# 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。

## 3.常用库

#### 3.1 nuppy

#### 3.2 twilio发送短信

<https://www.twilio.com/console>

#### 3.3 matplotlib

绘制图表

#### 3.4 logging

日志

#### 3.5 turtle

绘制图像的函数库

turtle.screensize(canvwidth=None, canvheight=None, bg=None)

参数分别为画布的宽(单位像素), 高, 背景颜色

turtle.setup(width=0.5, height=0.75, startx=None, starty=None)

参数:

width, height: 输入宽和高为整数时, 表示像素; 为小数时, 表示占据电脑屏幕的比例

(startx, starty): 这一坐标表示 矩形窗口左上角顶点的位置, 如果为空,则窗口位于屏幕中心

turtle.pensize() ：设置画笔的宽度；

turtle.pencolor() ：没有参数传入返回当前画笔颜色；传入参数设置画笔颜色,可以是字符串如"green", "red",也可以是RGB 3元组。

turtle.speed(speed) ：设置画笔移动速度,画笔绘制的速度范围[0,10]整数, 数字越大越快

操纵海龟绘图有着许多的命令，这些命令可以划分为3种：运动命令，画笔控制命令和全局控制命令

参考链接：https://blog.csdn.net/ameng001/article/details/81182388

# 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>

### 1.2 kill函数

**#include <sys/types.h>**  
**#include <signal.h>**

**int kill(pid\_t pid, int sig);**

**参数：**  
**pid：可能选择有以下四种**

**1. pid大于零时，pid是信号欲送往的进程的标识。**  
**2. pid等于零时，信号将送往所有与调用kill()的那个进程属同一个使用组的进程。**  
**3. pid等于-1时，信号将送往所有调用进程有权给其发送信号的进程，除了进程1(init)。**  
**4. pid小于-1时，信号将送往以-pid为组标识的进程。**

**sig：准备发送的信号代码，假如其值为零则没有任何信号送出，但是系统会执行错误检查，通常会利用sig值为零来检验某个进程是否仍在执行。**  
**返回值说明： 成功执行时，返回0。失败返回-1，errno被设为以下的某个值 EINVAL：指定的信号码无效（参数 sig 不合法） EPERM；权限不够无法传送信号给指定进程 ESRCH：参数 pid 所指定的进程或进程组不存在**

**sig的值有：**  
**1) SIGHUP**  
**本信号在用户终端连接(正常或非正常)结束时发出, 通常是在终端的控制进程结束时, 通知同一session内的各个作业, 这时它们与控制终端不再关联。**

**登录Linux时，系统会分配给登录用户一个终端(Session)。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于这个 Session。当用户退出Linux登录时，前台进程组和后台有对终端输出的进程将会收到SIGHUP信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。不过可以捕获这个信号，比如wget能捕获SIGHUP信号，并忽略它，这样就算退出了Linux登录，wget也能继续下载。**

**此外，对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件。**

**2) SIGINT**  
**程序终止(interrupt)信号, 在用户键入INTR字符(通常是Ctrl-C)时发出，用于通知前台进程组终止进程。**

**3) SIGQUIT**  
**和SIGINT类似, 但由QUIT字符(通常是Ctrl-\)来控制. 进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号。**

**4) SIGILL**  
**执行了非法指令. 通常是因为可执行文件本身出现错误, 或者试图执行数据段. 堆栈溢出时也有可能产生这个信号。**

**5) SIGTRAP**  
**由断点指令或其它trap指令产生. 由debugger使用。**

**6) SIGABRT**  
**调用abort函数生成的信号。**

**7) SIGBUS**  
**非法地址, 包括内存地址对齐(alignment)出错。比如访问一个四个字长的整数, 但其地址不是4的倍数。它与SIGSEGV的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间)。**

