目录

Python核心编程

语法高级

系统编程

网络编程

正则表达式

1.1 属性与方法

1.1.1 类属性与实例属性

- 类属性不会因实例不同而不同
- 实例属性通常需要类__init__方法初始化
- 实例可以调类属性、类不可以调实例属性

```
"""
类与对象区别
类实例与对象实例区别
"""


class AI(object):
    """
    AI父类
    """
    # AI有是否存活类属性
    isAlive = False

    def __init__(self, _hp, _mp):
        # AI 有血量和魔法值实例属性
        self.hp = _hp
        self.mp = _mp


if __name__ == "__main__":
    # 实例a1、a2中hp、mp属性不同
    a1 = AI(100, 50)
    a2 = AI(60, 120)
    print(a1.mp, a2.mp)
    print(a1.isAlive, a2.isAlive)
    # 实例可以调类属性
    print(AI.isAlive)
    # 类名不可以调实例属性错误
    print(AI.mp)
```

1.1.2 类方法、静态方法、实例方法

- 实例方法用来操作实例属性
- 类方法可以访问到类的内容
- 静态方法不可以访问类内容和实例内容

```
""
类与对象区别
类方法 静态方法 实例方法
"""


class AI(object):
    """
    AI类
```

```
    """
    def __init__(self, _speed):
        self.speed = _speed

    # 声明类方法run
    def run(self):
        print("run速度为%d" % self.speed)

    # 声明静态方法
    @staticmethod
    def dead():
        print("AI死亡")

    # 声明类方法
    @classmethod
    def printclassinfo(cls):
        print("类文档%s" % cls.__doc__)


if __name__ == "__main__":
    a1 = AI(50)
    a1.run()
    a1.dead()
    a1.printclassinfo()
    AI.dead()
```

1.1.3 为什么要使用类属性、类方法

- 为了减少多个类实例所产生的内存空间
- 类方法与类属性属于类，而不属于某个实例

1.2 Python是动态语言

1.2.1 动态语言的定义

是在运行时可以改变其结构的语言

运行中新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。

Python、JavaScript、PHP 为动态语言，不需要编译，直接由解释器解释执行，效率低下

C 、 C++、Java、C#不属于动态语言，需要经过编译成中间语言或者机器语言在进行执行

- 动态的添加类属性

通过类名

```
class AI(object):

    def __init__(self, _hp):
        self.hp = _hp


a1 = AI(50)
print(a1.hp)
AI.isalive = False #通过类名动态的添加类属性
print(AI.isalive)
print(a1.isalive)实例也拥有了该类属性
```

- 动态的添加实例属性

通过实例名

```
class AI(object):

    def __init__(self, _hp):
        self.hp = _hp


a1 = AI(50)
print(a1.hp)
a1.mp = 80 #动态的添加魔法值属性
print(a1.mp)
```

如果随意添加岂不是封装不安全了？

可以通过__slots__限制添加的内容

```
class AI(object):

    __slots__ = ('hp', 'mp',)


a1 = AI()
a1.hp = 50
a1.mp = 100
# 因为__slots__限制了只能添加hp mp
```

```
# 添加attack会报错
a1.attack =100
```

- 动态的添加实例方法

通过实例名结合types模块

```
实例名.实例方法名= types.MethodType(定义好的实例方法名, 实例)
```

```
import types
class AI(object):
    def __init__(self):
        pass


# 定义实例方法
def move(self):
    print("move")

a1 = AI()
a1.move = types.MethodType(move, a1)
a1.move()
```

- 动态的添加类方法、静态方法

通过类名

```
import types
class AI(object):
    def __init__(self):
        pass


#定义类方法
@classmethod
def attack(cls):
    print("attack")

# 定义静态方法
@staticmethod
def dead():
    print("dead")


a1 = AI()
AI.attack = attack
AI.attack()
AI.dead = dead
AI.dead()
```

动态的删除属性、方法

```
del a1.move
a1.move()
```

```
delattr(a1, "move")
a1.move()
```

1.3 生成器

1.3.1 什么是生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。

在**Python**中，这种一边循环一边计算的机制，称为生成器：**generator**

1.3.2 生成器的创建

方法**1**

把一个列表生成式的 **[ ]** 改成 **( )**

```
"""
生成器
"""
L = [x*x for x in range(5)]
# 列表类型为class 'list
print(L, type(L))
G = (x*x for x in range(5))
# 生成器类型为 class 'generator
print(G, type(G))
# 可以使用next获取元素
print(next(G))
print("----")
# 可以使用for对生成器遍历
for r in G:
    print(r)
```

方法**2**

用函数来创建

打印裴波拉契算法（从第三项开始均为前两项的和）

```
def fib(times):
    n = 0
    a, b = 0, 1
    yield a
    yield b
    while n<times:
        # 0 1 1 2 3 5 8
        a, b = b, b+a
        print(b)
        n += 1
    print("finish")

r = fib(5)
print(r)
```

只需要使用将print改为yield即可使用生成器

```
def fib(times):
    n = 0
    a, b = 0, 1
    yield a
    yield b
    while n<times:
        # 0 1 1 2 3 5 8
        a, b = b, b+a
        yield b
        n += 1
    return "finish"

r = fib(5)
print(r)
# r为生成器对象
```

获取函数返回值需要捕获异常

```
def fib(times):
    n = 0
    a, b = 0, 1
    yield a
    yield b
    while n<times:
        # 0 1 1 2 3 5 8
        a, b = b, b+a
        yield b
        n += 1
    return "finish"

r = fib(5)
print(r)

while True:
    try:
        x = next(r)
        print(x)
    except StopIteration as e:
        print(e.value)
        break
```

例子

```
def gen():
    n = 0
    while n < 5:
        yield n
        n += 1

r = gen()
print(r, type(r))
for r1 in r:
    print(r1)
```

1.3.3 生成器的特点：

- 节约内存

- 迭代到下一次的调用时，所使用的参数都是第一次所保留下的，即是说，在整个所有函数调用的参数都是第一次所调用时保留的，而不是新创建的

1.4 可迭代、迭代器

1.4.1 可迭代对象

可以直接作用于 for 循环的数据类型有以下几种：

一类是集合数据类型，如 list 、 tuple 、 dict 、 set 、 str 等；

一类是 generator ，包括生成器和带 yield 的generator function。

这些可以直接作用于 for 循环的对象统称为可迭代对象： Iterable 。

1.4.2 判断是否可以迭代

```
可以使用 isinstance() 判断一个对象是否是 Iterable 对象：

In [50]: from collections.abc import Iterable

In [51]: isinstance([], Iterable)
Out[51]: True

In [52]: isinstance({}, Iterable)
Out[52]: True

In [53]: isinstance('abc', Iterable)
Out[53]: True

In [54]: isinstance((x for x in range(10)), Iterable)
Out[54]: True

In [55]: isinstance(100, Iterable)
Out[55]: False
```

1.4.3 迭代器

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

```
可以使用 isinstance() 判断一个对象是否是 Iterator 对象：

In [56]: from collections.abc import Iterator

In [57]: isinstance((x for x in range(10)), Iterator)
Out[57]: True

In [58]: isinstance([], Iterator)
Out[58]: False

In [59]: isinstance({}, Iterator)
Out[59]: False

In [60]: isinstance('abc', Iterator)
Out[60]: False

In [61]: isinstance(100, Iterator)
Out[61]: False
```

1.4.4 iter()函数

list 、 dict 、 str是可迭代的（可以for遍历）不是迭代器（没有next方法）

生成器是可迭代的也是迭代器

把 list 、 dict 、 str 等 Iterable 变成 Iterator 可以使用 iter() 函数：

```
In [62]: isinstance(iter([]), Iterator)
Out[62]: True

In [63]: isinstance(iter('abc'), Iterator)
Out[63]: True
```

1.4.5 参考

```
"""
迭代器与可迭代
可迭代（可以for遍历）
迭代器（有next方法）
"""
from collections.abc import Iterator, Iterable
G = (x*x for x in range(5))
print(G, type(G))
print(isinstance(G, Iterator))
print(isinstance(G, Iterable))
print(next(G)) #可以next是迭代器

for r in G: #可以for是可迭代
    print(r)

L = [1, 2, 3, 4, 5]
print(isinstance(L, Iterable)) #True
print(isinstance(L, Iterator)) #False

L = iter(L)
print(isinstance(L, Iterator))# True
```

1.5 闭包

1.5.1 什么是闭包

在函数内部再定义一个函数，内部函数用到了外边函数的变量，并且外部函数将内部函数的引用返回，那么将这个函数以及用到的一些变量称之为闭包

1.5.2 例子

```
def func1(a):
    def func2(b):
        return a+b
    return func2

f1 = func1(100)
f2 = func1(50)
print(f1(1))
print(f1(1))
```

1.5.3 闭包思考

闭包似优化了变量，原来需要类对象完成的工作，闭包也可以完成

由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内存

1.6 装饰器

- 装饰器的作用：在不改变原函数的情况下给函数增加功能！
- 装饰器由闭包和语法糖组成

1.6.1 案例分析

用户查询商品列表业务逻辑

```
def selectgoods():
    print("开始查询商品列表")
selectgoods()
```

调用selectgoods方法即可查询商品列表

业务更改

在查询商品列表之前需要添加权限验证

```
def selectgoods():
    name = input("请输入用户名")
    if name == "zzy":
        print("开始查询商品")
    else:
        print("无权查看")
selectgoods()
```

功能实现呃，但是改变了原有函数的

在增加新功能的情况下不改变原有业务逻辑实现，违背了开闭原则

1.6.2 使用装饰器

@函数名 是python的一种语法糖。

```
def checkaccess(sg):
    def check():
        name = input("输入用户名")
        if name == "zzy":
            sg()
        else:
            print("无权查看")
    return check



@checkaccess
def selectgoods():
    print("开始查询商品")
selectgoods()
```

使用装饰器之后则selectgoods函数没有发生变化

1.6.3 @函数名 执行逻辑

检测到需要执行的函数selectgoods 拥有装饰器@checkaccess

不执行selectgoods 而是将selectgoods 作为参数传入checkaccess方法，并且执行checkaccess 方法 此时sg =selectgoods

将checkaccess的执行结果返回，即将check方法的引用返回，即实际执行的是check方法

1.6.4 装饰器使用场景

引入日志

函数执行时间统计

执行函数前预备处理

执行函数后清理功能

权限校验等场景

缓存

1.7 浅拷贝 深拷贝

1.7.1 '=' 一般意义的复制

我们所说的一般意义的"等于号"相当于引用，即原始队列改变，被赋值的队列也会作出相同的改变。

直接赋值,传递对象的引用而已,原始列表改变，被赋值的队列也会做相同的改变

```
list1 = [1,2,[3,4]]
print(list1)
list2 = list1
print(list2)
list1.append(5)
print(list1,list2)
list1[2].append(3.5)
print(list1,list2)
```

1.7.2 浅拷贝copy

copy是浅拷贝，并不会产生一个独立的对象单独存在，他只是将原有的数据块打上一个新标签，所以当其中一个标签被改变的时候，数据块就会发生变化，另一个标签也会随之改变。这就和我们寻常意义上的复制有所不同了。

这里需要注意，对于简单对象来说，深拷贝、浅copy并没有区分，改变原始队列，赋值队列并不会改变。

但是对于嵌套队列来说，改变了子对象的值，浅拷贝会随之变化，深拷贝不会。

```
import copy
list1 = [1,2,[3,4]]
print(list1)
list2 = copy.copy(list1)
print(list2)
# list1.append(5)
# print(list1,list2)
list1[2].append(3.5)
print(list1,list2)
```

1.7.3 深拷贝 deepcopy

Python中的copy动作，对于一个复杂对象的子对象并不会完全复制，什么是复杂对象的子对象呢？就比如序列里的嵌套序列，字典里的嵌套序列等都是复杂对象的子对象。

对于子对象，浅拷贝动作会把它当作一个公共镜像存储起来，所有对他的复制都被当成一个引用，所以说当其中一个引用将镜像改变了之后另一个引用使用镜像的时候镜像已经被改变了。

对于子对象，深拷贝动作会将复杂对象的每一层复制一个单独的个体出来，因而二者完全独立。

```
import copy
list1 = [1,2,[3,4]]
print(list1)
list2 = copy.deepcopy(list1)
print(list2)
# list1.append(5)
# print(list1,list2)
list1[2].append(3.5)
print(list1,list2)
```

1.9 元类

1.9.1 什么是元类

元类就是用来创建类的“东西”。

Python中类也是一种对象。

```
#定义一个类
# Python解释器在执行的时候会创建一个对象
class AI(object):
    pass
```

```
使用type方法查看类类型<class 'type'>
```

```
print(type(AI))
# <class 'type'>
```

type拥有创建类的功能

1.9.2 使用type创建类

type可以动态的创建类。

type(类名, 由父类名称组成的元组，包含属性的字典)

```
AI = type("AI", (), {"hp": 100})
print(AI)
print(AI.__class__)
a1 = AI()
print(a1.__class__)
```

```
#NPC继承AI
NPC = type("NPC", (AI,), {"canMove": False})
print(NPC)
print(NPC.__bases__)
print(NPC.hp)
```

