# Python语言

## Python基础

### 1.1基础功能

#### 1.1.1 打印

打印：print("hello world") 既可以用“”也可以用‘。

print('The quick brown fox', 'jumps over', 'the lazy dog') 遇到逗号“,”会输出一个空格。

#### 1.1.2 输入

func\_num = input()；

print(func\_num)；

这是因为 input()返回的数据类型是 str，str 不能直接和整数比较，必须先把 str 转换成整数。Python 提供了 int()函数来完成这件事情：s = input('birth: ')；birth = int(s)；

#### 1.1.3 基础

#：注释。

其他每一行都是一个语句，当语句以冒号:结尾时，缩进的语句视为代码块。

请务必注意，Python 程序是大小写敏感的，如果写错了大小写。

Python使用缩进组织代码块，请务必遵守约定俗成的习惯，坚持使用4个空格的缩进。

#### 1.1.4 数据类型

整数，浮点型

字符串：\转义‘。\n\t等。以\\表示的字符就是\。允许用 r''\\\\“表示''内部的字符串默认不转义。用'''...'''的格式表示多行内容。

布尔型

空值：不代表是0

变量：//，称为地板除，两个整数的除法仍然是整数：

格式化： '%2d-%02d' % (3, 1) ' 3-01'

#### 1.1.5 编码

中国制定了 GB2312 编码，用来把中文编进去。Unicode 标准也在不断发展，但最常用的是用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要 4 个字节）。本着节约的精神，又出现了把 Unicode 编码转化为“可变长编码”的 UTF-8 编码。ASCII编码实际上可以被看成是UTF-8编码的一部分，所以，大量只支持ASCII编码的历史遗留软件可以在 UTF-8 编码下继续工作。



提供了 ord()函数获取字符的整数表示，

chr()函数把编码转换为对应的字符：

以 Unicode 表示的 str 通过 encode()方法可以编码为指定的 bytes， '中文'.encode('utf-8')

str 包含多少个字符，可以用 len()函数

#!/usr/bin/env python3# -\*- coding: utf-8 -\*-

#### 1.1.6 list tuple

Python 内置的一种数据类型是列表：list。list是一种有序的集合，可以随时添加和删除其中的元素。

1. classmates = ['Michael', 'Bob', 'Tracy']。
2. 用 len()函数可以获得 list 元素的个数。
3. classmates[0]。如果要取最后一个元素，除了计算索引位置外，还可以用-1 做索引，直

接获取最后一个元素。

1. list 是一个可变的有序表，所以，可以往 list 中追加元素到末尾：classmates.append('Adam')。
2. 可以把元素插入到指定的位置，比如索引号为 1 的位置：classmates.insert(1, 'Jack')。
3. 要删除 list 末尾的元素，用 pop()方法：classmates.pop()。
4. 要删除指定位置的元素，用 pop(i)方法，其中 i 是索引位置：classmates.pop(1)。
5. 某个元素替换成别的元素，可以直接赋值给对应的索引位置：classmates[1] = 'Sarah'。
6. list 里面的元素的数据类型也可以不同，比如： L = ['Apple', 123, True]。
7. list 元素也可以是另一个 list，比如： s = ['python', 'java', ['asp', 'php'], 'scheme']。len(s)

4要注意 s 只有 4 个元素，其中 s[2]又是一个 list，如果拆开写就更容易理解了。

tuple另一种有序列表叫元组：tuple。tuple 和 list 非常类似，但是 tuple 一旦初始化就不能修改。classmates = ('Michael', 'Bob', 'Tracy')。classmates 这个 tuple 不能变了，它也没有 append()，insert()这样的方法。其他获取元素的方法和 list 是一样的，你可以正常地使用

classmates[0]，classmates[-1]，但不能赋值成另外的元素。不可变的 tuple 有什么意义？因为 tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。

要定义一个只有 1 个元素的 tuple，如果你这么定义t = (1)。定义的不是 tuple，是 1 这个数！这是因为括号()既可以表示 tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python 规定，这种情况下，按小括号进行计算，计算结果自然是 1。 t = (，1)。Python 在显示只有 1 个元素的 tuple 时，也会加一个逗号,，以免你误解成数学计算意义上的括号。

一个“可变的”tuple：t = ('a', 'b', ['A', 'B'])。tuple所谓的“不变”是说，tuple 的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个 list，就不能改成指向其他对象，但指向的这个 list 本身是可变的！

#### 1.1.7 if

根据 Python 的缩进规则，如果 if 语句判断是 True，就把缩进的两行 print语句执行了，否则，什么也不做。也可以给 if 添加一个 else 语句，意思是，如果 if 判断是 False，不要执行 if 的内容，去把 else 执行了。注意不要少写了冒号:。

age = 3

if age >= 18:

print('adult')

elif age >= 6:

print('teenager')

else:

print('kid')

#### 1.1.8 for

Python 的循环有两种，一种是 for...in 循环，依次把 list 或 tuple 中的每个元素迭代出来，

names = ['Michael', 'Bob', 'Tracy']

for name in names:

print(name)

所以 for x in ...循环就是把每个元素代入变量 x，然后执行缩进块的语句。

Python提供一个 range()函数，可以生成一个整数序列，再通过 list()函数可以转换为 list。比如 range(5)生成的序列是从 0 开始小于 5 的整数：list(range(5))。

第二种循环是 while 循环，只要条件满足，就不断循环，条件不满足时退出循环。比如我们要计算 100 以内所有奇数之和，可以用 while 循环实现：

sum = 0

n = 99

while n > 0:

sum = sum + n

n = n - 2

print(sum)

#### 1.1.9 dict set

Dict

Python 内置了字典：dict 的支持，dict 全称 dictionary，在其他语言称为 map，使用键-值（key-value）存储，具有极快的查找速度。 d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}。d['Michael']。

假设字典包含了 1 万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在 list 中查找元素的方法，list 越大，查找越慢。第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字。无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。dict 就是第二种实现方式。由于一个 key 只能对应一个 value，所以，多次对一个 key 放入 value，后面的值会把前面的值冲掉。如果 key 不存在，dict 就会报错。要避免 key 不存在的错误，有两种办法，一是通过 in 判断 key 是否存在。二是通过 dict 提供的 get 方法，如果 key 不存在，可以返回 None，或者自己指定的 value。

要删除一个 key，用 pop(key)方法，对应的 value 也会从 dict 中删除。

请务必注意，dict 内部存放的顺序和 key 放入的顺序是没有关系的。和 list 比较，dict 有以下几个特点：

1. 查找和插入的速度极快，不会随着 key 的增加而增加；

2. 需要占用大量的内存，内存浪费多。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用 dict 非常重要，需要牢记的第一条就是 dict 的 key 必须是不可变对象。

Set

set 和 dict 类似，也是一组 key 的集合，但不存储 value。由于 key 不能重复，所以，在 set 中，没有重复的 key。要创建一个 set，需要提供一个 list 作为输入集合：s = set([1, 2, 3])。注意，传入的参数[1, 2, 3]是一个 list，而显示的{1, 2, 3}只是告诉你这个 set 内部有 1，2，3 这 3 个元素，显示的顺序也不表示 set 是有序的。重复元素在 set 中自动被过滤。

通过 add(key)方法可以添加元素到 set 中，可以重复添加，但不会有效果。

通过 remove(key)方法可以删除元素。set 可以看成数学意义上的无序和无重复元素的集合，因此，两个 set 可以做数学意义上的交集、并集等操作。

s1 = set([1, 2, 3])

>>> s2 = set([2, 3, 4])

>>> s1 & s2

{2, 3}

>>> s1 | s2

{1, 2, 3, 4}

set 和 dict 的唯一区别仅在于没有存储对应的 value，但是，set 的原理和dict 一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证 set 内部“不会有重复元素”。试试把 list 放入set，看看是否会报错。

str 是不变对象，而 list 是可变对象。

a = 'abc'

>>> a.replace('a', 'A')

'Abc'

>>> a

'abc'

所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

要始终牢记的是，a 是变量，而'abc'才是字符串对象！有些时候，我们经常说，对象 a 的内容是'abc'，但其实是指，a 本身是一个变量，它指向的对象的内容才是'abc'。当我们调用 a.replace('a', 'A')时，实际上调用方法 replace 是作用在字符串对象'abc'上的，而这个方法虽然名字叫 replace，但却没有改变字符串'abc'的内容。相反，replace 方法创建了一个新字符串'Abc'并返回，如果我们用变量 b 指向该新字符串，就容易理解了，变量 a 仍指向原有的字符串'abc'，但变量 b 却指向新字符串'Abc'了。

#### 1.1.10 数据类型转换

Int float str bool

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”： a = abs # 变量 a 指向 abs 函数 a(-1) # 所以也可以通过 a 调用 ab

#### 1.1.11 函数默认参数

如果没有return，函数执行完毕返回none。Retrun none可以简写为return。

如果你已经把 my\_abs()的函数定义保存为 abstest.py 文件了，那么，可以在该文件的当前目录下启动 Python 解释器，用 from abstest import my\_abs 来导入 my\_abs()函数，注意 abstest 是文件名（不含.py 扩展名）。

数据类型检查可以用内置函数 isinstance()实现：

if not isinstance(x, (int, float)):

raise TypeError('bad operand type')

函数可以返回多个值x, y = move(100, 100, 60, math.pi / 6) return nx, ny r = move(100, 100, 60, math.pi / 6)这样也行。返回是一个tuple。在语法上，返回一个 tuple 可以省略括

号，而多个变量可以同时接收一个 tuple，按位置赋给对应的值，所以，Python 的函数返回多值其实就是返回一个 tuple，但写起来更方便。

def power(x, n=2):第二参数默认值2兼容。当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。默认参数必须指向不变对象！否在多次调用默认值会变，用list就会变。为什么要设计 str、None 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

注：由于 abs 函数实际上是定义在\_\_builtin\_\_模块中的，所以要让修改abs 变量的指向在其它模块也生效，要用\_\_builtin\_\_.abs = 10。

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

#### 1.1.12 函数可变参数

可变参数：输入参数个数不确定，可以用list tupledef calc(numbers): calc([1, 2, 3])

可以将函数参数变为可变从参数def calc(\*numbers):定义可变参数和定义一个 list 或 tuple 参数相比，仅仅在参数前面加了一个\*号。在函数内部，参数 numbers 接收到的是一个 tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括 0 个

参数： calc(1, 3, 5, 7)。如果要传入list或者tuple，这么调用calc(\*nums)。\*nums 表示把 nums 这个 list 的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

关键字参数：关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个 dict。

>>> person('Bob', 35, city='Beijing')

name: Bob age: 35 other: {'city': 'Beijing'}

>>> person('Adam', 45, gender='M', job='Engineer')

name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}

关键字参数有什么用？它可以扩展函数的功能。比如，在 person 函数里，我们保证能接收到 name 和 age 这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

>>> extra = {'city': 'Beijing', 'job': 'Engineer'

>>> person('Jack', 24, \*\*extra)

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

命名关键字参数：关键字参数可以通过if in检查。或者用如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 city 和 job 作为关键字参数。和关键字参数\*\*kw 不同，命名关键字参数需要一个特殊分隔符\*，\*后面的参数被视为命名关键字参数。这种方式定义的函数如下：

def person(name, age, \*, city, job):

print(name, age, city, job)

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

person('Jack', 24, city='Beijing', job='Engineer')

Jack 24 Beijing Engineer

def person(name, age, \*, city='Beijing', job):这样定义即有默认值。

使用命名关键字参数时，要特别注意，\*不是参数，而是特殊分隔符。

在 Python 中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这 5 种参数都可以组合使用，除了可变参数无法和命名关键字参数混合。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数/命名关键字参数和关键字参数。

def f1(a, b, c=0, \*args, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, \*, d, \*\*kw):

print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)

#### 1.1.13 切片

取list tuple 一部分数据可用切片silce。

L = ['Michael' 'Sarah' 'Trac' 'Bob' 'Jack'] L[0:3] ['Michael', 'Sarah', 'Tracy'] L[0:3]表示，从索引 0 开始取，直到索引 3 为止，但不包括索引 3。即索引 0，1，2，正好是 3 个元素。

(0, 1, 2, 3, 4, 5)[:3] 'ABCDEFG'[::2]

#### 1.1.14 迭代和列表表达式

默认情况下，dict 迭代的是 key。如果要迭代 value，可以用 for value in d.values()，如果要同时迭代 key 和 value，可以用 for k, v in d.items()。那么，如何判断一个对象是可迭代对象呢？方法是通过 collections 模块的 Iterable 类型判断。迭代都可以用for进行使用。

Python 内置的 enumerate 函数可以把一个 list 变成索引-元素对，这样就可以在 for 循环中同时迭代索引和元素本身：

>>> for i, value in enumerate(['A', 'B', 'C']):

... print(i, value)

列表表达式【 】写列表生成式时，把要生成的元素 x \* x 放到前面，后面跟 for 循环，

就可以把 list 创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

[d for d in os.listdir('.')]

[x \* x for x in range(1, 11) if x % 2 == 0]

生成list

#### 1.1.15 生成器generator

定义生成器一个方法，创建生成器很简单将列表生成式中【】改为（）。g = (x \* x for x in range(10))

print(next(g));打印值

for n in g:

... print(n)

定义生成器另一个方法如果一个函数定义中包含 yield 关键字，那么这个函数就不再是一个普通函数，而是一个 generator：

def fib(max):

n, a, b = 0, 0, 1

while n < max:

yield b

a, b = b, a + b

n = n + 1

return 'done'

generator 和函数的执行流程不一样。函数是顺序执行，遇到 return 语句或者最后一行函数语句就返回。而变成generator 的函数，在每次调用 next()的时候执行，遇到 yield 语句返回，再次执行时从上次返回的 yield 语句处继续执行。要理解 generator 的工作原理，它是在 for 循环的过程中不断计算出下一个元素，并在适当的条件结束 for 循环。对于函数改成的 generator 来说，遇到 return 语句或者执行到函数体最后一行语句，就是结束 generator的指令，for 循环随之结束。

#### 1.1.16 迭代器

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。

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

isinstance([], Iterable)

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：

isinstance([], Iterator)

生成器都是 Iterator 对象，但 list、dict、str 虽然是 Iterable，却不是Iterator。

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

#### 1.1.17 map reduce

Map：ap()函数接收两个参数，一个是函数，一个是 Iterable，map 将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator 返回。

>>> def f(x):

... return x \* x

...

