单例

目标

- 单例设计模式
- `__new__` 方法
- Python 中的单例

01. 单例设计模式

- 设计模式
  - **设计模式** 是 **前人工作的总结和提炼**，通常，被人们广泛流传的设计模式都是针对 **某一特定问题** 的成熟的解决方案
  - 使用 **设计模式** 是为了可重用代码、让代码更容易被他人理解、保证代码可靠性
- 单例设计模式
  - **目的** —— 让 **类** 创建的对象，在系统中 **只有 唯一的一个实例**
  - 每一次执行 `类名()` 返回的对象，**内存地址是相同的**

单例设计模式的应用场景

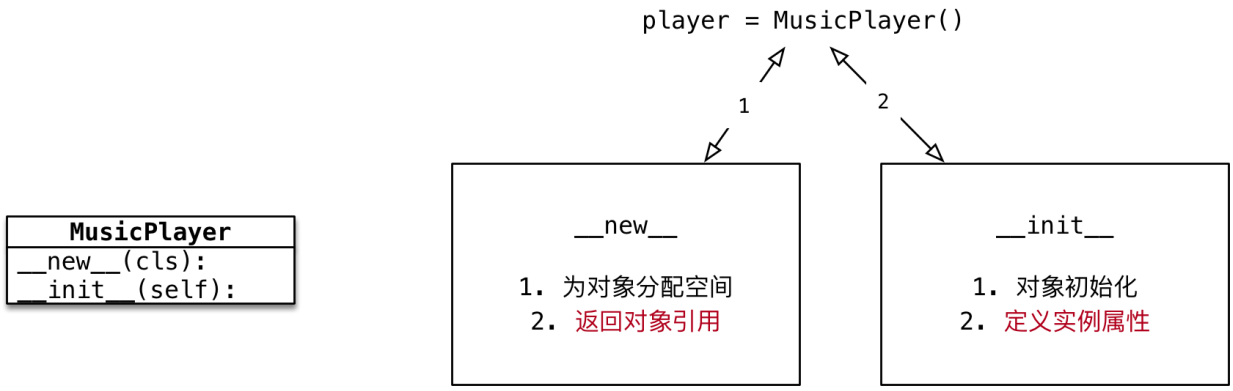
- **音乐播放** 对象
- **回收站** 对象
- **打印机** 对象
- ……

02. `__new__` 方法

- 使用 **类名()** 创建对象时，`Python` 的解释器 **首先** 会 调用 `__new__` 方法为对象 **分配空间**
- `__new__` 是一个 由 `object` 基类提供的 **内置的静态方法**，主要作用有两个：
  - 1) 在内存中为对象 **分配空间**
  - 2) **返回** 对象的引用
- `Python` 的解释器获得对象的 **引用** 后，将引用作为 **第一个参数**，传递给 `__init__` 方法

> 重写 `__new__` 方法 的代码非常固定！

- 重写 `__new__` 方法 一**定要** `return super().__new__(cls)`
- 否则 Python 的解释器 **得不到** 分配了空间的 **对象引用，就不会调用对象的初始化方法**
- 注意：`__new__` 是一个静态方法，在调用时需要 **主动传递** `cls` 参数



示例代码

```python class MusicPlayer(object):

```

```
def __new__(cls, *args, **kwargs):
    # 如果不返回任何结果，
    return super().__new__(cls)

def __init__(self):
    print("初始化音乐播放对象")
```

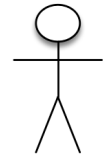
player = MusicPlayer()

print(player)

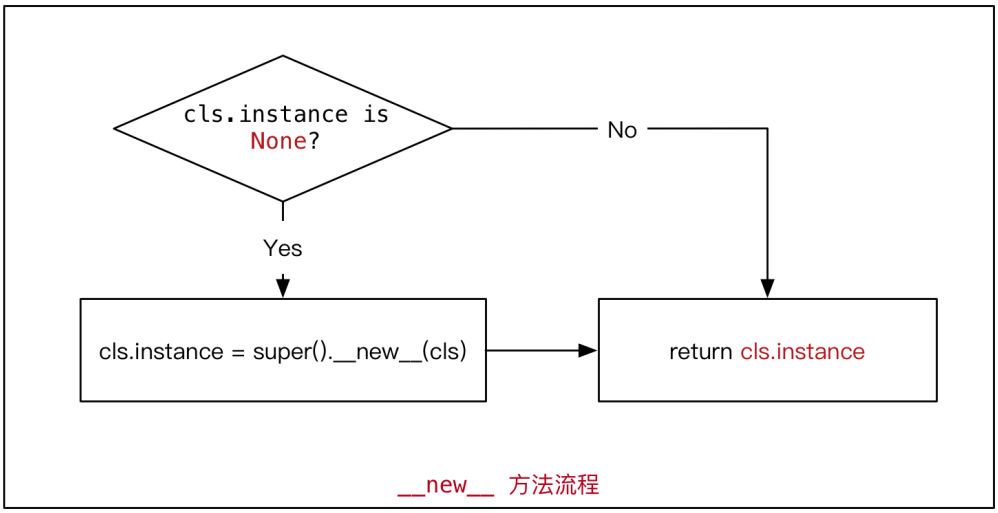
```

## 03. Python 中的单例

- **单例** —— 让 **类** 创建的对象，在系统中 **只有 唯一的一个实例**
  1. 定义一个 **类属性**，初始值是 `None`，用于记录 **单例对象的引用**
  2. 重写 `__new__` 方法
  3. 如果 **类属性** `is None`，调用父类方法分配空间，并在类属性中记录结果
  4. 返回 **类属性** 中记录的 **对象引用**



**类属性**
**instance**




