# 第十章 面向对象三大特征

本节所讲内容：

10.1 类的内置属性和方法

10.2 封装

10.3 继承

10.4 方法重写

10.5 多态

10.6 魔法方法

10.7 单例模式

## 10.1 类的内置属性和方法

### 10.1.1 类的内置属性

\_\_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（包含了一个由所有父类组成的元组）

class Parent:  
 *'所有人类的基类'* def \_\_init\_\_(self,name,age):  
 self.name = name  
 self.age = age  
 def set\_info(self):  
 print('set\_info')  
 def print\_info(self):  
 print(self.name,self.age)  
p = Parent('for',18)  
p.print\_info()  
print('类的属性',Parent.\_\_dict\_\_)  
print('类的文档字符串',Parent.\_\_doc\_\_)  
print('类的姓名',Parent.\_\_name\_\_)  
print('类定义所在的模块',Parent.\_\_module\_\_)  
print('类的所有父类构成的元素',Parent.\_\_bases\_\_)

运行结果如下：

for 18

类的属性 {'\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Parent' objects>, 'set\_info': <function Parent.set\_info at 0x000002A360CD76A8>, 'print\_info': <function Parent.print\_info at 0x000002A369B9E9D8>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Parent' objects>, '\_\_module\_\_': '\_\_main\_\_', '\_\_init\_\_': <function Parent.\_\_init\_\_ at 0x000002A360CD7840>, '\_\_doc\_\_': '所有人类的基类'}

类的文档字符串 所有人类的基类

类的姓名 Parent

类定义所在的模块 \_\_main\_\_

类的所有父类构成的元素 (<class 'object'>,)10.1.2 类方法

1. 类方法

类方法是类对象所拥有的方法，需要用修饰器@classmethod来标识其为类方法.

对于类方法，第一个参数接收的必须是类对象，一般以cls作为第一个参数（当然可以用其他名称的变量作为其第一个参数，但是大部分人都习惯以'cls'作为第一个参数的名字，就最好用'cls'了）

访问类方法：能够通过实例对象和类对象（类名）去访问。

class People(object):  
 *#类属性* country = 'china'  
*#类方法，用classmethod来进行修饰* @classmethod  
 def getCountry(cls):  
 return cls.country  
p = People()  
print(p.getCountry()) *# 可以用做实例对象引用*print(People.getCountry()) *# 可以通过类对象引用*

运行结果如下：

china

china

类方法还有一个用途就是可以对类属性进行修改：

class People(object):  
 *#类属性* country = 'china'  
 *#类方法，用classmethod来进行修饰* @classmethod  
 *# cls 用来接收类对象例如：People* def getCountry(cls):  
 return cls.country  
 @classmethod  
 *# cls 用来接收类对象例如：People* def setCountry(cls,country):  
 *#更改类属性，People.country = country* cls.country = country  
p = People()  
*#可以用过实例对象引用*print (p.getCountry())  
*#可以通过类对象引用*print (People.getCountry())  
*#更改类属性为japan*p.setCountry('japan')  
*#可以用过实例对象引用*print (p.getCountry())  
*#可以通过类对象引用*print (People.getCountry())

运行结果如下：

china

china

japan

japan

分析：结果显示在用类方法对类属性修改之后，通过类对象和实例对象访问都发生了改变。

### 10.1.3 静态方法

需要通过修饰器@staticmethod来进行修饰，另外注意静态方法不需要定义cls参数来接收类对象，它是通过类对象直接调用类属性。

class People(object):  
 country = 'china'  
 @staticmethod*#静态方法* def getCountry():  
 print(People.country)  
 def test(self):  
 print(self.country)  
*#类对象访问静态方法*People.getCountry()  
*#实例化一个对象*P = People()  
*#调用静态方法*P.getCountry()  
*#调用实例方法*P.test()

运行结果如下：

china

china

china

思考题：

简单的游戏，利用一个类创建Game类！

要求：

1. 玩家可以自己输入自己的名字和姓名。
2. 登录后开始游戏就会打印“帮助信息”
3. 帮助信息的下方会出现最高分
4. 启动游戏会打印欢迎字句
5. 利用类方法，静态方法完成

class Game:

def \_\_init\_\_(self,player\_name):

self.player\_name = player\_name

*#类对象直接调用*

Game.show\_help()

Game.show\_top\_score()