添加实例方法、静态方法、类方法

```
#定义普通方法
def speak(self):
    print("Welcome")
#定义静态方法
@staticmethod
def attack():
    print("Attack")
#定义类方法
@classmethod
def dead(cls):
    print("Dead")
NPC = type("NPC", (AI,), {"canMove": False, "speak": speak, "attack": attack, "dead":dead})
npc = NPC()
```

```
npc.speak()
NPC.attack()
NPC.dead()
```

1.9.3 自定义元类

- 元类的调用

```
class NPC(AI):
    pass
```

Python在创建NPC类时先不在内存中创建、首先查看NPC中有__metaclass__这个属性吗

如果有，Python会通过__metaclass__创建一个名字为NPC的类(对象)

如果Python没有找到__metaclass__，它会继续在AI（父类）中寻找__metaclass__属性，并尝试做和前面同样的操作。

如果Python在任何父类中都找不到__metaclass__，它就会在模块层次中去寻找__metaclass__，并尝试做同样的操作。

如果还是找不到__metaclass__,Python就会用内置的type来创建这个类对象

- 例子：实现所有的属性均需要以类名开始

```
def renameattribute(classname, parentclass, attributedic):
    newattribute = {}
    for key, value in attributedic.items():
        if not key.startswith("__"):
            key = classname.lower() + key
            newattribute[key] = value
    # for key, value in newattribute.items():
    #     print(key, value)
    return type(classname, parentclass, newattribute)

class AI(object, metaclass=renameattribute):
    isalive = False

ai = AI()
print(hasattr(ai, "isalive")) #false
print(hasattr(ai, "aiisalive")) #true
```

1.9.4 使用元类好处：

- 拦截类的创建
- 修改类
- 返回修改之后的类

1.10 垃圾回收

高级语言一般都有垃圾回收机制，其中c、c++使用的是用户自己管维护内存的方式，这种方式比较自由，但如果回收不当也会引起垃内存泄露等问题。而python采用的是引用计数机制为主，标记-清理和分代收集两种机制为辅的策略。

1.10.1 引用计数

python中一切皆对象，所以python底层计数结构地就可以抽象为：

```
引用计数结构体{
引用计数
引用的对象
}
```

对象什么情况下会被删除：引用计数=0

什么情况下引用计数+1，什么情况下-1，当引用次数为0时，肯定就是需要进行回收的时刻。

- 引用计数**+1**的情况

```
1、对象被创建时，例如a="hello zzy"
2、对象被copy引用时，例如 b=a，此时a引用计数+1
3、对象被作为参数，传入到一个函数中时
4、对象作为一个子元素，存储到容器中时，例如 list=[a,b]
```

- 引用计数**-1**的情况

```
1、对象别名被显示销毁，例如 del a
2、对象引用被赋予新的对象，例如b=c，此时b引用计数-1（对照引用计数+1的情况下的第二点来看）
3、一个函数离开他的作用域，例如函数执行完成，它的引用参数的引用计数-1
4、对象所在容器被销毁，或者从容器中删除。
```

- 例子

```
import sys
print(sys.getrefcount("hello zzy"))
a = "hello zzy"
print(sys.getrefcount("hello zzy"))
b = "hello zzy"
print(sys.getrefcount("hello zzy"))
c = b
print(sys.getrefcount("hello zzy"))
def pp(str):
    print(sys.getrefcount(str))
pp(a)
print(sys.getrefcount("hello zzy"))

del c
print(sys.getrefcount("hello zzy"))
del b
print(sys.getrefcount("hello zzy"))
del a
print(sys.getrefcount("hello zzy"))
```

引用计数机制优点

1、简单、直观

2、实时性，只要没有了引用就释放资源。

引用计数机制缺点

1、维护引用计数需要消耗一定的资源

2、循环应用时，无法回收。也正是因为这个原因，才需要通过标记-清理和分代收集机制来辅助引用计数机制。

```
while True:
    l1 = [x for x in range(1000)]
    l2 = [x for x in range(1000)]
    l1.append(l2)
    l2.append(l1)
    del l1
    del l2
```

1.10.2 标记-清除

由上面内容我们可以知道，引用计数机制有两个缺点，缺点1还可以勉强让人接受，缺点2如果不解决，肯定会引起内存泄露，为了解决这个问题，引入了标记删除。

我们先来看个实例，从实例中领会标记删除：

```
a=[1,2]#假设此时a的引用为1
b=[3,4]#假设此时b的引用为1
#循环引用
a.append(b)#b的引用+1=2
b.append(a)//a的引用+1=2

假如现在需要删除a,应该如何回收呢？


c=[5,6]#假设此时c的引用为1
d=[7,8]#假设此时d的引用为1
#循环引用
c.append(d)#c的引用+1=2
d.append(c)#d的引用+1=2

假如现在需要同时删除c、d，应该如何回收呢？
```

首先我们应该已经知道，不管上面两种情况的哪一个都无法只通过计数来完成回收，因为随便删除一个变量，它的引用只会-1，变成1，还是大于0，不会回收，为了解决这个问题，开始看标记删除来大展神威吧。

```
python标记删除是通过两个容器来完成的：死亡容器、存活容器。

首先，我们先来分析情况2，删除c、d
删除后，c的引用为1，d的引用为1，根据引用计数，还无法删除

标记删除第一步：对执行删除操作后的每个引用-1，此时c的引用为0，d的引用为0，把他们都放到死亡容器内。把那些引用仍然大于0的放到存活容器内。

标记删除第二步：遍历存活容器，查看是否有的存活容器引用了死亡容器内的对象，如果有就把该对象从死亡容器内取出，放到存活容器内。
由于c、d都没有对象引用他们了，所以经过这一步骤，他们还是在死亡组。

标记删除第三部：将死亡组所有对象删除。
这样就完成了对从c、d的删除。
```

同样道理，我们来分析：只删除a的过程：

```
标记删除第一步：对执行删除操作后的每个引用-1，那么a的引用就是0，b的引用为1，将a放到死亡容器，将b放到存活容器。
标记删除第二步：循环存活容器，发现b引用a，复活a：将a放到存活容器内。
标记删除第三步：删除死亡容器内的所有对象。
```

综上所说，发现对于循环引用，必须将循环引用的双发对象都删除，才可以被回收。

1.10.3 分代收集

经过上面的【标记-清理】方法，已经可以保证对垃圾的回收了，但还有一个问题，【标记-清理】什么时候执行比较好呢，是对所有对象都同时执行吗？

同时执行很显然不合理，我们知道，存活越久的对象，说明他的引用更持久（好像是个屁话，引用不持久就被删除了），为了更合理的进行【标记-删除】，就需要对对象进行分代处理，思路很简单：

```
1、新创建的对象做为0代
2、每执行一个【标记-删除】，存活的对象代数就+1
3、代数越高的对象（存活越持久的对象），进行【标记-删除】的时间间隔就越长。这个间隔，江湖人称阀值。
```

1.10.4 小整数对象池与intern机制

由于整数使用广泛，为了避免为整数频繁销毁、申请内存空间，引入了小整数对象池。[-5,257)是提前定义好的，不会销毁，单个字母也是。

那对于其他整数，或者其他字符串的不可变类型，如果存在重复的多个，例如：

```
a1="mark"
a2="mark"
a3="mark"
a4="mark"
....
a1000="mark"
```

如果每次声明都开辟出一段空间，很显然不合理，这个时候python就会使用intern机制，靠引用计数来维护。

总计：

```
1、小整数[-5，257)：共用对象，常驻内存
2、单个字符：共用对象，常驻内存
3、单个单词等不可变类型，默认开启intern机制，共用对象，引用计数为0时销毁。
```

1.1.10 内建模块、函数、属性

1.10.1 内建模块

Python有一套很有用的标准库(standard library)。标准库会随着Python解释器，一起安装在你的电脑中的。 它是Python的一个组成部分。这些标准库是Python为你准备好的利器，可以让编程事半功倍。

- 常用标准库

标准库	说明
builtins	内建函数默认加载
os	操作系统接口
sys	Python自身的运行环境
functools	常用的工具
json	编码和解码 JSON 对象
logging	记录日志，调试
multiprocessing	多进程
threading	多线程
copy	拷贝
time	时间
datetime	日期和时间
calendar	日历
hashlib	加密算法
random	生成随机数
re	字符串正则匹配
socket	标准的 BSD Sockets API
shutil	文件和目录管理
glob	基于文件通配符搜索

- 常用扩展库

扩展库	说明
requests	使用的是 urllib3，继承了urllib2的所有特性
urllib	基于http的高层库
scrapy	爬虫
beautifulsoup4	HTML/XML的解析器
celery	分布式任务调度模块
redis	缓存
Pillow(PIL)	图像处理
xlsxwriter	仅写excle功能,支持xlsx
xlwt	仅写excle功能,支持xls ,2013或更早版office

xlrd	仅读excle功能
elasticsearch	全文搜索引擎
pymysql	数据库连接库
mongoengine/pymongo	mongodbpython接口
matplotlib	画图
numpy/scipy	科学计算
django/tornado/flask	web框架
xmltodict	xml 转 dict
SimpleHTTPServer	简单地HTTP Server,不使用Web框架
gevent	基于协程的Python网络库
fabric	系统管理
pandas	数据处理库
scikit-learn	机器学习库

1.10.2 内建函数

Build-in Function,启动python解释器，输入 `dir(__builtins__)` , 可以看到很多python解释器启动后默认加载的属性和函数，这些函数称之为内建函数， 这些函数因为在编程时使用较多，cpython解释器用c语言实现了这些函数，启动解释器 时默认加载。

这些函数数量众多，不宜记忆，开发时不是都用到的，待用到时再help(function), 查看如何使用，或结合百度查询即可，在这里介绍些常用的内建函数。

```
import builtins
print(dir(builtins))
```

dir

在 Python 中，有大量的内置模块，模块中的定义（例如：变量、函数、类）众多，不可能全部都记住，这时 dir() 函数就非常有用了。

dir() 是一个内置函数，用于列出对象的所有属性及方法。在 Python 中，一切皆对象，模块也不例外，所以模块也可以使用 dir()。除了常用定义外，其它的不需要全部记住它，交给 dir() 就好了。

```
import math
print(dir(math))
```

range

```
range(stop) -> list of integers
range(start, stop[, step]) -> list of integers
```

start:计数从start开始。默认是从0开始。例如range（5）等价于range（0， 5）;

stop:到stop结束，但不包括stop.例如：range（0， 5） 是[0, 1, 2, 3, 4]没有5

step:每次跳跃的间距，默认为1。例如：range（0， 5） 等价于 range(0, 5, 1)

python3中range返回一个迭代值，如果想得到列表,可通过list函数

```
a = range(5)
list(a)
```

创建列表的另外一种方法

```
testList = [x+2 for x in range(5)]
```

map

map函数会根据提供的函数对指定序列做映射

```
map(...)
        map(function, sequence[, sequence, ...]) -> list
```

function:是一个函数

sequence:是一个或多个序列,取决于function需要几个参数

返回值是一个list

```
def f1( x, y ):
    return (x,y)

l1 = [ 0, 1, 2, 3, 4, 5, 6 ]
l2 = [ 'Sun', 'M', 'T', 'W', 'T', 'F', 'S' ]
l3 = map( f1, l1, l2 )
print(list(l3))
#结果为:[(0, 'Sun'), (1, 'M'), (2, 'T'), (3, 'W'), (4, 'T'), (5, 'F'), (6, 'S')]
```

filter

filter函数会对指定序列执行过滤操作

```
filter(...)
    filter(function or None, sequence) -> list, tuple, or string
```

function:接受一个参数，返回布尔值True或False

sequence:序列可以是str，tuple，list

filter函数会对序列参数sequence中的每个元素调用function函数，最后返回的结果包含调用结果为True的元素。

```
def fun(x):
    return x % 2 != 0

l1 = [1, 2, 5, 7, 9]
l = filter(fun, l1)
for r in l:
    print(r)
```

reduce

reduce函数，reduce函数会对参数序列中元素进行累积

```
reduce(...)
```

```
    reduce(function, sequence[, initial]) -> value
```

function:该函数有两个参数

sequence:序列可以是str，tuple，list

initial:固定初始值

reduce依次从sequence中取一个元素，和上一次调用function的结果做参数再次调用function。 第一次调用function时，如果提供initial参数，会以sequence中的第一个元素和initial 作为参数调用function，否则会以序列sequence中的前两个元素做参数调用function。 注意function函数不能为None。

```
from _functools import reduce
def fun(x, y):
    return x + y
l = [1, 3, 5, 7]
result = reduce(fun,l)
print(result)
```

sorted

sorted 会对指定的序列排序

```
l = [1, 3, 5, 7, 2]

l1 = sorted(l, reverse=1)
print(l1)
```

1.10.3 内建属性

常用专有属性	说明	触发方式
`__init__`	构造初始化函数	创建实例后,赋值时使用,在 `__new__` 后
`__class__`	实例所在的类	实例. `__class__`
`__str__`	实例字符串表示,可读性	print(类实例),如没实现，使用repr结果
`__repr__`	实例字符串表示,准确性	类实例 回车 或者 print(repr(类实例))
`__del__`	析构	del删除实例
`__dict__`	实例自定义属性	实例拥有属性
`__doc__`	类文档,子类不继承	help(类或实例)
`__getattribute__`	属性访问拦截器	访问实例属性时
`__bases__`	类的所有父类构成元素	`类名.__bases__`