>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> list(r)

[1, 4, 9, 16, 25, 36, 49, 64, 81]

Reduce：reduce 把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce 把结果继续和序列的下一个元素做累积计算，其效果就是：

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4) 可用以求和。

#### 1.1.18 filter 过滤序列

和 map()类似，filter()也接收一个函数和一个序列。和 map()不同的时，filter()把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素。

Filter过滤素数。注意到 filter()函数返回的是一个 Iterator，也就是一个惰性序列，所以要强迫 filter()完成计算结果，需要用 list()函数获得所有结果并返回list。

#### 1.1.19 sorted

Python 内置的 sorted()函数就可以对 list 进行排序：

>>> sorted([36, 5, -12, 9, -21])

[-21, -12, 5, 9, 36]

此外，sorted()函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序，例如按绝对值大小排序：

>>> sorted([36, 5, -12, 9, -21], key=abs)

[5, 9, -12, -21, 36]

sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)

#### 1.1.20 高阶函数 闭包

**函数作为返回值**：高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返

回。

def lazy\_sum(\*args):

def sum():

ax = 0

for n in args:

ax = ax + n

return ax

return sum

当我们调用 lazy\_sum()时，返回的并不是求和结果，而是求和函数。

请再注意一点，当我们调用 lazy\_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数。

在这个例子中，我们在函数 lazy\_sum 中又定义了函数 sum，并且，内部函数 sum 可以引用外部函数 lazy\_sum 的参数和局部变量，当 lazy\_sum 返回函数 sum 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

**闭包：**注意到返回的函数在其定义内部引用了局部变量 args，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

def count():

fs = []

for i in range(1, 4):

def f():

return i\*i

fs.append(f)

return fs

f1, f2, f3 = count()

9 9 9

def count():

def f(j):

def g():

return j\*j

return g

fs = []

for i in range(1, 4):

fs.append(f(i)) # f(i)立刻被执行，因此 i 的当前值被传入 f()

return fs

1 4 9

#### 1.1.21 匿名函数

关键字 lambda 表示匿名函数，冒号前面的 x 表示函数参数。

lambda x: x \* x

def f(x):

return x \* x

匿名函数有个限制，就是只能有一个表达式，不用写 return，返回值就是该表达式的结果。

>>> f = lambda x: x \* x

>>> f

<function <lambda> at 0x101c6ef28>

>>> f(5)

25

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

#### 1.1.22 装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

函数对象有一个\_\_name\_\_属性，可以拿到函数的名字。现在，假设我们要增强 now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 now()函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。本质上，decorator 就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的 decorator，可以定义如下：

def log(func):

def wrapper(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

@log

def now():

print('2015-3-25')

观察上面的 log，因为它是一个 decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助 Python 的@语法，把 decorator 置于函数的定义处：调用 now()函数，不仅会运行 now()函数本身，还会在运行 now()函数前。把@log 放到 now()函数的定义处，相当于执行了语句：now = log(now)。由于 log()是一个 decorator，返回一个函数，所以，原来的 now()函数仍然存在，只是现在同名的 now 变量指向了新的函数，于是调用 now()将执行新函数，即在 log()函数中返回的 wrapper()函数。

还可以用log（“”）添加额外信息，就是两层嵌套。now = log('execute')(now)



@unique 装饰器可以帮助我们检查保证没有重复值

#### 1.1.23 偏函数

Python 的 functools 模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低

函数调用的难度。而偏函数也可以做到这一点。举例如下：int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换。

functools.partial 就是帮助我们创建一个偏函数的，不需要我们自己定义 int2()，可以直接使用下面的代码创建一个新的函数 int2

>>> import functools

>>> int2 = functools.partial(int, base=2)

>>> int2('1000000')

64

>>> int2('1010101')

85

所以，简单总结 functools.partial 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

最后，创建偏函数时，实际上可以接收函数对象、\*args 和\*\*kw 这 3 个

参数。

#### 1.1.24 模块

在 Python 中，一个.py 文件就称之为一个模块（Module）。使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突点这里查看 Python 的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python 又引入了按目录来组织模块的方法，称为包（Package）。现在，假设我们的 abc 和 xyz 这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如 mycompany，按照如下目录存放。

请注意，每一个包目录下面都会有一个\_\_init\_\_.py 的文件，这个文件是必须存在的，否则，Python 就把这个目录当成普通目录，而不是一个包。\_\_init\_\_.py 可以是空文件，也可以有 Python 代码，因为\_\_init\_\_.py 本身就是一个模块，而它的模块名就是 mycompany。

Sys：模块sys 模块有一个 argv 变量，用 list 存储了命令行的所有参数。argv 至少有一个元素，因为第一个参数永远是该.py 文件的名称，例如：运行 python3 hello.py Michael。

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python 中，是通过\_前缀来实现的。正常的函数和变量名是公开的（public），可以被直接引用。

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，hello 模块定义的文档注释也可以用特殊变量\_\_doc\_\_访问，我们自己的变量一般不要用这种变量名。类似\_xxx 和\_\_xxx 这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc 等；

我们在模块里公开 greeting()函数，而把内部逻辑用 private 函数隐藏起来了，这样，调用 greeting()函数不用关心内部的 private 函数细节，这也是一种非常有用的代码封装和抽象的方法。

#### 1.1.25 第三方模块

安装第三方模块，是通过包管理工具pip完成。一般来说，第三方库都会在 Python 官方的 pypi.python.org 网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者 pypi 上搜索，比如 Pillow 的名称叫 Pillow，因此，安装 Pillow 的命令就是：

pip install Pillow 图像

Numpy库 科学计算

Jinja2 用于生成文本模块工具

Sys.path 添加搜索目录，运行完结束。

#### 1.1.26 面对对象

在 Python 中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

class Student(object):

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

self.name = name

self.score = score

def print\_score(self):

print('%s: %s' % (self.name, self.score))

bart = Student('Bart Simpson', 59)

lisa = Student('Lisa Simpson', 87)

bart.print\_score()

class 后面紧接着是类名，即 Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用 object 类，这是所有类最终都会继承的类。

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把 name，score 等属性绑上去。注意到\_\_init\_\_方法的第一个参数永远是 self，表示创建的实例本身，

因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到 self，因为 self就指向创建的实例本身。

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在 Python 中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问.

class Student(object):

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

self.\_\_name = name

self.\_\_score = score

需要注意的是，在 Python 中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是 private 变量，所以，不能用\_\_name\_\_、\_\_score\_\_这样的变量名。双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name 是因为 Python 解释器对外把\_\_name 变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name 来访问\_\_name 变量。

在 OOP 程序设计中，当我们定义一个 class 的时候，可以从某个现有的class 继承，新的 class 称为子类（Subclass），而被继承的 class 称为基类、父类或超类（Base class、Super class）。

判断一个变量是否是某个类型可以用 isinstance()判断。

#### 1.1.27 获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用 type()来判断对象类型，使用 type()函数， type(123)。

使用 isinstance()，一个对象是否是某种类型。

使用 dir()，如果要获得一个对象的所有属性和方法，可以使用 dir()函数，它返回

一个包含字符串的 list，比如，获得一个 str 对象的所有属性和方法。

仅仅把属性和方法列出来是不够的，配合 getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态。

hasattr(obj, 'x') #有属性'x'

setattr(obj, 'y', 19) #设置一个属性'y'

getattr(obj, 'y') #获取属性'y'

#### 1.1.28 slot @property \_str\_ \_iter\_ \_\_getitem\_\_ \_\_getattr\_\_

Slot：如果我们想要限制实例的属性怎么办？比如，只允许对 Student实例添加 name 和 age 属性。为了达到限制的目的，Python 允许在定义 class 的时候，定义一个特殊的\_\_slots\_\_变量，来限制该 class 实例能添加的属性：

class Student(object):

\_\_slots\_\_ = ('name', 'age') # 用 tuple 定义允许绑定的属性名称

由于'score'没有被放到\_\_slots\_\_中，所以不能绑定 score 属性，试图绑定 score 将得到 AttributeError 的错误

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的

Property：有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的 Python 程序员来说，这是必须要做到的！还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python 内置的@property 装饰器就是负责把一个方法变成属性调用的。

@property 的实现比较复杂，我们先考察如何使用。把一个 getter 方法变成属性，只需要加上@property 就可以了，此时，@property 本身又创建了另一个装饰器@score.setter，负责把一个 setter 方法变成属性赋值。还可以定义只读属性，只定义 getter 方法，不定义 setter 方法就是一个只读属性：

class Student(object):

@property

def birth(self):

return self.\_birth

@birth.setter

def birth(self, value):

self.\_birth = value

@property

def age(self):

return 2015 - self.\_birth

上面的 birth 是可读写属性，而 age 就是一个只读属性，因为 age 可以根据 birth 和当前时间计算出来。

def \_\_getattr\_\_(self, attr):

if attr=='score':

return 99

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。现在很多网站都搞 REST API，比如新浪微博、豆瓣啥的，调用 API 的URL 类似如果要写 SDK，给每个 URL 对应的 API 都写一个方法，那得累死，而且，API 一旦改动，SDK 也要改。利用完全动态的\_\_getattr\_\_，我们可以写出一个链式调用.

#### 1.1.29 枚举类 元类

更好的方法是为这样的枚举类型定义一个 class 类型，然后，每个常量

都是 class 的一个唯一实例。Python 提供了 Enum 类来实现这个功能：

from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',

'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))

我们说 class 的定义是运行时动态创建的，而创建 class 的方法就是使用type()函数。type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，

我们可以通过 type()函数创建出 Hello 类，而无需通过 class Hello(object)...的定义：

要创建一个 class 对象，type()函数依次传入 3 个参数。正常情况下，我们都用 class Xxx...来定义类，但是，type()函数也允许

我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

除了使用 type()动态创建类以外，要控制类的创建行为，还可以使用metaclass。metaclass，直译为元类，简单的解释就是：当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。正常情况下，你不会碰到需要使用 metaclass 的情况。

#### 1.1.30 调试 测试

Logging模块 assert print 启动 Python 解释器时可以用-O 参数来关闭 assert。

logging.basicConfig(level=logging.INFO)

logging.info('logging info');

logging.error('logging error');

启动 Python 的调试器 pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序然后启动：

$ python3 -m pdb err.py

>

/Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()

-> s = '0'

以参数-m pdb 启动后，pdb 定位到下一步要执行的代码-> s = '0'。输入命令 l 来查看代码

类似gdb

测试：为了编写单元测试，我们需要引入 Python 自带的 unittest 模块。对每一类测试都需要编写一个 test\_xxx()方法。由于 unittest.TestCase提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 assertEqual()。setUp()和 tearDown()方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在 setUp()方法中连接数据库，在 tearDown()方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码。

#### 1.1.31 IO stringIO byteIO 操作文件和目录

第一种是 CPU 等着，也就是程序暂停执行后续代码，等 100M 的数据在 10 秒后写入磁盘，再接着往下执行，这种模式称为同步 IO；

另一种方法是 CPU 不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步 IO。

很明显，使用异步 IO 来编写程序性能会远远高于同步 IO，但是异步 IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步 IO 的复杂度远远高于同步 IO。

f = open('/Users/michael/test.txt', 'r')。如果文件打开成功，接下来，调用 read()方法可以一次读取文件的全部内容，Python 把内容读到内存，用一个 str 对象表示。调用 read()会一次性读取文件的全部内容，如果文件有 10G，内存就爆了，所以，要保险起见，可以反复调用 read(size)方法，每次最多读取size 个字节的内容。另外，调用 readline()可以每次读取一行内容，调用 readlines()一次读取所有内容并按行返回 list。因此，要根据需要决

定怎么调用。f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')

>>> f = open('/Users/michael/test.txt', 'w')

>>> f.write('Hello, world!')