*#静态方法*

@staticmethod

def show\_help():

print('--------------')

print('|查看帮助信息|')

print('--------------')

*#类方法*

@classmethod

def show\_top\_score(cls):

print('--------------')

print('|最高分数:1111|')

print('--------------')

def start\_name(self):

print('%s开始游戏'%(self.player\_name))

tom = Game('for')

tom.start\_name()

运行结果如下：

--------------

|查看帮助信息|

--------------

--------------

|最高分数:1111|

--------------

for开始游戏

对方法的总结：

1、从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，那么通过cls引用的必定是类对象的属性和方法；

2、而实例方法的第一个参数是实例对象self，那么通过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。

3、静态方法@staticmethod中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用

|  |  |  |
| --- | --- | --- |
| 方法类别 | 语法 | 描述 |
| 类方法 | @classmethod | 第一个形参cls。默认传递 |
| 静态方法 | @staticmethod | 没有默认传递的形参 |
| 对象方法（成员方法） | def  方法名 | 第一个形参self，默认传递 |

### 10.1.4 类的私有属性和方法

给属性前面两个下划线代表私有化，属性和方法同样适用。

例如：

私有类属性(\_\_age）；

私有实例属性（self.\_\_name）；

私有实例方法（\_\_func()）。

我们要牢记私有的属性和方法只能在类的内部调用不能在外部调用。

让我们看下具体的例子

1、私有类属性

class Parent:  
 *#私有类属性* \_\_age= 10  
 def hello(self):  
 print(Parent.\_\_age)  
p = Parent()  
p.hello()

运行结果如下：

10

外部访问：

class Parent:

*#私有类属性*

\_\_age= 10

def hello(self):

print(Parent.\_\_age)

p = Parent()

p.hello()

print(Parent.\_\_age)

报错：  
Traceback (most recent call last):

File "E:/workspace/PIL\_test/01\_class\_test.py", line 172, in <module>

print(Parent.\_\_age)

AttributeError: type object 'Parent' has no attribute '\_\_age'

2、私有实例属性

class Parent:  
 def \_\_init\_\_(self,name,age):  
 self.\_\_name = name  
 self.age = age  
 def hello(self):  
 print(self.\_\_name)  
*#创建P这个对象*P = Parent('xuegod',18)  
*#调用对象的属性*P.hello() *# 可以*print(P.\_\_name) *#报错*

运行结果如下：  
xuegod

File "E:/workspace/importTest/test02.py", line 45, in <module>

print(P.\_\_name)#报错

AttributeError: 'Parent' object has no attribute '\_\_name'

3、私有实例方法

给方法前面加\_\_(类的私有化，只能在类的内部调用)

class Parent:  
 def \_\_init\_\_(self,name,age):  
 self.name = name  
 self.age = age  
 def \_\_func(self):  
 print('我是私有实例方法')  
 def hello(self):  
 self.\_\_func()  
*#创建P这个对象*P = Parent('xuegod',18)  
*#调用对象的属性*P.hello() *#可以*P.\_\_func() *#报错*

运行结果如下：

我是私有实例方法

Traceback (most recent call last):

File "E:/workspace/importTest/test02.py", line 47, in <module>

P.\_\_func()#报错

AttributeError: 'Parent' object has no attribute '\_\_func'

总结：

根据前面的知识可以得出以下结论：

如果需要在类外修改类属性，必须通过实例对象去引用然后进行修改。

通过实例对象去引用，会产生一个同名的实例属性，这种方式修改的是实例属性，而不会影响到类属性，并且之后如果通过实例对象去引用该名称的属性，实例属性会强制屏蔽掉类属性，即引用的是实例属性，除非删除了该实例属性

|  |  |  |
| --- | --- | --- |
| 属性叫法 | 变量叫法 | 描述 |
| 类属性（私有和公有） | 类变量 | 所有对象共享同一份类属性。 |
| 实例属性（私/公） | 成员变量 | 每个不同对象，有不一样值的实例属性 |

## 10.2 封装

### 10.2.1 封装定义

封装是实现面向对象程序设计的第一步，封装就是将数据（属性）或函数（方法）等集合在一个个的中类中（class），被封装的对象通常被称为抽象数据类型。

通俗的讲，就是隐藏一些属性和实现的细节，给用户提供一个公共接口使用。

1. 例如：

import datetime  
print(help(datetime))

…

class tzinfo:  
 def tzname(self, dt: Optional[datetime]) -> str: ...  
 def utcoffset(self, dt: Optional[datetime]) -> Optional[timedelta]: ...  
 def dst(self, dt: Optional[datetime]) -> Optional[timedelta]: ...  
 def fromutc(self, dt: datetime) -> datetime: ...