```python class MusicPlayer(object):

```python
# 定义类属性记录单例对象引用
instance = None

def __new__(cls, *args, **kwargs):

    # 1. 判断类属性是否已经被赋值
    if cls.instance is None:
        cls.instance = super().__new__(cls)

    # 2. 返回类属性的单例引用
    return cls.instance
```

```

### 只执行一次初始化工作

- 在每次使用 `类名()` 创建对象时，`Python` 的解释器都会自动调用两个方法：
  - `__new__` 分配空间
  - `__init__` 对象初始化
- 在上一小节对 `__new__` 方法改造之后，每次都会得到 **第一次被创建对象的引用**
- 但是：**初始化方法还会被再次调用**

**需求**

- 让 **初始化动作** 只被 **执行一次**

**解决办法**

1. 定义一个类属性 `init_flag` 标记是否 **执行过初始化动作**，初始值为 `False`
2. 在 `__init__` 方法中，判断 `init_flag`，如果为 `False` 就执行初始化动作
3. 然后将 `init_flag` 设置为 `True`
4. 这样，再次 **自动** 调用 `__init__` 方法时，**初始化动作就不会被再次执行** 了

```python class MusicPlayer(object):

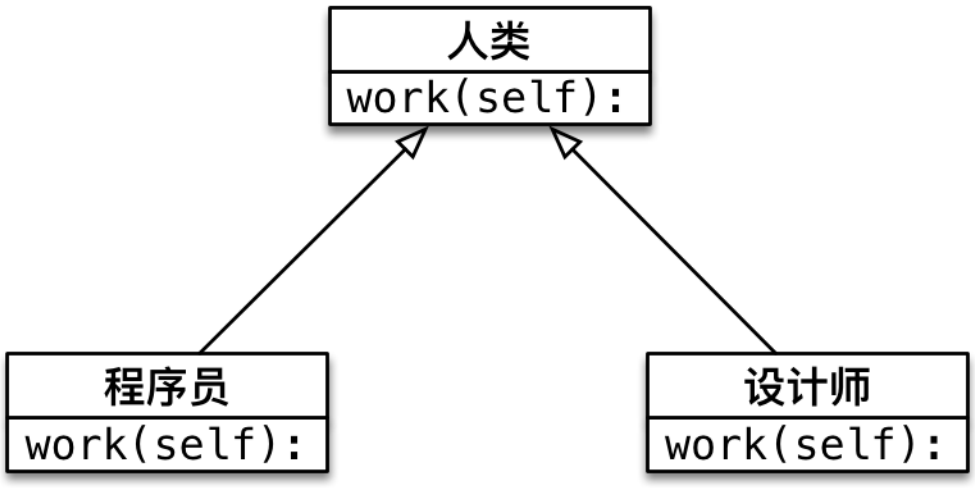
```python
# 记录第一个被创建对象的引用
instance = None
# 记录是否执行过初始化动作
init_flag = False

def __new__(cls, *args, **kwargs):

    # 1. 判断类属性是否是空对象
```

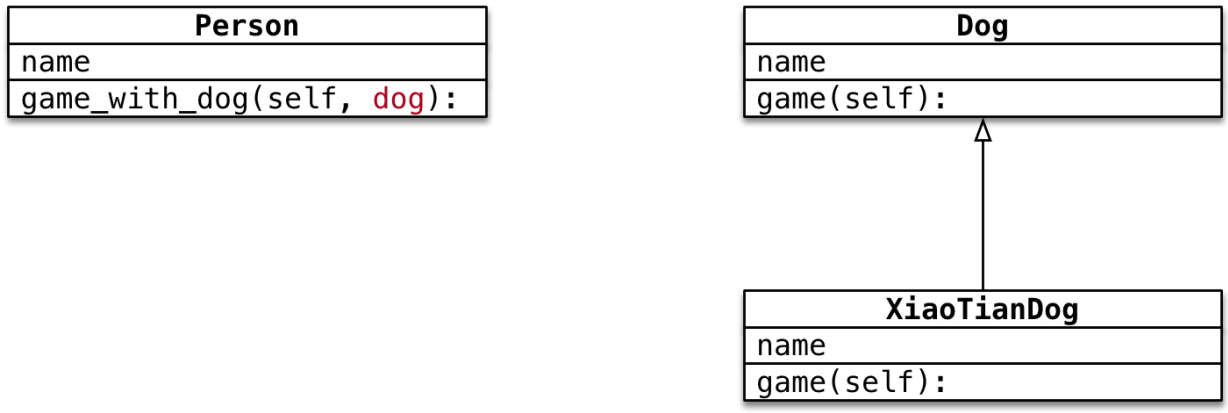
```
    if cls.instance is None:
        # 2. 调用父类的方法，为第一个对象分配空间
        cls.instance = super().__new__(cls)

    # 3. 返回类属性保存的对象引用
    return cls.instance

def __init__(self):

    if not MusicPlayer.init_flag:
        print("初始化音乐播放器")

        MusicPlayer.init_flag = True
```

创建多个对象

player1 = MusicPlayer() print(player1)

player2 = MusicPlayer() print(player2)

```

# 多态

## 目标

- 多态

**面向对象三大特性**

1. **封装** 根据 **职责** 将 **属性** 和 **方法** **封装** 到一个抽象的 **类** 中
   - 定义类的准则
2. **继承** **实现代码的重用**，相同的代码不需要重复的编写
   - 设计类的技巧
   - 子类针对自己特有的需求，编写特定的代码
3. **多态** 不同的 **子类对象** 调用相同的 **父类方法**，产生不同的执行结果
   - **多态** 可以 **增加代码的灵活度**
   - 以 **继承** 和 **重写父类方法** 为前提
   - 是调用方法的技巧，**不会影响到类的内部设计**




## 多态案例演练

**需求**

1. 在 `Dog` 类中封装方法 `game`
   - 普通狗只是简单的玩耍
2. 定义 `XiaoTianDog` 继承自 `Dog`，并且重写 `game` 方法
   - 哮天犬需要在天上玩耍
3. 定义 `Person` 类，并且封装一个 **和狗玩** 的方法
   - 在方法内部，直接让 **狗对象** 调用 `game` 方法
```