>>> f.close()

很多时候，数据读写不一定是文件，也可以在内存中读写。StringIO 顾名思义就是在内存中读写 str。要把 str 写入 StringIO，我们需要先创建一个 StringIO，然后，像文件一样写入即可。

StringIO操作的只能是str，如果要操作二进制数据，就需要使用BytesIO。

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如 dir、cp 等命令。其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python 内置的 os 模块也可以直接调用操作系统提供的接口函数。

os.environ，os.environ.get('PATH')，os.path.abspath('.')，os.mkdir('/Users/michael/testdir')，os.rmdir('/Users/michael/testdir')，os.path.split()，os.path.splitext()可以直接让你得到文件扩展名。>>> os.rename('test.txt', 'test.py')

# 删掉文件:

>>> os.remove('test.py')

幸运的是 shutil 模块提供了 copyfile()的函数，你还可以在 shutil 模块中找到很多实用函数，它们可以看做是 os 模块的补充。 [x for x in os.listdir('.') if os.path.isdir(x)]。

#### 1.1.32 序列化

我们把变量从内存中变成可存储或传输的过程称之为序列化，在 Python中叫 pickling，在其他语言中也被称之为 serialization，marshalling，flattening 等等，都是一个意思。反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即 unpickling。Python 提供了 pickle 模块来实现序列化。

import pickle

d = dict(name='Bob', age=20, score=88)

>>> pickle.dumps(d)

pickle.dumps()方法把任意对象序列化成一个 bytes，然后，就可以把这

个 bytes 写入文件。或者用另一个方法 pickle.dump()直接把对象序列化

后写入一个 file-like Object。当我们要把对象从磁盘读到内存时，可以先把内容读到一个 bytes，然后用 pickle.loads()方法反序列化出对象，也可以直接用 pickle.load()方法从一个 file-like Object 中直接反序列化出对象。我们打开另一个Python 命令行来反序列化刚才保存的对象。Pickle 的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于 Python，并且可能不同版本的 Python 彼此都不兼容，因此，只能用 Pickle 保存那些不重要的数据，不能成功地反序列化也没关系。

JSON：如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如 XML，但更好的方法是序列化为 JSON，因为 JSON 表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘。或者通过网络传输。JSON 不仅是标准格式，并且比 XML 更快，而且可以直接在 Web 页面中读取，非常方便。JSON 表示的对象就是标准的 JavaScript 语言的对象，JSON 和 Python内置的数据类型对应如下：

JSON 类型 Python 类型

{} dict

[] list

"string" str

1234.56 int 或 float

true/false True/False

null None

dumps()方法返回一个 str，内容就是标准的 JSON。json.dumps(d)。要把 JSON 反序列化为 Python 对象，用 loads()或者对应的 load()方法，前者把 JSON 的字符串反序列化，后者从 file-like Object 中读取字符串并反序列化：json.loads(json\_str)。

#### 1.1.33 进程 线程

子进程永远返回 0，而父进程返回子进程的 ID。这样做的理由是，一个父进程可以 fork 出很多子进程，所以，父进程要记下每个子进程的 ID，而子进程只需要调用 getppid()就可以拿到父进程的 ID。Python 的 os 模块封装了常见的系统调用，其中就包括 fork，可以在

Python 程序中轻松创建子进程。 Windows 没有 fork 调用。multiprocessing 模块就是跨平台版本的多进程模块。

如果要启动大量的子进程，可以用进程池的方式批量创建子进程。p = Pool(4)。由于 Pool 的默认大小是 CPU 的核数，如果你不幸拥有 8 核 CPU，你要提交至少 9 个子进程才能看到上面的等待效果。p.apply\_async(long\_time\_task, args=(i,))。subprocess 模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

subprocess 模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。r = subprocess.call(['nslookup', 'www.python.org'])。Process 之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python 的 multiprocessing 模块包装了底层的机制，提供了Queue、Pipes 等多种方式来交换数据。

# 写数据进程执行的代码:

def write(q):

print('Process to write: %s' % os.getpid())

for value in ['A', 'B', 'C']:

print('Put %s to queue...' % value)

q.put(value)

time.sleep(random.random())

# 读数据进程执行的代码:

def read(q):

print('Process to read: %s' % os.getpid())

while True:

value = q.get(True)

print('Get %s from queue.' % value)

if \_\_name\_\_=='\_\_main\_\_':

# 父进程创建 Queue，并传给各个子进程：

q = Queue()

pw = Process(target=write, args=(q,))

pr = Process(target=read, args=(q,))

# 启动子进程 pw，写入:

pw.start()

# 启动子进程 pr，读取:

pr.start()

# 等待 pw 结束:

pw.join()

# pr 进程里是死循环，无法等待其结束，只能强行终止:

pr.terminate()

Python 的标准库提供了两个模块：\_thread 和 threading，\_thread 是低级模块，threading 是高级模块，对\_thread 进行了封装。绝大多数情况下，我们只需要使用 threading 这个高级模块。t = threading.Thread(target=loop, name='LoopThread')

t.start()

t.join()

由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过threading.Lock()来实现。当多个线程同时执行 lock.acquire()时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。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。一个 ThreadLocal 变量虽然是全局变量，但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。

#### 1.1.34 正则表达式

在正则表达式中，如果直接给出字符，就是精确匹配。用\d 可以匹配一个数字，\w 可以匹配一个字母或数字。

要匹配变长的字符，在正则表达式中，用\*表示任意个字符（包括 0 个），用+表示至少一个字符，用?表示 0 个或 1 个字符，用{n}表示 n 个字符，用{n,m}表示 n-m 个字符。

如果要匹配'010-12345'的正则是\d{3}\-\d{3,8}。

要做更精确地匹配，可以用[]表示范围，比如： [0-9a-zA-Z\\_]可以匹配一个数字、字母或者下划线； [0-9a-zA-Z\\_]+可以匹配至少由一个数字、字母或者下划线组成的字符串，比如'a100'，'0\_Z'，'Py3000'等等；

^表示行的开头，^\d 表示必须以数字开头。

$表示行的结束，\d$表示必须以数字结束。

Python 提供 re 模块，包含所有正则表达式的功能。由于 Python 的字符串本身也用\转义。import re >>> re.match(r'^\d{3}\-\d{3,8}$', '010-12345')。

re.split(r'\s+', 'a b c') 切分字符串

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用()表示的就是要提取的分组（Group）。注意到 group(0)永远是原始字符串，group(1)、group(2)……表示第 1、2、……个子串。

当我们在 Python 中使用正则表达式时，re 模块内部会干两件事情：

1. 编译正则表达式，如果正则表达式的字符串本身不合法，会报错；

2. 用编译后的正则表达式去匹配字符串。

>>> import re

# 编译:

>>> re\_telephone = re.compile(r'^(\d{3})-(\d{3,8})$')

# 使用：

>>> re\_telephone.match('010-12345').groups()

('010', '12345')

>>> re\_telephone.match('010-8086').groups()

('010', '8086')

#### 1.1.35 内建模块

datetime 是 Python 处理日期和时间的标准库

now = datetime.now() # 获取当前 datetime

dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建 datetime

dt.timestamp() # 把 timestamp 转换为 datetime

collections 是 Python 内建的一个集合模块，提供了许多有用的集合类。namedtuple 是一个函数，它用来创建一个自定义的 tuple 对象，并且规定了 tuple 元素的个数，并可以用属性而不是索引来引用 tuple 的某个元素。

deque 是为了高效实现插入和删除操作的双向列表，适合用于队列和栈。使用 list 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 list 是线性存储，数据量大的时候，插入和删除效率很低。

OrderedDict：使用 dict 时，Key 是无序的。在对 dict 做迭代时，我们无法确定 Key的顺序。如果要保持 Key 的顺序，可以用 OrderedDict。

Defaultdict：使用 dict 时，如果引用的 Key 不存在，就会抛出 KeyError。如果希望key 不存在时，返回一个默认值，就可以用 defaultdict。

Counter 是一个简单的计数器，例如，统计字符出现的个数。

Base64 是一种用 64 个字符来表示任意二进制数据的方法。所以，Base64 编码会把 3 字节的二进制数据编码为 4 字节的文本数据，长度增加 33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。如果要编码的二进制数据不是 3 的倍数，最后会剩下 1 个或 2 个字节怎么办？Base64 用\x00 字节在末尾补足后，再在编码的末尾加上 1 个或 2个=号，表示补了多少字节，解码的时候，会自动去掉。>>> import base64

>>> base64.b64encode(b'binary\x00string')

b'YmluYXJ5AHN0cmluZw=='

>>> base64.b64decode(b'YmluYXJ5AHN0cmluZw==')

b'binary\x00string'

hashlib 提供了常见的摘要算法，如 MD5，SHA1 等等。

itertools 提供了非常有用的用于操作迭代对象的函数。

Python 提供了 HTMLParser 来非常方便地解析 HTML，只需简单几行代码。

urllib 提供了一系列用于操作 URL 的功能。urllib 的 request 模块可以非常方便地抓取 URL 内容，也就是发送一个GET 请求到指定的页面，然后返回 HTTP 的响应。

PIL：操作图像。

virtualenv 就是用来为一个应用创建一套“隔离”的 Python 运行环境。

#### 1.1.36 if \_\_name\_\_ == '\_\_main\_\_'

一个python的文件有两种使用的方法，第一是直接作为脚本执行，第二是import到其他的python脚本中被调用（模块重用）执行。因此if \_\_name\_\_ == 'main': 的作用就是控制这两种情况执行代码的过程，在if \_\_name\_\_ == 'main': 下的代码只有在第一种情况下（即文件作为脚本直接执行）才会被执行，而import到其他脚本中是不会被执行的。

每个python模块（python文件，也就是此处的test.py和import\_test.py）都包含内置的变量\_\_name\_\_,当运行模块被执行的时候，\_\_name\_\_等于文件名（包含了后缀.py）；如果import到其他模块中，则\_\_name\_\_等于模块名称（不包含后缀.py）。而“\_\_main\_\_”等于当前执行文件的名称（包含了后缀.py）。进而当模块被直接执行时，\_\_name\_\_ == 'main'结果为真。同样举例说明，我们在test.py脚本的if \_\_name\_\_=="\_\_main\_\_":之前加入print \_\_name\_\_，即将\_\_name\_\_打印出来。文件内容和结果如下。

#### 1.1.37 gui模块

我们编写的 Python 代码会调用内置的 Tkinter，Tkinter 封装了访问 Tk的接口。from tkinter import \*。pack()方法把 Widget 加入到父容器中，并实现布局。pack()是最简单的布局，grid()可以实现更复杂的布局。

from tkinter import \*

import tkinter.messagebox as messagebox

class Application(Frame):

def \_\_init\_\_(self, master=None):

Frame.\_\_init\_\_(self, master)

self.pack()

self.createWidgets()

def hello(self):

name = self.nameInput.get() or 'world';

messagebox.showinfo('Message', 'Hello, %s' % name)

def createWidgets(self):

#self.helloLabel = Label(self, text='Hello, world!');

#self.helloLabel.pack();

#self.quitButton = Button(self, text='Quit', command=self.quit);

#self.quitButton.pack();

self.nameInput = Entry(self)

self.nameInput.pack()

self.alertButton = Button(self, text='Hello',command=self.hello)

self.alertButton.pack()

def function\_tkinter():

app = Application();

# 设置窗口标题:

app.master.title('Hello World');

# 主消息循环:

app.mainloop();

#### 1.1.38 网络编程

import socket

# 创建一个 socket:

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 建立连接:

s.connect(('www.sina.com.cn', 80))

s.send(b'GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection:

close\r\n\r\n')

d = s.recv(1024)

接收数据时，调用 recv(max)方法，一次最多接收指定的字节数，因此，在一个 while 循环中反复接收，直到 recv()返回空数据，表示接收完毕，退出循环。

# 关闭连接:

s.close()

s.bind(('127.0.0.1', 9999))

紧接着，调用 listen()方法开始监听端口，传入的参数指定等待连接的最大数量：

s.listen(5)

SMTP 发送邮件import smtplib

Python 内置一个 poplib 模块，实现了 POP3 协议，可以直接用来收邮件。

#### 1.1.39 数据库

要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection；连接到数据库后，需要打开游标，称之为 Cursor，通过 Cursor 执行 SQL语句，然后，获得执行结果。import sqlite3

def function\_sqllite():

# 连接到 SQLite 数据库,数据库文件是 test.db,如果文件不存在，会自动在当前目录创建:

conn = sqlite3.connect('test.db');

# 创建一个 Cursor:

cursor = conn.cursor();

#cursor.execute('create table user (id varchar(20) primary key, name varchar(20))');

#cursor.execute('insert into user (id, name) values (\'1\',\'Michael\')');

print(cursor.rowcount);

cursor.execute('select \* from user where id=?', '1');

values = cursor.fetchall();

print(values);

cursor.close();

conn.commit();# 提交事务:

conn.close();# 关闭 Connection:

使用 Cursor 对象执行 select 语句时，通过 featchall()可以拿到结果集。结果集是一个 list，每个元素都是一个 tuple，对应一行记录

#### 1.1.40 web框架

1.1.34 除了 Flask，常见的 Python Web 框架还有：

 Django：全能型 Web 框架；

 web.py：一个小巧的 Web 框架；

 Bottle：和 Flask 类似的 Web 框架；

 Tornado：Facebook 的开源异步 Web 框架。

除了 Jinja2，常见的模板还有：

 Mako：用<% ... %>和${xxx}的一个模板；

 Cheetah：也是用<% ... %>和${xxx}的一个模板；

 Django：Django 是一站式框架，内置一个用{% ... %}和{{ xxx }}

的模板。

#### 1.1.41 异步

asyncio 的编程模型就是一个消息循环。我们从 asyncio 模块中直接获取一个 EventLoop 的引用，然后把需要执行的协程扔到 EventLoop 中执行，就实现了异步 IO。

asyncio 提供了完善的异步 IO 支持；异步操作需要在 coroutine 中通过 yield from 完成；多个 coroutine 可以封装成一组 Task 然后并发执行。

asyncio 实现了 TCP、UDP、SSL 等协议，aiohttp 则是基于 asyncio 实现的 HTTP 框架。

协程，又称微线程，纤程。英文名 Coroutine。看起来 A、B 的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

Python 对协程的支持是通过 generator 实现的。在 generator 中，我们不但可以通过 for 循环来迭代，还可以不断调用next()函数获取由 yield 语句返回的下一个值。

import threading

import asyncio

@asyncio.coroutine

def hello():

print('Hello world! (%s)' % threading.currentThread())

yield from asyncio.sleep(1)

print('Hello again! (%s)' % threading.currentThread())

loop = asyncio.get\_event\_loop()

tasks = [hello(), hello()]

loop.run\_until\_complete(asyncio.wait(tasks))

loop.close()

Coroutine函数是多线程的。可见 3 个连接由一个线程通过 coroutine 并发完成。

只需要做两步简单的替换：

1. 把@asyncio.coroutine 替换为 async；

2. 把 yield from 替换为 await。

async def hello():

print("Hello world!")

r = await asyncio.sleep(1)

print("Hello again!")

