容于某基类，定义一个接口类Interface，接口类中定义了一些接口名（就是函数名）且并未实现接口的功能，子类继承接口类，并且实现接口中的功能；

[复制代码](javascript:void(0);)

class Interface:#定义接口Interface类来模仿接口的概念，python中压根就没有interface关键字来定义一个接口。

def read(self): #定接口函数read

pass

def write(self): #定义接口函数write

pass

class Txt(Interface): #文本，具体实现read和write

def read(self):

print('文本数据的读取方法')

def write(self):

print('文本数据的读取方法')

class Sata(Interface): #磁盘，具体实现read和write

def read(self):

print('硬盘数据的读取方法')

def write(self):

print('硬盘数据的读取方法')

class Process(Interface):

def read(self):

print('进程数据的读取方法')

def write(self):

print('进程数据的读取方法')

[复制代码](javascript:void(0);)

实践中，

继承的第一种含义意义并不很大，甚至常常是有害的。因为它使得子类与基类出现强耦合。

继承的第二种含义叫“接口继承”。接口继承实质上是要求“做出一个良好的抽象，这个抽象规定了一个兼容接口，使得外部调用者无需关心具体细节，可一视同仁的处理实现了特定接口的所有对象”——这在程序设计上，叫做归一化。

归一化使得高层的外部使用者可以不加区分的处理所有接口兼容的对象集合——就好象linux的泛文件概念一样，所有东西都可以当文件处理，不必关心它是内存、磁盘、网络还是屏幕（当然，对底层设计者，当然也可以区分出“字符设备”和“块设备”，然后做出针对性的设计：细致到什么程度，视需求而定）。

* 为什么要使用接口？

接口提取了一群类共同的函数，可以把接口当做一个函数的集合，然后让子类去实现接口中的函数。

这么做的意义在于归一化，什么叫归一化，就是只要是基于同一个接口实现的类，那么所有的这些类产生的对象在使用时，从用法上来说都一样。

归一化，让使用者无需关心对象的类是什么，只需要的知道这些对象都具备某些功能就可以了，这极大地降低了使用者的使用难度。

比如：我们定义一个动物接口，接口里定义了有跑、吃、呼吸等接口函数，这样老鼠的类去实现了该接口，松鼠的类也去实现了该接口，由二者分别产生一只老鼠和一只松鼠送到你面前，即便是你分别不到底哪只是什么鼠你肯定知道他俩都会跑，都会吃，都能呼吸。

再比如：我们有一个汽车接口，里面定义了汽车所有的功能，然后由本田汽车的类，奥迪汽车的类，大众汽车的类，他们都实现了汽车接口，这样就好办了，大家只需要学会了怎么开汽车，那么无论是本田，还是奥迪，还是大众我们都会开了，开的时候根本无需关心我开的是哪一类车，操作手法（函数调用）都一样。

**6.抽象类**

* 抽象类

抽象类是一个特殊的类，它的特殊之处在于只能被继承，不能被实例化

* 为什么要有抽象类

如果说类是从一堆对象中抽取相同的内容而来的，那么抽象类就是从一堆类中抽取相同的内容而来的，内容包括数据属性和函数属性。

比如我们有香蕉的类，有苹果的类，有桃子的类，从这些类抽取相同的内容就是水果这个抽象的类，你吃水果时，要么是吃一个具体的香蕉，要么是吃一个具体的桃子。你永远无法吃到一个叫做水果的东西。

从设计角度去看，如果类是从现实对象抽象而来的，那么抽象类就是基于类抽象而来的。

从实现角度来看，抽象类与普通类的不同之处在于：抽象类中只能有抽象方法（没有实现功能），该类不能被实例化，只能被继承，且子类必须实现抽象方法。这一点与接口有点类似，但其实是不同的

* 在python中实现抽象类

[复制代码](javascript:void(0);)

import abc #利用abc模块实现抽象类

class All\_file(metaclass=abc.ABCMeta):

all\_type='file'

@abc.abstractmethod #定义抽象方法，无需实现功能

def read(self):

'子类必须定义读功能'

pass

@abc.abstractmethod #定义抽象方法，无需实现功能

def write(self):