案例小结

- `Person` 类中只需要让 **狗对象** 调用 `game` 方法，而不关心具体是 **什么狗**
  - `game` 方法是在 `Dog` 父类中定义的
- 在程序执行时，传入不同的 **狗对象** 实参，就会产生不同的执行效果

> **多态** 更容易编写出出通用的代码，做出通用的编程，以适应需求的不断变化！

```python class Dog(object):

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

def game(self):
    print("%s 蹦蹦跳跳的玩耍..." % self.name)
```

class XiaoTianDog(Dog):

```
def game(self):
    print("%s 飞到天上去玩耍..." % self.name)
```

class Person(object):

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

def game_with_dog(self, dog):

    print("%s 和 %s 快乐的玩耍..." % (self.name, dog.name))

    # 让狗玩耍
    dog.game()
```

1. 创建一个狗对象

wangcai = Dog("旺财")

wangcai = XiaoTianDog("飞天旺财")

2. 创建一个小明对象

xiaoming = Person("小明")

3. 让小明调用和狗玩的方法

xiaoming.game*with*dog(wangcai)

```

# 继承

## 目标

- 单继承
- 多继承

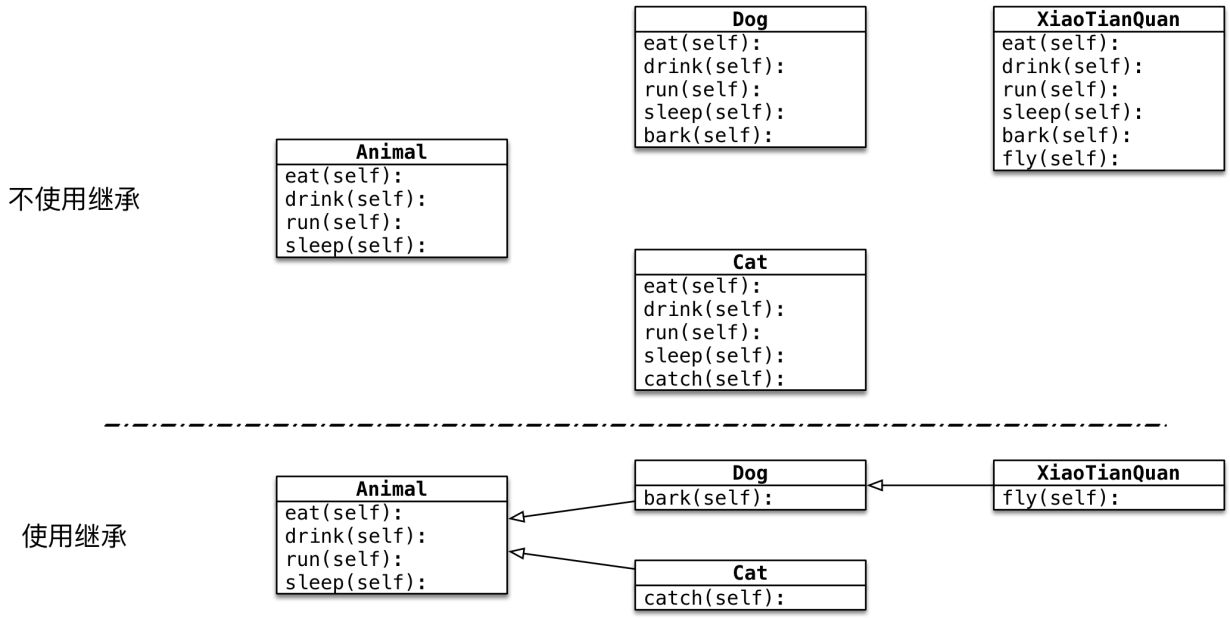
**面向对象三大特性**

1. **封装** 根据 **职责** 将 **属性** 和 **方法** **封装** 到一个抽象的 **类** 中
2. **继承** **实现代码的重用**，相同的代码不需要重复的编写
3. **多态** 不同的对象调用相同的方法，产生不同的执行结果，**增加代码的灵活度**

# 01. 单继承

## 1.1 继承的概念、语法和特点

**继承的概念：子类** 拥有 **父类** 的所有 **方法** 和 **属性**




### 1) 继承的语法

```python class 类名(父类名):

```
pass
```

```

- **子类** 继承自 **父类**，可以直接 **享受** 父类中已经封装好的方法，不需要再次开发
- **子类** 中应该根据 **职责**，封装 **子类特有的** **属性和方法**

2) 专业术语

- `Dog` 类是 `Animal` 类的**子类**，`Animal` 类是 `Dog` 类的**父类**，**`Dog` 类从 `Animal` 类继承**
- `Dog` 类是 `Animal` 类的**派生类**，`Animal` 类是 `Dog` 类的**基类**，**`Dog` 类从 `Animal` 类派生**