**8) SIGFPE**  
**在发生致命的算术运算错误时发出. 不仅包括浮点运算错误, 还包括溢出及除数为0等其它所有的算术的错误。**

**9) SIGKILL**  
**用来立即结束程序的运行. 本信号不能被阻塞、处理和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号。**

**10) SIGUSR1**  
**留给用户使用**

**11) SIGSEGV**  
**试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据.**

**12) SIGUSR2**  
**留给用户使用**

**13) SIGPIPE**  
**管道破裂。这个信号通常在进程间通信产生，比如采用FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到SIGPIPE信号。此外用Socket通信的两个进程，写进程在写Socket的时候，读进程已经终止。**

**14) SIGALRM**  
**时钟定时信号, 计算的是实际的时间或时钟时间. alarm函数使用该信号.**

**15) SIGTERM**  
**程序结束(terminate)信号, 与SIGKILL不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出，shell命令kill缺省产生这个信号。如果进程终止不了，我们才会尝试SIGKILL。**

**17) SIGCHLD**  
**子进程结束时, 父进程会收到这个信号。**

**如果父进程没有处理这个信号，也没有等待(wait)子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程。这种情况我们应该避免(父进程或者忽略SIGCHILD信号，或者捕捉它，或者wait它派生的子进程，或者父进程先终止，这时子进程的终止自动由init进程来接管)。**