…

### 10.2.2 调用封装

调用被封装的内容时，两种方式

1、通过实例对象直接调用

2、通过实例方法利用self间接调用

案例：

class Parent:  
 *'人类文档说明'* sex = '男'  
 def \_\_init\_\_(self,name,age):  
 self.name = name  
 self.age = age  
 *#通过利用self间接调用* def print\_info(self):  
 print(self.name,self.age)  
For = Parent('for',18)  
*#通过对象直接调用*print(For.name,For.age,For.sex)  
For.print\_info()

运行结果如下：

for 18 男

for 18

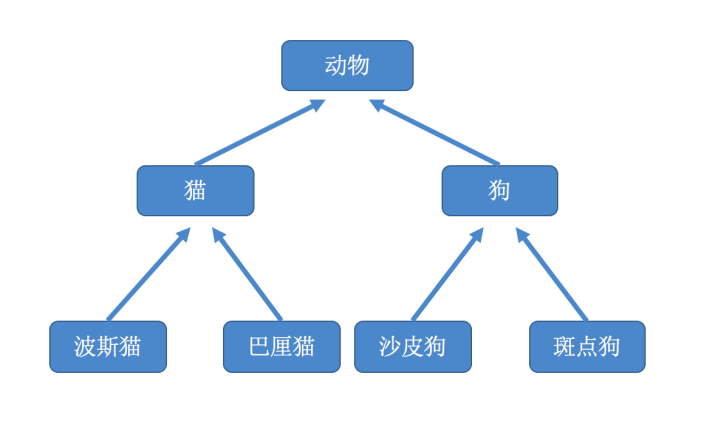
## 10.3 继承

### 10.3.1 继承的介绍

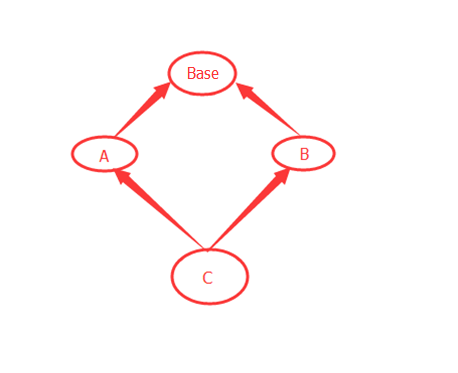
在现实生活中，继承一般指的是子女继承父辈的财产，如下图



在程序中，继承描述的是事物之间的所属关系，例如猫和狗都属于动物，程序中便可以描述为猫和狗继承自动物；同理，波斯猫和巴厘猫都继承自猫，而沙皮狗和斑点狗都继承狗，如下如所示：



### 10.3.2 继承示例

class Animal:  
 def eat(self):  
 print('吃……')  
 def drink(self):  
 print('喝……')  
class Dog(Animal):  
 def eat(self):  
 print('狗吃……')  
 def call(self):  
 print('汪汪叫……')  
class Cat(Animal):   
 def catch(self):  
 print('抓老鼠……')  
*#实例化一个对象*dog = Dog()  
*#调用父类的方法*dog.eat()  
tom = Cat()  
tom.eat()  
*#调用对象本身的方法*tom.catch()

运行结果如下：

狗吃……

吃……

抓老鼠……

分析：Dog类继承Animal类，当调用方法的时候，先调用Dog类自身的方法，没有才去调用父类的方法，同理Cat类也一样,可以认为就近原则。

如果两个类同时继承一个类，比如Cat(Animal)此时，Cat中的一些行为（非父类）不能被Dog使用（dog和cat是并行的关系）。

### 10.3.3 继承中的私有属性和方法

class Animal:

def \_\_init\_\_(self,name='动物',color = '黄色'):

self.\_\_name = name

self.color =color

def eat(self):

print('吃……')

def drink(self):

print('喝……')

def \_\_test(self):

print(self.color)

def print\_func(self):

print(self.\_\_name)

print(self.color)

class Dog(Animal):

def eat(self):

print('狗吃……')

def call(self):

print('汪汪叫……')