### 1.2 额外功能

#### 1.2.1打包

【Terminal】打开终端，输入命令pip install pyinstaller

输入命令 pyinstaller，回车显示安装成功

输入命令 pyinstaller --console --onefile jingyan\_frame.py

## python 项目

### xlrd xlwt操作excel

import xlrd;

import xlwt;

相关列子在py代码中可以参考一下。主要有设置样式，保存 读取 sheet等。

### 豆瓣250抓包保存excel

#### request下载页面

get请求

参数也可以传递列表

payload = {'key1': 'value1', 'key2': ['value2', 'value3']}

r = requests.get('http://httpbin.org/get', params=payload)  
print(r.url)  
<http://httpbin.org/get?key1=value1&key2=value2&key2=value3>

r.text 返回headers中的编码解析的结果，可以通过r.encoding = 'gbk'来变更解码方式

r.content返回二进制结果

r.json()返回JSON格式，可能抛出异常

r.status\_code

r.raw返回原始socket respons，需要加参数stream=True

r = requests.get('https://api.github.com/events', stream=True)

r.raw

<requests.packages.urllib3.response.HTTPResponse object at 0x101194810>

都是r的方法

传入header cookie

data = requests.get('http://movie.douban.com/top250/', headers=headers, stream=True)

url = 'http://httpbin.org/cookies'

r = requests.get(url, cookies=dict(cookies\_are='working'))

r.text

'{"cookies": {"cookies\_are": "working"}}'

Post请求

传递表单

r = requests.post('http://httpbin.org/post', data = {'key':'value'})

传递文件

url = '<http://httpbin.org/post>'  
>>> files = {'file': open('report.xls', 'rb')}

>>> r = requests.post(url, files=files)

Get携带session

session，自动保存cookies，可以设置请求参数，下次请求自动带上请求参数

s = requests.Session()

s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')

r = s.get('http://httpbin.org/cookies')

print(r.text)

# '{"cookies": {"sessioncookie": "123456789"}}'

SSL证书验证

>>> requests.get('https://kennethreitz.com', verify=True)

requests.exceptions.SSLError: hostname 'kennethreitz.com' doesn't match either of '\*.herokuapp.com', 'herokuapp.com'

Stream

如果设置stream为True，请求连接不会被关闭，除非读取所有数据或者调用Response.close。

Content

resp.text返回的是Unicode型的数据。

resp.content返回的是bytes型也就是二进制的数据。

（resp.json()返回的是json格式数据）

r.raw返回原始socket respons

#### 2.2.2 BeautifulSoup从网页抓取数据

Beautiful Soup支持Python标准库中的HTML解析器,还支持一些第三方的解析器，如果我们不安装它，则 Python 会使用 Python默认的解析器，lxml 解析器更加强大，速度更快，推荐安装。

soup = BeautifulSoup(doc, 'html.parser')

|  |  |  |  |
| --- | --- | --- | --- |
| 解析器 | 使用方法 | 优势 | 劣势 |
| Python标准库 | BeautifulSoup(markup, “html.parser”) | Python的内置标准库  执行速度适中  文档容错能力强 | Python 2.7.3 or 3.2.2)前 的版本中文档容错能力差 |
| lxml HTML 解析器 | BeautifulSoup(markup, “lxml”) | 速度快  文档容错能力强 | 需要安装C语言库 |
| lxml XML 解析器 | BeautifulSoup(markup, [“lxml”, “xml”])BeautifulSoup(markup, “xml”) | 速度快  唯一支持XML的解析器 | 需要安装C语言库 |
| html5lib | BeautifulSoup(markup, “html5lib”) | 最好的容错性  以浏览器的方式解析文档  生成HTML5格式的文档 | 速度慢  不依赖外部扩展 |

Beautiful Soup将复杂HTML文档转换成一个复杂的树形结构,每个节点都是Python对象,所有对象可以归纳为4种:Tag NavigableString BeautifulSoup Comment

1. Tag 是什么？通俗点讲就是 HTML 中的一个个标签，如下使用：

print soup.title

#<title>The Dormouse's story</title>

print soup.head

#<head><title>The Dormouse's story</title></head>

print soup.a

#<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>

print soup.p

#<p class="title" name="dromouse"><b>The Dormouse's story</b></p>

对于 Tag，它有两个重要的属性，是 name 和 attrs。

如果我们想要单独获取某个属性，可以这样，例如我们获取它的 class 叫什么

print soup.p['class']

#['title']

还可以这样，利用get方法，传入属性的名称，二者是等价的

print soup.p.get('class')

#['title']

我们可以对这些属性和内容等等进行修改，例如

soup.p['class']="newClass"

print soup.p

#<p class="newClass" name="dromouse"><b>The Dormouse's story</b></p>

还可以对这个属性进行删除，例如

del soup.p['class']

print soup.p

#<p name="dromouse"><b>The Dormouse's story</b></p>

1. NavigableString

既然我们已经得到了标签的内容，那么问题来了，我们要想获取标签内部的文字怎么办呢？很简单，用 .string 即可，例如

print soup.p.string

#The Dormouse's story

这样我们就轻松获取到了标签里面的内容，想想如果用正则表达式要多麻烦。它的类型是一个 NavigableString，翻译过来叫 可以遍历的字符串，不过我们最好还是称它英文名字吧。

print type(soup.p.string)

#<class 'bs4.element.NavigableString'>

来检查一下它的类型

print type(soup.p.string)

#<class 'bs4.element.NavigableString'>

1. BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容.大部分时候,可以把它当作 Tag 对象，是一个特殊的 Tag，我们可以分别获取它的类型，名称，以及属性来感受一下

1. Comment

Comment 对象是一个特殊类型的 NavigableString 对象，其实输出的内容仍然不包括注释符号，但是如果不好好处理它，可能会对我们的文本处理造成意想不到的麻烦。

我们找一个带注释的标签

print soup.a

print soup.a.string

print type(soup.a.string)

运行结果如下

<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>

Elsie

<class 'bs4.element.Comment'>

a 标签里的内容实际上是注释，但是如果我们利用 .string 来输出它的内容，我们发现它已经把注释符号去掉了，所以这可能会给我们带来不必要的麻烦。

另外我们打印输出下它的类型，发现它是一个 Comment 类型，所以，我们在使用前最好做一下判断，判断代码如下

if type(soup.a.string)==bs4.element.Comment:

print soup.a.string

上面的代码中，我们首先判断了它的类型，是否为 Comment 类型，然后再进行其他操作，如打印输出。

**遍历文档树**

1. 直接子节点

要点：.contents属性

tag 的 .content 属性可以将tag的子节点以列表的方式输出。它返回的不是一个 list，不过我们可以通过遍历获取所有子节点。输出方式为列表，我们可以用列表索引来获取它的某一个元素。soup.head.contents[0]。

.children返回的不是一个 list，不过我们可以通过遍历获取所有子节点。我们打印输出 .children 看一下，可以发现它是一个 list 生成器对象

1. 所有子孙节点

.contents 和 .children 属性仅包含tag的直接子节点，.descendants 属性可以对所有tag的子孙节点进行递归循环，和 children类似，我们也需要遍历获取其中的内容。

for child in soup.descendants:

print child

1. 节点内容

知识点：.string 属性

如果tag只有一个 NavigableString 类型子节点,那么这个tag可以使用 .string 得到子节点。如果一个tag仅有一个子节点,那么这个tag也可以使用 .string 方法,输出结果与当前唯一子节点的 .string 结果相同。

通俗点说就是：如果一个标签里面没有标签了，那么 .string 就会返回标签里面的内容。如果标签里面只有唯一的一个标签了，那么 .string 也会返回最里面的内容。

如果tag包含了多个子节点,tag就无法确定，string 方法应该调用哪个子节点的内容, .string 的输出结果是 None。

1. 多个内容

知识点： .strings .stripped\_strings 属性

.strings

获取多个内容，不过需要遍历获取，比如下面的例子

for string in soup.strings:

print(repr(string))

.stripped\_strings

输出的字符串中可能包含了很多空格或空行,使用 .stripped\_strings 可以去除多余空白内容。

（5）父节点

知识点： .parent 属性

（6）全部父节点

知识点：.parents 属性

通过元素的 .parents 属性可以递归得到元素的所有父辈节点。

（7）兄弟节点

知识点：.next\_sibling .previous\_sibling 属性

兄弟节点可以理解为和本节点处在统一级的节点，.next\_sibling 属性获取了该节点的下一个兄弟节点，.previous\_sibling 则与之相反，如果节点不存在，则返回 None

注意：实际文档中的tag的 .next\_sibling 和 .previous\_sibling 属性通常是字符串或空白，因为空白或者换行也可以被视作一个节点，所以得到的结果可能是空白或者换行

1. 全部兄弟节点

知识点：.next\_siblings .previous\_siblings 属性

通过 .next\_siblings 和 .previous\_siblings 属性可以对当前节点的兄弟节点迭代输出

（9）前后节点

知识点：.next\_element .previous\_element 属性

与 .next\_sibling .previous\_sibling 不同，它并不是针对于兄弟节点，而是在所有节点，不分层次

比如 head 节点为

<head><title>The Dormouse's story</title></head>

那么它的下一个节点便是 title，它是不分层次关系的

（10）所有前后节点

知识点：.next\_elements .previous\_elements 属性

通过 .next\_elements 和 .previous\_elements 的迭代器就可以向前或向后访问文档的解析内容,就好像文档正在被解析一样

**搜索文档树**

（1）find\_all( name , attrs , recursive , text , \*\*kwargs )

find\_all() 方法搜索当前tag的所有tag子节点,并判断是否符合过滤器的条件。

1）name 参数

name 参数可以查找所有名字为 name 的tag,字符串对象会被自动忽略掉

A.传字符串

最简单的过滤器是字符串.在搜索方法中传入一个字符串参数,Beautiful Soup会查找与字符串完整匹配的内容,下面的例子用于查找文档中所有的<b>标签

soup.find\_all('b')

# [<b>The Dormouse's story</b>]

B.传正则表达式

如果传入正则表达式作为参数,Beautiful Soup会通过正则表达式的 match() 来匹配内容.下面例子中找出所有以b开头的标签,这表示<body>和<b>标签都应该被找到

import re

for tag in soup.find\_all(re.compile("^b")):

print(tag.name)

C.传列表

如果传入列表参数,Beautiful Soup会将与列表中任一元素匹配的内容返回.下面代码找到文档中所有<a>标签和<b>标签

soup.find\_all(["a", "b"])

D.传 True

True 可以匹配任何值,下面代码查找到所有的tag,但是不会返回字符串节点

for tag in soup.find\_all(True):

print(tag.name)

E.传方法

如果没有合适过滤器,那么还可以定义一个方法,方法只接受一个元素参数 [4] ,如果这个方法返回 True 表示当前元素匹配并且被找到,如果不是则反回 False

下面方法校验了当前元素,如果包含 class 属性却不包含 id 属性,那么将返回 True:

def has\_class\_but\_no\_id(tag):

return tag.has\_attr('class') and not tag.has\_attr('id')

def has\_class\_but\_no\_id(tag):

return tag.has\_attr('class') and not tag.has\_attr('id')

将这个方法作为参数传入 find\_all() 方法,将得到所有<p>标签:

soup.find\_all(has\_class\_but\_no\_id)

2）keyword 参数

注意：如果一个指定名字的参数不是搜索内置的参数名,搜索时会把该参数当作指定名字tag的属性来搜索,如果包含一个名字为 id 的参数,Beautiful Soup会搜索每个tag的”id”属性

soup.find\_all(id='link2')

如果传入 href 参数,Beautiful Soup会搜索每个tag的”href”属性

soup.find\_all(href=re.compile("elsie"))

# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]

使用多个指定名字的参数可以同时过滤tag的多个属性

soup.find\_all(href=re.compile("elsie"), id='link1')

# [<a class="sister" href="http://example.com/elsie" id="link1">three</a>]

在这里我们想用 class 过滤，不过 class 是 python 的关键词，这怎么办？加个下划线就可以

soup.find\_all("a", class\_="sister")

有些tag属性在搜索不能使用,比如HTML5中的 data-\* 属性

data\_soup = BeautifulSoup('<div data-foo="value">foo!</div>')

data\_soup.find\_all(data-foo="value")

但是可以通过 find\_all() 方法的 attrs 参数定义一个字典参数来搜索包含特殊属性的tag

data\_soup.find\_all(attrs={"data-foo": "value"})

3）text 参数

通过 text 参数可以搜搜文档中的字符串内容.与 name 参数的可选值一样, text 参数接受 字符串 , 正则表达式 , 列表, True

soup.find\_all(text="Elsie")

4）limit 参数

find\_all() 方法返回全部的搜索结构,如果文档树很大那么搜索会很慢.如果我们不需要全部结果,可以使用 limit 参数限制返回结果的数量.效果与SQL中的limit关键字类似,当搜索到的结果数量达到 limit 的限制时,就停止搜索返回结果.

文档树中有3个tag符合搜索条件,但结果只返回了2个,因为我们限制了返回数量

5）recursive 参数

调用tag的 find\_all() 方法时,Beautiful Soup会检索当前tag的所有子孙节点,如果只想搜索tag的直接子节点,可以使用参数 recursive=False .