**18) SIGCONT**  
**让一个停止(stopped)的进程继续执行. 本信号不能被阻塞. 可以用一个handler来让程序在由stopped状态变为继续执行时完成特定的工作. 例如, 重新显示提示符**

**19) SIGSTOP**  
**停止(stopped)进程的执行. 注意它和terminate以及interrupt的区别:该进程还未结束, 只是暂停执行. 本信号不能被阻塞, 处理或忽略.**

**20) SIGTSTP**  
**停止进程的运行, 但该信号可以被处理和忽略. 用户键入SUSP字符时(通常是Ctrl-Z)发出这个信号**

**21) SIGTTIN**  
**当后台作业要从用户终端读数据时, 该作业中的所有进程会收到SIGTTIN信号. 缺省时这些进程会停止执行.**

**22) SIGTTOU**  
**类似于SIGTTIN, 但在写终端(或修改终端模式)时收到.**

**23) SIGURG**  
**有”紧急”数据或out-of-band数据到达socket时产生.**

**24) SIGXCPU**  
**超过CPU时间资源限制. 这个限制可以由getrlimit/setrlimit来读取/改变。**

**25) SIGXFSZ**  
**当进程企图扩大文件以至于超过文件大小资源限制。**

**26) SIGVTALRM**  
**虚拟时钟信号. 类似于SIGALRM, 但是计算的是该进程占用的CPU时间.**

**27) SIGPROF**  
**类似于SIGALRM/SIGVTALRM, 但包括该进程用的CPU时间以及系统调用的时间.**

**28) SIGWINCH**  
**窗口大小改变时发出.**

**29) SIGIO**  
**文件描述符准备就绪, 可以开始进行输入/输出操作.**

**30) SIGPWR**  
**Power failure**

**31) SIGSYS**  
**非法的系统调用。**

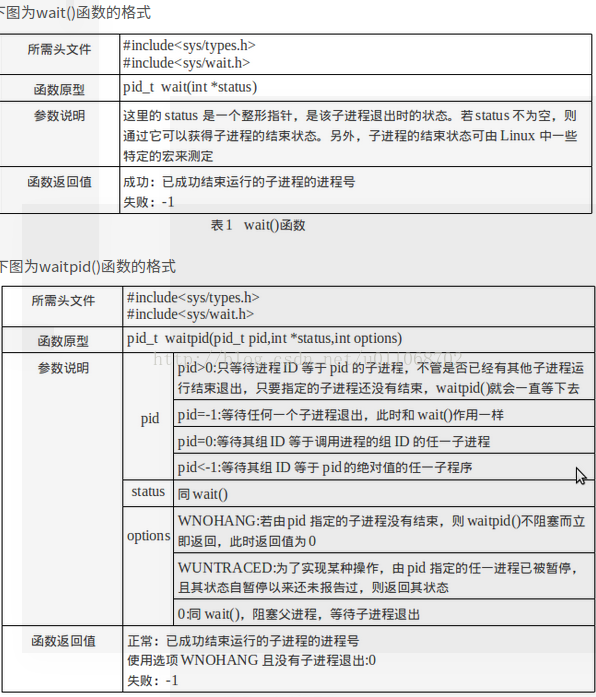
**在以上列出的信号中，程序不可捕获、阻塞或忽略的信号有：SIGKILL,SIGSTOP**  
**不能恢复至默认动作的信号有：SIGILL,SIGTRAP**  
**默认会导致进程流产的信号有：SIGABRT,SIGBUS,SIGFPE,SIGILL,SIGIOT,SIGQUIT,SIGSEGV,SIGTRAP,SIGXCPU,SIGXFSZ**  
**默认会导致进程退出的信号有：SIGALRM,SIGHUP,SIGINT,SIGKILL,SIGPIPE,SIGPOLL,SIGPROF,SIGSYS,SIGTERM,SIGUSR1,SIGUSR2,SIGVTALRM**  
**默认会导致进程停止的信号有：SIGSTOP,SIGTSTP,SIGTTIN,SIGTTOU**  
**默认进程忽略的信号有：SIGCHLD,SIGPWR,SIGURG,SIGWINCH**

### 1.3 waitpid 和 wait函数

1)  wait()函数用于使父进程（也就是调用wait()的进程）阻塞，直到一个子进程结束或者该进程接收到了一个指定的信号为止。如果该父进程没有子进程或者它的子进程已经结束，则wait()函数就会立即返回。

2) waitpid()的作用和wait()一样，但它并不一定要等待第一个终止的子进程（它可以指定需要等待终止的子进程），它还有若干选项，如可提供一个非阻塞版本的 wait()功能，也能支持作业控制。实际上，wait()函数只是 waitpid()函数的一个特例，在Linux 内部实现 wait()函数时直接调用的就是waitpid()函数。

Wati等待父进程中子进程结束或者子进程手动指定信号。



### 1.4 exec( )函数族

在Linux中要使用exec函数族。系统调用execve（）对当前进程进行替换，替换者为一个指定的程序，其參数包含文件名称（filename）、參数列表（argv）以及环境变量（envp）。exec函数族当然不止一个。但它们大致同样，在Linux中，它们各自是：execl，execlp。execle，execv，execve和execvp，以下我仅仅以execlp为例，其他函数到底与execlp有何差别，请通过man exec命令来了解它们的详细情况。 一个进程一旦调用exec类函数，它本身就"死亡"了。系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号。也就是说，对系统而言，还是同一个进程，只是已经是还有一个程序了。（只是exec类函数中有的还同意继承环境变量之类的信息。）

## 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>

### 软件安装

#### LibCurl

1.下载 或者使用sudo apt install curl安装

下载链接：<https://curl.haxx.se/download/>

Github链接：<https://github.com/curl/curl>

2.解压到当前目录（或者 http://www.linuxidc.com/Linux/2014-08/106022.htm）

tar -zxf curl-7.20.0.tar.gz

3.进入解压后的目录内

cd curl-7.17.1

4.配置，指定安装的目录，这里是“/usr/local/curl”

./configure --prefix=/usr/local/curl

Make

Make install

1. 卸载

#### 4.2 libjson

linux下json库的编译及例程链接：<https://blog.csdn.net/renzha0401/article/details/6740205>

linux下json库的编译及例程：<https://www.cnblogs.com/zendu/p/4980312.html>

1、下载JsonCpp

http://sourceforge.net/projects/jsoncpp/files/

2、解压jsconcpp

tar -zvxf jsoncpp-src-0.5.0.tar.gz

我的解压到了/opt/JSON/jsoncpp-src-0.5.