'子类必须定义写功能'

pass

# class Txt(All\_file):

# pass

#

# t1=Txt() #报错,子类没有定义抽象方法

class Txt(All\_file): #子类继承抽象类，但是必须定义read和write方法

def read(self):

print('文本数据的读取方法')

def write(self):

print('文本数据的读取方法')

class Sata(All\_file): #子类继承抽象类，但是必须定义read和write方法

def read(self):

print('硬盘数据的读取方法')

def write(self):

print('硬盘数据的读取方法')

class Process(All\_file): #子类继承抽象类，但是必须定义read和write方法

def read(self):

print('进程数据的读取方法')

def write(self):

print('进程数据的读取方法')

wenbenwenjian=Txt()

yingpanwenjian=Sata()

jinchengwenjian=Process()

#这样大家都是被归一化了,也就是一切皆文件的思想

wenbenwenjian.read()

yingpanwenjian.write()

jinchengwenjian.read()

print(wenbenwenjian.all\_type)

print(yingpanwenjian.all\_type)

print(jinchengwenjian.all\_type)

[复制代码](javascript:void(0);)

输出

文本数据的读取方法

硬盘数据的读取方法

进程数据的读取方法

file

file

file

 补充：Python的abc模块

abc模块作用：Python本身不提供抽象类和接口机制，要想实现抽象类，可以借助abc模块。ABC是Abstract Base Class的缩写。

模块中的类和函数：

abc.ABCMeta　　这是用来生成抽象基础类的元类。由它生成的类可以被直接继承。

abc.abstractmethod(function)　　表明抽象方法的生成器

abc.abstractproperty([fget[,fset[,fdel[,doc]]]])　　表明一个抽象属性

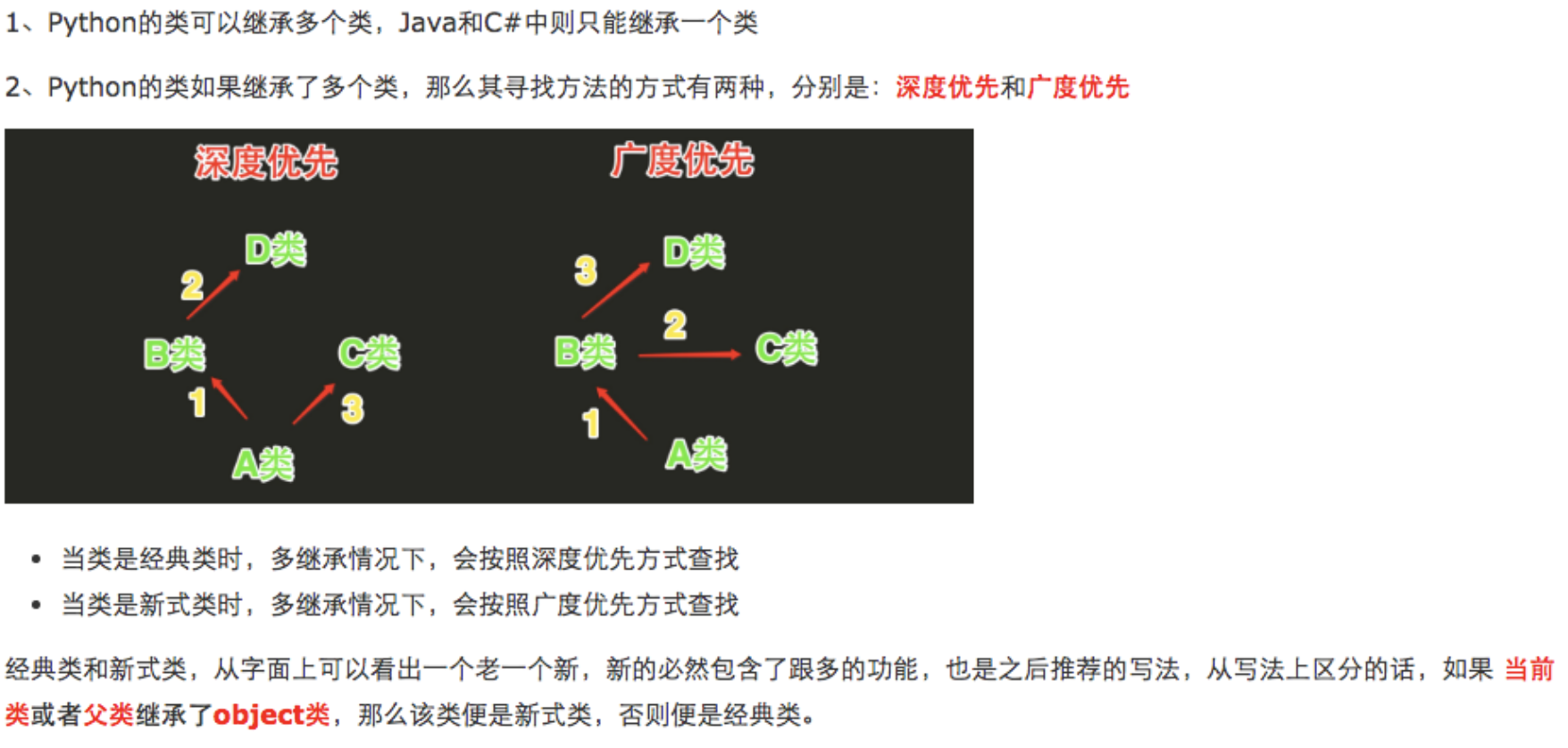
* 抽象类与接口

抽象类的本质还是类，指的是一组类的相似性，包括数据属性（如all\_type）和函数属性（如read、write），而接口只强调函数属性的相似性。

抽象类是一个介于类和接口直接的一个概念，同时具备类和接口的部分特性，可以用来实现归一化设计 。

**7.继承的顺序**

* 继承顺序



[复制代码](javascript:void(0);)

class A(object):

def test(self):

print('from A')

class B(A):

def test(self):

print('from B')

class C(A):

def test(self):

print('from C')

class D(B):

def test(self):

print('from D')

class E(C):