（2）find( name , attrs , recursive , text , \*\*kwargs )

它与 find\_all() 方法唯一的区别是 find\_all() 方法的返回结果是值包含一个元素的列表,而 find() 方法直接返回结果

（3）find\_parents() find\_parent()

find\_all() 和 find() 只搜索当前节点的所有子节点,孙子节点等. find\_parents() 和 find\_parent() 用来搜索当前节点的父辈节点,搜索方法与普通tag的搜索方法相同,搜索文档搜索文档包含的内容

（4）find\_next\_siblings() find\_next\_sibling()

这2个方法通过 .next\_siblings 属性对当 tag 的所有后面解析的兄弟 tag 节点进行迭代, find\_next\_siblings() 方法返回所有符合条件的后面的兄弟节点,find\_next\_sibling() 只返回符合条件的后面的第一个tag节点

（5）find\_previous\_siblings() find\_previous\_sibling()

这2个方法通过 .previous\_siblings 属性对当前 tag 的前面解析的兄弟 tag 节点进行迭代, find\_previous\_siblings() 方法返回所有符合条件的前面的兄弟节点, find\_previous\_sibling() 方法返回第一个符合条件的前面的兄弟节点

（6）find\_all\_next() find\_next()

这2个方法通过 .next\_elements 属性对当前 tag 的之后的 tag 和字符串进行迭代, find\_all\_next() 方法返回所有符合条件的节点, find\_next() 方法返回第一个符合条件的节点

（7）find\_all\_previous() 和 find\_previous()

这2个方法通过 .previous\_elements 属性对当前节点前面的 tag 和字符串进行迭代, find\_all\_previous() 方法返回所有符合条件的节点, find\_previous()方法返回第一个符合条件的节点

**CSS选择器**

我们在写 CSS 时，标签名不加任何修饰，类名前加点，id名前加 #，在这里我们也可以利用类似的方法来筛选元素，用到的方法是 soup.select()，返回类型是 list

（1）通过标签名查找

print soup.select('title')

（2）通过类名查找

print soup.select('.sister')

（3）通过 id 名查找

print soup.select('#link1')

（4）组合查找

组合查找即和写 class 文件时，标签名与类名、id名进行的组合原理是一样的，例如查找 p 标签中，id 等于 link1的内容，二者需要用空格分开

print soup.select('p #link1')

直接子标签查找

print soup.select("head > title")

（5）属性查找

查找时还可以加入属性元素，属性需要用中括号括起来，注意属性和标签属于同一节点，所以中间不能加空格，否则会无法匹配到。

print soup.select('a[class="sister"]')

同样，属性仍然可以与上述查找方式组合，不在同一节点的空格隔开，同一节点的不加空格

print soup.select('p a[href="http://example.com/elsie"]')

以上的 select 方法返回的结果都是列表形式，可以遍历形式输出，然后用 get\_text() 方法来获取它的内容。

#### 2.2.3 参考链接：

<https://cuiqingcai.com/1319.html>

### Scrapy简单入门及实例讲解

#### 2.3.1 scrapy介绍

Scrapy是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 其可以应用在数据挖掘，信息处理或存储历史数据等一系列的程序中。其最初是为了页面抓取 (更确切来说, 网络抓取 )所设计的， 也可以应用在获取API所返回的数据(例如 Amazon Associates Web Services ) 或者通用的网络爬虫。Scrapy用途广泛，可以用于数据挖掘、监测和自动化测试。

Scrapy 使用了 Twisted异步网络库来处理网络通讯。整体架构大致如下：



Scrapy主要包括了以下组件：

**引擎(Scrapy)**  
用来处理整个系统的数据流, 触发事务(框架核心)

**调度器(Scheduler)**  
用来接受引擎发过来的请求, 压入队列中, 并在引擎再次请求的时候返回. 可以想像成一个URL（抓取网页的网址或者说是链接）的优先队列, 由它来决定下一个要抓取的网址是什么, 同时去除重复的网址

**下载器(Downloader)**  
用于下载网页内容, 并将网页内容返回给蜘蛛(Scrapy下载器是建立在twisted这个高效的异步模型上的)

**爬虫(Spiders)**  
爬虫是主要干活的, 用于从特定的网页中提取自己需要的信息, 即所谓的实体(Item)。用户也可以从中提取出链接,让Scrapy继续抓取下一个页面

**项目管道(Pipeline)**  
负责处理爬虫从网页中抽取的实体，主要的功能是持久化实体、验证实体的有效性、清除不需要的信息。当页面被爬虫解析后，将被发送到项目管道，并经过几个特定的次序处理数据。

**下载器中间件(Downloader Middlewares)**  
位于Scrapy引擎和下载器之间的框架，主要是处理Scrapy引擎与下载器之间的请求及响应。

**爬虫中间件(Spider Middlewares)**  
介于Scrapy引擎和爬虫之间的框架，主要工作是处理蜘蛛的响应输入和请求输出。

**调度中间件(Scheduler Middewares)**  
介于Scrapy引擎和调度之间的中间件，从Scrapy引擎发送到调度的请求和响应。

Scrapy运行流程大概如下：

1.引擎从调度器中取出一个链接(URL)用于接下来的抓取

2.引擎把URL封装成一个请求(Request)传给下载器

3.下载器把资源下载下来，并封装成应答包(Response)

4,爬虫解析Response

5.解析出实体（Item）,则交给实体管道进行进一步的处理

6.解析出的是链接（URL）,则把URL交给调度器等待抓取

#### 2.3.1 scrpay在pycharm实例抓美剧

需要安装scrapy wheel lxml pyopenssl Twisted pywin32

Scrapy爬虫的运行需要到命令行下运行，在pychram中左下角有个Terminal,点开就可以在Pycharm下进入命令行，默认是在项目目录下的，要运行项目。

1、创建工程

scrapy startproject movie

2、创建爬虫程序

cd movie

scrapy genspider meiju meijutt.com

Tips:在创建爬虫时使用模板更加方便一些，如：

scrapy genspider [-t template] <name> <domain> 即：scrapy genspider testDemoSpider baidu.com

3、自动创建目录及文件

Spiders

4、文件说明：

scrapy.cfg 项目的配置信息，主要为Scrapy命令行工具提供一个基础的配置信息。（真正爬虫相关的配置信息在settings.py文件中）

items.py 设置数据存储模板，用于结构化数据，如：Django的Model

pipelines 数据处理行为，如：一般结构化的数据持久化

settings.py 配置文件，如：递归的层数、并发数，延迟下载等

spiders 爬虫目录，如：创建文件，编写爬虫规则

注意：一般创建爬虫文件时，以网站域名命名

5、设置数据存储模板

items.py

import scrapy

class MovieItem(scrapy.Item):

# define the fields for your item here like:

# name = scrapy.Field()

name = scrapy.Field()

1. 编写爬虫

meiju.py

# -\*- coding: utf-8 -\*-

import scrapy

from movie.items import MovieItem

class MeijuSpider(scrapy.Spider):

name = "meiju"

allowed\_domains = ["meijutt.com"]

start\_urls = ['http://www.meijutt.com/new100.html']

def parse(self, response):

movies = response.xpath('//ul[@class="top-list fn-clear"]/li')

for each\_movie in movies:

= item['name'] = each\_movie.xpath('./h5/a/@title').extract()[0]

yield item

7、设置配置文件

settings.py增加如下内容

ITEM\_PIPELINES = {'movie.pipelines.MoviePipeline':100}

8、编写数据处理脚本

pipelines.py

class MoviePipeline(object):

def process\_item(self, item, spider):

with open("my\_meiju.txt",'a') as fp:

fp.write(item['name'].encode("utf8") + '\n')

9、执行爬虫

cd movie

scrapy crawl meiju --nolog

scrapy crawl 爬虫名 ，即可运行爬虫。

### python+tensorflow+opencv人脸识别

参考链接：<https://blog.csdn.net/qq_42633819/article/details/81191308>

参考github：<https://github.com/Hironsan/BossSensor>

参考链接：<https://blog.csdn.net/mdzzname/article/details/78239973>

Anaconda里面集成了很多关于python科学计算的第三方库，主要是安装方便，如果不使用anaconda，那么安装起来会很麻烦，各个库之间的依赖性就很难连接的很好，简单点说就是Anaconda已经包含了python,而且安装非常快捷方便。

在安装Anaconda的时候特别需要注意的是，有人到官网以后看到了最新的Anacoda3立马就下了最新的，需要说明的是，Anaconda2和3分别对应的是python版本的2.7和3.6,python的2和3的差别是比较明显的，然而opencv目前支持的python版本是2.7的，如何使用Anaconda3的话，在pycharm中调用的时候会出现无法导入opencv库的情况！！

#### 2.4.1 搭建环境

##### 1.Anaconda2安装

Anaconda2的下载：<https://www.anaconda.com/distribution/>

进行安装

##### 2.opencv安装

opencv库下载好以后，进入安装目录的build\python\2.7的目录，这里面有两个目录,一个是64位的一个是32位的，里面都包含着一个cv2.pyd文件，复制与你安装Anaconda位数相同的里面的cv2.pyd文件，将其粘贴到Anaconda安装目录的Lib\site-packages目录下。

如果不行得话，可以手动安装，先先下载一个whl文件，关于opencv的，下载地址在这里：<https://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv。>



当时下载红色的那个（对应的是cv3.4版，python3.6版）

现在应该3.4.1了，最新的肯定没问题，只是pip 时候名字对应，跟自己python3.x，win\_x对应就好 win32是32位，amd64是64位。

下完了，放到conda对应的文件夹里：

可以在pycharm terminal在输入

cd C:\Anaconda3\Lib\site-packages\

输入打开命令行：

pip install opencv\_python-3.4.6-cp37-cp37m-win\_amd64.whl

差不多完成了。

##### 3.tensorflow安装

TensorFlow即可以支持CPU，也可以支持CPU+GPU。前者的环境需求简单，后者需要额外的支持。具备了上述安装条件（CPU或GPU）之后，TensorFlow可以通过两种方式进行安装。一是”native” pip，二是Anaconda。

**Anaconda配置tensorflow**

1. 检查Anaconda是否成功安装：conda --version
2. 检测目前安装了哪些环境：conda info --envs
3. 检查目前有哪些版本的python可以安装：conda search --full-name python
4. 安装不同版本的python：conda create --name tensorflow python=3.6最好安装3.6或者3.5
5. 按照提示，激活之：activate tensorflow
6. 确保名叫tensorflow的环境已经被成功添加：conda info --envs
7. 检查新环境中的python版本：python --version
8. 退出当前环境：deactivate
9. 切换环境：activate tensorflow

**安装tensorflow**

1.一种尝试

安装CPU版本输入

安装GPU版本输入

Pip install --ignore-installed --upgrade tensorflow-gpu

（网络不好，多下载几次就成功了）

2.另一种尝试：pip install tensorflow

3.配置pycharm中使用tensorflw得python.exe

安装完设置pycharm选择anaconda下anaconda2/envs/python.exe

File-》setting-》project interpreter-》选择设置-》add-》conda environment-》exsitingt environmeng-》选择anaconda2/envs/tensorflow下python.exe。

参考链接：<https://blog.csdn.net/Eppley/article/details/79297503>

##### 4.sklearn等搭建

自2007年发布以来，scikit-learn已经成为Python重要的机器学习库了。scikit-learn简称sklearn，支持包括分类、回归、降维和聚类四大机器学习算法。还包含了特征提取、数据处理和模型评估三大模块。

  sklearn是Scipy的扩展，建立在NumPy和matplotlib库的基础上。利用这几大模块的优势，可以大大提高机器学习的效率。

  sklearn拥有着完善的文档，上手容易，具有着丰富的API，在学术界颇受欢迎。sklearn已经封装了大量的机器学习算法，包括LIBSVM和LIBINEAR。同时sklearn内置了大量数据集，节省了获取和整理数据集的时间。

在anaconda2里面添加tensorflow sklearn等库，一个利用pip install 有时候网络不好 还有一种是用anaconda2 navigater界面

安装sklearn可以通过pip install安装

pip install keras

pip install Keras-2.2.4-py2.py3-none-any.whl 可以下载在：<https://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv，然后通过pip安装也可以通过pip> uninstall卸载。

pip install sklearn

#### 2.4.2 函数讲解

选择使用人脸识别分类器

classfier = cv2.CascadeClassifier("./haarcascade\_frontalface\_alt2.xml")

Xml位于opencv\\build\\etc\\haarcascades\\haarcascade\_frontalface\_alt2.xml

人脸检测器（默认）：haarcascade\_frontalface\_default.xml

人脸检测器（快速Harr）：haarcascade\_frontalface\_alt2.xml

人脸检测器（侧视）：haarcascade\_profileface.xml

眼部检测器（左眼）：haarcascade\_lefteye\_2splits.xml

眼部检测器（右眼）：haarcascade\_righteye\_2splits.xml

嘴部检测器：haarcascade\_mcs\_mouth.xml

鼻子检测器：haarcascade\_mcs\_nose.xml

身体检测器：haarcascade\_fullbody.xml

人脸检测器（快速LBP）：lbpcascade\_frontalface.xml

人脸检测，1.2和2分别为图片缩放比例和需要检测的有效点数

faceRects = classfier.detectMultiScale(grey, scaleFactor = 1.2, minNeighbors = 3, minSize = (32, 32))

这个函数完成对人脸的识别以及用一个框框给框起来，其中grey是要识别的图像数据，转化为灰度可以减少计算量。scaleFactor：图像缩放比例，可以理解为同一个物体与相机距离不同，其大小亦不同，必须将其缩放到一定大小才方便识别，该参数指定每次缩放的比例。minNeighbors：对特征检测点周边多少有效点同时检测，这样可避免因选取的特征检测点太小而导致遗漏。minSize：特征检测点的最小值。

1. pycharm中配置解析器

安装完设置pycharm选择anaconda下python.exe

File-》setting-》project interpreter-》选择设置-》add-》conda environment-》exsitingt environmeng-》选择anaconda2得安装路径下ptyho.exe。

# LINUX

## Linux系统函数

### 1.1 select函数

在Linux中，我们可以使用select函数实现I/O端口的复用，传递给 select函数的参数会告诉内核：

•我们所关心的文件描述符

•对每个描述符，我们所关心的状态。(我们是要想从一个文件描述符中读或者写，还是关注一个描述符中是否出现异常)

•我们要等待多长时间。(我们可以等待无限长的时间，等待固定的一段时间，或者根本就不等待)

从 select函数返回后，内核告诉我们一下信息：

•对我们的要求已经做好准备的描述符的个数

•对于三种条件哪些描述符已经做好准备.(读，写，异常)

有了这些返回信息，我们可以调用合适的I/O函数(通常是 read 或 write)，并且这些函数不会再阻塞。

|  |
| --- |
| #include <sys/select.h>  **int** select(**int maxfdp1, fd\_set \*readset, fd\_set \*writeset, fd\_set \*exceptset,struct timeval \*timeout);** |

返回：做好准备的文件描述符的个数，超时为0，错误为 -1.