3、进入到jsoncpp解压目录下，执行命令：

scons platform=linux-gcc

4、将/jsoncpp-src-0.5.0/include/目录下的json文件夹拷贝到/usr/include/

5、将jsoncpp-src-0.5.0/libs/Linux-gcc-4.9.1/目录下的libjson\_linux-gcc-4.9.1\_libmt.a 拷贝到/usr/local/lib/下，并为了方便使用，将其重命名为libjson.a

## 4. Linux运维相关

### 4.1 参考连接

1. linux相关

<https://www.linuxprobe.com/chapter-00.html>

1. Linux就该这么学

<https://www.linuxprobe.com/docs/LinuxProbe.pdf>

### 4.2 Linux就该这么学

#### 4.2.1 虚拟机网络连接

桥接模式：相当于在物理主机与虚拟机网卡之间架设了一座桥梁，从而可以通过物理主机的网卡访问外网。

NAT模式：让VM虚拟机的网络服务发挥路由器的作用，使得通过虚拟机软件模拟的主机可以通过物理主机访问外网，在真机中NAT虚拟机网卡对应的物理网卡是VMnet8。

仅主机模式：仅让虚拟机内的主机与物理主机通信，不能访问外网，在真机中仅主机模式模拟网卡对应的物理网卡是VMnet1

#### 4.2.2 常用的RPM软件包命令

|  |  |
| --- | --- |
| 安装软件的命令格式 | rpm -ivh filename.rpm |
| 升级软件的命令格式 | rpm -Uvh filename.rpm |
| 卸载软件的命令格式 | rpm -e filename.rpm |
| 查询软件描述信息的命令格式 | rpm -qpi filename.rpm |
| 列出软件文件信息的命令格式 | rpm -qpl filename.rpm |
| 查询文件属于哪个RPM的命令格式 | rpm -qf filename |

#### 4.2.3 常见的Yum命令

|  |  |
| --- | --- |
| 命令 | 作用 |
| yum repolist all | 列出所有仓库 |
| yum list all | 列出仓库中所有软件包 |
| yum info 软件包名称 | 查看软件包信息 |
| yum install 软件包名称 | 安装软件包 |
| yum reinstall 软件包名称 | 重新安装软件包 |
| yum update 软件包名称 | 升级软件包 |
| yum remove 软件包名称 | 移除软件包 |
| yum clean all | 清除所有仓库缓存 |
| yum check-update | 检查可更新的软件包 |
| yum grouplist | 查看系统中已经安装的软件包组 |
| yum groupinstall 软件包组 | 安装指定的软件包组 |
| yum groupremove 软件包组 | 移除指定的软件包组 |
| yum groupinfo 软件包组 | 查询指定的软件包组信息 |

#### 4.2.4 Systemd初始化进程

Linux操作系统的开机过程是这样的，即从BIOS开始，然后进入Boot Loader，再加载系统内核，然后内核进行初始化，最后启动初始化进程。初始化进程作为Linux系统的第一个进程，它需要完成Linux系统中相关的初始化工作，为用户提供合适的工作环境。

初始化进程服务System V init，正式采用全新的systemd初始化进程服务

systemd与System V init的区别以及作用

|  |  |  |
| --- | --- | --- |
| System V init运行级别 | systemd目标名称 | 作用 |
| 0 | runlevel0.target, poweroff.target | 关机 |
| 1 | runlevel1.target, rescue.target | 单用户模式 |
| 2 | runlevel2.target, multi-user.target | 等同于级别3 |
| 3 | runlevel3.target, multi-user.target | 多用户的文本界面 |
| 4 | runlevel4.target, multi-user.target | 等同于级别3 |
| 5 | runlevel5.target, graphical.target | 多用户的图形界面 |
| 6 | runlevel6.target, reboot.target | 重启 |
| emergency | emergency.target | 紧急Shell |