def test(self):

print('from E')

class F(D,E):

# def test(self):

# print('from F')

pass

f1=F()

f1.test()

print(F.\_\_mro\_\_) #只有新式才有这个属性可以查看线性列表，经典类没有这个属性

#新式类继承顺序:F->D->B->E->C->A

#经典类继承顺序:F->D->B->A->E->C

#python3中统一都是新式类

#pyhon2中才分新式类与经典类

[复制代码](javascript:void(0);)

输出

from D

(<class '\_\_main\_\_.F'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.E'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>)

* 继承原理

对于你定义的每一个类，python会计算出一个方法解析顺序(MRO)列表，这个MRO列表就是一个简单的所有基类的线性顺序列表，例如

>>> F.mro() #等同于F.\_\_mro\_\_

[<class '\_\_main\_\_.F'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.E'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

为了实现继承,python会在MRO列表上从左到右开始查找基类,直到找到第一个匹配这个属性的类为止。  
而这个MRO列表的构造是通过一个C3线性化算法来实现的。我们不去深究这个算法的数学原理,它实际上就是合并所有父类的MRO列表并遵循如下三条准则:  
1.子类会先于父类被检查  
2.多个父类会根据它们在列表中的顺序被检查  
3.如果对下一个类存在两个合法的选择,选择第一个父类

**8.子类中调用父类的方法**

子类继承了父类的方法，然后想进行修改，注意了是基于原有的基础上修改，那么就需要在子类中调用父类的方法

* 方法一：父类名.父类方法()

[复制代码](javascript:void(0);)

class Vehicle: #定义交通工具类

Country='China'

def \_\_init\_\_(self,name,speed,load,power):

self.name=name

self.speed=speed

self.load=load

self.power=power

def run(self):

print('开动啦...')

class Subway(Vehicle): #地铁

def \_\_init\_\_(self,name,speed,load,power,line):

Vehicle.\_\_init\_\_(self,name,speed,load,power)

self.line=line

def run(self):

print('地铁%s号线欢迎您' %self.line)

Vehicle.run(self)

line13=Subway('中国地铁','180m/s','1000人/箱','电',13)

line13.run()

[复制代码](javascript:void(0);)

输出

地铁13号线欢迎您

开动啦...