首先我们先看一下最后一个参数。它指明我们要等待的时间：

struct timeval{

long tv\_sec; /\*秒 \*/

long tv\_usec; /\*微秒 \*/

}

有三种情况：

timeout == NULL 等待无限长的时间。等待可以被一个信号中断。当有一个描述符做好准备或者是捕获到一个信号时函数会返回。如果捕获到一个信号， select函数将返回 -1,并将变量 erro设为 EINTR。

timeout->tv\_sec == 0 &&timeout->tv\_usec == 0不等待，直接返回。加入描述符集的描述符都会被测试，并且返回满足要求的描述符的个数。这种方法通过轮询，无阻塞地获得了多个文件描述符状态。

timeout->tv\_sec !=0 ||timeout->tv\_usec!= 0 等待指定的时间。当有描述符符合条件或者超过超时时间的话，函数返回。在超时时间即将用完但又没有描述符合条件的话，返回 0。对于第一种情况，等待也会被信号所中断。

中间的三个参数 readset, writset, exceptset,指向描述符集。这些参数指明了我们关心哪些描述符，和需要满足什么条件(可写，可读，异常)。一个文件描述集保存在 fd\_set 类型中。fd\_set类型变量每一位代表了一个描述符。我们也可以认为它只是一个由很多二进制位构成的数组，所以fd\_set每一位代表一个fd。如下图所示：



对于 fd\_set类型的变量我们所能做的就是声明一个变量，为变量赋一个同种类型变量的值，或者使用以下几个宏来控制它：

|  |  |
| --- | --- |
|  | #include <sys/select.h>  **int** FD\_ZERO(**int fd, fd\_set \*fdset);**  **int** FD\_CLR(**int fd, fd\_set \*fdset);**  **int** FD\_SET(**int fd, fd\_set \*fd\_set);**  **int** FD\_ISSET(**int fd, fd\_set \*fdset);</span>** |

D\_ZERO宏将一个 fd\_set类型变量的所有位都设为 0，使用FD\_SET将变量的某个位置位。清除某个位时可以使用 FD\_CLR，我们可以使用 FD\_SET来测试某个位是否被置位。

当声明了一个文件描述符集后，必须用FD\_ZERO将所有位置零。之后将我们所感兴趣的描述符所对应的位置位，操作如下：

fd\_set rset;

int fd;

FD\_ZERO(&rset);

FD\_SET(fd, &rset);

FD\_SET(stdin, &rset);

select返回后，用FD\_ISSET测试给定位是否置位：

|  |  |
| --- | --- |
|  | **if**(FD\_ISSET(fd, &rset)  { ... }</span> |

具体解释select的参数：

（1）intmaxfdp是一个整数值，是指集合中所有文件描述符的范围，即所有文件描述符的最大值加1，不能错。

说明：对于这个原理的解释可以看上边fd\_set的详细解释，fd\_set是以位图的形式来存储这些文件描述符。maxfdp也就是定义了位图中有效的位的个数。

（2）fd\_set\*readfds是指向fd\_set结构的指针，这个集合中应该包括文件描述符，我们是要监视这些文件描述符的读变化的，即我们关心是否可以从这些文件中读取数据了，如果这个集合中有一个文件可读，select就会返回一个大于0的值，表示有文件可读；如果没有可读的文件，则根据timeout参数再判断是否超时，若超出timeout的时间，select返回0，若发生错误返回负值。可以传入NULL值，表示不关心任何文件的读变化。

（3）fd\_set\*writefds是指向fd\_set结构的指针，这个集合中应该包括文件描述符，我们是要监视这些文件描述符的写变化的，即我们关心是否可以向这些文件中写入数据了，如果这个集合中有一个文件可写，select就会返回一个大于0的值，表示有文件可写，如果没有可写的文件，则根据timeout参数再判断是否超时，若超出timeout的时间，select返回0，若发生错误返回负值。可以传入NULL值，表示不关心任何文件的写变化。

（4）fd\_set\*errorfds同上面两个参数的意图，用来监视文件错误异常文件。

（5）structtimeval\* timeout是select的超时时间，这个参数至关重要，它可以使select处于三种状态，第一，若将NULL以形参传入，即不传入时间结构，就是将select置于阻塞状态，一定等到监视文件描述符集合中某个文件描述符发生变化为止；第二，若将时间值设为0秒0毫秒，就变成一个纯粹的非阻塞函数，不管文件描述符是否有变化，都立刻返回继续执行，文件无变化返回0，有变化返回一个正值；第三，timeout的值大于0，这就是等待的超时时间，即 select在timeout时间内阻塞，超时时间之内有事件到来就返回了，否则在超时后不管怎样一定返回，返回值同上述。

说明：

函数返回：

（1）当监视的相应的文件描述符集中满足条件时，比如说读文件描述符集中有数据到来时，内核(I/O)根据状态修改文件描述符集，并返回一个大于0的数。

（2）当没有满足条件的文件描述符，且设置的timeval监控时间超时时，select函数会返回一个为0的值。

（3）当select返回负值时，发生错误。

理解select模型：

理解select模型的关键在于理解fd\_set,为说明方便，取fd\_set长度为1字节，fd\_set中的每一bit可以对应一个文件描述符fd。则1字节长的fd\_set最大可以对应8个fd。

（1）执行fd\_set set;FD\_ZERO(&set);则set用位表示是0000,0000。

（2）若fd＝5,执行FD\_SET(fd,&set);后set变为0001,0000(第5位置为1)

（3）若再加入fd＝2，fd=1,则set变为0001,0011

（4）执行select(6,&set,0,0,0)阻塞等待

（5）若fd=1,fd=2上都发生可读事件，则select返回，此时set变为0000,0011。注意：没有事件发生的fd=5被清空。

基于上面的讨论，可以轻松得出select模型的特点：

（1)可监控的文件描述符个数取决与sizeof(fd\_set)的值。我这边服务器上sizeof(fd\_set)＝512，每bit表示一个文件描述符，则我服务器上支持的最大文件描述符是512\*8=4096。据说可调，另有说虽然可调，但调整上限受于编译内核时的变量值。

（2）将fd加入select监控集的同时，还要再使用一个数据结构array保存放到select监控集中的fd，一是用于再select返回后，array作为源数据和fd\_set进行FD\_ISSET判断。二是select返回后会把以前加入的但并无事件发生的fd清空，则每次开始 select前都要重新从array取得fd逐一加入（FD\_ZERO最先），扫描array的同时取得fd最大值maxfd，用于select的第一个参数。

（3）可见select模型必须在select前循环array（加fd，取maxfd），select返回后循环array（FD\_ISSET判断是否有时间发生）。

基本原理：



 select()系统调用顺序如下：sys\_select() à core\_sys\_select() à do\_select() à fop->poll()



参考链接：

<https://www.cnblogs.com/ccsccs/articles/4224253.html>

## linux命令

## gdb

### 3.1gdb调试

对C/C++程序的调试，需要在编译前就加上-g选项:

$g++ -g hello.cpp -o hello

调试core文件(core是程序非法执行后core dump后产生的文件):

$gdb <program> <core dump file>

$gdb program core.11127

运行

run：简记为 r ，其作用是运行程序，当遇到断点后，程序会在断点处停止运行，等待用户输入下一步的命令。

continue （简写c ）：继续执行，到下一个断点处（或运行结束）

next：（简写 n），单步跟踪程序，当遇到函数调用时，也不进入此函数体；此命令同 step 的主要区别是，step 遇到用户自定义的函数，将步进到函数中去运行，而 next 则直接调用函数，不会进入到函数体内。

step（简写s）：单步调试如果有函数调用，则进入函数；与命令n不同，n是不进入调用的函数的

until：当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

until+行号： 运行至某行，不仅仅用来跳出循环

finish： 运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。

call 函数(参数)：调用程序中可见的函数，并传递“参数”，如：call gdb\_test(55)

quit：简记为 q ，退出gdb

设置断点

break n （简写b n）:在第n行处设置断点

（可以带上代码路径和代码名称： b OAGUPDATE.cpp:578）

b fn1 if a＞b：条件断点设置

break func（break缩写为b）：在函数func()的入口处设置断点，如：break cb\_button

delete 断点号n：删除第n个断点

disable 断点号n：暂停第n个断点

enable 断点号n：开启第n个断点

clear 行号n：清除第n行的断点

info b （info breakpoints） ：显示当前程序的断点设置情况

delete breakpoints：清除所有断点：

查看源代码

list ：简记为 l ，其作用就是列出程序的源代码，默认每次显示10行。

list 行号：将显示当前文件以“行号”为中心的前后10行代码，如：list 12

list 函数名：将显示“函数名”所在函数的源代码，如：list main

list ：不带参数，将接着上一次 list 命令的，输出下边的内容。

打印表达式

print 表达式：简记为 p ，其中“表达式”可以是任何当前正在被测试程序的有效表达式，比如当前正在调试C语言的程序，那么“表达式”可以是任何C语言的有效表达式，包括数字，变量甚至是函数调用。

print a：将显示整数 a 的值

print ++a：将把 a 中的值加1,并显示出来

print name：将显示字符串 name 的值

print gdb\_test(22)：将以整数22作为参数调用 gdb\_test() 函数

print gdb\_test(a)：将以变量 a 作为参数调用 gdb\_test() 函数

display 表达式：在单步运行时将非常有用，使用display命令设置一个表达式后，它将在每次单步进行指令后，紧接着输出被设置的表达式及值。如： display a

watch 表达式：设置一个监视点，一旦被监视的“表达式”的值改变，gdb将强行终止正在被调试的程序。如： watch a

whatis ：查询变量或函数

info function： 查询函数

扩展info locals： 显示当前堆栈页的所有变量

查询运行信息

where/bt ：当前运行的堆栈列表；

bt backtrace 显示当前调用堆栈

up/down 改变堆栈显示的深度

set args 参数:指定运行时的参数

show args：查看设置好的参数

info program： 来查看程序的是否在运行，进程号，被暂停的原因。

分割窗口

layout：用于分割窗口，可以一边查看代码，一边测试：

layout src：显示源代码窗口

layout asm：显示反汇编窗口

layout regs：显示源代码/反汇编和CPU寄存器窗口

layout split：显示源代码和反汇编窗口

Ctrl + L：刷新窗口

### 3.2gdb 调试coredump文件过程

gdb 调试coredump文件过程：

**第一步：首先需要一个进程的coredump文件，怎么搞出coredump文件呢？**

1、 ps -fax|grep                 进程名称 找到进程的pid

2、gdb -p pid                     调试进程

3、gcore coredump名称        则生成core文件

**第二步：找出coredump文件的应用程序**

1、gdb -c corefile   使用gdb调试core文件

2、info auxv          索引31对应的是core文件的应用程序

**第三部：gdb使用应用程序调试coredump文件**

gdb  coredump应用程序  coredump文件     调试coredump文件

通过以上三步就可以调试coredump文件了

**通过以下命令调试coredump文件**

info threads 显示所有线程

bt 显示线程堆栈信息

thread thread\_num   切换线程

frame num  切换栈

info r 显示当前帧的寄存器信息 （每一帧的寄存器信息都是不相同的）

**readelf应用coredump**

readelf -h 读取coredump文件头

readelf -wl 读取应用程序debug\_line

readelf -wf 读取应用程序fde和cie信息

参考链接：https://www.cnblogs.com/wangjian8888/p/5950912.html

# CMake

## 3.1 CMake

### 3.1.1 介绍

CMake是一种跨平台编译工具，比make更为高级。CMake主要是编写CMakeLists.txt文件，然后用cmake命令将CMakeLists.txt文件转化为make所需要的makefile文件，最后用make命令编译源码生成可执行程序。

cmake 指向CMakeLists.txt所在的目录，例如cmake .. 表示CMakeLists.txt在当前目录的上一级目录。

### 3.1.2 cmake编译

cmake会生成很多中间文件，所以建议新建一个文件夹存放，分四个步骤：

1. mkdir bulid   //新建文件夹存放编译的中间文件
2. cd bulid //进入bulide文件夹
3. cmake .. //因为CMakeLists.txt文件在当前目录的外面，所以cmake命令需要使用cmake ..
4. Make //make命令根据makefile文件编译源码，在bulid目录下生成一个可执行文件，运行获取结果

CMakeLists.txt主要包含以上的7个步骤：

#1.cmake verson，指定cmake版本

cmake\_minimum\_required(VERSION 3.13)

#2.project name，指定项目的名称，一般和项目的文件夹名称FirstProject对应PROJECT(FirstProject)

#3.head file path，头文件目录

INCLUDE\_DIRECTORIES()

#4.source directory，源文件目录

AUX\_SOURCE\_DIRECTORY(src DIR\_SRCS)

#5.set environment variable，设置环境变量，编译用到的源文件全部都要放到这里，否则编译能够通过，但是执行的时候会出现各种问题，比如"symbol lookup error xxxxx , undefined symbol"

SET(TEST\_MATH) 需要的源文件

#6.add executable file，添加要编译的可执行文件

ADD\_EXECUTABLE(${PROJECT\_NAME} ${TEST\_MATH})

#7.add link library，添加可执行文件所需要的库（命名规则：lib+name+.so）,就添加该库的名称TARGET\_LINK\_LIBRARIES(${PROJECT\_NAME} m)

### 3.1.3 cmake语法

1. #： 注释

2.变量：使用set命令显式定义及赋值，在非if语句中，使用${}引用，if中直接使用变量名引用；后续的set命令会清理变量原来的值；${FS\_BUILD\_BINARY\_PREFIX}

3.command (args ...)： #命令不分大小写，参数使用空格分隔，使用双引号引起参数中空格

4.set(var a;b;c) <=> set(var a b c)：#定义变量var并赋值为a;b;c这样一个string list

5.Add\_executable(${var}) <=> Add\_executable(a b c): #变量使用${xxx}引用

6.条件语句：

if(var) #var非empty 0 N No OFF FALSE... #非运算使用NOT

…

else()/elseif() … endif(var)

7. 循环语句

Set(VAR a b c)

Foreach(f ${VAR}) …Endforeach(f)

8. 循环语句

WHILE() … ENDWHILE()

### 3.1.4cmake变量

CMAKE\_C\_COMPILER：指定C编译器

CMAKE\_CXX\_COMPILER：

CMAKE\_C\_FLAGS：编译C文件时的选项，如-g；也可以通过add\_definitions添加编译选项

EXECUTABLE\_OUTPUT\_PATH：可执行文件的存放路径

LIBRARY\_OUTPUT\_PATH：库文件路径

CMAKE\_BUILD\_TYPE:：build 类型(Debug, Release, ...)，CMAKE\_BUILD\_TYPE=Debug

BUILD\_SHARED\_LIBS：Switch between shared and static libraries

内置变量的使用：

>> 在CMakeLists.txt中指定，使用set

>> cmake命令中使用，如cmake -DBUILD\_SHARED\_LIBS=OFF

### 3.1.5cmake命令

**project** (HELLO)   #指定项目名称，生成的VC项目的名称；

>>使用${HELLO\_SOURCE\_DIR}表示项目根目录

**include\_directories**：指定头文件的搜索路径，相当于指定gcc的-I参数

>> include\_directories (${HELLO\_SOURCE\_DIR}/Hello)  #增加Hello为include目录

**link\_directories**：动态链接库或静态链接库的搜索路径，相当于gcc的-L参数

       >> link\_directories (${HELLO\_BINARY\_DIR}/Hello)     #增加Hello为link目录

**add\_subdirectory**：包含子目录

       >> add\_subdirectory (Hello)

**add\_executable**：编译可执行程序，指定编译，好像也可以添加.o文件

       >> add\_executable (helloDemo demo.cxx demo\_b.cxx)   #将cxx编译成可执行文件——

**add\_definitions**：添加编译参数

>> add\_definitions(-DDEBUG)将在gcc命令行添加DEBUG宏定义；

>> add\_definitions( “-Wall -ansi –pedantic –g”)

**target\_link\_libraries**：添加链接库,相同于指定-l参数，添加动态库和静态库。

>> target\_link\_libraries(demo Hello) #将可执行文件与Hello连接成最终文件demo

**add\_library**:

>> add\_library(Hello hello.cxx)  #将hello.cxx编译成静态库如libHello.a

**add\_custom\_target**:

**message**( status|fatal\_error, “message”):在执行CMAKE时打印消息。message([SEND\_ERROR | STATUS | FATAL\_ERROR] "message to display" ...)