#### 4.2.5 systemctl管理服务的启动、重启、停止、重载、查看状态等常用命令

设置开机启动

|  |  |  |
| --- | --- | --- |
| System V init命令（RHEL 6系统） | systemctl命令（RHEL 7系统） | 作用 |
| service foo start | systemctl start foo.service | 启动服务 |
| service foo restart | systemctl restart foo.service | 重启服务 |
| service foo stop | systemctl stop foo.service | 停止服务 |
| service foo reload | systemctl reload foo.service | 重新加载配置文件（不终止服务） |
| service foo status | systemctl status foo.service | 查看服务状态 |

systemctl设置服务开机启动、不启动、查看各级别下服务启动状态等常用命令

|  |  |  |
| --- | --- | --- |
| System V init命令（RHEL 6系统） | systemctl命令（RHEL 7系统） | 作用 |
| chkconfig foo on | systemctl enable foo.service | 开机自动启动 |
| chkconfig foo off | systemctl disable foo.service | 开机不自动启动 |
| chkconfig foo | systemctl is-enabled foo.service | 查看特定服务是否为开机自启动 |
| chkconfig --list | systemctl list-unit-files --type=service | 查看各个级别下服务的启动与禁用情况 |

#### 4.2.6 linux常用命令

命令对象一般是指要处理的文件、目录、用户等资源，而命令参数可以用长格式（完整的选项名称），也可以用短格式（单个字母的缩写），两者分别用--与-作为前缀

##### 1 Man:命令自身的帮助信息

|  |  |
| --- | --- |
| 空格键 | 向下翻一页 |
| PaGe down | 向下翻一页 |
| PaGe up | 向上翻一页 |
| home | 直接前往首页 |
| end | 直接前往尾页 |
| / | 从上至下搜索某个关键词，如“/linux” |
| ? | 从下至上搜索某个关键词，如“?linux” |
| n | 定位到下一个搜索到的关键词 |
| N | 定位到上一个搜索到的关键词 |
| q | 退出帮助文档 |

##### echo命令

echo命令用于在终端输出字符串或变量提取后的值，格式为“echo [字符串 | $变量]”。

例如，把指定字符串“Linuxprobe.com”输出到终端屏幕的命令为：

echo Linuxprobe.Com

我们使用$变量的方式提取变量SHELL的值，并将其输出到屏幕上

echo $SHELL

##### date命令

date命令用于显示及设置系统的时间或日期，格式为“date [选项] [+指定的格式]”。

|  |  |
| --- | --- |
| %t | 跳格[Tab键] |
| %H | 小时（00～23） |
| %I | 小时（00～12） |
| %M | 分钟（00～59） |
| %S | 秒（00～59） |
| %j | 今年中的第几天 |

按照“年-月-日 小时:分钟:秒”的格式查看当前系统时间的date命令如下所示：

date "+%Y-%m-%d %H:%M:%S"

将系统的当前时间设置为2017年9月1日8点30分的date命令如下所示

date -s "20170901 8:30:00"

##### 重启

reboot命令用于重启系统，其格式为reboot。由于重启计算机这种操作会涉及硬件资源的管理权限，因此默认只能使用root管理员来重启

poweroff命令用于关闭系统，其格式为poweroff。该命令与reboot命令相同，都会涉及硬件资源的管理权限，因此默认只有root管理员才可以关闭电脑

##### wget命令

wget命令用于在终端中下载网络文件，格式为“wget [参数] 下载地址”。

|  |  |
| --- | --- |
| 参数 | 作用 |
| -b | 后台下载模式 |
| -P | 下载到指定目录 |
| -t | 最大尝试次数 |
| -c | 断点续传 |
| -p | 下载页面内所有资源，包括图片、视频等 |
| -r | 递归下载 |

##### Ps 命令

ps命令用于查看系统中的进程状态，格式为“ps [参数]”。

|  |  |
| --- | --- |
| 参数 | 作用 |
| -a | 显示所有进程（包括其他用户的进程） |
| -u | 用户以及其他详细信息 |
| -x | 显示没有控制终端的进程 |