* 方法二：super()

[复制代码](javascript:void(0);)

class Vehicle: #定义交通工具类

Country='China'

def \_\_init\_\_(self,name,speed,load,power):

self.name=name

self.speed=speed

self.load=load

self.power=power

def run(self):

print('开动啦...')

class Subway(Vehicle): #地铁

def \_\_init\_\_(self,name,speed,load,power,line):

#super(Subway,self) 就相当于实例本身 在python3中super()等同于super(Subway,self)

super().\_\_init\_\_(name,speed,load,power)

self.line=line

def run(self):

print('地铁%s号线欢迎您' %self.line)

super(Subway,self).run()

class Mobike(Vehicle):#摩拜单车

pass

line13=Subway('中国地铁','180m/s','1000人/箱','电',13)

line13.run()

[复制代码](javascript:void(0);)

注意：

当你使用super()函数时,Python会在MRO列表上继续搜索下一个类。只要每个重定义的方法统一使用super()并只调用它一次,那么控制流最终会遍历完整个MRO列表,每个方法也只会被调用一次

使用super调用的所有属性，都是从MRO列表当前的位置往后找，千万不要通过看代码去找继承关系，一定要看MRO列表

**五、多态与多态性**

**1.多态**

多态指的是一类事物有多种形态，（一个抽象类有多个子类，因而多态的概念依赖于继承）。

序列类型有多种形态：字符串，列表，元组。

动物有多种形态：人，狗，猪

[复制代码](javascript:void(0);)

import abc

class Animal(metaclass=abc.ABCMeta): #同一类事物:动物

@abc.abstractmethod

def talk(self):

pass

class People(Animal): #动物的形态之一:人

def talk(self):

print('say hello')

class Dog(Animal): #动物的形态之二:狗

def talk(self):

print('say wangwang')

class Pig(Animal): #动物的形态之三:猪

def talk(self):

print('say aoao')

[复制代码](javascript:void(0);)

文件有多种形态：文本文件，可执行文件

[复制代码](javascript:void(0);)

import abc

class File(metaclass=abc.ABCMeta): #同一类事物:文件

@abc.abstractmethod

def click(self):

pass

class Text(File): #文件的形态之一:文本文件

def click(self):

print('open file')

class ExeFile(File): #文件的形态之二:可执行文件

def click(self):

print('execute file')

[复制代码](javascript:void(0);)

**2.多态性**

多态性是指具有不同功能的函数可以使用相同的函数名，这样就可以用一个函数名调用不同功能的函数。

比如：老师.下课铃响了（），学生.下课铃响了()，老师执行的是下班操作，学生执行的是放学操作，虽然二者消息一样，但是执行的效果不同。

* 静态多态性：如任何类型都可以用运算符+进行运算
* 动态多态性：

[复制代码](javascript:void(0);)

import abc

class Animal(metaclass=abc.ABCMeta): #同一类事物:动物

@abc.abstractmethod

def talk(self):

pass

class People(Animal): #动物的形态之一:人

def talk(self):

print('say hello')

class Dog(Animal): #动物的形态之二:狗

def talk(self):

print('say wangwang')

class Pig(Animal): #动物的形态之三:猪

def talk(self):

print('say aoao')

def func(animal):

animal.talk()

people1=People()

pig1=Pig()

dog1=Dog()

func(people1)

func(pig1)

func(dog1)

[复制代码](javascript:void(0);)

多态性是‘一个接口（函数func），多种实现（如f.click()）’

**3.为什么要使用多态性？**

* 增加了程序的灵活性

以不变应万变，不论对象千变万化，使用者都是同一种形式去调用，如func(animal)

* 增加了程序可扩展性

通过继承animal类创建了一个新的类，使用者无需更改自己的代码，还是用func(animal)去调用

[复制代码](javascript:void(0);)

import abc

class Animal(metaclass=abc.ABCMeta): #同一类事物:动物

@abc.abstractmethod

def talk(self):

pass

class Cat(Animal): #属于动物的另外一种形态：猫

def talk(self):

print('say miao')

def func(animal): #对于使用者来说，自己的代码根本无需改动

animal.talk()

cat1=Cat() #实例出一只猫

func(cat1) #甚至连调用方式也无需改变，就能调用猫的talk功能

[复制代码](javascript:void(0);)