*#实例化一个对象*

dog = Dog()

*#访问父类的方法*

dog.drink()

*#访问父类的私有方法*

dog.\_\_test() *#报错*

*#调用父类的属性*

print(dog.color)

print(dog.\_\_name)*#报错*

运行结果如下：

喝……

File "E:/workspace/importTest/test02.py", line 71, in <module>

dog.\_\_test()#错误

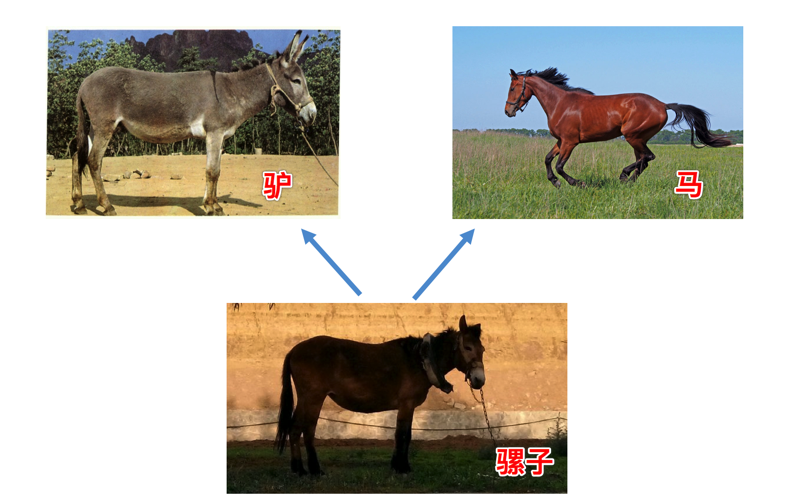
AttributeError: 'Dog' object has no attribute '\_\_test'

说明：

私有的属性、方法，不会被子类继承，也不能被访问。

### 10.3.4 多继承

多继承概念



从图中能够看出，所谓多继承，即子类有多个父类，并且具有它们的特征

Python中多继承的格式如下:

#定义一个Base类

class Base(object):  
 def test(self):  
 print('base')  
class A(Base):  
 def test1(self):  
 print('test1')  
class B(Base):  
 def test2(self):  
 print('test2')  
class C(B,A):  
 pass  
c = C()  
c.test()  
c.test1()  
c.test2()  
print(c.\_\_class\_\_.\_\_mro\_\_)

运行结果如下：

Base

test1

test2

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.Base'>, <class 'object'>)

案例二：

class Base:  
 def \_\_init\_\_(self):  
 print('base')  
 def test(self):  
 print('base\_test')  
class A(Base):  
 def \_\_init\_\_(self):  
 print('A')  
class B(Base):  
 def \_\_init\_\_(self):  
 print('B')  
 def test(self):  
 print('b\_test')  
class C(A,B):  
 pass  
c =C()  
c.test()  
*#访问顺序*print(c.\_\_class\_\_.\_\_mro\_\_)

运行结果如下：

A

b\_test

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.Base'>, <class 'object'>)

说明：

1、python中是可以多继承的并且父类中的方法、属性，子类也都会继承

2、python3中的多继承遵循mro算法顺序当中的，先广度后深度的原理。

3、新式类只会执行一次\_\_init\_\_()文件

### 10.3.5 总结

子类在继承的时候，在定义类时，小括号()中为父类的名字。

子类继承后，父类的属性、方法，都会被继承给子类。

私有的属性，不能通过对象直接访问，但是可以通过方法访问。

私有的方法，不能通过对象直接访问。

私有的属性、方法，不会被子类继承，也不能被访问。

## 10.4 方法的重写

### 10.4.1 什么是重写？

我们在学习多态之前我们要学习一下，python中方法的重写，所谓重写，就是子类中，有一个和父类相同名字的方法，在子类中的方法会覆盖掉父类中同名的方法。

class Dog(object):  
 def print\_self(self):  
 print('Hello,I am dog')  
class Tom(Dog):  
 def print\_self(self):  
 *#调用父类的方法  
 # super().print\_self()  
 # Dog.print\_self(self)* print('Hi I am tom')  
T = Tom()  
T.print\_self()

### 10.4.2 调用父类的方法

如何调用父类的方法呢？

固定写法：

1. super(子类名，self).方法名（参数）
2. super().方法名（参数）
3. 父类名.方法名(self,参数)