3) 继承的传递性

- `C` 类从 `B` 类继承，`B` 类又从 `A` 类继承
- 那么 `C` 类就具有 `B` 类和 `A` 类的所有属性和方法

子类 拥有 **父类** 以及 **父类的父类** 中封装的所有 **属性** 和 **方法**

提问

哮天犬 能够调用 `Cat` 类中定义的 `catch` 方法吗？

答案

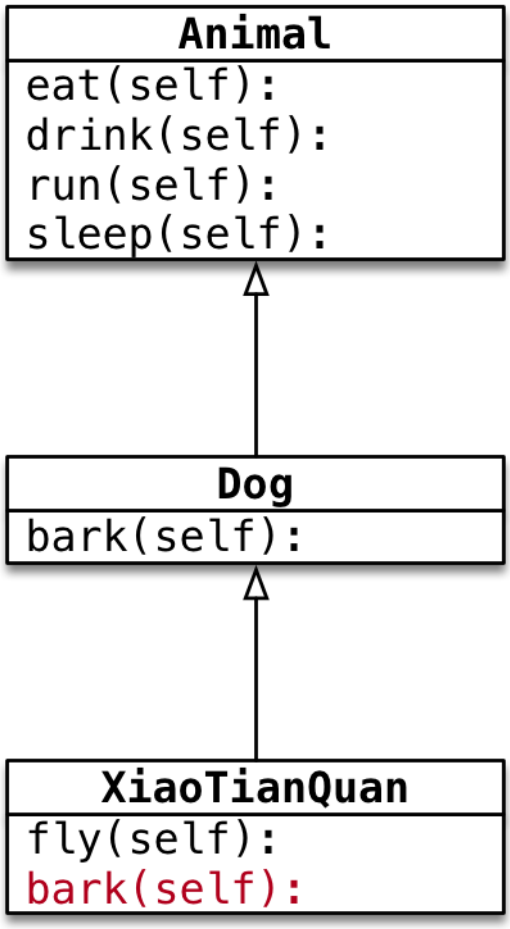
不能，因为 **哮天犬** 和 `Cat` 之间没有 **继承** 关系

1.2 方法的重写

- **子类** 拥有 **父类** 的所有 **方法** 和 **属性**
- **子类** 继承自 **父类**，可以直接 **享受** 父类中已经封装好的方法，不需要再次开发

应用场景

- 当 **父类** 的方法实现不能满足子类需求时，可以对方法进行 **重写(override)**



重写 父类方法有两种情况：

1. **覆盖** 父类的方法
2. 对父类方法进行 **扩展**

1) 覆盖父类的方法

- 如果在开发中，**父类的方法实现** 和 **子类的方法实现，完全不同**
- 就可以使用 **覆盖** 的方式，**在子类中 重新编写** 父类的方法实现

> 具体的实现方式，就相当于在 **子类中** 定义了一个 **和父类同名的方法并且实现**

重写之后，在运行时，**只会调用** 子类中重写的方法，而不再会调用 **父类封装的方法**

2) 对父类方法进行 扩展

- 如果在开发中，**子类的方法实现** 中 **包含** **父类的方法实现**
  - **父类原本封装的方法实现** 是 **子类方法的一部分**
- 就可以使用 **扩展** 的方式
  1. **在子类中 重写** 父类的方法
  2. 在需要的位置使用 `super().父类方法` 来调用父类方法的执行
  3. 代码其他的位置针对子类的需求，编写 **子类特有的代码实现**

关于 `super`

- 在 `Python` 中 `super` 是一个 **特殊的类**
- `super()` 就是使用 `super` 类创建出来的对象
- **最常** 使用的场景就是在 **重写父类方法时**，调用 **在父类中封装的方法实现**

调用父类方法的另外一种方式（知道）

> 在 `Python 2.x` 时，如果需要调用父类的方法，还可以使用以下方式：

`python 父类名.方法(self)`

- 这种方式，目前在 `Python 3.x` 还支持这种方式
- 这种方法 **不推荐使用**，因为一旦 **父类发生变化**，方法调用位置的 **类名** 同样需要修改

提示

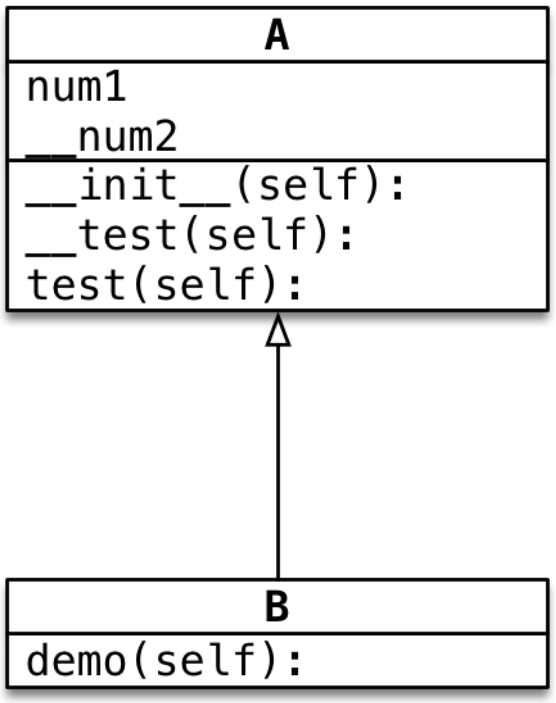
- 在开发时，`父类名` 和 `super()` 两种方式不要混用
- 如果使用 **当前子类名** 调用方法，会形成递归调用，**出现死循环**

1.3 父类的 私有属性 和 私有方法

1. **子类对象 不能** 在自己的方法内部，**直接** 访问 父类的 **私有属性** 或 **私有方法**
2. **子类对象** 可以通过 **父类** 的 **公有方法 间接** 访问到 **私有属性** 或 **私有方法**

> - **私有属性、方法** 是对象的隐私，不对外公开，**外界** 以及 **子类** 都不能直接访问
> - **私有属性、方法** 通常用于做一些内部的事情