输出

say miao

这样我们新增了一个形态Cat，由Cat类产生的实例cat1，使用者可以在完全不需要修改自己代码的情况下。使用和人、狗、猪一样的方式调用cat1的talk方法，即func(cat1)

**六、封装**

**1.封装的定义**

“封装”就是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（即类）。

封装的目的是增强安全性和简化编程，使用者不必了解具体的实现细节，而只是要通过外部接口，一特定的访问权限来使用类的成员。

保护隐私和隔离复杂度。

**2.封装的层面**

第一个层面的封装（什么都不用做）：创建类和对象会分别创建二者的名称空间，我们只能用类名.或者obj.的方式去访问里面的名字，这本身就是一种封装。

>>> r1.nickname

'草丛伦'

>>> Riven.camp

'Noxus'

注意：对于这一层面的封装（隐藏），类名.和实例名.就是访问隐藏属性的接口

第二个层面的封装：类中把某些属性和方法隐藏起来(或者说定义成私有的)，只在类的内部使用、外部无法访问，或者留下少量接口（函数）供外部访问。

在python中用双下划线的方式实现隐藏属性（设置成私有的），类中所有双下划线开头的名称如\_\_x都会自动变形成：\_类名\_\_x的形式**：**

[复制代码](javascript:void(0);)

class Foo:

\_\_x=1 #\_Foo\_\_x

def \_\_test(self): #\_Foo\_\_test

print('from test')

print(Foo.\_\_dict\_\_)

print(Foo.\_Foo\_\_x)

[复制代码](javascript:void(0);)

输出

{'\_Foo\_\_test': <function Foo.\_\_test at 0x10e842400>, '\_\_module\_\_': '\_\_main\_\_', '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Foo' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Foo' objects>, '\_\_doc\_\_': None, '\_Foo\_\_x': 1}

1

[复制代码](javascript:void(0);)

class Teacher:

def \_\_init\_\_(self,name,age):

self.\_\_name=name

self.\_\_age=age

def tell\_info(self):

print('姓名:%s,年龄:%s' %(self.\_\_name,self.\_\_age))

def set\_info(self,name,age):

if not isinstance(name,str):

raise TypeError('姓名必须是字符串类型')

if not isinstance(age,int):

raise TypeError('年龄必须是整型')

self.\_\_name=name

self.\_\_age=age

t=Teacher('hexin',18)

t.tell\_info()

[复制代码](javascript:void(0);)

输出

姓名:hexin,年龄:18

这种自动变形的特点：

1.类中定义的\_\_x只能在内部使用，如self.\_\_x，引用的就是变形的结果。

2.这种变形其实正是针对外部的变形，在外部是无法通过\_\_x这个名字访问到的。

3.在子类定义的\_\_x不会覆盖在父类定义的\_\_x，因为子类中变形成了：\_子类名\_\_x,而父类中变形成了：\_父类名\_\_x，即双下滑线开头的属性在继承给子类时，子类是无法覆盖的。

注意：对于这一层面的封装（隐藏），我们需要在类中定义一个函数（接口函数）在它内部访问被隐藏的属性，然后外部就可以使用了.

这种变形需要注意的问题是：

1.这种机制也并没有真正意义上限制我们从外部直接访问属性，知道了类名和属性名就可以拼出名字：\_类名\_\_属性，然后就可以访问了，如a.\_A\_\_N

2.变形的过程只在类的定义是发生一次,在定义后的赋值操作，不会变形

3.在继承中，父类如果不想让子类覆盖自己的方法，可以将方法定义为私有的

**3.特性（property）**

property是一种特殊的属性，访问它时会执行一段功能（函数）然后返回值

[复制代码](javascript:void(0);)

import math

class Circle:

def \_\_init\_\_(self,radius): #圆的半径radius

self.radius=radius

@property

def area(self):

return math.pi \* self.radius\*\*2 #计算面积

@property

def perimeter(self):

return 2\*math.pi\*self.radius #计算周长

c=Circle(10)

print(c.radius)

print(c.area) #可以向访问数据属性一样去访问area,会触发一个函数的执行,动态计算出一个值

print(c.perimeter)