2.1 进程线程对比

根本区别：

- 进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位
- 每个进程都有独立的代码和数据空间、同一类线程共享代码和数据空间
- 在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行
- 内存分配方面：系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源
- 包含关系：没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的

优缺点：

- 线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护，而进程正相反。

2.2 进程

2.2.1 进程vs 程序

编写完毕的代码，在没有运行的时候，称为程序

正在运行着的代码，称为进程

进程除了包含代码之外还要有代码的运行环境

2.2.2 fork()

python的os模块封装了常见的系统相关，其中就包括fork（Windows不支持），可以在python程序中轻松创建子进程

```
import os
pid = os.fork()
```

os.fork()方法执行一次回返回两次，操作系统自动把当前进程（父进程）赋值一份（子进程），然后在父进程和子进程内分别执行

- 子进程返回0
- 父进程返回子进程的id

```
import os
pid = os.fork()
print("do")
if pid == 0:
    print("child process pid %d" % pid)
else:
    print("parent process pid %d" % pid)
```

getpid()、getppid()

```
import os

pid = os.fork()
print("do")
if pid == 0:
    # print("child process id %d" % pid)
    print("current child process pid %d parent pid %d" % (os.getpid(), os.getppid()))
else:
    # print("parent process id %d" % pid)
    print("current parent process pid %d child pid %d" % (os.getpid(), pid))
```

输出结果

```
do
current parent process pid 4296 child pid 4297
do
current child process pid 4297 parent pid 4296
```

子进程可以创建子进程么

```
import os
print("do")
pid1 = os.fork()
if pid1 == 0:
    print("child1 pid %d parent pid %d" % (os.getpid(), os.getppid()))
    pid2 = os.fork()
    if pid2 == 0:
        print("child2 pid %d parent pid %d" % (os.getpid(), os.getppid()))
    else:
        print("parent pid %d child2 pid %d" % (os.getppid(), pid2))
else:
    print("parent pid %d child1 pid %d" % (os.getpid(), pid1))
```

输出结果

```
do
parent pid 4529 child1 pid 4530
child1 pid 4530 parent pid 4529
parent pid 4529 child2 pid 4531
child2 pid 4531 parent pid 4530
```

父子进程的执行没有先后顺序

多进程修改全局变量

在多进程中每个进程都复制父进程的所有数据，互不影响

```
import os
pid = os.fork()
num = 1
print("do", num)
if pid == 0:
    num+=1
    print(num)
else:
    num+=1
    print(num)
```

输出结果

```
do 1
2
do 1
2
```

2.2.3 multiprocessing模块

windows下如何编写多进程引用程序，python跨平台模块multiprocessing

```
from multiprocessing import Process
import os
def fun():
    print("child pid %d" % os.getpid())

print("parent pid %d" % os.getpid())
p = Process(target=fun)
```

```
print("has not exec")
p.start()
p.join()
print("exec finish")
```

join方法可以等待子线程执行结束在往下运行

Process语法结构如下：

Process([group [, target [, name [, args [, kwargs]]]]])

target：表示这个进程实例所调用对象；

args：表示调用对象的位置参数元组；

kwargs：表示调用对象的关键字参数字典；

name：为当前进程实例的别名；

Process类常用方法：

is_alive()：判断进程实例是否还在执行；

join([timeout])：是否等待进程实例执行结束，或等待多少秒；

start()：启动进程实例（创建子进程）；

run()：如果没有给定target参数，对这个对象调用start()方法时，就将执行对象中的run()方法；

terminate()：不管任务是否完成，立即终止；

Process类常用属性：

name：当前进程实例别名，默认为Process-N，N为从1开始递增的整数；

pid：当前进程实例的PID值(调用过start之后有int值)；

```
from multiprocessing import Process
import os
import time
def fun(name, age , **kwargs1):
    print(name, age, kwargs1)
    time.sleep(1)
    print(name, age, kwargs1)
    time.sleep(1)
    print(name, age, kwargs1)
    time.sleep(1)
    print(name, age, kwargs1)
print("parent pid %d" % os.getpid())
p = Process(name="child", target=fun, args=("zzy", 20), kwargs={"isalive": False})
p.start()
print("p name %s pid %d" % (p.name, int(p.pid) ))
time.sleep(2)
p.terminate()
p.join()
print("child finish")
```

封装Process类

当Process构造没有target参数参数化，调用start方法会执行对象的run方法

```
from multiprocessing import Process
```

```
def run():
    print("run")

p = Process()
p.run = run

p.start()
p.join()
print("finish")
```

封装类继承Process类并在内部创建run方法

```
from multiprocessing import Process
import os

class MyProcess(Process):
    def __init__(self):
        # init parent class
        Process.__init__()
    def run(self):
        print("pid %d" % os.getpid())

p = MyProcess()
p.start()
p.join()
print("finish")
```

2.2.4 进程池 Pool

当需要创建的子进程数量不多时，可以直接利用multiprocessing中的Process动态成生多个进程，但如果是上百甚至上千个目标，手动的去创建进程的工作量巨大，此时就可以用到multiprocessing模块提供的Pool方法。

初始化进程池时需要给定最大进程数，当请求Pool时会根据当前池子中获取进程，如果有可用进程则执行，如果没有可用进程则等待，直到有可用进程

```
from multiprocessing import Pool
import os
import time
def fun(name, age, **kwargs):
    print("pid %d" % os.getpid())
    print(name, age, kwargs)
    time.sleep(1)
pool = Pool(5)
for i in range(20):
    pool.apply_async(fun, ("zzy", 20), {"isalive": False})
pool.close()
pool.join()
print("finish")
```

multiprocessing.Pool常用函数解析：

apply_async(func[, args[, kwds]]) ：使用非阻塞方式调用func（并执行，堵塞方式必须等待上一个进程退出才能执行下一个进程），args为传递给func的参数列表，kwds为传递给func的关键字参数列表；

apply(func[, args[, kwds]])：使用阻塞方式调用func

close()：关闭Pool，使其不再接受新的任务；

terminate()：不管任务是否完成，立即终止；

join()：主进程阻塞，等待子进程的退出， 必须在close或terminate之后使用；

```
from multiprocessing import Pool
import os
import time
def fun(name, age, **kwargs):
    print("pid %d" % os.getpid())
    time.sleep(1)
pool = Pool(5)
for i in range(20):
    pool.apply_async(fun, ("zzy", 20), {"isalive": False})
pool.close()
time.sleep(1)
pool.terminate()
# pool.join()
print("finish")
```

阻塞、非阻塞

```
from multiprocessing import Pool
import os
import time

def fun(i):
    print("run %d" % i)
    time.sleep(0.2)
pool = Pool(5)
start = time.time()
for i in range(20):
    # pool.apply_async(fun, (i,))
    pool.apply(fun, (i,))
end =time.time()
print(end-start)
pool.close()
pool.join()
print("finish")
```

2.2.5 Queue 进程间通信

Queue用法

put get

put_nowait get_nowait

常用方法 size() empty() full() get()

默认block参数为true阻塞执行

```
from multiprocessing import Queue
from multiprocessing import ProcessError
q = Queue(5)
q.put(1)
q.put(2)
q.put(3)
q.put(4)
```

```
q.put(5)

q.put(6) #no except
# q.put(6, False) ==q.put_nowait(6) except
```

使用**Queue**共享数据

```
from multiprocessing import Process
from multiprocessing import Pool
from multiprocessing import Queue
from multiprocessing import Value
import time
def read(q):
    while True:
        print("pqueue size %d" % q.qsize())
        r = q.get()
        print("get %d" % r)
        print("pqueue size %d" % q.qsize())
def write(q):
    for r in range(5):
        q.put(r)
        print("put %d " % r)
        print("queue size %d" % q.qsize())
        time.sleep(2)

q = Queue(10)
q.put(-1)
q.put(-2)
pr = Process(target=read, args=(q,))
pw = Process(target=write, args=(q,))
pr.start()
pw.start()

pr.join()
pw.join()

print("finish")
```

进程池下共享**Queue**

```
from multiprocessing import Pool
from multiprocessing import Manager
import time
def read(q):
    while True:
            print("pqueue size %d" % q.qsize())
            r = q.get()
            print("get %d" % r)
            print("pqueue size %d" % q.qsize())
            time.sleep(0.1)
def write(q):
    for r in range(5):
        q.put(r)
        print("put %d " % r)
        print("queue size %d" % q.qsize())
        time.sleep(2)
q = Manager().Queue(5)
q.put(-2)
```

```
q.put(-1)
pool = Pool(5)

pool.apply_async(write, (q,))
pool.apply_async(read, (q,))
pool.close()
pool.join()
print("finish")
```

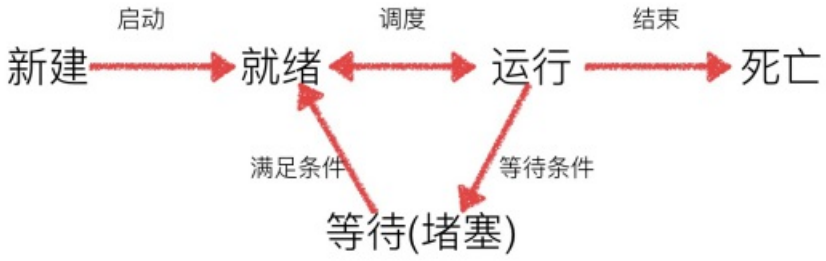
2.3 多线程-threading

2.3.1 使用threading模块

多线程并发花费时间要短

创建好的线程，需要调用start()启动

线程状态




```
import time
def prt():
    print("hello")
    time.sleep(1)

start = time.time()
for r in range(5):
    prt()
end = time.time()
print(end-start)

import threading
start = time.time()
for i in range(5):
    t = threading.Thread(target=prt)
    t.start()
end = time.time()
print(end-start)
```

2.3.2 获取线程数

```
import threading
import time
def fun():
    time.sleep(1)
    print("hello")
t = threading.Thread(target=fun)
t.start()
# print(threading.enumerate())
while True:
    leng = len(threading.enumerate())
    print("num%d" % leng)
    time.sleep(0.2)
```

2.3.3 threading封装

没有使用target参数运行对象内部run方法

```
import threading
import time
def fun():
    time.sleep(1)
    print("hello")
t = threading.Thread()
t.run = fun
t.start()
```

封装类

```
import threading
import time
class MyThread(threading.Thread):
    def run(self):
        print("run")

t = MyThread()
t.start()
```

thread 执行顺序不确定

2.3.4 threading共享全局变量

```
import threading
num = 1
def fun1():
    num = 2
    print(num)
fun1()
print(num)
def fun2():
    global num
    num = 3
    print(num)
fun2()
print(num)
def fun3():
    global num
    num = 4
    print(num)
t = threading.Thread(target=fun3)
t.start()
print(num)
```

一个进程内的线程直接可以共享全局变量

多个线程同时操作共享变量造成数据混乱

```
import threading
num = 0
def fun1():
    for r in range(1000000):
```

```
        global num
        num += 1

p1 = threading.Thread(target=fun1)
p1.start()
# time.sleep(2)   注释取消会改变结果

class MyThread(threading.Thread):
    def run(self):
        for r in range(1000000):
            global num
            num += 1

t = MyThread()
t.start()
time.sleep(2)
print(num)
```

2.3.5 互斥锁

当多个线程几乎同时修改某一个共享数据的时候，需要进行同步控制

线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁。

互斥锁为资源引入一个状态：锁定/非锁定。

某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源。

互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

互斥锁状态

```
# create
mutex = threading.Lock()
# lock
mutex.acquire()
# release
mutex.release()
```

mutex.acquire()默认blocking为True，表示当前线程去获取加锁资源时如果加锁不成功则阻塞直到加锁成功

使用互斥锁

```
import threading
import time
num = 0
def fun1():
    global num
    for r in range(1000000):
        mutex.acquire()
        num += 1
        mutex.release()
def fun2():
    global num
    for r in range(1000000):
        mutex.acquire()
        num += 1
        mutex.release()
```

```
mutex = threading.Lock()
t1 = threading.Thread(target=fun1)
t1.start()
# time.sleep(2)
t2 = threading.Thread(target=fun1)
t2.start()

time.sleep(2)
print(num)
```

优缺点

锁的好处：

确保了某段关键代码只能由一个线程从头到尾完整地执行

锁的坏处：

阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了

由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁

死锁

```
import threading
import time
def fun1():
    if mutex1.acquire():
        time.sleep(2)
        if mutex2.acquire():
            print("finish")
def fun2():
    if mutex2.acquire():
        time.sleep(2)
        if mutex1.acquire():
            print("finish")
mutex1 = threading.Lock()
mutex2 = threading.Lock()
t1 = threading.Thread(target=fun1)
t2 = threading.Thread(target=fun2)
t1.start()
t2.start()
```