示例

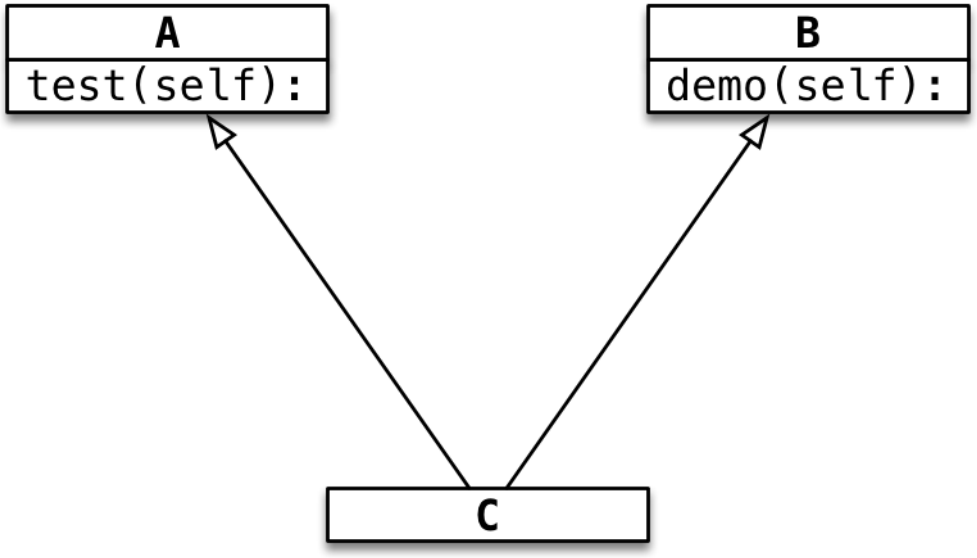


- `B` 的对象不能直接访问 `__num2` 属性
- `B` 的对象不能在 `demo` 方法内访问 `__num2` 属性
- `B` 的对象可以在 `demo` 方法内，调用父类的 `test` 方法
- 父类的 `test` 方法内部，能够访问 `__num2` 属性和 `__test` 方法

02. 多继承

概念

- **子类** 可以拥有 **多个父类**，并且具有 **所有父类** 的 **属性** 和 **方法**
- 例如：**孩子** 会继承自己 **父亲** 和 **母亲** 的 **特性**



语法

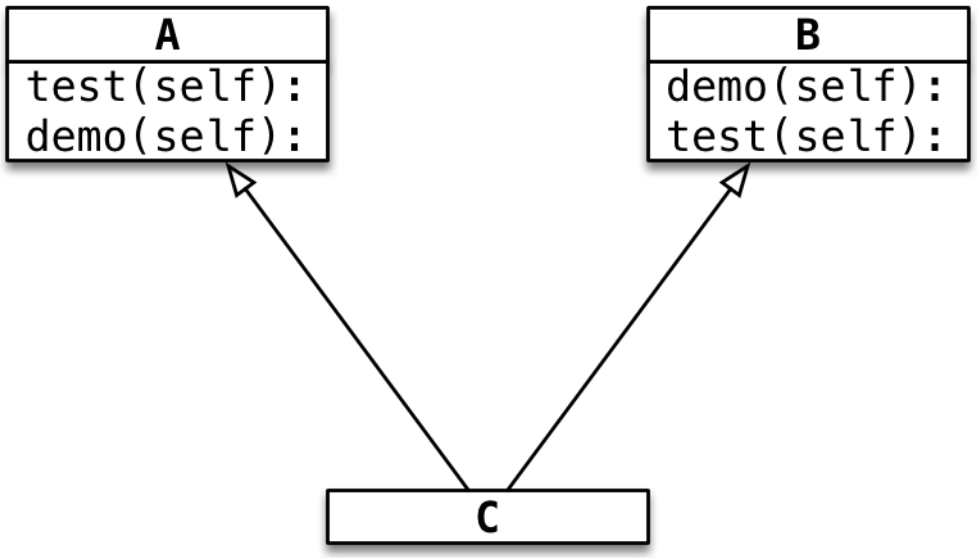
```
python class 子类名(父类名1, 父类名2...) pass
```

2.1 多继承的使用注意事项

问题的提出

- 如果 **不同的父类** 中存在 **同名的方法，子类对象** 在调用方法时，会调用 **哪一个父类中** 的方法呢？

> 提示：**开发时，应该尽量避免这种容易产生混淆的情况！** —— 如果 **父类之间** 存在 **同名的属性或者方法**，应该 **尽量避免** 使用多继承



Python 中的 MRO —— 方法搜索顺序（知道）

- `Python` 中针对 **类** 提供了一个 **内置属性** `__mro__` 可以查看 **方法** 搜索顺序
- MRO 是 `method resolution order`，主要用于 **在多继承时判断 方法、属性 的调用 路径**

```
python print(C.__mro__)
```

输出结果

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
```

- 在搜索方法时，是按照 `__mro__` 的输出结果 **从左至右** 的顺序查找的
- 如果在当前类中 **找到方法，就直接执行，不再搜索**
- 如果 **没有找到，就查找下一个类** 中是否有对应的方法，**如果找到，就直接执行，不再搜索**
- 如果找到最后一个类，还没有找到方法，程序报错

2.2 新式类与旧式（经典）类

> `object` 是 `Python` 为所有对象提供的 **基类**，提供有一些内置的属性和方法，可以使用 `dir` 函数查看

- **新式类**：以 `object` 为基类的类，**推荐使用**
- **经典类**：不以 `object` 为基类的类，**不推荐使用**

- 在 `Python 3.x` 中定义类时，如果没有指定父类，会 **默认使用** `object` 作为该类的 **基类** —— `Python 3.x` 中定义的类都是 **新式类**

- 在 `Python 2.x` 中定义类时，如果没有指定父类，则不会以 `object` 作为 **基类**

> **新式类** 和 **经典类** 在多继承时 —— 会影响到方法的搜索顺序

为了保证编写的代码能够同时在 `Python 2.x` 和 `Python 3.x` 运行！ 今后在定义类时，**如果没有父类，建议统一继承自** `object`