在Linux系统中，有5种常见的进程状态，分别为运行、中断、不可中断、僵死与停止，其各自含义如下所示。

R（运行）：进程正在运行或在运行队列中等待。

S（中断）：进程处于休眠中，当某个条件形成后或者接收到信号时，则脱离该 状态。

D（不可中断）：进程不响应系统异步信号，即便用kill命令也不能将其中断。

Z（僵死）：进程已经终止，但进程描述符依然存在, 直到父进程调用wait4()系统函数后将进程释放。

T（停止）：进程收到停止信号后停止运行。

##### top命令

top命令用于动态地监视进程活动与系统负载等信息，其格式为top。

top命令相当强大，能够动态地查看系统运维状态，完全将它看作Linux中的“强化版的Windows任务管理器”。

##### 8.pidof命令

pidof命令用于查询某个指定服务进程的PID值，格式为“pidof [参数] [服务名称]”。

##### 9.kill命令

kill命令用于终止某个指定PID的服务进程，格式为“kill [参数] [进程PID]”。

killall命令用于终止某个指定名称的服务所对应的全部进程，格式为：“killall [参数] [服务名称]”。通常来讲，复杂软件的服务程序会有多个进程协同为用户提供服务，如果逐个去结束这些进程会比较麻烦，此时可以使用killall命令来批量结束某个服务程序带有的全部进程。如果我们在系统终端中执行一个命令后想立即停止它，可以同时按下Ctrl + C组合键（生产环境中比较常用的一个快捷键），这样将立即终止该命令的进程。或者，如果有些命令在执行时不断地在屏幕上输出信息，影响到后续命令的输入，则可以在执行命令时在末尾添加上一个&符号，这样命令将进入系统后台来执行。

##### ifconfig命令

ifconfig命令用于获取网卡配置与网络状态等信息，格式为“ifconfig [网络设备] [参数]”。

##### uname命令

uname命令用于查看系统内核与系统版本等信息，格式为“uname [-a]”。在使用uname命令时，一般会固定搭配上-a参数来完整地查看当前系统的内核名称、主机名、内核发行版本、节点名、系统时间、硬件名称、硬件平台、处理器类型以及操作系统名称等信息。

如果要查看当前系统版本的详细信息，则需要查看redhat-release文件，其命令以及相应的结果如下：cat /etc/redhat-release

##### uptime命令

uptime用于查看系统的负载信息，格式为uptime。

uptime命令真的很棒，它可以显示当前系统时间、系统已运行时间、启用终端数量以及平均负载值等信息。平均负载值指的是系统在最近1分钟、5分钟、15分钟内的压力情况（下面加粗的信息部分）；负载值越低越好，尽量不要长期超过1，在生产环境中不要超过5。

##### free命令

free用于显示当前系统中内存的使用量信息，格式为“free [-h]”。

##### who命令

who用于查看当前登入主机的用户终端信息，格式为“who [参数]”。

##### last命令

last命令用于查看所有系统的登录记录，格式为“last [参数]”。

##### history命令

history命令用于显示历史执行过的命令，格式为“history [-c]”。

history命令应该是作者最喜欢的命令。执行history命令能显示出当前用户在本地计算机中执行过的最近1000条命令记录。如果觉得1000不够用，还可以自定义/etc/profile文件中的HISTSIZE变量值。在使用history命令时，如果使用-c参数则会清空所有的命令历史记录。还可以使用“!编码数字”的方式来重复执行某一次的命令。

##### sosreport命令

sosreport命令用于收集系统配置及架构信息并输出诊断文档，格式为sosreport。当Linux系统出现故障需要联系技术支持人员时，大多数时候都要先使用这个命令来简单收集系统的运行状态和服务配置信息，以便让技术支持人员能够远程解决一些小问题，亦或让他们能提前了解某些复杂问题。在下面的输出信息中，加粗的部分是收集好的资料压缩文件以及校验码，将其发送给技术支持人员即可

##### 查看文档

Cat：查看较少得文档

More：查看较多文档

Head查看开头

Tail查看n行

##### 19.tr命令

tr命令用于替换文本文件中的字符，格式为“tr [原始字符] [目标字符]”。

cat anaconda-ks.cfg | tr [a-z] [A-Z]

##### wc命令

wc命令用于统计指定文本的行数、字数、字节数，格式为“wc [参数] 文本”。

##### stat命令

stat命令用于查看文件的具体存储信息和时间等信息，格式为“stat 文件名称”。stat命令可以用于查看文件的存储信息和时间等信息，

##### cut命令

cut命令用于按“列”提取文本字符，格式为“cut [参数] 文本”。

##### diff命令

diff命令用于比较多个文本文件的差异，格式为“diff [参数] 文件”。

在使用diff命令时，不仅可以使用--brief参数来确认两个文件是否不同，还可以使用-c参数来详细比较出多个文件的差异之处，这绝对是判断文件是否被篡改的有力神器。例如，先使用cat命令分别查看diff\_A.txt和diff\_B.txt文件的内容，然后进行比较.