使用互斥锁完成线程同步

通过多把锁完成控制线程的执行顺序

```
import threading
mutex1 = threading.Lock()
mutex2 = threading.Lock()
mutex2.acquire()
mutex3 = threading.Lock()
mutex3.acquire()
mutex4 = threading.Lock()
mutex4.acquire()
def fun1():
    while True:
        if mutex1.acquire():
            print("fun1 do")
```

```
            mutex2.release()
def fun2():
    while True:
        if mutex2.acquire():
            print("fun2 do")
            mutex3.release()
def fun3():
    while True:
        if mutex3.acquire():
            print("fun3 do")
            mutex4.release()

t1 = threading.Thread(target=fun1)
t2 = threading.Thread(target=fun2)
t3 = threading.Thread(target=fun3)
t1.start()
t2.start()
t3.start()

if mutex4.acquire():
    print("finish")
```

可以使用互斥锁完成多个任务，有序的进程工作，这就是线程的同步

2.3.5 ThreadLocal

一个ThreadLocal变量是全局变量，每个线程都只能读写自己线程的独立副本，互不干扰。ThreadLocal解决了参数在一个线程中各个函数之间互相传递的问题

```
import threading
import time
local_var = threading.local()
# print(local_var, type(local_var))
def fun1():
    local_var.stdo1 = 60
    print(local_var.stdo1)
def fun2():
    local_var.stdo1 = 100
    print(local_var.stdo1)

t1 = threading.Thread(target=fun1)
t2 = threading.Thread(target=fun2)
t2.start()
time.sleep(1)
t1.start()
```

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

3.1 网络编程

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。

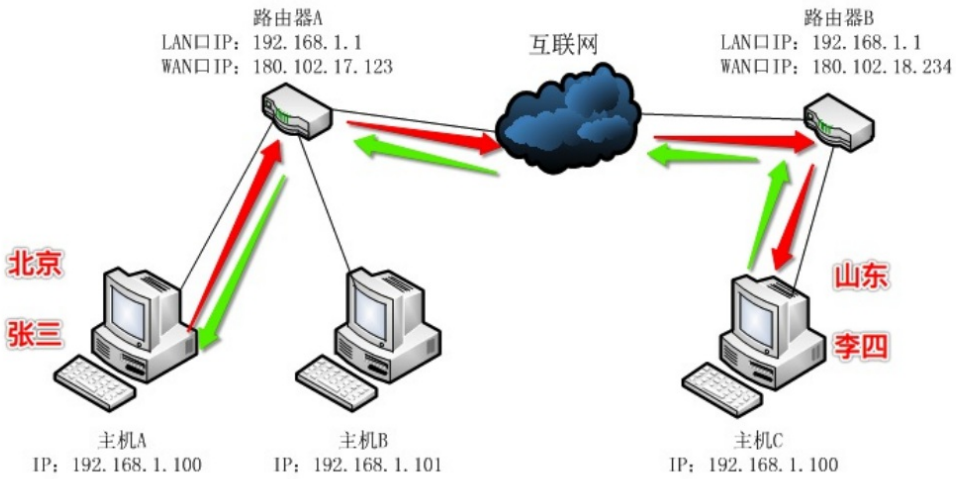
网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

网络通信是两台计算机上的两个进程之间的通信。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。

网络编程对所有开发语言都是一样的，Python也不例外。用Python进行网络编程，就是在Python程序本身这个进程内，连接别的服务器进程的通信端口进行通信。



3.2 TCP/IP简介

3.2.1 什么是协议

计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，就需要使用互联网协议簇（Internet Protocol Suite）就是通用协议标准。

Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

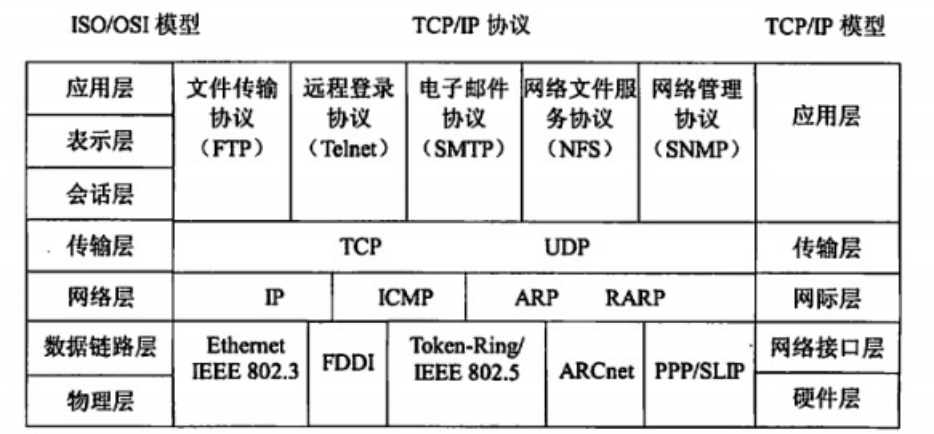
因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称**TCP/IP**协议。

3.2.2 TCP/IP协议族

TCP/IP提供点对点的链接机制，将数据应该如何封装、定址、传输、路由以及在目的地如何接收，都加以标准化。

它通常将软件通信过程抽象化为四个抽象层，采取协议堆栈的方式，分别实现出不同通信协议。协议套组下的各种协议，依其功能不同，被分别归属到这四个层次结构之中，常被视为是简化的七层OSI模型。

OSI模型，即开放式通信系统互联参考模型（Open System Interconnection Reference Model），是国际标准化组织（ISO）提出的一个试图使各种计算机在世界范围内互连为网络的标准框架，简称OSI。



TCP/IP 模型与 OSI 模型的对比

网际层也称网络层

网络连接层也称链路层

链路层：处理与电缆或其他传输媒介的物理接口

网络层：处理数据在网络中的活动

- ip协议->网络互连协议
  用途：将多个包在网络中联系起来，传输数据包（不可靠传输），最基本功能就是寻址和分段功能，不提供端到端，路由到路由的确认，不提供重发和流量控制。是计算机网络能狗相互通信的基本规则。出错则像ICMP报告，ICMP在IP模块中实现

- ICMP协议
  用途：面向无连接协议，用于传输错误报告控制信息（控制信息是指网络不通畅，主机是否到达，路由是否可用的这些网络本身的消息，不涉及用户传输的数据）
- ARP协议->地址解析协议
  用途：根据IP地址获取物理地址的协议（即MAC地址）。在同一子网内通过ARP协议可以实现数据包的互相传递。不在一个子网内则无法获得MAC地址，只有通过网关去处理。
- RARP协议->反转地址协议
  用途：和ARP协议相反，将主机的物理地址转换成IP地址。
- BOOTP协议->引导程序协议
  用途：用于无盘工作站的局域网中，可以无盘工作站从一个中心服务器上获得IP地址。

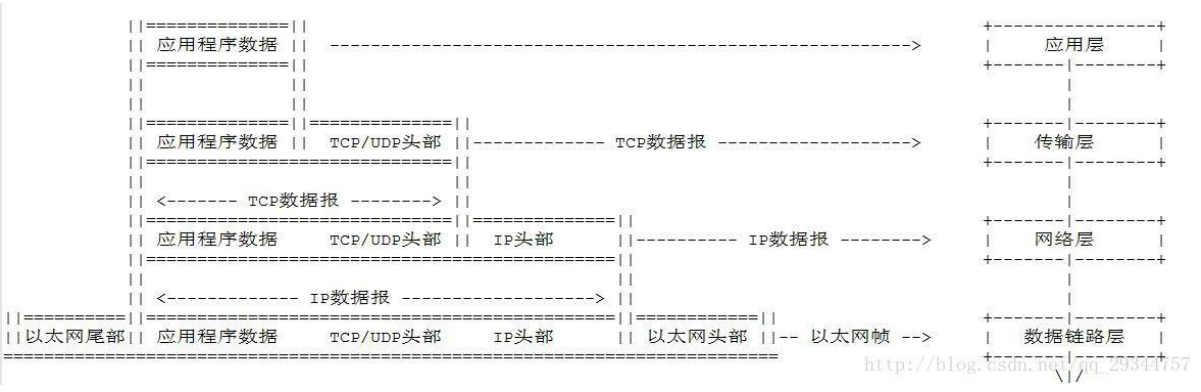
传输层：提供两台主机间端到端的通信

- TCP协议->传输控制协议
  用途：主要用于网间传输的协议，分割处理报文并把结果包传到IP层，并接收处理IP曾传到的数据包。
- UDP协议->用户数据协议
  用途：主要用于需要在计算器之间传输数据的应用，将网络数据流浪压缩成数据包。

应用层：用于不同的应用程序

- NET协议->网络地址转换协议
  用途：实现内网IP地址和公网地址之间的相互转换。将大量的内网IP转换成一个或者少量的公网IP
- FTP协议->文件传输协议
  用途：通过FTP协议在FTP客户端访问FTP服务端，默认使用20和21端口，20用于传输数据，21用于传输控制信息。
- HTTP协议->超文本传输协议
  用途：是用于从WWW服务器传输超文本到本地浏览器的传输协议。是客户端浏览器或其他程序与Web服务器之间的应用层通信协议。
- TELNET协议
  用途：是Internet远程登陆服务的标准协议和主要方式，为用户提供了在本地计算机上完成远程主机工作的能力。
- SMTP协议->简单邮件传输协议
  用途：控制邮件传输的规则，以及邮件的中转方式。
- DNS协议
  用途：定义域名规则，将域名和IP相互映射

3.2.3 数据封包解包



3.2.4 IP地址

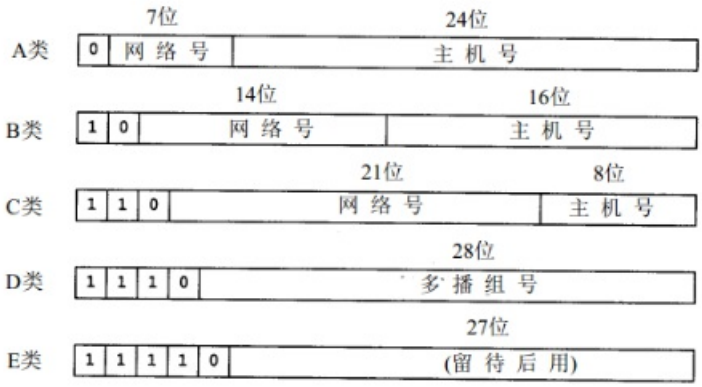
互联网上的每个接口必须有一个唯一的Internet地址（也称作IP地址）。IP地址长32 bit。

IP地址由两部分组成，即网络地址和主机地址。

网络地址表示其属于互联网的哪一个网络，主机地址表示其属于该网络中的哪一台主机。

IP地址根据网络号和主机号来分，分为A、B、C三类及特殊地址D、E。 全0和全1的都保留不用。

五类不同的互联网地址格式如下：



A类取值范围 0-127

B类取值范围128-191

C类取值范围192-223

D类取值范围224-239

E类取值范围240-247

D类IP是一个专门保留的地址。
它并不指向特定的网络。

目前这一类地址被用在多点广播（Multicast）中
多点广播地址用来一次寻址一组计算机

E类地址保留，仅作实验和开发使用

通常我们判断一个ip地址是A类还是B类C类，几乎都是看子网掩码255.255.255.0，明摆着前三位网络位，第四位主机位，不是C类还能是什么对吧？

回送地址：127.0.0.1。 也是本机地址，等效于localhost或本机IP。

在这么多网络IP中，国际规定有一部分IP地址是用于我们的局域网使用，也
就
是属于私网IP，不在公网中使用的，它们的范围是

```
10.0.0.0～10.255.255.255
172.16.0.0～172.31.255.255
192.168.0.0～192.168.255.255
```

通过IP地址的网络号和主机号已经可以识别网络接口，进而访问主机，但是人们最喜欢还是使用主机名，所以我们需要提供一个：域名系统（**DNS**），它给我们提供了主机名和IP地址之间的映射信息。

3.2.5 端口

那么TCP/IP协议中的端口指的是什么呢？

端口就好一个房子的门，是出入这间房子的必经之路。

端口号

端口是通过端口号来标记的，端口号只有整数，范围是从0到65535(2的16次方)

系统预留端口号

预留端口号是广泛使用的端口号，方位从0-1023 开发中不要使用

如http的80端口 ftp的21端口

总结

通过IP在网络内寻找主机、通过端口寻找主机中的应用程序

3.2.6 子网掩码

子网掩码只有一个作用，就是将某个IP地址划分成网络地址和主机地址两部分

子网掩码的设定必须遵循规定的规则。

与IP地址相同，子网掩码的■度也是32位，

左边是■络位，用二进制数字“1”表示；

右边是主机位，用二进制数字“0”表示。

通常我们判断一个ip地址是A类还是B类C类，几乎都是看子网掩码255.255.255.0，明摆着前三位网络位，第四位主机位，不是C类还能是什么对吧？

设置

子网掩码应该根据网络的规模进■设置。如果一个网络的规模不超过254台电脑，采用“255.255.255.0”作为子掩码就可以了，现在大多数局域网都不会超过这

个数字，因此“255.255.255.0”是最常■的IP地址子网掩码；假如在一所大学具有1500多台电脑，这种规模的局域网可以使用“255.255.0.0”。

3.2.7 默认网关

是一个可直接到达的 IP路由器的 IP地址

配置默认网关可以在 IP 路由表中创建一个默认路径。 一台主机可以有多个网关。默认网关的意思是一台主机如果找不到可用的网关，就把数据包发给默认指定的网关，由这个网关来处理数据包。现在主机使用的网关，一般指的是默认网关。