[复制代码](javascript:void(0);)

输出

10

314.1592653589793

62.83185307179586

注意：此时的特性arear和perimeter不能被赋值

c.area=3 #为特性area赋值

'''

抛出异常:

AttributeError: can't set attribute

'''

为什么要使用特性？

将一个类的函数定义成特性以后，对象再去使用的时候obj.name,根本无法察觉自己的name是执行了一个函数然后计算出来的，这种特性的使用方式遵循了统一访问的原则

补充：面向对象的封装有三种方式:

【public】  
这种其实就是不封装,是对外公开的  
【protected】  
这种封装方式对外不公开,但对朋友(friend)或者子类公开  
【private】  
这种封装对谁都不公开

python并没有在语法上把它们三个内建到自己的class机制中，在C++里一般会将所有的所有的数据都设置为私有的，然后提供set和get方法（接口）去设置和获取，在python中通过property方法可以实现

[复制代码](javascript:void(0);)

class Foo:

def \_\_init\_\_(self,val):

self.\_\_NAME=val #将所有的数据属性都隐藏起来

@property

def name(self):

return self.\_\_NAME #obj.name访问的是self.\_\_NAME(这也是真实值的存放位置)

@name.setter

def name(self,value):

if not isinstance(value,str): #在设定值之前进行类型检查

raise TypeError('%s must be str' %value)

self.\_\_NAME=value #通过类型检查后,将值value存放到真实的位置self.\_\_NAME

@name.deleter

def name(self):

raise TypeError('Can not delete')

f=Foo('egon')

print(f.name)

# f.name=10 #抛出异常'TypeError: 10 must be str'

del f.name #抛出异常'TypeError: Can not delete'

[复制代码](javascript:void(0);)

一种property的古老用法

[复制代码](javascript:void(0);)

class Foo:

def \_\_init\_\_(self,val):

self.\_\_NAME=val #将所有的数据属性都隐藏起来

def getname(self):

return self.\_\_NAME #obj.name访问的是self.\_\_NAME(这也是真实值的存放位置)

def setname(self,value):

if not isinstance(value,str): #在设定值之前进行类型检查

raise TypeError('%s must be str' %value)

self.\_\_NAME=value #通过类型检查后,将值value存放到真实的位置self.\_\_NAME

def delname(self):

raise TypeError('Can not delete')

name=property(getname,setname,delname) #不如装饰器的方式清晰

[复制代码](javascript:void(0);)

**4.封装与扩展性**

封装在于明确区分内外，使得类实现者可以修改封装内的东西而不影响外部调用者的代码；而外部使用用者只知道一个接口(函数)，只要接口（函数）名、参数不变，使用者的代码永远无需改变。

[复制代码](javascript:void(0);)

#类的设计者

class Room:

def \_\_init\_\_(self,name,owner,width,length,high):

self.name=name

self.owner=owner

self.\_\_width=width

self.\_\_length=length

self.\_\_high=high

def tell\_area(self): #对外提供的接口，隐藏了内部的实现细节，此时我们想求的是面积

return self.\_\_width \* self.\_\_length

[复制代码](javascript:void(0);)

#使用者

>>> r1=Room('卧室','hexin',20,20,20)

>>> r1.tell\_area() #使用者调用接口tell\_area

400

要计算体积

[复制代码](javascript:void(0);)

#类的设计者，轻松的扩展了功能，而类的使用者完全不需要改变自己的代码

class Room:

def \_\_init\_\_(self,name,owner,width,length,high):

self.name=name

self.owner=owner

self.\_\_width=width

self.\_\_length=length

self.\_\_high=high

def tell\_area(self): #对外提供的接口，隐藏内部实现，此时我们想求的是体积,内部逻辑变了,只需求修该下列一行就可以很简答的实现,而且外部调用感知不到,仍然使用该方法，但是功能已经变了

return self.\_\_width \* self.\_\_length \* self.\_\_high

[复制代码](javascript:void(0);)

对于仍然在使用tell\_area接口的人来说，根本无需改动自己的代码，就可以用上新功能

>>> r1.tell\_area()