注：有参数传参，没有就（）

class Dog(object):  
 def \_\_init\_\_(self,name):  
 self.name = name  
 self.color = '黄色'  
class Tom(Dog):  
 def \_\_init\_\_(self,name):  
 *# super(Tom,self).\_\_init\_\_(name)  
 # Dog.\_\_init\_\_(self,name)* super().\_\_init\_\_(name)  
 def getNaem(self):  
 print(self.name)  
T = Tom('Lucky')  
T.getNaem()

注：super不是代表父类，而是按照继承顺序的下一个类，初学者可以认为，super.\_\_init\_\_()整合在一其就是可对父类中的方法操作。

## 10.5 多态

### 10.5.1 多态定义

Python 不支持多态，也用不到多态。

多态的概念是应用于Java和C#这一类强类型语言中，而Python崇尚“鸭子类型”。

什么叫鸭子类型？

鸭子类型：是一种动态类型的风格。一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由James Whitcomb Riley提出的鸭子测试。

鸭子测试”可以这样表述：   
“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

所谓多态：定义时的类型和运行时的类型不一样，此时就成为多态，更通俗的讲就是多种形态，不同对象调用相同方法的时候产生不同的结果。

Python中要是通过类的继承以及方法的重写来实现该功能.

作用：

主要用作封装，达到灵活调用的目的、

让编程更加灵活；

提高程序的可拓展性。

class Dog(object):  
 def print\_self(self):  
 print('Hello,I am dog')  
class Tom(Dog):  
 def print\_self(self):  
 *#调用父类的方法  
 # super().print\_self()  
 # Dog.print\_self(self)* print('Hi I am tom')  
def func(object):  
 object.print\_self()  
tom = Tom()  
func(tom)

#定义一个实例  
dog = Dog()  
func(dog)

运行结果如下：

Hi I am tom

## 10.6 魔法方法(面试题)

魔法方法就如同它的名字一样神奇，总能在你需要的时候为你提供某种方法来让你的想法实现。魔法方法是指Python内部已经包含的，被双下划线所包围的方法，这些方法在进行特定的操作时会自动被调用，它们是Python面向对象下智慧的结晶。初学者掌握Python的魔法方法也就变得尤为重要了。

使用Python的魔法方法可以使Python的自由度变得更高，当不需要重写时魔法方法也可以在规定的默认情况下生效，在需要重写时也可以让使用者根据自己的需求来重写部分方法来达到自己的期待。而且众所周知Python是支持面向对象的语言Python的基本魔法方法就使得Python在面对对象方面做得更好。

### 10.6.1 \_\_del\_\_()方法

创建对象后，python解释器默认调用\_\_init\_\_()方法；

当删除一个对象时，python解释器也会默认调用一个方法，这个方法为\_\_del\_\_()方法

当内存中构建一个对象数据的时候回调\_\_init\_\_()方法，

当内存中销毁（释放）一个对象时回调\_\_del\_\_()方法。

例子：

class Hero:  
 def \_\_del\_\_(self):  
 print("英雄已阵亡")  
*#创建一个实例*man1 = Hero()print('程序结束')

运行结果如下：  
程序结束

英雄已阵亡

class Hero:  
 def \_\_del\_\_(self):  
 print("英雄已阵亡")  
*#创建一个实例*man1 = Hero()  
*#指向这个实例的引用计数为2*man2 = man1  
*#删除引用计数*del man1  
*#删除引用计数*del man2  
print('程序结束')

运行结果如下：  
英雄已阵亡

程序结束

class Hero:  
 def \_\_del\_\_(self):  
 print("英雄已阵亡")  
*#创建一个实例*man1 = Hero()  
*#指向这个实例的引用计数为2*man2 = man1  
*#删除引用计数*del man1  
print('程序结束')

运行结果如下：

程序结束

英雄已阵亡

总结：

1. 当有1个变量保存了对象的引用时，此对象的引用计数就会加1
2. 当使用del删除变量指向的对象时，如果对象的引用计数不是1，比如有3个引用计数，那么此时只会让这个引用计数减1，即变为2，当再次调用del时，变为1，如果再调用1次del，此时会真的把对象进行删除
3. 只有真正删除这个hero()实例对象时候才会回调del魔法方法