3.2.8 LAN、WAN、WLAN

局域网（Local Area Network，LAN）

广域网 Wide Area Network，WAN）

无线局域网（Wireless LAN, WLAN）

3.2.9 Socket简介

本地的进程间通信有很多种方式，例如队列
、同步（互斥锁、条件变量等）

以上通信方式都是在一台机器上不同进程之间的通信方式，那么问题来了
网络中进程之间如何通信？

网络中进程之间如何通信

主要解决的问题是如何唯一标识一个进程，否则通信无从谈起！

在本地可以通过进程PID来唯一标识一个进程，但是在网络中这是行不通的。

其实TCP/IP协议族已经帮我们解决了这个问题，网络层的“ip地址”可以唯一标识网络中的主机，传输层的“协议+端口”可以唯一标识主机中的应用程序（进程）。

这样利用ip地址，协议，端口就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互

什么是**socket**

socket(简称套接字) 是进程间通信的一种方式，它与其他进程间通信的一
个主要不同是：
它能实现不同主机间的进程间通信，我们网络上各种各样的服务大多都是基于Socket来完成通信的。

例如我们每天浏览的、QQ 聊天、收发Email等等

创建**socket**

在Python中使用socket模块的函数socke就可以完成：socket.socket(AddressFamily, Type)

说明：

函数socket.socket创建一个socket，返回该 socket的描述符，该函数带有两个参数：

Address Family：可以选择AF_INET（用于 Internet 进程间通信）或者AF_UNIX（用于同一台机器进程间通信）

实际工作中常用AF_INET

Type：套接字类型，可以是SOCK_STREAM（流式套接字，主要用于TCP协议）或者SOCK_DGRAM（数据报套接字，主要用于UDP协
议）

```
import    socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
print'SocketCreated'

import    socket
s=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
print'SocketCreated'
```

常用方法

socket套接字常用操作

s.bind() 绑定(主机名称、端口)到一个套接字上

s.listen() 设置并启动TCP监听

s.accept() 等待客户端连接

s.connect() 连接指定服务器

s.recv() 接受TCP消息

s.send() 发送TCP消息

s.recvfrom() 接受UDP消息

s.sendto() 发送UDP消息

s.close() 关闭套接字对象

3.3 UDP编程

TCP是建立可靠连接，并且通信双方都可以以流的形式发送数据。相对TCP，UDP则是面向无连接的协议。

使用UDP协议时，不需要建立连接，只需要知道对方的IP地址和端口号，就可以直接发数据包。但是，能不能到达就不知道了。

虽然用UDP传输数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。

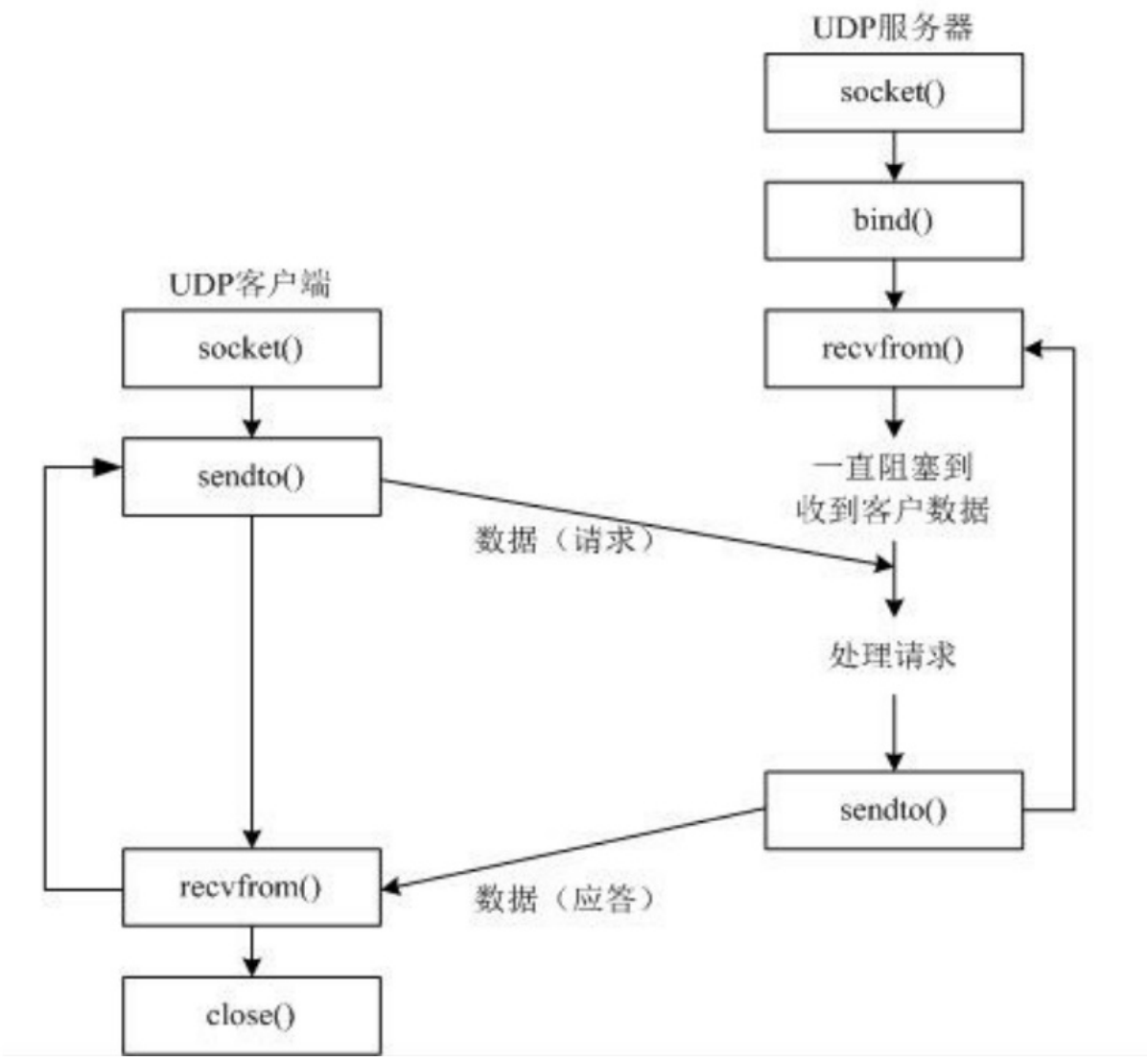
适用情况

UDP是面向消息的协议，通信时不需要建立连接，数据的传输当然是不可靠
的，UDP一般■于多点通信和实时的数据业务，例如

语音广播、视频、QQ、TFTP(简单文件传送）、SNMP（简单网络管理协议）、RIP（路由信息协议，如报告股票市场，航空信息）

DNS(域名解释）

2.1 通信流程



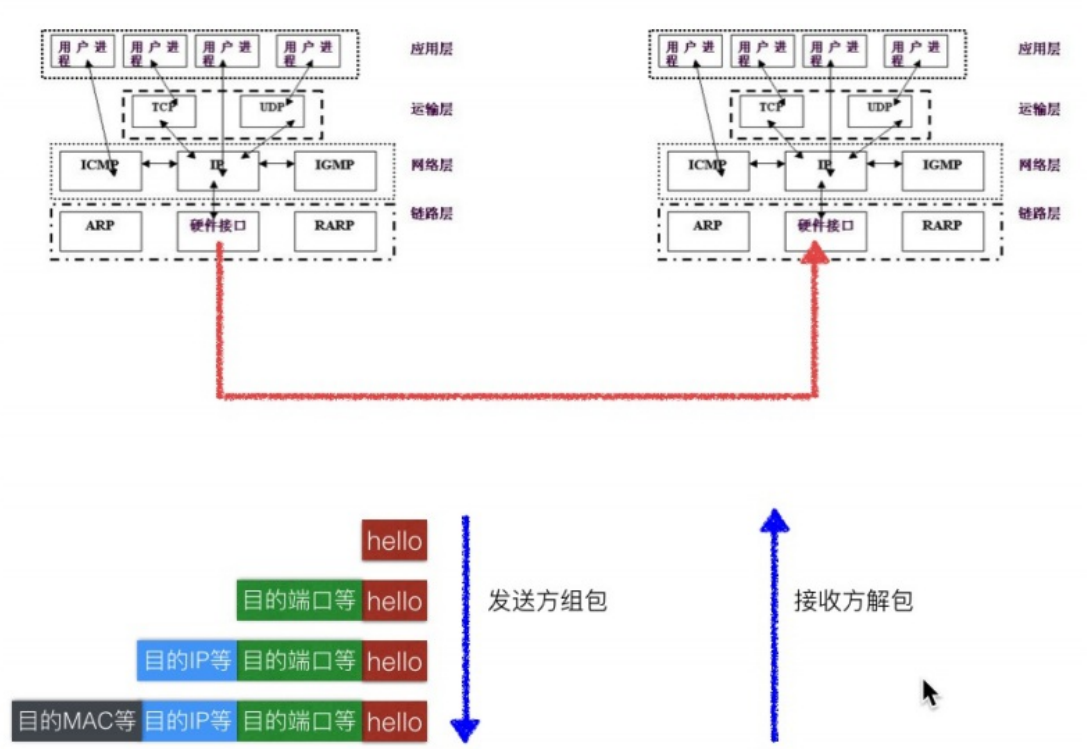
2.2 客户端实现

```
from socket import *
sc = socket(AF_INET, SOCK_DGRAM)
addr = ('127.0.0.1', 8989)
while True:
    datastr = input("输入发送内容")
    if datastr.__eq__("exit"):
        break
    dataencode = datastr.encode(encoding="utf-8")
    sc.sendto(dataencode, addr)
    receivecode, saddr = sc.recvfrom(1024)
    receivedata = receivecode.decode(encoding="utf-8")
    print(receivedata, saddr)
sc.close()
print("finish")
```

2.2 服务端实现

```
from socket import *
ss = socket(AF_INET, SOCK_DGRAM)
saddr = ('127.0.0.1', 8989)
ss.bind(saddr)
while True:
    dataencode, caddr = ss.recvfrom(1024)
    datastr = dataencode.decode(encoding="utf-8")
    print(datastr)
    str = input("输入发送内容")
    if str.__eq__("exit"):
        break
    strcode = str.encode(encoding="utf-8")
    ss.sendto(strcode, caddr)
ss.close()
print("finish")
```

2.3 UDP通信过程



2.4 UDP总结

UDP是TCP/IP协议族中的各种协议能够完成不同机器上的程序间的数据通信

UDP服务器、客户端

UDP的服务器和客户端的区分：往往是通过请求服务和提供服务来进行区分

请求服务的电脑称为：客户端

提供服务的电脑称为：服务器

UDP绑定问题

一般情况下，服务器端，需要绑定端口，目的是为了让其他的客户端能够正确发送到此进程

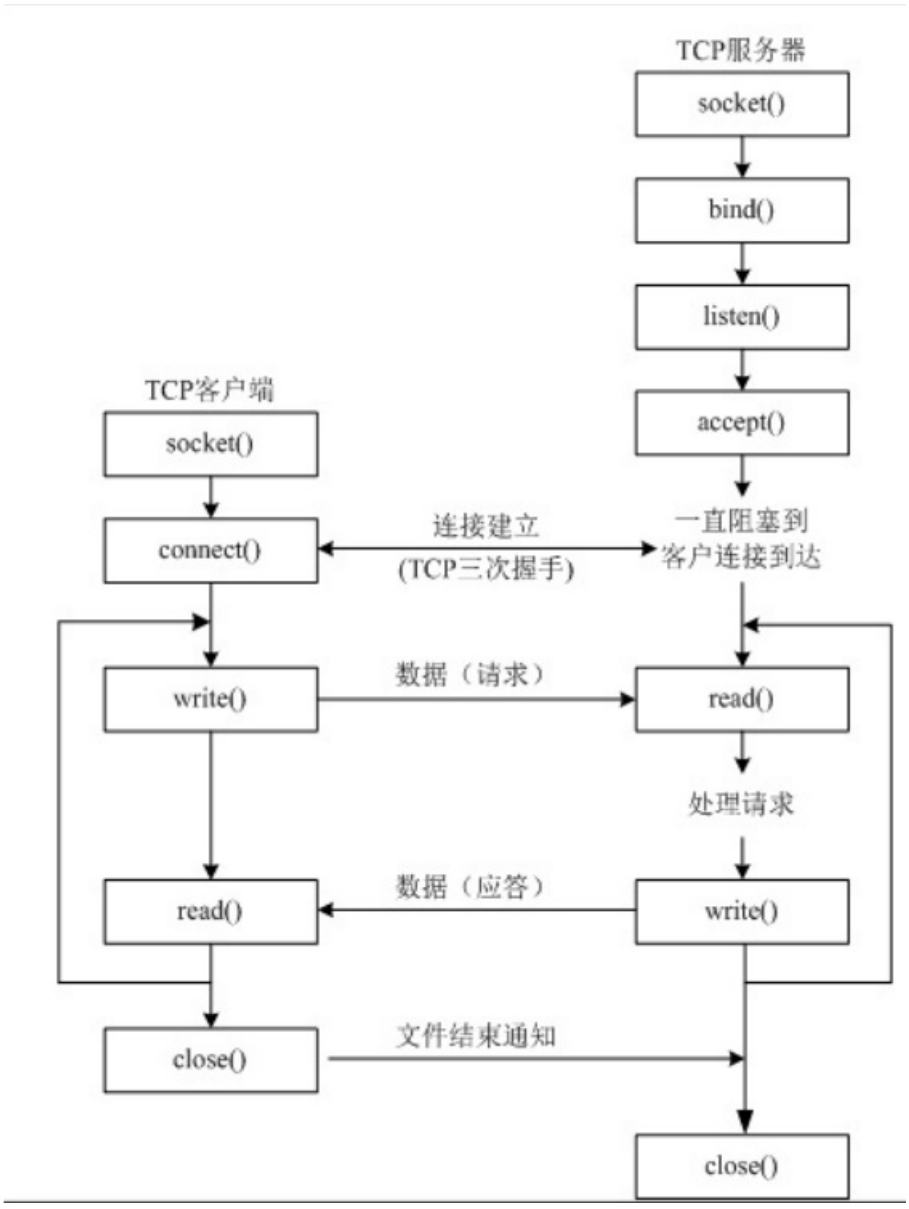
客户端，一般不需要绑定，目的是让操作系统随机分配，这样就不会因为需要绑定的端口被占用导致程序用法运用的情况