```
python class 类名(object): pass
```

类属性和类方法

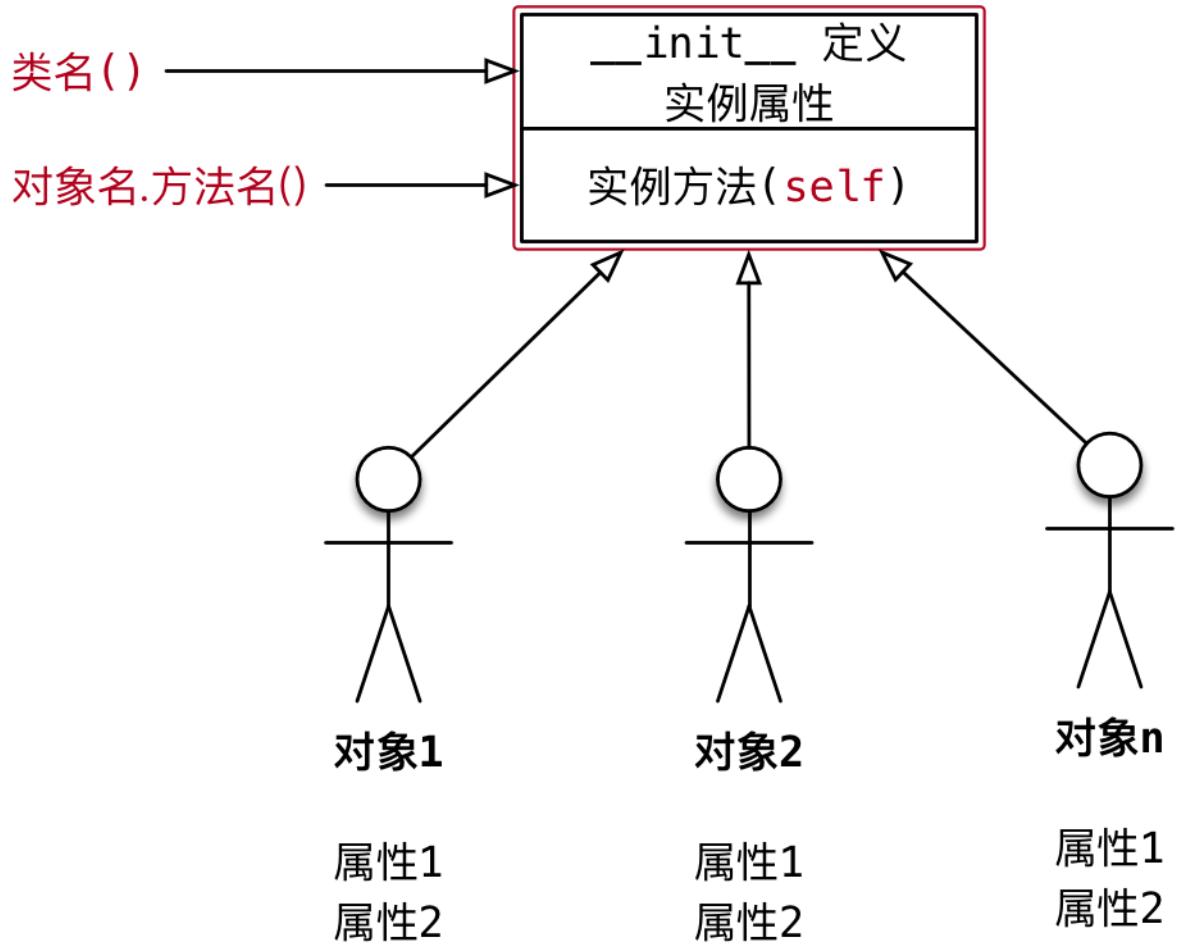
目标

- 类的结构
- 类属性和实例属性
- 类方法和静态方法

01. 类的结构

1.1 术语 —— 实例

1. 使用面相对象开发，**第 1 步** 是设计 **类**
2. 使用 **类名()** 创建对象，**创建对象** 的动作有两步：
   - 1) 在内存中为对象 **分配空间**
   - 2) 调用初始化方法 `__init__` 为 **对象初始化**
3. 对象创建后，**内存** 中就有了一个对象的 **实实在在** 的存在 —— **实例**



因此，通常也会把：

1. 创建出来的 **对象** 叫做 **类** 的 **实例**

2. 创建对象的 **动作** 叫做 **实例化**
3. **对象的属性** 叫做 **实例属性**
4. **对象调用的方法** 叫做 **实例方法**

在程序执行时：

1. 对象各自拥有自己的 **实例属性**
2. 调用对象方法，可以通过 `self.`
   - 访问自己的属性
   - 调用自己的方法

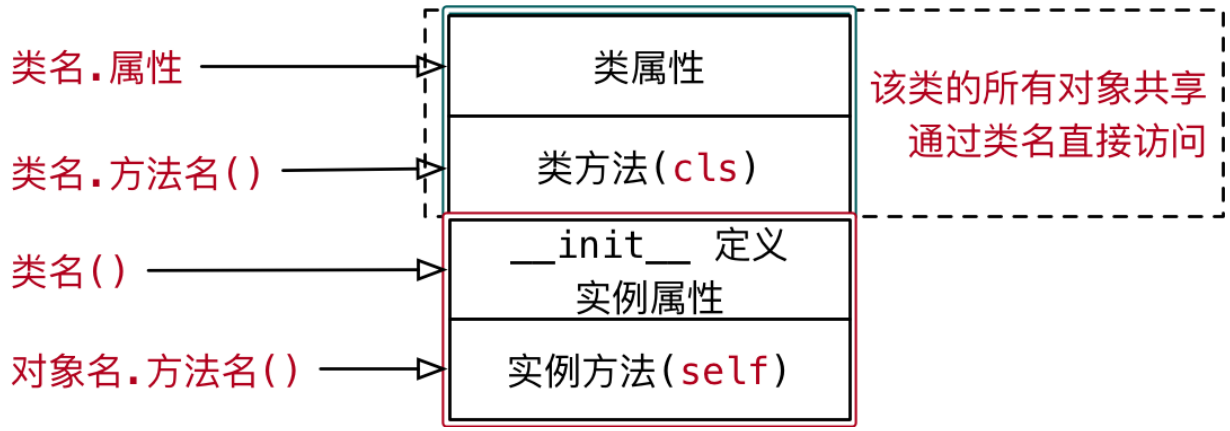
结论

- **每一个对象** 都有自己 **独立的内存空间，保存各自不同的属性**
- **多个对象的方法，在内存中只有一份，在调用方法时，需要把对象的引用** 传递到方法内部

1.2 类是一个特殊的对象

> `Python` 中 **一切皆对象**：
>
> - `class AAA:` 定义的类属于 **类对象**
> - `obj1 = AAA()` 属于 **实例对象**

- 在程序运行时，**类** 同样 **会被加载到内存**
- 在 `Python` 中，**类** 是一个特殊的对象 —— **类对象**
- 在程序运行时，**类对象** 在内存中 **只有一份**，使用 **一个类** 可以创建出 **很多个对象实例**
- 除了封装 **实例** 的 **属性** 和 **方法**外，**类对象** 还可以拥有自己的 **属性** 和 **方法**
  1. **类属性**
  2. **类方法**
- 通过 **类名.** 的方式可以 **访问类的属性** 或者 **调用类的方法**



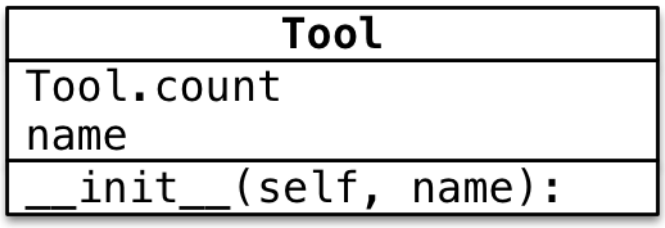
02. 类属性和实例属性

2.1 概念和使用

- **类属性** 就是给 **类对象** 中定义的 **属性**
- 通常用来记录 **与这个类相关** 的特征
- **类属性 不会用于**记录 **具体对象的特征**

示例需求

- 定义一个 **工具类**
- 每件工具都有自己的 `name`
- **需求** —— 知道使用这个类，创建了多少个工具对象？