8000

**七、绑定方法与非绑定方法**

**1.定义**

类中定义的函数分成两大类：

* 绑定方法（绑定给谁，谁来调用就自动将它本身当作第一个参数传入）：

1. 绑定到类的方法：用classmethod装饰器装饰的方法。为类量身定制

类.boud\_method(),自动将类当作第一个参数传入 （其实对象也可调用，但仍将类当作第一个参数传入）

2. 绑定到对象的方法：没有被任何装饰器装饰的方法。 为对象量身定制

对象.boud\_method(),自动将对象当作第一个参数传入  （属于类的函数，类可以调用，但是必须按照函数的规则来，没有自动传值那么一说）

* 非绑定方法：用staticmethod装饰器装饰的方法

1. 不与类或对象绑定，类和对象都可以调用，但是没有自动传值那么一说。就是一个普通工具而已

注意：与绑定到对象方法区分开，在类中直接定义的函数，没有被任何装饰器装饰的，都是绑定到对象的方法，可不是普通函数，对象调用该方法会自动传值，而staticmethod装饰的方法，不管谁来调用，都没有自动传值一说

**2.staticmethod**

statimethod不与类或对象绑定，谁都可以调用，没有自动传值效果，python为我们内置了函数staticmethod来把类中的函数定义成静态方法。

[复制代码](javascript:void(0);)

import hashlib

import time

class MySQL:

def \_\_init\_\_(self,host,port):

self.id=self.create\_id()

self.host=host

self.port=port

@staticmethod

def create\_id(): #就是一个普通工具

m=hashlib.md5(str(time.clock()).encode('utf-8'))

return m.hexdigest()

print(MySQL.create\_id) #<function MySQL.create\_id at 0x0000000001E6B9D8> #查看结果为普通函数

conn=MySQL('127.0.0.1',3306)

print(conn.create\_id) #<function MySQL.create\_id at 0x00000000026FB9D8> #查看结果为普通函数

[复制代码](javascript:void(0);)

**3.classmethod**

classmehtod是给类用的，即绑定到类，类在使用时会将类本身当做参数传给类方法的第一个参数（即便是对象来调用也会将类当作第一个参数传入），python为我们内置了函数classmethod来把类中的函数定义成类方法

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

HOST='127.0.0.1'

PORT=3306

DB\_PATH=r'C:\Users\Administrator\PycharmProjects\test\面向对象编程\test1\db'

settings.py

[复制代码](javascript:void(0);)

import settings

import hashlib

import time

class MySQL:

def \_\_init\_\_(self,host,port):

self.host=host

self.port=port

@classmethod

def from\_conf(cls):

print(cls)

return cls(settings.HOST,settings.PORT)

print(MySQL.from\_conf) #<bound method MySQL.from\_conf of <class '\_\_main\_\_.MySQL'>>

conn=MySQL.from\_conf()

print(conn.host,conn.port)

conn.from\_conf() #对象也可以调用，但是默认传的第一个参数仍然是类

[复制代码](javascript:void(0);)

比较staticmethod和classmethod的区别

[复制代码](javascript:void(0);)

import settings

class MySQL:

def \_\_init\_\_(self,host,port):

self.host=host

self.port=port

@staticmethod

def from\_conf():

return MySQL(settings.HOST,settings.PORT)

# @classmethod

# def from\_conf(cls):

# return cls(settings.HOST,settings.PORT)

def \_\_str\_\_(self):

return '就不告诉你'

class Mariadb(MySQL):

def \_\_str\_\_(self):

return '主机:%s 端口:%s' %(self.host,self.port)

m=Mariadb.from\_conf()

print(m) #我们的意图是想触发Mariadb.\_\_str\_\_,但是结果触发了MySQL.\_\_str\_\_的执行，打印就不告诉你

[复制代码](javascript:void(0);)

定义MySQL类

　　1.对象有id、host、port三个属性

　　2.定义工具create\_id，在实例化时为每个对象随机生成id，保证id唯一

　　3.提供两种实例化方式，方式一：用户传入host和port 方式二：从配置文件中读取host和port进行实例化

　　4.为对象定制方法，save和get，save能自动将对象序列化到文件中，文件名为id号，文件路径为配置文件中DB\_PATH;get方法用来从文件中反序列化出对象