**set\_target\_properties**( ... ): lots of properties... OUTPUT\_NAME, VERSION, ....

**link\_libraries**( lib1 lib2 ...): All targets link with the same set of libs

**IF语句中使用字串或数字比较**：数字比较LESS、GREATER、EQUAL，字串比STRLESS、STRGREATER、STREQUAL。

### 3.1.6 cmake基础

CMakeLists.txt 的语法比较简单，由命令、注释和空格组成，其中命令是不区分大小写的。符号 # 后面的内容被认为是注释。命令由命令名称、小括号和参数组成，参数之间使用空格进行间隔。

PROJECT\_BINARY\_DIR=全路径/build

PROJECT\_SOURCE\_DIR=全路径/src

一个是src源文件目录，一个是build目录

**多源多目标**

cmake\_minimum\_required (VERSION 2.8)：说明CMake 最低版本号要求

project (Demo2)：项目信息

aux\_source\_directory(. DIR\_SRCS)：查找当前目录下的所有源文件，并将名称保存到 DIR\_SRCS 变量。

add\_executable(Demo ${DIR\_SRCS})：指定生成目标

add\_subdirectory(math)：添加 math 子目录，指明本项目包含一个子目录 math，这样 math 目录下的 CMakeLists.txt 文件和源代码也会被处理。

target\_link\_libraries(Demo MathFunctions)：添加链接库使用命令 target\_link\_libraries 指明可执行文件 main 需要连接一个名为 MathFunctions 的链接库。

add\_library (MathFunctions ${DIR\_LIB\_SRCS})： 将 src 目录中的源文件编译为静态链接库。

**自定义编译选项**

CMake 允许为项目增加编译选项，从而可以根据用户的环境和需求选择最合适的编译方案。

# CMake 最低版本号要求

cmake\_minimum\_required (VERSION 2.8)

# 项目信息

project (Demo4)

# 加入一个配置头文件，用于处理 CMake 对源码的设置

configure\_file (

"${PROJECT\_SOURCE\_DIR}/config.h.in"

"${PROJECT\_BINARY\_DIR}/config.h"

)

# 是否使用自己的 MathFunctions 库

option (USE\_MYMATH

"Use provided math implementation" ON)

# 是否加入 MathFunctions 库

if (USE\_MYMATH)

include\_directories ("${PROJECT\_SOURCE\_DIR}/math")

add\_subdirectory (math)

set (EXTRA\_LIBS ${EXTRA\_LIBS} MathFunctions)

endif (USE\_MYMATH)

# 查找当前目录下的所有源文件

# 并将名称保存到 DIR\_SRCS 变量

aux\_source\_directory(. DIR\_SRCS)

# 指定生成目标

add\_executable(Demo ${DIR\_SRCS})

target\_link\_libraries (Demo ${EXTRA\_LIBS})

configure\_file命令：用于加入一个配置头文件 config.h ，这个文件由 CMake 从 config.h.in 生成，通过这样的机制，将可以通过预定义一些参数和变量来控制代码的生成config.h.in->config.h。#cmakedefine USE\_MYMATH。先写config.h.in，在cmakelist中使用configure命令生成config.h，分别将 USE\_MYMATH 设为 ON 和 OFF 得到的结果，#define USE\_MYMATH，/\* #undef USE\_MYMATH \*/。

option命令：添加了一个 USE\_MYMATH 选项，并且默认值为 ON 。

USE\_MYMATH：变量的值来决定是否使用我们自己编写的 MathFunctions 库。

修改 main.cc 文件，让其根据 USE\_MYMATH 的预定义值来决定是否调用标准库还是 MathFunctions 库。在代码中也可以用这个预定义值。

**安装和测试**

CMake 也可以指定安装规则，以及添加测试。这两个功能分别可以通过在产生 Makefile 后使用 make install 和 make test 来执行。在 CMake 里，这样的工作同样只需要简单的调用几条命令。

定制安装规则

Install命令：

# 指定 MathFunctions 库的安装路径 module目录下cmakelist

install (TARGETS MathFunctions DESTINATION bin)

install (FILES MathFunctions.h DESTINATION include)

# 指定安装路径 根目录下cmakelist

install (TARGETS Demo DESTINATION bin)

install (FILES "${PROJECT\_BINARY\_DIR}/config.h" DESTINATION include)

通过上面的定制，生成的 Demo 文件和 MathFunctions 函数库 libMathFunctions.o 文件将会被复制到 ./bin 中，而 MathFunctions.h 和生成的 config.h 文件则会被复制到 ./include 中。

add\_test：添加测试同样很简单。CMake 提供了一个称为 CTest 的测试工具。我们要做的只是在项目根目录的 CMakeLists 文件中调用一系列的 add\_test 命令。

# 启用测试

enable\_testing()

# 测试程序是否成功运行 5 2 应该是入参

add\_test (test\_run Demo 5 2)

# 测试帮助信息是否可以正常提示

add\_test (test\_usage Demo)

set\_tests\_properties (test\_usage

PROPERTIES PASS\_REGULAR\_EXPRESSION "Usage: .\* base exponent")

# 测试 5 的平方

add\_test (test\_5\_2 Demo 5 2)

set\_tests\_properties (test\_5\_2

PROPERTIES PASS\_REGULAR\_EXPRESSION "is 25")

PASS\_REGULAR\_EXPRESSION 用来测试输出是否包含后面跟着的字符串。

macro ：用宏减少重复行动

# 定义一个宏，用来简化测试工作

macro (do\_test arg1 arg2 result)

add\_test (test\_${arg1}\_${arg2} Demo ${arg1} ${arg2})

set\_tests\_properties (test\_${arg1}\_${arg2}

PROPERTIES PASS\_REGULAR\_EXPRESSION ${result})

endmacro (do\_test)

# 使用该宏进行一系列的数据测试

do\_test (5 2 "is 25")

do\_test (10 5 "is 100000")

do\_test (2 10 "is 1024")

**支持 gdb**

添加编译选项：让 CMake 支持 gdb 的设置也很容易，只需要指定 Debug 模式下开启 -g 选项：

set(CMAKE\_BUILD\_TYPE "Debug")

set(CMAKE\_CXX\_FLAGS\_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")

set(CMAKE\_CXX\_FLAGS\_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")

**添加环境检查**

有时候可能要对系统环境做点检查，例如要使用一个平台相关的特性的时候。在这个例子中，我们检查系统是否自带 pow 函数。如果带有 pow 函数，就使用它；否则使用我们定义的 power 函数。

添加 CheckFunctionExists 宏：首先在顶层 CMakeLists 文件中添加 CheckFunctionExists.cmake 宏，并调用 check\_function\_exists 命令测试链接器是否能够在链接阶段找到 pow 函数。

# 检查系统是否支持 pow 函数

include (${CMAKE\_ROOT}/Modules/CheckFunctionExists.cmake)

check\_function\_exists (pow HAVE\_POW)

将上面这段代码放在 configure\_file 命令前。

**添加版本号**

首先修改顶层 CMakeLists 文件，在 project 命令之后加入如下两行：

set (Demo\_VERSION\_MAJOR 1)

set (Demo\_VERSION\_MINOR 0)

分别指定当前的项目的主版本号和副版本号。

之后，为了在代码中获取版本信息，我们可以修改 config.h.in 文件，添加两个预定义变量：

// the configured options and settings for Tutorial

#define Demo\_VERSION\_MAJOR @Demo\_VERSION\_MAJOR@

#define Demo\_VERSION\_MINOR @Demo\_VERSION\_MINOR@

**生成安装包**

习如何配置生成各种平台上的安装包，包括二进制安装包和源码安装包。为了完成这个任务，我们需要用到 CPack ，它同样也是由 CMake 提供的一个工具，专门用于打包。

首先在顶层的 CMakeLists.txt 文件尾部添加下面几行：

# 构建一个 CPack 安装包

include (InstallRequiredSystemLibraries)

set (CPACK\_RESOURCE\_FILE\_LICENSE

"${CMAKE\_CURRENT\_SOURCE\_DIR}/License.txt")

set (CPACK\_PACKAGE\_VERSION\_MAJOR "${Demo\_VERSION\_MAJOR}")

set (CPACK\_PACKAGE\_VERSION\_MINOR "${Demo\_VERSION\_MINOR}")

include (CPack)

上面的代码做了以下几个工作：

1.导入 InstallRequiredSystemLibraries 模块，以便之后导入 CPack 模块；

2.设置一些 CPack 相关变量，包括版权信息和版本信息，其中版本信息用了上一节定义的版本号；

3.导入 CPack 模块。

接下来的工作是像往常一样构建工程，并执行 cpack 命令。

生成二进制安装包：cpack -C CPackConfig.cmake

生成源码安装包：cpack -C CPackSourceConfig.cmak

### 3.1.7迁移CMake

CMake 可以很轻松地构建出在适合各个平台执行的工程环境。而如果当前的工程环境不是 CMake ，而是基于某个特定的平台，是否可以迁移到 CMake 呢？答案是可能的。下面针对几个常用的平台，列出了它们对应的迁移方案。

autotools

[am2cmake](https://projects.kde.org/projects/kde/kdesdk/kde-dev-scripts/repository/revisions/master/changes/cmake-utils/scripts/am2cmake" \t "https://www.hahack.com/codes/cmake/_blank) 可以将 autotools 系的项目转换到 CMake，这个工具的一个成功案例是 KDE 。

[Alternative Automake2CMake](http://emanuelgreisen.dk/stuff/kdevelop_am2cmake.php.tgz" \t "https://www.hahack.com/codes/cmake/_blank) 可以转换使用 automake 的 KDevelop 工程项目。

[Converting autoconf tests](http://www.cmake.org/Wiki/GccXmlAutoConfHints" \t "https://www.hahack.com/codes/cmake/_blank)

qmake

[qmake converter](http://www.cmake.org/Wiki/CMake:ConvertFromQmake" \t "https://www.hahack.com/codes/cmake/_blank) 可以转换使用 QT 的 qmake 的工程。

Visual Studio

[vcproj2cmake.rb](http://vcproj2cmake.sf.net/" \t "https://www.hahack.com/codes/cmake/_blank) 可以根据 Visual Studio 的工程文件（后缀名是 .vcproj 或 .vcxproj）生成 CMakeLists.txt 文件。

[vcproj2cmake.ps1](http://nberserk.blogspot.com/2010/11/converting-vc-projectsvcproj-to.html" \t "https://www.hahack.com/codes/cmake/_blank) vcproj2cmake 的 PowerShell 版本。

[folders4cmake](http://sourceforge.net/projects/folders4cmake/" \t "https://www.hahack.com/codes/cmake/_blank) 根据 Visual Studio 项目文件生成相应的 “source\_group” 信息，这些信息可以很方便的在 CMake 脚本中使用。支持 Visual Studio 9/10 工程文件。

CMakeLists.txt 自动推导

[gencmake](http://websvn.kde.org/trunk/KDE/kdesdk/cmake/scripts/" \t "https://www.hahack.com/codes/cmake/_blank) 根据现有文件推导 CMakeLists.txt 文件。

[CMakeListGenerator](http://www.vanvelzensoftware.com/postnuke/index.php?name=Downloads&req=viewdownload&cid=7" \t "https://www.hahack.com/codes/cmake/_blank) 应用一套文件和目录分析创建出完整的 CMakeLists.txt 文件。仅支持 Win32 平台。

相关链接

[官方主页](http://www.cmake.org/" \t "https://www.hahack.com/codes/cmake/_blank)

[官方文档](http://www.cmake.org/cmake/help/cmake2.4docs.html" \t "https://www.hahack.com/codes/cmake/_blank)

[官方教程](http://www.cmake.org/cmake/help/cmake_tutorial.html" \t "https://www.hahack.com/codes/cmake/_blank)

[Wiki](http://www.cmake.org/Wiki/CMake" \l "Basic_CMakeLists.txt_from-scratch-generator" \t "https://www.hahack.com/codes/cmake/_blank)

[FAQ](http://www.cmake.org/Wiki/CMake_FAQ" \t "https://www.hahack.com/codes/cmake/_blank)

[bug tracker](http://www.cmake.org/Bug" \t "https://www.hahack.com/codes/cmake/_blank)