```python class Tool(object):
```

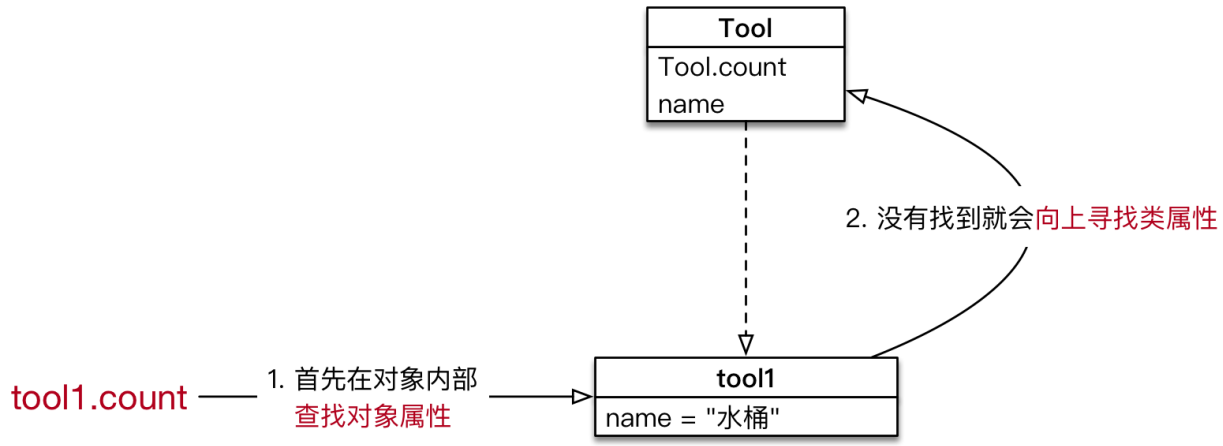
```
# 使用赋值语句，定义类属性，记录创建工具对象的总数
count = 0

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

    # 针对类属性做一个计数+1
    Tool.count += 1
```

创建工具对象

tool1 = Tool("斧头") tool2 = Tool("榔头") tool3 = Tool("铁锹")

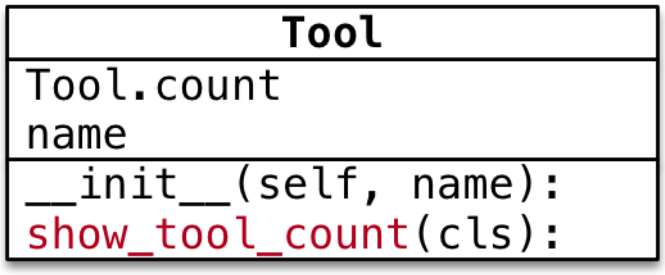
知道使用 Tool 类到底创建了多少个对象？

print("现在创建了 %d 个工具" % Tool.count)