[复制代码](javascript:void(0);)

#settings.py内容

'''

HOST='127.0.0.1'

PORT=3306

DB\_PATH=r'C:\Users\Administrator\PycharmProjects\test\面向对象编程\test1\db'

'''

import settings

import hashlib

import time

import random

import pickle

import os

class MySQL:

def \_\_init\_\_(self,host,port):

self.id=self.create\_id()

self.host=host

self.port=port

def save(self):

file\_path=r'%s%s%s' %(settings.DB\_PATH,os.sep,self.id)

pickle.dump(self,open(file\_path,'wb'))

def get(self):

file\_path = r'%s%s%s' % (settings.DB\_PATH, os.sep, self.id)

return pickle.load(open(file\_path,'rb'))

@staticmethod

def create\_id():

m=hashlib.md5(str(time.clock()).encode('utf-8')) #查看clock源码注释，指的是cpu真实时间，不要用time.time()，否则会出现id重复

return m.hexdigest()

@classmethod

def from\_conf(cls):

print(cls)

return cls(settings.HOST,settings.PORT)

# print(MySQL.from\_conf) #<bound method MySQL.from\_conf of <class '\_\_main\_\_.MySQL'>>

conn=MySQL.from\_conf()

# print(conn.id)

print(conn.create\_id())

print(MySQL.create\_id())

# print(conn.id)

# conn.save()

# obj=conn.get()

# print(obj.id)

[复制代码](javascript:void(0);)

小练习

[复制代码](javascript:void(0);)

import time

class Date:

def \_\_init\_\_(self,year,month,day):

self.year=year

self.month=month

self.day=day

@staticmethod

def now(): #用Date.now()的形式去产生实例,该实例用的是当前时间

t=time.localtime() #获取结构化的时间格式

return Date(t.tm\_year,t.tm\_mon,t.tm\_mday) #新建实例并且返回

@staticmethod

def tomorrow():#用Date.tomorrow()的形式去产生实例,该实例用的是明天的时间

t=time.localtime(time.time()+86400)

return Date(t.tm\_year,t.tm\_mon,t.tm\_mday)

a=Date('1987',11,27) #自己定义时间

b=Date.now() #采用当前时间

c=Date.tomorrow() #采用明天的时间

print(a.year,a.month,a.day)

print(b.year,b.month,b.day)

print(c.year,c.month,c.day)

[复制代码](javascript:void(0);)

输出

1987 11 27

2017 6 15

2017 6 16

[复制代码](javascript:void(0);)

import time

class Date:

def \_\_init\_\_(self,year,month,day):

self.year=year

self.month=month

self.day=day

@staticmethod

def now():

t=time.localtime()

return Date(t.tm\_year,t.tm\_mon,t.tm\_mday)

class EuroDate(Date):

def \_\_str\_\_(self):

return 'year:%s month:%s day:%s' %(self.year,self.month,self.day)

e=EuroDate.now()

print(e) #我们的意图是想触发EuroDate.\_\_str\_\_,但是结果为

'''

输出结果:

<\_\_main\_\_.Date object at 0x1013f9d68>

'''

因为e就是用Date类产生的,所以根本不会触发EuroDate.\_\_str\_\_,解决方法就是用classmethod

import time

class Date:

def \_\_init\_\_(self,year,month,day):

self.year=year

self.month=month

self.day=day

# @staticmethod

# def now():

# t=time.localtime()

# return Date(t.tm\_year,t.tm\_mon,t.tm\_mday)

@classmethod #改成类方法

def now(cls):

t=time.localtime()

return cls(t.tm\_year,t.tm\_mon,t.tm\_mday) #哪个类来调用,即用哪个类cls来实例化

class EuroDate(Date):

def \_\_str\_\_(self):

return 'year:%s month:%s day:%s' %(self.year,self.month,self.day)

e=EuroDate.now()

print(e) #我们的意图是想触发EuroDate.\_\_str\_\_,此时e就是由EuroDate产生的,所以会如我们所愿

'''

输出结果:

year:2017 month:3 day:3

'''

[复制代码](javascript:void(0);)