10.6.2 \_\_new\_\_方法

1. \_\_new\_\_至少要有一个参数cls，代表要实例化的类(并不是对象)，此参数在实例化时由Python解释器自动提供。
2. \_\_new\_\_必须要有返回值，返回实例化出来的实例，这点在自己实现\_\_new\_\_时要特别注意，可以return object的\_\_new\_\_出来的实例。
3. \_\_init\_\_有一个参数self，就是这个\_\_new\_\_返回的实例，\_\_init\_\_在\_\_new\_\_的基础上可以完成一些其它初始化的动作，\_\_init\_\_不需要返回值。
4. 我们可以将类比作制造商，\_\_new\_\_方法就是前期的原材料购买环节，\_\_init\_\_方法就是在有原材料的基础上，加工，初始化商品环节。
5. 简单来说\_\_new\_\_方法，就是创建一个实例。

class A(object):  
 def \_\_init\_\_(self):  
 print('这是init方法')  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 print('这是new方法')  
 return object.\_\_new\_\_(cls)  
A()

运行结果如下：  
 这是new方法

这是init方法

我们来看下cls到底是什么？

class A(object):  
 def \_\_init\_\_(self,name,age):  
 self.name = name  
 self.age = age  
 *#加入我们要生产的参数* def \_\_new\_\_(cls,name,age):  
 print(cls)  
 ret = object.\_\_new\_\_(cls)  
 return ret  
a = A('for',18)  
print(a.name)

运行结果如下：

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

for

### 10.6.3． \_\_str\_\_()方法

\_\_str\_\_：在将对象转换成字符串str(对象)测试的时候，打印对象的信息。

\_\_str\_\_方法必须要return一个字符串类型的返回值，作为对实例对象的字符串描述。

\_\_str\_\_实际上是被print函数默认调用的，当要print（实例对象）时，默认调用\_\_str\_\_方法，将其字符串描述返回。

当你打印一个类的时候，那么print首先调用的就是类里面的定义的\_\_str\_\_。

class Test(object):  
 def \_\_init\_\_(self,world):  
 self.world = world  
 def \_\_str\_\_(self):  
 print('\_\_str\_\_')  
 return 'world is %s'%self.world  
t = Test('for')  
*# print(str(t))*print(t)

print(t)#直接敲t 的时候str是不会执行的。

运行结果如下：

\_\_str\_\_

world is for

## 10.7 单例模式（面试题）

### 10.7.1 单例是什么

举个常见的单例模式例子，我们日常使用的电脑上都有一个回收站，在整个操作系统中，回收站只能有一个实例，整个系统都使用这个唯一的实例，而且回收站自行提供自己的实例。因此回收站是单例模式的应用。

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。

### 10.7.2 创建单例-保证只有1个对象

# 实例化一个单例

class Singleton(object):  
 \_\_instance = None  
 def \_\_new\_\_(cls,name ,age):  
 *#如果\_\_instance没有赋值则一直为none  
 #接下来我们利用new方法，创造出一个对象，并且更改\_\_instance的值  
 #保证私有属性\_\_instance不为空！然后经过逻辑判断只为* if not cls.\_\_instance:*#判断是否\_\_instance为空，为空就生成* cls.\_\_instance = object.\_\_new\_\_(cls)  
 *#\_\_instance不为空，直接返回* return cls.\_\_instance  
*#第一个实例对象*a = Singleton('for',18)  
*#第二个实例对象*b = Singleton('bin',8)  
print(id(a))  
print(id(b))  
a.age = 19  
print(b.age)

运行结果：

1798590570336

1798590570336

19

创建单例时，只执行1次\_\_init\_\_方法

# 实例化一个单例

class SingLeton(object):  
 \_\_instance = None  
 \_\_first\_init = False  
 def \_\_new\_\_(cls,\*args,\*\*kwargs):  
 if not cls.\_\_instance:  
 cls.\_\_instance = object.\_\_new\_\_(cls)  
 return cls.\_\_instance  
 def \_\_init\_\_(self,name,age):  
 if not self.\_\_first\_init:  
 self.name = name  
 self.age = age  
 *#更改属性的私有类属性，并把它设置为True，下个对象调用时就不会执行这个if判断中的语句。* SingLeton.\_\_first\_init = True  
a = SingLeton('for',18)  
b = SingLeton('django',10)  
print(id(a))  
print(id(b))  
print(b.age)