邮件列表：

[cmake on Gmane](http://dir.gmane.org/gmane.comp.programming.tools.cmake.user" \t "https://www.hahack.com/codes/cmake/_blank)

[http://www.mail-archive.com/cmake@cmake.org/](http://www.mail-archive.com/cmake@cmake.org/" \t "https://www.hahack.com/codes/cmake/_blank)

[http://marc.info/?l=cmake](http://www.mail-archive.com/cmake@cmake.org/" \t "https://www.hahack.com/codes/cmake/_blank)

其他推荐文章

[在 linux 下使用 CMake 构建应用程序](http://www.ibm.com/developerworks/cn/linux/l-cn-cmake/" \t "https://www.hahack.com/codes/cmake/_blank)

[cmake的一些小经验](http://www.cppblog.com/skyscribe/archive/2009/12/14/103208.aspx" \t "https://www.hahack.com/codes/cmake/_blank)

[Packaging Software with CPack](http://www.kitware.com/media/archive/kitware_quarterly0107.pdf" \t "https://www.hahack.com/codes/cmake/_blank)

[视频教程: 《Getting Started with CMake》](http://www.youtube.com/watch?v=CLvZTyji_Uw" \t "https://www.hahack.com/codes/cmake/_blank)

### 3.1.8参考链接：

1.CMake 入门实战

<https://www.hahack.com/codes/cmake/#>

2.cmake 简介

<https://www.cnblogs.com/lidabo/p/7359422.html>

# C++

## 4.1 c++

部分C++基础在面试中

## 4.2 reactor设计模式

### 4.2.1参考链接

1.Reactor模式详解

<https://www.cnblogs.com/winner-0715/p/8733787.html>

2.Reactor模式

<https://www.jianshu.com/p/eef7ebe28673>

### 4.2.2 reactor介绍

反应器设计模式(Reactor pattern)是一种为处理并发服务请求，并将请求提交到一个或

者多个服务处理程序的事件设计模式。当客户端请求抵达后，服务处理程序使用多路分配策略，由一个非阻塞的线程来接收所有的请求，然后派发这些请求至相关的工作线程进行处理。

### 4.2.3 什么场景下使用Reactor模式？

对于高并发系统，常会使用Reactor模式，其代替了常用的多线程处理方式，节省系统的资源，提高系统的吞吐量。

下面用比较直观的形式来介绍这种模式的使用场景。

以餐厅为例，每一个人就餐就是一个事件，顾客会先看下菜单，然后点餐，处理这些就餐事件需要服务人员。就像一个网络服务会有很多的请求，服务器会收到每个请求，然后指派工作线程去处理一样，在多线程处理方式下：

 一个人来就餐，一个服务员去服务，然后客人会看菜单，点菜。 服务员将菜单给后厨。

二个人来就餐，二个服务员去服务……

五个人来就餐，五个服务员去服务……

这类似多线程的处理方式，一个事件到来，就会有一个线程为其服务。很显然这种方式在人少的情况下会有很好的用户体验，每个客人都感觉自己享有了最好的服务。同一时间会来10个客人，老板很开心，但是只有5个服务员，这样就不能一对一服务了，有些客人就不能马上享有服务员为其服务了。综合考虑了一下，老板决定就使用10个服务人员的线程池！

但是这样又有一个比较严重的缺点：如果正在接受服务员服务的客人点菜很慢，其他的

客人可能就要等好长时间了。有些脾气火爆的客人可能就等不了走人了。

这样，那就引入了Reactor模式。那么，Reactor模式是如何处理这个问题呢？老板后来发现，客人点菜比较慢，大部服务员都在等着客人点菜，其实干的活不是太多。老板之所以能当老板当然有点不一样的地方，终于发现了一个新的方法，那就是：当客人点菜的时候，服务员就可以去招呼其他客人了，等客人点好了菜，直接招呼一声“服务员”，马上就有个服务员过去服务。在用了这个新方法后，老板进行了一次裁员，只留了一个服务员！这就是用单个线程来做多线程的事。实际的餐馆都是用的Reactor模式在服务。

### 4.2.4 reactor处理流程

在处理web请求时，通常有两种体系结构，分别为：thread-based architecture（基于线程）、event-driven architecture（事件驱动）。

actor设计模式是event-driven architecture的一种实现方式，处理多个客户端并发的向服务端请求服务的场景。每种服务在服务端可能由多个方法组成。reactor会解耦并发请求的服务并分发给对应的事件处理器来处理。总体图示如下：



reactor主要由以下几个角色构成：handle、Synchronous Event Demultiplexer、Initiation Dispatcher、Event Handler、Concrete Event Handler。

Handlehandle在linux中一般称为文件描述符，而在window称为句柄，两者的含义一样。handle是事件的发源地。比如一个网络socket、磁盘文件等。而发生在handle上的事件可以有connection、ready for read、ready for write等。

Synchronous Event Demultiplexer同步事件分离器，本质上是系统调用。比如linux中的select、poll、epoll等。比如，select方法会一直阻塞直到handle上有事件发生时才会返回。 Event Handler事件处理器，其会定义一些回调方法或者称为钩子函数，当handle上有事件发生时，回调方法便会执行，一种事件处理机制。

Concrete Event Handler具体的事件处理器，实现了Event Handler。在回调方法中会实现具体的业务逻辑。

Initiation Dispatcher初始分发器，也是reactor角色，提供了注册、删除与转发event handler的方法。当Synchronous Event Demultiplexer检测到handle上有事件发生时，便会通知initiation dispatcher调用特定的event handler的回调方法。

处理流程

1. 当应用向Initiation Dispatcher注册Concrete Event Handler时，应用会标识出该事件处理器希望Initiation Dispatcher在某种类型的事件发生发生时向其通知，事件与handle关联 2. Initiation Dispatcher要求注册在其上面的Concrete Event Handler传递内部关联的handle，该handle会向操作系统标识
2. 当所有的Concrete Event Handler都注册到 Initiation Dispatcher上后，应用会调用handle\_events方法来启动Initiation Dispatcher的事件循环，这时Initiation Dispatcher会将每个Concrete Event Handler关联的handle合并，并使用Synchronous Event Demultiplexer来等待这些handle上事件的发生
3. 当与某个事件源对应的handle变为ready时，Synchronous Event Demultiplexer便会通知 Initiation Dispatcher。比如tcp的socket变为ready for reading
4. Initiation Dispatcher会触发事件处理器的回调方法。当事件发生时， Initiation Dispatcher会将被一个“key”（表示一个激活的handle）定位和分发给特定的Event Handler的回调方法
5. Initiation Dispatcher调用特定的Concrete Event Handler的回调方法来响应其关联的handle上发生的事件

### 4.2.5多线程IO的致命缺陷

多线程并发模式，一个连接一个线程的缺点是：缺点在于资源要求太高，系统中创建线程是需要比较高的系统资源的，如果连接数太高，系统无法承受，而且，线程的反复创建-销毁也需要代价。改进方法是：采用基于事件驱动的设计，当有事件触发时，才会调用处理器进行数据处理。使用Reactor模式，对线程的数量进行控制，一个线程处理大量的事件。

### 4.2.6 单/多线程Reactor

单Reactor单线程模型：Reactor线程是个多面手，负责多路分离套接字，Accept新连接，并分派请求到Handler处理器中。Reactor和Hander 处于一条线程执行。



单线程模式的缺点:

1、 当其中某个 handler 阻塞时， 会导致其他所有的 client 的 handler 都得不到执行， 并且更严重的是， handler 的阻塞也会导致整个服务不能接收新的 client 请求(因为 acceptor 也被阻塞了)。 因为有这么多的缺陷， 因此单线程Reactor 模型用的比较少。这种单线程模型不能充分利用多核资源，所以实际使用的不多。

2、因此，单线程模型仅仅适用于handler 中业务处理组件能快速完成的场景。

多线程的Reactor：在线程Reactor模式基础上，做如下改进：

（1）将Handler处理器的执行放入线程池，多线程进行业务处理。

（2）而对于Reactor而言，可以仍为单个线程。如果服务器为多核的CPU，为充分利用系统资源，可以将Reactor拆分为两个线程。

一个简单的图如下：





### 4.2.7 Reactor模式详解

在学习Reactor模式之前，我们需要对“I/O的四种模型”以及“什么是I/O多路复用”进行简单的介绍，因为Reactor是一个使用了同步非阻塞的I/O多路复用机制的模式。

## 4.3 Proactor模式

### 4.3.1 同步 异步 阻塞 非阻塞

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中Reactor模式用于同步I/O，而Proactor运用于异步I/O操作。

在比较这两个模式之前，我们首先的搞明白几个概念，什么是阻塞和非阻塞，什么是同步和异步,同步和异步是针对应用程序和内核的交互而言的，同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，而异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

一般来说I/O模型可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞IO。

同步阻塞IO：

在此种方式下，用户进程在发起一个IO操作以后，必须等待IO操作的完成，只有当真正完成了IO操作以后，用户进程才能运行。JAVA传统的IO模型属于此种方式！

同步非阻塞IO:

在此种方式下，用户进程发起一个IO操作以后边可返回做其它事情，但是用户进程需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的CPU资源浪费。其中目前JAVA的NIO就属于同步非阻塞IO。

异步阻塞IO：

此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄，从而提高系统的并发性！

异步非阻塞IO:

在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。目前Java中还没有支持此种IO模型。

搞清楚了以上概念以后，我们再回过头来看看，Reactor模式和Proactor模式。首先来看看Reactor模式，Reactor模式应用于同步I/O的场景。我们分别以读操作和写操作为例来看看Reactor中的具体步骤：

读取操作：

1. 应用程序注册读就需事件和相关联的事件处理器

2. 事件分离器等待事件的发生

3. 当发生读就需事件的时候，事件分离器调用第一步注册的事件处理器

4. 事件处理器首先执行实际的读取操作，然后根据读取到的内容进行进一步的处理

写入操作类似于读取操作，只不过第一步注册的是写就绪事件。

下面我们来看看Proactor模式中读取操作和写入操作的过程：

读取操作：

1. 应用程序初始化一个异步读取操作，然后注册相应的事件处理器，此时事件处理器不关注读取就绪事件，而是关注读取完成事件，这是区别于Reactor的关键。

2. 事件分离器等待读取操作完成事件

3. 在事件分离器等待读取操作完成的时候，操作系统调用内核线程完成读取操作，并将读取的内容放入用户传递过来的缓存区中。这也是区别于Reactor的一点，Proactor中，应用程序需要传递缓存区。

4. 事件分离器捕获到读取完成事件后，激活应用程序注册的事件处理器，事件处理器直接从缓存区读取数据，而不需要进行实际的读取操作。

Proactor中写入操作和读取操作，只不过感兴趣的事件是写入完成事件。

从上面可以看出，Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，而Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的IO设备.

综上所述，同步和异步是相对于应用和内核的交互方式而言的，同步需要主动去询问，而异步的时候内核在IO事件发生的时候通知应用程序，而阻塞和非阻塞仅仅是系统在调用系统调用的时候函数的实现方式而已。

说到阻塞,首先得说说I/O等待。I/O等待是不可避免的,那么既然有了等待,就会有阻塞,但是注意,我们说的阻塞是指当前发起I/O操作的进程被阻塞。同步阻塞I/O便是指,当进程调用某些涉及I/O操作的系统调用或库函数时,比如accept()(注意accept也算在了i/o操作)、send()、recv()等,进程便暂停下来,等待I/O操作完成再继续运行。这是一种简单而有效的I/O模型,它可以和多进程结合起来有效的利用CPU资源,但是代价就是多进程的大量内存开销。

比如我们使用非阻塞recv()接收网络数据的时候,如果网卡缓冲区中没有可接收的数据,函数就及时返回,告诉进程没有数据可读了。相比于阻塞I/O,这种非阻塞I/O结合反复的轮询来尝试数据是否就绪,防止进程被阻塞,最大的好处便在于可以在一个进程里同时处理多个I/O操作。但正是由于需要进程执行多次的轮询来查看数据是否就绪,这花费了大量的CPU时间,使得进程处于忙碌等待状态。

非阻塞I/O一般只针对网络I/O有效,我们只要在socket的选项设置中使用O\_NONBLOCK即可,这样对于该socket的send()或recv()便采用非阻塞方式。如果服务器想要同时接收多个TCP连接的数据,就必须轮流对每个socket调用接收数据的方法,比如recv()。不管这些socket有没有可以接收的数据,都要询问一遍,假如大部分socket并没有数据可以接收,那么进程便会浪费很多CPU时间用于检查这些socket,这显然不是我们所希望看到的。

阻塞和非阻塞是指当进程访问的数据如果尚未就绪,进程是否需要等待,简单说这相当于函数内部的实现区别,也就是未就绪时是直接返回还是等待就绪;

而同步和异步是指访问数据的机制,同步一般指主动请求并等待I/O操作完毕的方式,当数据就绪后在读写的时候必须阻塞(区别就绪与读写二个阶段,同步的读写必须阻塞),异步则指主动请求数据后便可以继续处理其它任务,随后等待I/O,操作完毕的通知,这可以使进程在数据读写时也不阻塞。(等待"通知")

### 4.3.2 Reactor和Proactor模式

Reactor模式



Proactor模式



从上面可以看出，Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，而Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的IO设备.





堵塞、非堵塞的区别是在于第一阶段，即数据准备阶段。无论是堵塞还是非堵塞，都是用应用主动找内核要数据，而read数据的过程是‘堵塞’的，直到数据读取完。

同步、异步的区别在于第二阶段，若由请求者主动的去获取数据，则为同步操作，需要说明的是：read/write操作也是‘堵塞’的，直到数据读取完。

若数据的read都由kernel内核完成了(在内核read数据的过程中，应用进程依旧可以执行其他的任务)，这就是异步操作。

I/O多路复用是指使用一个线程来检查多个文件描述符(Socket)的就绪状态，比如调用select和poll函数，传入多个文件描述符，如果有一个文件描述符就绪，则返回，否则阻塞直到超时。得到就绪状态后进行真正的操作可以在同一个线程里执行，也可以启动线程执行(比如使用线程池)。

## 4.4 muduo框架