3.4 TCP编程

TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。

TCP通信模型中，在通信开始之前，一定要先建立相关的链接，才能发送数据，类似于生活中，"打电话""

3.2 TCP通信流程



生活中的电话机

如果想让别人能更够打通咱们的电话获取相应服务的话，需要做以下几件事
情：

买个手机

插上手机卡

设计手机为正常接听状态（即能够响铃）

静静的等着别打拨打

tcp服务器

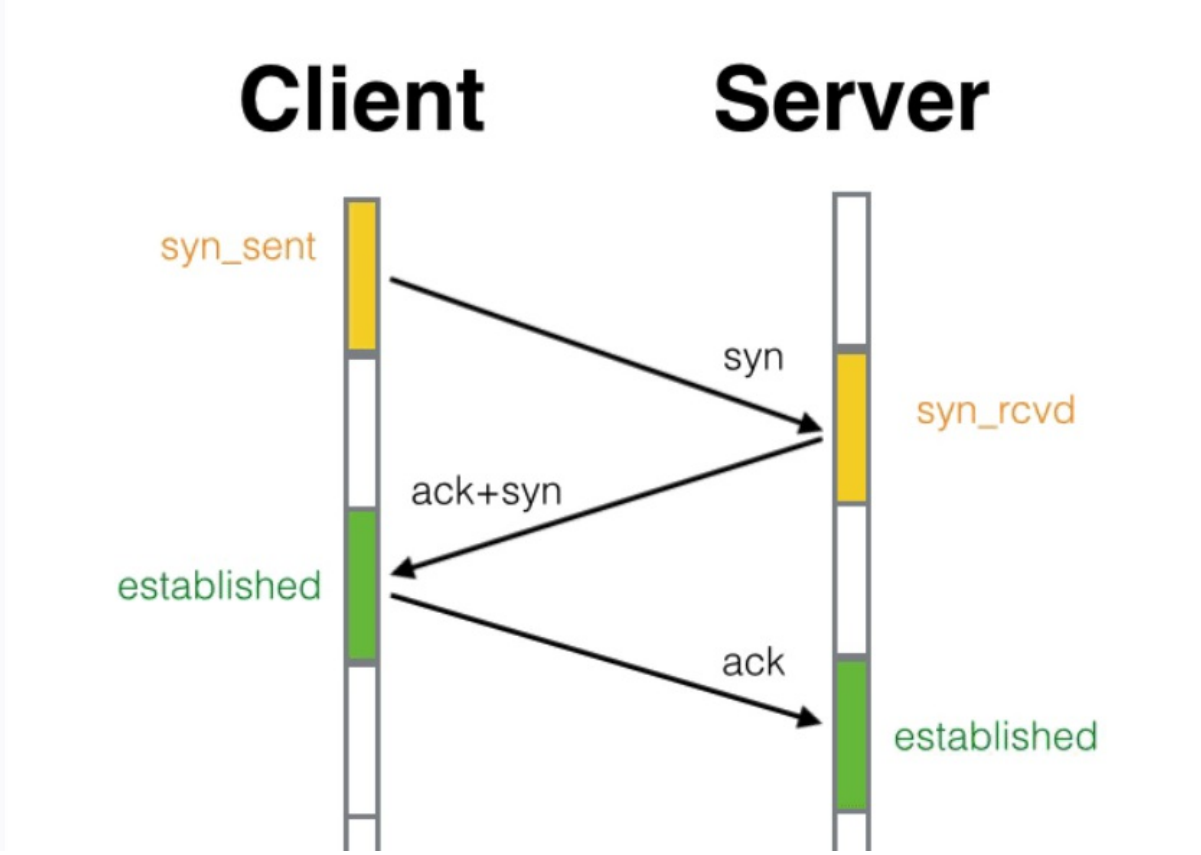
如同上面的电话机过程一样，在程序中，如果想要完成一个tcp服务器的功能，需要的流程如下：

1. socket创建一个套接字
2. bind绑定ip和port
3. listen使套接字变为可以被动链接
4. accept等待客户端的链接
5. recv/send接收发送数据

3.3 三次握手、四次挥手

TCP 连接三次握手

TCP 三次握手就好比两个人在街上隔着50米看见了对方，但是因为雾霾等原因不能100%确认，所以要通过招手的方式相互确定对方是否认识自己。



张三首先向李四招手(syn)，李四看到张三向自己招手后，向对方点了点头挤出了一个微笑(ack)。张三看到李四微笑后确认了李四成功辨认出了自己(进入established状态)。

但是李四还有点狐疑，向四周看了一看，有没有可能张三是在看别人呢，他也需要确认一下。

所以李四也向张三招了招手(syn)，张三看到李四向自己招手后知道对方是在寻求自己的确认，于是也点了点头挤出了微笑(ack)，李四看到对方的微笑后确认了张三就是在向自己打招呼(进入established状态)。

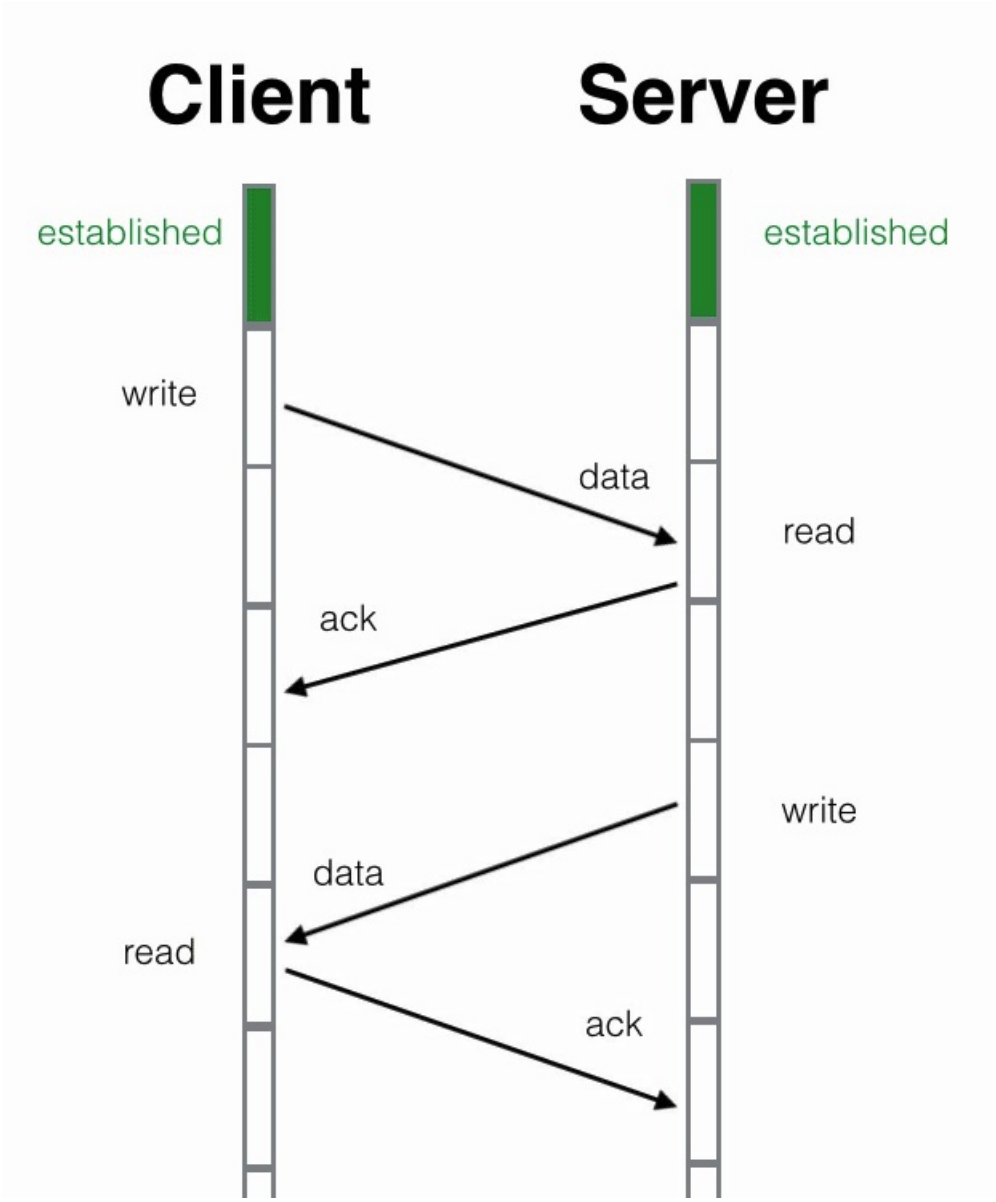
我们看到这个过程中一共是四个动作，张三招手--李四点头微笑--李四招手--张三点头微笑。

其中李四连续进行了2个动作，先是点头微笑(回复对方)，然后再次招手(寻求确认)，实际上可以将这两个动作合一，招手的同时点头和微笑(syn+ack)。

于是四个动作就简化成了三个动作，张三招手--李四点头微笑并招手--张三点头微笑。这就是三次握手的本质，中间的一次动作是两个动作的合并。

TCP数据传输

TCP 数据传输就是两个人隔空对话，差了一点距离，所以需要对方反复确认听见了自己的话。

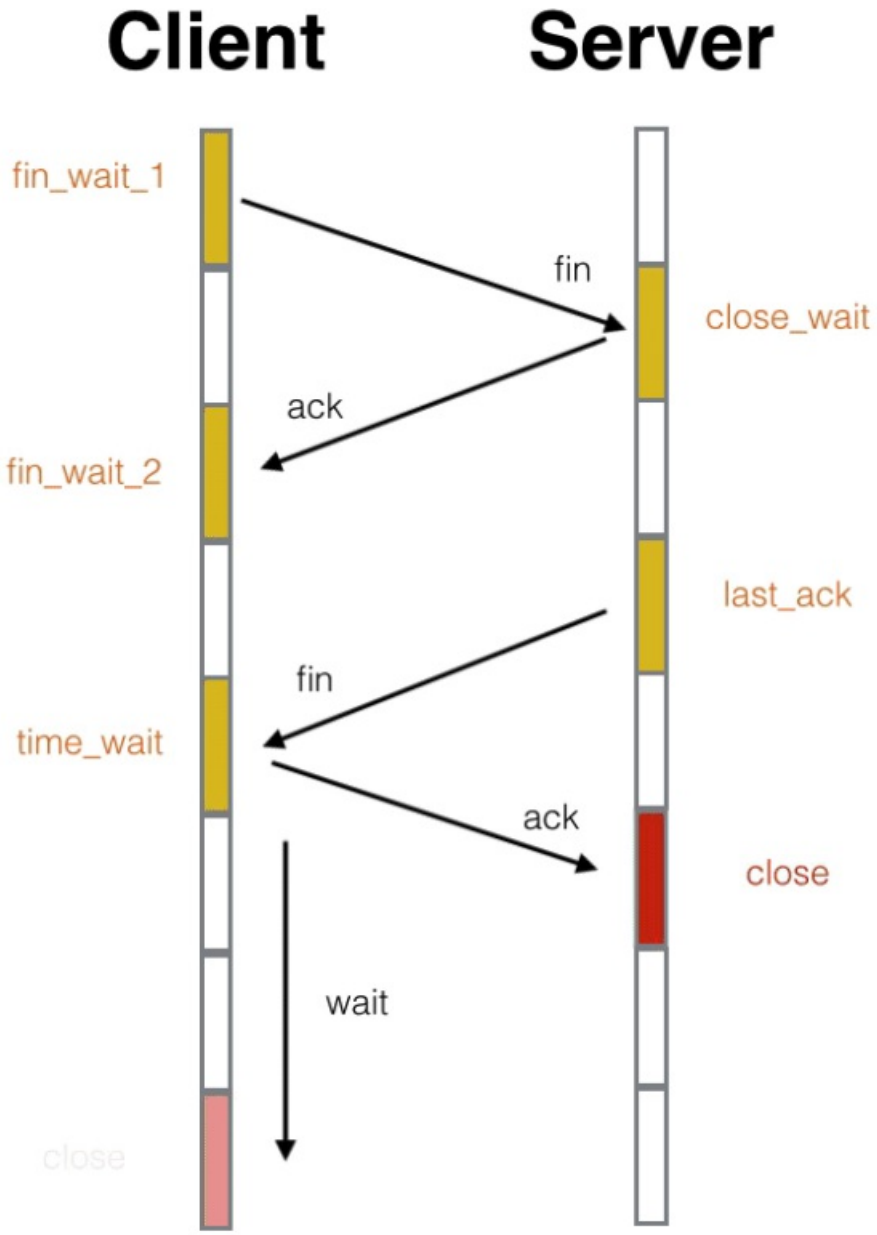


张三喊了一句话(data)，李四听见了之后要向张三回复自己听见了(ack)。

如果张三喊了一句，半天没听到李四回复，张三就认为自己的话被大风吹走了，李四没听见，所以需要重新喊话，这就是tcp重传。

TCP 四次挥手

TCP断开链接的过程和建立链接的过程比较类似，只不过中间的两部并不总是会合成一步走，所以它分成了4个动作，张三挥手(fin)——李四伤感地微笑(ack)——李四挥手(fin)——张三伤感地微笑(ack)。