```

### 2.2 属性的获取机制（科普）

- 在 `Python` 中 **属性的获取** 存在一个 **向上查找机制**




- 因此，要访问类属性有两种方式：
  1. **类名.类属性**
  2. **对象.类属性** （不推荐）

**注意**

- 如果使用 `对象.类属性 = 值` 赋值语句，只会 **给对象添加一个属性**，而不会影响到 **类属性的值**

## 03. 类方法和静态方法

### 3.1 类方法

- **类属性** 就是针对 **类对象** 定义的属性
  - 使用 **赋值语句** 在 `class` 关键字下方可以定义 **类属性**
  - **类属性** 用于记录 **与这个类相关** 的特征
- **类方法** 就是针对 **类对象** 定义的方法
  - 在 **类方法** 内部可以直接访问 **类属性** 或者调用其他的 **类方法**

**语法如下**

`python @classmethod def 类方法名(cls): pass`

- 类方法需要用 **修饰器** `@classmethod` 来标识，**告诉解释器这是一个类方法**
- 类方法的 **第一个参数** 应该是 `cls`
  - 由 **哪一个类** 调用的方法，方法内的 `cls` 就是 **哪一个类的引用**
  - 这个参数和 **实例方法** 的第一个参数是 `self` 类似
  - **提示** 使用其他名称也可以，不过习惯使用 `cls`
- 通过 **类名.** 调用 **类方法，调用方法时**，不需要传递 `cls` 参数
- **在方法内部**
  - 可以通过 `cls.` **访问类的属性**
  - 也可以通过 `cls.` **调用其他的类方法**
```

示例需求

- 定义一个 **工具类**
- 每件工具都有自己的 `name`
- **需求** —— 在 **类** 封装一个 `show_tool_count` 的类方法，输出使用当前这个类，创建的对象个数



`python @classmethod def show_tool_count(cls): """显示工具对象的总数""" print("工具对象的总数 %d" % cls.count)`

> 在类方法内部，可以直接使用 `cls` 访问 **类属性** 或者 **调用类方法**

3.2 静态方法

- 在开发时，如果需要在 **类** 中封装一个方法，这个方法：
  - 既 **不需要** 访问 **实例属性** 或者调用 **实例方法**
  - 也 **不需要** 访问 **类属性** 或者调用 **类方法**
- 这个时候，可以把这个方法封装成一个 **静态方法**

语法如下

`python @staticmethod def 静态方法名(): pass`

- **静态方法** 需要用 **修饰器** `@staticmethod` 来标识，**告诉解释器这是一个静态方法**
- 通过 **类名.** 调用 **静态方法**

```python class Dog(object):

```python
# 狗对象计数
dog_count = 0

@staticmethod
def run():

    # 不需要访问实例属性也不需要访问类属性的方法
    print("狗在跑...")

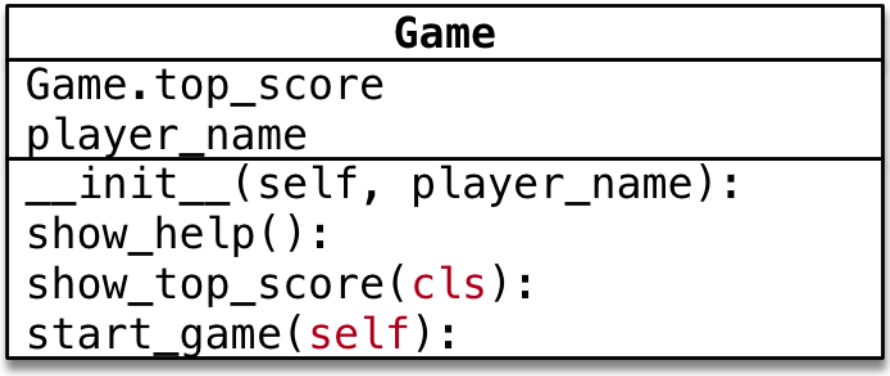
def __init__(self, name):
    self.name = name
```

```

## 3.3 方法综合案例

**需求**

1. 设计一个 `Game` 类
2. 属性：
   - 定义一个 **类属性** `top_score` 记录游戏的 **历史最高分**
   - 定义一个 **实例属性** `player_name` 记录 **当前游戏的玩家姓名**
3. 方法：
   - **静态方法** `show_help` 显示游戏帮助信息
   - **类方法** `show_top_score` 显示历史最高分
   - **实例方法** `start_game` 开始当前玩家的游戏
4. 主程序步骤
   - 1) 查看帮助信息
   - 2) 查看历史最高分
   - 3) 创建游戏对象，开始游戏




**案例小结**

1. **实例方法** —— 方法内部需要访问 **实例属性**
   - **实例方法** 内部可以使用 **类名.** 访问类属性
2. **类方法** —— 方法内部 **只** 需要访问 **类属性**
3. **静态方法** —— 方法内部，不需要访问 **实例属性** 和 **类属性**

**提问**

如果方法内部 即需要访问 **实例属性**，又需要访问 **类属性**，应该定义成什么方法？

**答案**

- 应该定义 **实例方法**
- 因为，**类只有一个**，在 **实例方法** 内部可以使用 **类名.** 访问类属性

```python class Game(object):

```python
# 游戏最高分，类属性
top_score = 0

@staticmethod
def show_help():
    print("帮助信息：让僵尸走进房间")

@classmethod
def show_top_score(cls):
    print("游戏最高分是 %d" % cls.top_score)

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

def start_game(self):
    print("[%s] 开始游戏..." % self.player_name)

    # 使用类名.修改历史最高分
    Game.top_score = 999
```

1. 查看游戏帮助

Game.show_help()

2. 查看游戏最高分

Game.show*top*score()

3. 创建游戏对象，开始游戏

game = Game("小明")

game.start_game()

4. 游戏结束，查看游戏最高分

Game.show*top*score()

```