输出结果：

1284844314696

1284844314696

18

附录：

|  |  |
| --- | --- |
| 魔法方法 | 含义 |
| 基本的魔法方法 | |
| \_\_new\_\_(cls[, ...]) | 1. \_\_new\_\_ 是在一个对象实例化的时候所调用的第一个方法 2. 它的第一个参数是这个类，其他的参数是用来直接传递给 \_\_init\_\_ 方法 3. \_\_new\_\_ 决定是否要使用该 \_\_init\_\_ 方法，因为 \_\_new\_\_ 可以调用其他类的构造方法或者直接返回别的实例对象来作为本类的实例，如果 \_\_new\_\_ 没有返回实例对象，则 \_\_init\_\_ 不会被调用 4. \_\_new\_\_ 主要是用于继承一个不可变的类型比如一个 tuple 或者 string |
| \_\_init\_\_(self[, ...]) | 构造器，当一个实例被创建的时候调用的初始化方法 |
| \_\_del\_\_(self) | 析构器，当一个实例被销毁的时候调用的方法 |
| \_\_call\_\_(self[, args...]) | 允许一个类的实例像函数一样被调用：x(a, b) 调用 x.\_\_call\_\_(a, b) |
| \_\_len\_\_(self) | 定义当被 len() 调用时的行为 |
| \_\_repr\_\_(self) | 定义当被 repr() 调用时的行为 |
| \_\_str\_\_(self) | 定义当被 str() 调用时的行为 |
| \_\_bytes\_\_(self) | 定义当被 bytes() 调用时的行为 |
| \_\_hash\_\_(self) | 定义当被 hash() 调用时的行为 |
| \_\_bool\_\_(self) | 定义当被 bool() 调用时的行为，应该返回 True 或 False |
| \_\_format\_\_(self, format\_spec) | 定义当被 format() 调用时的行为 |
| 有关属性 | |
| \_\_getattr\_\_(self, name) | 定义当用户试图获取一个不存在的属性时的行为 |
| \_\_getattribute\_\_(self, name) | 定义当该类的属性被访问时的行为 |
| \_\_setattr\_\_(self, name, value) | 定义当一个属性被设置时的行为 |
| \_\_delattr\_\_(self, name) | 定义当一个属性被删除时的行为 |
| \_\_dir\_\_(self) | 定义当 dir() 被调用时的行为 |
| \_\_get\_\_(self, instance, owner) | 定义当描述符的值被取得时的行为 |
| \_\_set\_\_(self, instance, value) | 定义当描述符的值被改变时的行为 |
| \_\_delete\_\_(self, instance) | 定义当描述符的值被删除时的行为 |
| 比较操作符 | |
| \_\_lt\_\_(self, other) | 定义小于号的行为：x < y 调用 x.\_\_lt\_\_(y) |
| \_\_le\_\_(self, other) | 定义小于等于号的行为：x <= y 调用 x.\_\_le\_\_(y) |
| \_\_eq\_\_(self, other) | 定义等于号的行为：x == y 调用 x.\_\_eq\_\_(y) |
| \_\_ne\_\_(self, other) | 定义不等号的行为：x != y 调用 x.\_\_ne\_\_(y) |
| \_\_gt\_\_(self, other) | 定义大于号的行为：x > y 调用 x.\_\_gt\_\_(y) |
| \_\_ge\_\_(self, other) | 定义大于等于号的行为：x >= y 调用 x.\_\_ge\_\_(y) |
| 算数运算符 | |
| \_\_add\_\_(self, other) | 定义加法的行为：+ |
| \_\_sub\_\_(self, other) | 定义减法的行为：- |
| \_\_mul\_\_(self, other) | 定义乘法的行为：\* |
| \_\_truediv\_\_(self, other) | 定义真除法的行为：/ |
| \_\_floordiv\_\_(self, other) | 定义整数除法的行为：// |
| \_\_mod\_\_(self, other) | 定义取模算法的行为：% |
| \_\_divmod\_\_(self, other) | 定义当被 divmod() 调用时的行为 |
| \_\_pow\_\_(self, other[, modulo]) | 定义当被 power() 调用或 \*\* 运算时的行为 |
| \_\_lshift\_\_(self, other) | 定义按位左移位的行为：<< |
| \_\_rshift\_\_(self, other) | 定义按位右移位的行为：>> |
| \_\_and\_\_(self, other) | 定义按位与操作的行为：& |
| \_\_xor\_\_(self, other) | 定义按位异或操作的行为：^ |
| \_\_or\_\_(self, other) | 定义按位或操作的行为：| |
| 反运算 | |