##### Touch mkdir cp mv rm file命令

touch命令用于创建空白文件或设置文件的时间，格式为“touch [选项] [文件]”。

mkdir命令用于创建空白的目录，格式为“mkdir [选项] 目录”。

mv命令用于剪切文件或将文件重命名，格式为“mv [选项] 源文件 [目标路径|目标文件名]”。

cp命令用于复制文件或目录，格式为“cp [选项] 源文件 目标文件”。

m命令用于删除文件或目录，格式为“rm [选项] 文件”。

file命令用于查看文件的类型，格式为“file 文件名”。

##### tar命令

tar命令用于对文件进行打包压缩或解压，格式为“tar [选项] [文件]”。

|  |  |
| --- | --- |
| 参数 | 作用 |
| -c | 创建压缩文件 |
| -x | 解开压缩文件 |
| -t | 查看压缩包内有哪些文件 |
| -z | 用Gzip压缩或解压 |
| -j | 用bzip2压缩或解压 |
| -v | 显示压缩或解压的过程 |
| -f | 目标文件名 |
| -p | 保留原始的权限与属性 |
| -P | 使用绝对路径来压缩 |
| -C | 指定解压到的目录 |

##### grep命令

grep命令用于在文本中执行关键词搜索，并显示匹配的结果，格式为“grep [选项] [文件]”。

|  |  |
| --- | --- |
| 参数 | 作用 |
| -b | 将可执行文件(binary)当作文本文件（text）来搜索 |
| -c | 仅显示找到的行数 |
| -i | 忽略大小写 |
| -n | 显示行号 |
| -v | 反向选择——仅列出没有“关键词”的行。 |

##### find命令

find命令用于按照指定条件来查找文件，格式为“find [查找路径] 寻找条件 操作”。

|  |  |
| --- | --- |
| 参数 | 作用 |
| -name | 匹配名称 |
| -perm | 匹配权限（mode为完全匹配，-mode为包含即可） |
| -user | 匹配所有者 |
| -group | 匹配所有组 |
| -mtime -n +n | 匹配修改内容的时间（-n指n天以内，+n指n天以前） |
| -atime -n +n | 匹配访问文件的时间（-n指n天以内，+n指n天以前） |
| -ctime -n +n | 匹配修改文件权限的时间（-n指n天以内，+n指n天以前） |
| -nouser | 匹配无所有者的文件 |
| -nogroup | 匹配无所有组的文件 |
| -newer f1 !f2 | 匹配比文件f1新但比f2旧的文件 |
| --type b/d/c/p/l/f | 匹配文件类型（后面的字幕字母依次表示块设备、目录、字符设备、管道、链接文件、文本文件） |
| -size | 匹配文件的大小（+50KB为查找超过50KB的文件，而-50KB为查找小于50KB的文件） |
| -prune | 忽略某个目录 |
| -exec …… {}\; | 后面可跟用于进一步处理搜索结果的命令（下文会有演示） |

#### 4.2.7 重定向 管道符 通配符

输入重定向是指把文件导入到命令中，而输出重定向则是指把原本要输出到屏幕的数据信息写入到指定文件中。

标准输入重定向（STDIN，文件描述符为0）：默认从键盘输入，也可从其他文件或命令中输入。

标准输出重定向（STDOUT，文件描述符为1）：默认输出到屏幕。

错误输出重定向（STDERR，文件描述符为2）：默认输出到屏幕。

|  |  |
| --- | --- |
| 命令 < 文件 | 将文件作为命令的标准输入 |
| 命令 << 分界符 | 从标准输入中读入，直到遇见分界符才停止 |
| 命令 < 文件1 > 文件2 | 将文件1作为命令的标准输入并将标准输出到文件2 |