上面有一个非常特殊的状态time_wait，它是主动关闭的一方在回复完对方的挥手后进入的一个长期状态，这个状态标准的持续时间是4分钟，4分钟后才会进入到closed状态，释放套接字资源。不过在具体实现上这个时间是可以调整的。

3.3 服务端实现

```
from socket import *
ss = socket(AF_INET, SOCK_STREAM)
saddr = ('127.0.0.1', 8989)
ss.bind(saddr)
ss.listen(100)
print("服务启动了")
while True:
    sc, caddr = ss.accept()
    receivebytes = sc.recv(1024)
    receivedata = receivebytes.decode("utf-8")
    print("收到：%s" % receivedata)
    sendenbytes = receivebytes
```

```
    sc.send(sendenbytes)
    # sc.close()
ss.close()
```

3.4 客户端实现

```
from socket import *
sc = socket(AF_INET, SOCK_STREAM)
saddr = ('127.0.0.1', 8989)
sc.connect(saddr)
while True:
    senddata = input("输入要发送的数据")
    if senddata.__eq__("exit"):
        sc.close()
        break
    sendbytes = senddata.encode("utf-8")
    sc.send(sendbytes)
    receivebytes = sc.recv(1024)
    receivedata = receivebytes.decode("utf-8")
    print(receivedata)
```

3.5 多线程、多进程网络编程

3.5.1 单进程服务器

只能有单个客户端连接

```
from socket import *
ss = socket(AF_INET, SOCK_STREAM)
saddr = ('127.0.0.1', 8989)
ss.bind(saddr)
ss.listen(5)
while True:
    print("等待客户端连接")
    sc, caddr = ss.accept()
    print("%s %d连接上了" % caddr)
    while True:
        recebytes = sc.recv(1024)
        recedata = recebytes.decode(encoding="utf-8")
        if len(recedata) > 0:
            print(recedata)
        else:
            print("%s %d断开了连接" % caddr)
            break

ss.close()
```

3.5.2 多进程服务器

同时连接多个客户端

```
from multiprocessing import Process
from socket import *

def readprocess(cs, caddr):
    while True:
        receivebytes = cs.recv(1024)
        receivedata = receivebytes.decode("utf-8")
        print("收到%s from %s %d" % (receivedata, caddr[0], caddr[1]))



if __name__ == "__main__":
    ss = socket(AF_INET, SOCK_STREAM)
    saddr = ('127.0.0.1', 8989)
    ss.bind(saddr)
    ss.listen(50)
    while True:
        cs, caddr = ss.accept()
        readp = Process(target=readprocess, args=(cs, caddr))
        readp.start()
```

3.5.3 多线程服务器

同时连接多个客户端

```
import threading
from socket import *

def readprocess(cs, caddr):
    while True:
        receivebytes = cs.recv(1024)
        receivedata = receivebytes.decode("utf-8")
        # print(type(caddr))
        print("收到%s from %s %d" % (receivedata, caddr[0], caddr[1]))

def listenprocess(ss):
    while True:
        cs, caddr = ss.accept()
        readsendthread = threading.Thread(target=readprocess, args=(cs, caddr))
        readsendthread.start()

if __name__ == "__main__":
    ss = socket(AF_INET, SOCK_STREAM)
    saddr = ('127.0.0.1', 8989)
    ss.bind(saddr)
    ss.listen(50)
    listenthread = threading.Thread(target=listenprocess, args=(ss,))
    listenthread.start()
```

3.5.4 多线程聊天服务器

```
import threading
from socket import *

users = []
def readprocess(cs, caddr):
    while True:
        receivebytes = cs.recv(1024)
        if len(receivebytes) > 0:
            receivedata = receivebytes.decode("utf-8")
            # print(type(caddr))
            print("收到%s from %s %d" % (receivedata, caddr[0], caddr[1]))
        else:
            break


def listenprocess(ss):
    while True:
        cs, caddr = ss.accept()
        readsendthread = threading.Thread(target=readprocess, args=(cs, caddr))
        readsendthread.start()
        user = {"ip": caddr[0], "port": caddr[1], "socket": cs}
        users.append(user)
        print("当前用户%d" % len(users))
        print(users)

if __name__ == "__main__":
    ss = socket(AF_INET, SOCK_STREAM)
    saddr = ('127.0.0.1', 8989)
    ss.bind(saddr)
    ss.listen(50)
    listenthread = threading.Thread(target=listenprocess, args=(ss,))
    listenthread.start()
```

```
    while True:
        sendstr = input("输入发送内容")
        sendbytes = sendstr.encode("utf-8")
        selectport = input("输入对方port")
        for r in users:
            if r["port"] == int(selectport):
                r["socket"].send(sendbytes)
```

3.6 应用层编程

前面描述的基于TCP/UDP协议的网络程序开发，主要是针对传输层协议的底层代码实现

在实际操作过程中，更多的情况是直接操作应用层的数据协议的网络程序开发，如文件传输

协议FTP，邮件协议SMTP等等

3.6.1 FTP编程

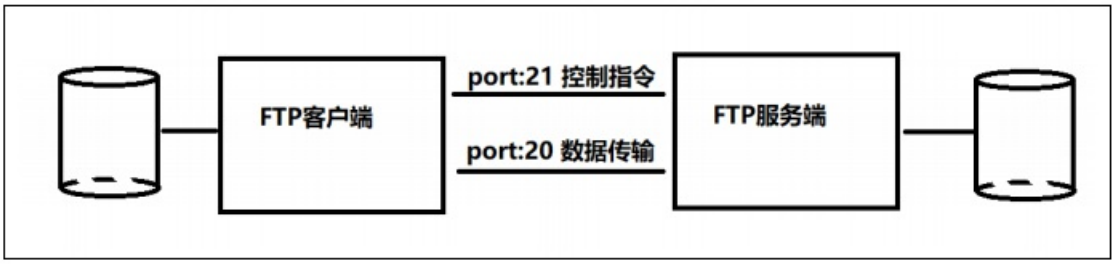
FTP：File Transfer Protocal 文件传输协议 ，要用于两台计算机之间进行文件的传输工作！

FTP协议底层采用的是TCP协议完成的网络数据传输

为了方便两台计算机上的文件正确
的进行交互，FTP协议封装了两个套接字完成文件操作

第一个工作在21端口的套接字专门用于传输控制指令

第二个工作在20端口的套接字专门传输具体文件数据



3.6.2 FTP客户端程序开发

FTP网络程序的开发，我们不需要关注服务端的程序，服务端的软件程序开发和文件管理操

作可以有大量的现成的工具去操作完成，而FTP如果作为应用软件的一部分功能，客户端

程序的操作才是应用软件中最常规的操作

PYTHON中提供了对FTP操作友好的支持，通过内建标准模块ftplib提供

针对FTP客户端的逻辑流程，进行如下步骤分析：

● 客户端——连接到服务器

● 客户端——账号+密码登录服务器

● 发出服务请求——控制指令、数据传输指令——处理响应数据

● 客户端退出

伪代码操作如下

```
from ftplib import FTP
ftp = FTP(“ftp.server.com”)
ftp.login(‘account’, ‘password’)
# 数据交互
ftp.quit()
```

3.6.3 FTP类型常见属性方法

login(user='anonymous', passwd='',acct='') 登录FTP服务器

pwd() 查看当前路径

cwd(path) 切换路径到指定的path路径

dir(path [,…[,cb]]) 显示path路径中文件的内容

nlst([path [,….]]) 类似dir()，返回文件名称列表

rename(old, name) 重命名old文件为new

retrlines(cmd [, cb]) 给定ftp命令，下载文本文件回调函数cb用于处理每一行文本

retrbinary(cmd, cb [, bs=8192 [,ra]])给定ftp命令，下载二进制文件回调函数cb处理每次读取的8k数据

storlines(cmd, f) 给定ftp命令，上传文本文件f

storbinary(cmd, f [, bs=8192]) 给定ftp命令，上传二进制文件f

delete(path) 删除path指定的某个文件

mkd(directory) 创建一个目录directory

rmd(directory) 删除指定的目录directory

quit() 关闭连接，退出FTP

3.6.4 FTP客户端实现

搭建好我们的服务器环境，指定连接的FTP服务器信息、文件路径信息

上传下载

```
import ftplib
def ftpconnect(host, username, passwd):
    ftp = ftplib.FTP(host=host, user=username, passwd=passwd)
    return ftp

def upload(ftp, localfile, remotefile):
    buffersize = 1024
    file = open(localfile, "rb")
    ftp.storbinary("STOR " + remotefile, file, buffersize)
    file.close()

def download(ftp, localfile, remotefile):
    buffersize = 1024
    file = open(localfile, "wb")
    ftp.retrbinary("RETR " + remotefile, file.write, buffersize)
    file.close()

if __name__ == "__main__":
    ftp = ftpconnect("localhost", "zzy", "123456")
    # upload(ftp, "d:/flashfxp.png", "newxp.png")
    download(ftp, "d:/newxpp.png", "newxp.png")
    ftp.quit()
```

3.7 SMTP/POP3/IMAP邮件收发

邮件收发操作中，经常会使用到一些和邮件传输相关的专业术语和协议，对比现实生活中的

邮局，可以快速的理解相关专业术语的含义

3.7.1 收发协议

POP3是Post Office Protocol 3的简称，即邮局协议的第3个版本,它规定怎样将个人计算机连接到Internet的邮件服务器和下载电子邮件的电子协议。它是因特网电子邮件的第一个离线协议标准,POP3允许用户从服务器上把邮件存储到本地主机（即自己的计算机）上,同时删除保存在邮件服务器上的邮件，而POP3服务器则是遵循POP3协议的接收邮件服务器，用来接收电子邮件的。

SMTP的全称是“Simple Mail Transfer Protocol”，即简单邮件传输协议。它是一组用于从源地址到目的地址传输邮件的规范，通过它来控制邮件的中转方式。SMTP 协议属于 TCP/IP 协议簇，它帮助每台计算机在发送或中转信件时找到下一个目的地。SMTP 服务器就是遵循 SMTP 协议的发送邮件服务器。
SMTP 认证，简单地说就是要求必须在提供了账户名和密码之后才可以登录 SMTP 服务器，这就使得那些垃圾邮件的散播者无可乘之机。增加 SMTP 认证的目的是为了使用户避免受到垃圾邮件的侵扰。

IMAP全称是Internet Mail Access Protocol，即交互式邮件存取协议，它是跟POP3类似邮件访问标准协议之一。不同的是，开启了IMAP后，您在电子邮件客户端收取的邮件仍然保留在服务器上，同时在客户端上的操作都会反馈到服务器上，如：删除邮件，标记已读等，服务器上的邮件也会做相应的动作。所以无论从浏览器登录邮箱或者客户端软件登录邮箱，看到的邮件以及状态都是一致的。

网易163免费邮箱相关服务器信息：

服务器名称	服务器地址	SSL协议端口号	非SSL协议端口号
IMAP	imap.163.com	993	143
SMTP	smtp.163.com	465/994	25
POP3	pop.163.com	995	110

3.7.2 专业术语

邮件传输所需要的组件（**MUA**、**MYA**、**MDA**）和相关协议**(SMTP)**

邮件服务器的传输过程，其几个重要组件如下：

MUA（Mail User Agent）：即邮件客户端软件，如Windows的Outlook，客户通过它来浏览、写和收邮件；

MTA（Mail Transfer Agent）：当你在MUA（如outlook）上点发邮件时，其实它是把邮件发到MTA(如SendMail, Postfix)，通过MTA实现发邮件的功能；