| \_\_radd\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rsub\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rmul\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rtruediv\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rfloordiv\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rmod\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rdivmod\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rpow\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rlshift\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rrshift\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_rxor\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| \_\_ror\_\_(self, other) | （与上方相同，当左操作数不支持相应的操作时被调用） |
| 增量赋值运算 | |
| \_\_iadd\_\_(self, other) | 定义赋值加法的行为：+= |
| \_\_isub\_\_(self, other) | 定义赋值减法的行为：-= |
| \_\_imul\_\_(self, other) | 定义赋值乘法的行为：\*= |
| \_\_itruediv\_\_(self, other) | 定义赋值真除法的行为：/= |
| \_\_ifloordiv\_\_(self, other) | 定义赋值整数除法的行为：//= |
| \_\_imod\_\_(self, other) | 定义赋值取模算法的行为：%= |
| \_\_ipow\_\_(self, other[, modulo]) | 定义赋值幂运算的行为：\*\*= |
| \_\_ilshift\_\_(self, other) | 定义赋值按位左移位的行为：<<= |
| \_\_irshift\_\_(self, other) | 定义赋值按位右移位的行为：>>= |
| \_\_iand\_\_(self, other) | 定义赋值按位与操作的行为：&= |
| \_\_ixor\_\_(self, other) | 定义赋值按位异或操作的行为：^= |
| \_\_ior\_\_(self, other) | 定义赋值按位或操作的行为：|= |
| 一元操作符 | |
| \_\_neg\_\_(self) | 定义正号的行为：+x |
| \_\_pos\_\_(self) | 定义负号的行为：-x |
| \_\_abs\_\_(self) | 定义当被 abs() 调用时的行为 |
| \_\_invert\_\_(self) | 定义按位求反的行为：~x |
|  | 类型转换 |
| \_\_complex\_\_(self) | 定义当被 complex() 调用时的行为（需要返回恰当的值） |
| \_\_int\_\_(self) | 定义当被 int() 调用时的行为（需要返回恰当的值） |
| \_\_float\_\_(self) | 定义当被 float() 调用时的行为（需要返回恰当的值） |
| \_\_round\_\_(self[, n]) | 定义当被 round() 调用时的行为（需要返回恰当的值） |
| \_\_index\_\_(self) | 1. 当对象是被应用在切片表达式中时，实现整形强制转换 2. 如果你定义了一个可能在切片时用到的定制的数值型,你应该定义 \_\_index\_\_ 3. 如果 \_\_index\_\_ 被定义，则 \_\_int\_\_ 也需要被定义，且返回相同的值 |
| 上下文管理（with 语句） | |
| \_\_enter\_\_(self) | 1. 定义当使用 with 语句时的初始化行为 2. \_\_enter\_\_ 的返回值被 with 语句的目标或者 as 后的名字绑定 |
| \_\_exit\_\_(self, exc\_type, exc\_value, traceback) | 1. 定义当一个代码块被执行或者终止后上下文管理器应该做什么 2. 一般被用来处理异常，清除工作或者做一些代码块执行完毕之后的日常工作 |
| 容器类型 | |
| \_\_len\_\_(self) | 定义当被 len() 调用时的行为（返回容器中元素的个数） |
| \_\_getitem\_\_(self, key) | 定义获取容器中指定元素的行为，相当于 self[key] |
| \_\_setitem\_\_(self, key, value) | 定义设置容器中指定元素的行为，相当于 self[key] = value |
| \_\_delitem\_\_(self, key) | 定义删除容器中指定元素的行为，相当于 del self[key] |
| \_\_iter\_\_(self) | 定义当迭代容器中的元素的行为 |
| \_\_reversed\_\_(self) | 定义当被 reversed() 调用时的行为 |
| \_\_contains\_\_(self, item) | 定义当使用成员测试运算符（in 或 not in）时的行为 |

总结：

10.1 类的内置属性和方法

10.2 封装

10.3 继承

10.4 方法重写

10.5 多态

10.6 魔法方法

10.7 单例模式