MDA（Mail Delivery Agent）：MDA是挂在MUA下的一个小功能，它的主要功能是分析邮件的表头，决定邮件去向。

邮件传输过程采用的协议如下：

SMTP（Simple Mail Transfer Protocal）

邮件接收所需要的组件（**MRA**）和协议（**POP3**和**IMAP**）

邮件接收需要的组件为：

MRA（Mail Retrival Agrent）：MUA向MRA发送请求接收邮件

邮件接收有两种协议：

POP3：当MUA收到邮件后，该协议将默认删除Mailbox里的内容

IMAP：当MUA收到邮件后，该协议不删除Mailbox里的内容，所以一定要对每个用户的Mailbox进行容量限制

3.7.3 发送邮件

smtplib模块

smtplib使用较为简单。以下是最基本的语法。

```
import smtplib

smtp = smtplib.SMTP()
smtp.connect('smtp.163.com,25')
smtp.login(username, password)
smtp.sendmail(sender, receiver, msg.as_string())
smtp.quit()
```

smtplib.SMTP()：实例化SMTP()

connect(host,port):host:指定连接的邮箱服务器。port：指定连接服务器的端口号，默认为25.

login(user,password):user:登录邮箱的用户名。password：登录邮箱的密码，

sendmail(from_addr,to_addrs,msg,...)

from_addr:邮件发送者地址

to_addrs:邮件接收者地址。字符串列表['接收地址1','接收地址2','接收地址3',...]或'接收地址'

msg：发送消息：邮件内容。一般是msg.as_string():as_string()是将msg(MIMEText对象或者MIMEMultipart对象)变为str。

quit():用于结束SMTP会话。

email模块

email模块下有mime包，mime英文全称为“Multipurpose Internet Mail Extensions”，

即多用途互联网邮件扩展，是目前互联网电子邮件普遍遵循的邮件技术规范。

该mime包下常用的有三个模块：text，image，multpart。

```
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
```

构造一个邮件对象就是一个Message对象，如果构造一个MIMEText对象，就表示一个文本邮件对象，如果构造一个MIMEImage对象，就表示一个作为附件的图片，要把多个对象组合起来，就用MIMEMultipart对象，而MIMEBase可以表示任何对象。它们的继承关系如下：

```
Message
+- MIMEBase
   +- MIMEMultipart
   +- MIMENonMultipart
      +- MIMEMessage
      +- MIMEText
```

```
      +- MIMEImage
```

邮件发送程序为了防止有些邮件阅读软件不能显示处理HTML格式的数据，通常都会用两类型分别为"text/plain"和"text/html"

构造MIMEText对象时，第一个参数是邮件正文，第二个参数是MIME的subtype，最后一定要用utf-8编码保证多语言兼容性。

```
添加文本
text_plain = MIMEText(text,'plain', 'utf-8')
添加html
text_html = MIMEText(html,'html', 'utf-8')
```

发送普通文本邮件

```
import smtplib
from email.mime.text import MIMEText
# smtp服务器
smtpserver = 'smtp.qiye.163.com'
# 发送方信息
user = 'zhangzhaoyu@qikux.com'
passwd = '********'
# 接收方信息
receiver = "496575233@qq.com"

message = MIMEText("我是 通过Python发送的", "plain", "utf-8")
message["from"] = user
message["to"] = receiver
message["subject"] = "测试邮件"

server = smtplib.SMTP(host=smtpserver, port=25)
server.login(user, passwd)
server.sendmail(from_addr=user, to_addrs=receiver, msg=message.as_string())
server.quit()
print("finish")
```

发送**html**

```
html = "<h1>标题1</h1>  我是 通过Python发送的  <i>作者</i>"
message = MIMEText(html, 'html', 'utf-8')
```

发送带附件邮件

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
serversmtp = "smtp.qiye.163.com"
user = 'zhangzhaoyu@qikux.com'
passwd = '*******'
receiver = 'zhangzhaoyu@qikux.com'
message = MIMEMultipart()
# message = MIMEText("我是文本", "plain", "utf-8")
message["from"] = user
message["to"] = receiver
message["subject"] = "带附件邮件"
# html附件
```

```
html = "<b>图片附件</b><img src='cid:imageid'/>"
msghtml = MIMEText(html, "html", "utf-8")
message.attach(msghtml)
# 图片附件
fileimage = open("f:/flashfxp.png", 'rb')
msgimage = MIMEImage(fileimage.read())
fileimage.close()
msgimage.add_header("Content-ID", "imageid")
message.attach(msgimage)
# 文件附件
filedoc = open("d:/zzy.docx", "rb")
msgfile = MIMEText(filedoc.read(), "base64", "utf-8")
filedoc.close()
msgfile["Content-Type"] = 'application/octet-stream'
# 这里的filename可以任意写，写什么名字，邮件中显示什么名字
msgfile["Content-Disposition"] = 'attachment; filename="test.docx"'
message.attach(msgfile)
```

多个收件人

message["to"]后面跟的是字符串，使用逗号分隔

toaddr 后面跟的是列表

3.1 正则表达式

正则表达式，又称规则表达式。（英语：Regular Expression，在代码中常简写为regex、regexp或RE），计算机科学的一个概念。正则表达式通常被用来检索、替换那些符合某个模式(规则)的文本。

3.1.1目的

给定一个正则表达式和另一个字符串，我们可以达到如下的目的：

- 给定的字符串是否符合正则表达式的过滤逻辑（称作“匹配”）：
- 可以通过正则表达式，从字符串中获取我们想要的特定部分。

3.1.2 特点

正则表达式的特点是：

- 灵活性、逻辑性和功能性非常强；
- 可以迅速地用极简单的方式达到字符串的复杂控制。
- 对于刚接触的人来说，比较晦涩难懂。

3.2 re模块操作

在Python中通过re模块来完成正则表达式操作

3.2.1 re模块常用方法

re.match

```
re.match(pattern, string, flags=0)
```

```
result = re.match("hello", "hellozzy world")
print(result, result.group(), type(result))
```

在字符串开头匹配pattern，如果匹配成功（可以是空字符串）返回对应的match对象,否则返回None。

re.search

```
re.search(pattern, string, flags=0)
```

```
result = re.search("hello", "2018hellozzy world")
print(result.group())
```

扫描整个字符串string，找到与正则表达式pattern的第一个匹配（可以是空字符串），并返回一个对应的match对象。如果没有匹配返回None.

re.fullmatch

```
re.fullmatch(pattern, string, flags=0)
```

```
result = re.fullmatch("hello", "hello1")
print(result)
```

string是否整个和pattern匹配，如果是返回对应的match对象,否则返回None。

re.findall

```
findall(pattern, string, flags=0)
```

```
result = re.findall("hello", "zzy hello china hello world")
print(result, type(result))
```

返回列表

re.split

```
re.split(pattern, string, maxsplit=0, flags=0)
```

```
result = re.split("hello", "zzy hello china hello world hello zhengzhou", 2)
```

```
print(result, type(result))
```

返回分割列表

re.sub

```
sub(pattern, repl, string, count=0, flags=0)
```

```
result = re.sub("hello", "hi", "zzy hello china hello world hello zhengzhou", 2)
print(result, type(result))
```

使用repl替换pattern匹配到的内容，最多匹配count次

re.iterator

```
finditer(pattern, string, flags=0)
```

```
result = re.finditer("hello", "hello world hello china")
print(result, type(result))
```

返回迭代器

re.compile

```
compile(pattern, flags=0)
```

```
pat = re.compile("hello")
print(pat, type(pat))
result = pat.search("helloworld")
print(result, type(result))
```

编译得到匹配模型

3.2.2 flags

re模块的一些函数中将flags作为可选参数，下面列出了常用的几个flag, 它们实际对应的是二进制数，

可以通过位或将他们组合使用。flags可能改变正则表达时的行为：

```
re.I re.IGNORECASE: 匹配中大小写不敏感

re.M  re.MULTILINE: "^"匹配字符串开始以及"\n"之后；"$"匹配"\n"之前以及字符串末尾。通常称为多行模式

re.S re.DOTALL: "."匹配任意字符，包括换行符。通常称为单行模式
```

如果要同时使用单行模式和多行模式，只需要将函数的可选参数flags设置为re.M | re.S即可。

```
result = re.match("hello", "HeLlo", flags=re.I)
print(result)
```

```
result = re.findall(".", "hello \n  china", flags=re.S)
```

```
# "." 可以匹配换行符
print(result)

result = re.findall(".", "hello \n  china", flags=re.M)
# "." 不可以匹配换行符
print(result)
```

3.3 单个字符

3.3.1 单字符匹配

字符	功能
.	匹配任意1个字符（除了\n）（单行模式合一匹配\n）
[ ]	匹配[ ]中列举的字符
\d	匹配数字，即0-9
\D	匹配非数字，即不是数字
\s	匹配空白，即 空格，tab键
\S	匹配非空白
\w	匹配单词字符，即a-z、A-Z、0-9、_
\W	匹配非单词字符

```
# result = re.findall(".", "hello \n  china", flags=re.M)
# # "." 不可以匹配换行符
# print(result)
```

```
result = re.findall("[0123].ello", "0hello 1hello 2 ello 88ello 888ello")
print(result)
```

```
result = re.findall("\dhello", "hello 1hello 5hello 0hello")
print(result)
```

```
result = re.findall("\Dhello", "xhello 1hello 5hello 0hello")
print(result)
```

```
result = re.findall("\shello", "hello hello     hello")
print(result)
```

```
result = re.findall("\Shello", "1hello shello     hello")
print(result)
```

```
result = re.findall("\wello", "hello aello5ello_ello .ello ^ello")
print(result)
```

```
result = re.findall("\Wello", "hello aello5ello_ello .ello ^ello")
print(result)
```

3.3.2 特殊贪婪字符 ？

正则匹配默认贪婪模式即匹配尽可能多个字符

```
result = re.findall("he*", "hee heeeee zzheee0")
print(result)
#结果为 ['hee', 'heeeee', 'heee']
```

当?出现在+、？、*、{m}之后开启非贪婪模式

```
result = re.findall("he*?", "hee heeeee zzheee0")
print(result)
#结果为 ['h', 'h', 'h']
```

3.4 表示数量

匹配多个字符的相关格式

字符	功能
*	匹配前一个字符出现0次或者无限次，即可有可无
+	匹配前一个字符出现1次或者无限次，即至少有1次
?	匹配前一个字符出现1次或者0次，即要么有1次，要么没有
{m}	匹配前一个字符出现m次
{m,}	匹配前一个字符至少出现m次
{m,n}	匹配前一个字符出现从m到n次

```
result = re.findall("hi*", "hi china hello china")
print(result)
```

```
result = re.findall("hi+", "hi china hello chiina")
print(result)
```

```
result = re.findall("hi?", "hi china hello chiina")
print(result)
```

```
result = re.findall("hi{2}", "hi china hello chiina")
print(result)
```

```
result = re.findall("hi{1}", "hi china hello chiina")
print(result)
```

```
result = re.findall("hi{1,2}", "hi china hello chiiina")
print(result)
```

3.5 表示边界

字符	功能
^	匹配字符串开头
$	匹配字符串结尾
\b	匹配一个单词的边界
\B	匹配非单词边界

```
result = re.findall("^hello", "hello world hello zhengzhou")
print(result)
```

```
result = re.findall("zhengzhou$", "hello world hello zhengzhou")
print(result)
```

```
# \b表示回退一定要 使用原始语句 r
result = re.findall(r"\bhello\b", "hello world hello zhengzhou sayhellook")
print(result)
```

```
# \b表示回退一定要 使用原始语句 r
result = re.findall(r"\Bhello\B", "hello world hello zhengzhou sayhellook")
print(result)
```

3.6 原始字符串

Python中字符串前面加上 r 表示原生字符串

与大多数编程语言相同，正则表达式里使用"\"作为转义字符，这就可能造成反斜杠困扰。

如果要查找以world作为单词边界的匹配编写如下代码

```
result = re.findall("\bhello\b", "hello world hello zhengzhou sayhellook")
print(result)
```

匹配失败 应为\b 需要转义

带有转义字符 的写法

```
result = re.findall("\\bhello\\b", "hello world hello zhengzhou sayhellook")
print(result)
```

效果可以达到，太麻烦

使用原始语句

```
result = re.findall(r"\bhello\b", "hello world hello zhengzhou sayhellook")
print(result)
```

Python里的原生字符串很好地解决了这个问题，有了原始字符串，你再也不用担心是不是漏写了反斜杠，写出来的表达式也更直观。

在比如

```
# \n 需要转义 如果没有转义则打印换行
print("hello \n world")
#\n 带有转义
print("hello \\n world")
#使用原始语句
print(r"hello \n world")
```

3.7 匹配分组

字符	功能	
\		匹配左右任意一个表达式
(ab)	将括号中字符作为一个分组	
`\num`	引用分组num匹配到的字符串	
`(?P<name1>) (P=name1)`	分组起别名 引用别名为name分组匹配到的字符串	

```
result = re.findall(r"\bhello\b|\bworld\b|\bhi\b", "hello world hi world")
print(result)
```

```
 retsult = re.match(r".*?@.*?.com", "496575233@qq.com")
 print(retsult.group())
 retsult = re.match(r"(.*?)@(.*?)(.com)", "496575233@qq.com")
 print(retsult.group(), retsult.group(1), retsult.group(2), retsult.group(3))
```

只可以使用原始字符串

```
# result = re.match(r"(hello).*?\1", "hello world hello china")
# # print(result.group(), result.group(1))
```

```
result = re.match(r"(?P<hname>)hello(?P=hname)", "hello world hello china")
print(result.group(), result.group("hname"))
```

3.8 练习

1、 F12提取网页html源文件中的网页标题，图片src地址，href链接地址存储为文件

2 、能不能通过python语言提供的库来获取网页源码？怎么获取