|  |  |
| --- | --- |
| 命令 > 文件 | 将标准输出重定向到一个文件中（清空原有文件的数据） |
| 命令 2> 文件 | 将错误输出重定向到一个文件中（清空原有文件的数据） |
| 命令 >> 文件 | 将标准输出重定向到一个文件中（追加到原有内容的后面） |
| 命令 2>> 文件 | 将错误输出重定向到一个文件中（追加到原有内容的后面） |
| 命令 >> 文件 2>&1  或 命令 &>> 文件 | 将标准输出与错误输出共同写入到文件中（追加到原有内容的后面） |

星号（\*）代表匹配零个或多个字符，问号（?）代表匹配单个字符，中括号内加上数字[0-9]代表匹配0～9之间的单个数字的字符，而中括号内加上字母[abc]则是代表匹配a、b、c三个字符中的任意一个字符。

反斜杠（\）：使反斜杠后面的一个变量变为单纯的字符串。

单引号（''）：转义其中所有的变量为单纯的字符串。

双引号（""）：保留其中的变量属性，不进行转义处理。

反引号（``）：把其中的命令执行后返回结果。

CMake表3-3                                       Linux系统中最重要的10个环境变量

|  |  |
| --- | --- |
| 变量名称 | 作用 |
| HOME | 用户的主目录（即家目录） |
| SHELL | 用户在使用的Shell解释器名称 |
| HISTSIZE | 输出的历史命令记录条数 |
| HISTFILESIZE | 保存的历史命令记录条数 |
| MAIL | 邮件保存路径 |
| LANG | 系统语言、语系名称 |
| RANDOM | 生成一个随机数字 |
| PS1 | Bash解释器的提示符 |
| PATH | 定义解释器搜索用户执行命令的路径 |
| EDITOR | 用户默认的文本编辑器 |

### 4.3 Vim编辑和shell命令

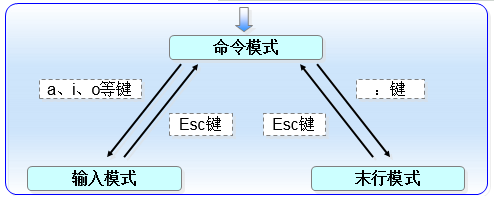
#### 4.3.1 vim

Vim编辑器中设置了三种模式—命令模式、末行模式和编辑模式。

命令模式：控制光标移动，可对文本进行复制、粘贴、删除和查找等工作。

输入模式：正常的文本录入。

末行模式：保存或退出文档，以及设置编辑环境。



命令模式

|  |  |
| --- | --- |
| 命令 | 作用 |
| dd | 删除（剪切）光标所在整行 |
| 5dd | 删除（剪切）从光标处开始的5行 |
| yy | 复制光标所在整行 |
| 5yy | 复制从光标处开始的5行 |
| n | 显示搜索命令定位到的下一个字符串 |
| N | 显示搜索命令定位到的上一个字符串 |
| u | 撤销上一步的操作 |
| p | 将之前删除（dd）或复制（yy）过的数据粘贴到光标后面 |

模式模式

|  |  |
| --- | --- |
| :w | 保存 |
| :q | 退出 |
| :q! | 强制退出（放弃对文档的修改内容） |
| :wq! | 强制保存退出 |
| :set nu | 显示行号 |
| :set nonu | 不显示行号 |
| :命令 | 执行该命令 |
| :整数 | 跳转到该行 |
| :s/one/two | 将当前光标所在行的第一个one替换成two |
| :s/one/two/g | 将当前光标所在行的所有one替换成two |
| :%s/one/two/g | 将全文中的所有one替换成two |
| ?字符串 | 在文本中从下至上搜索该字符串 |
| /字符串 | 在文本中从上至下搜索该字符串 |

打开文件到某一行 vim +3 xx.txt

打开到某一个字符 vim +/asd xx.txt

#### 4.3.2 shell

Shell脚本命令的工作方式有两种：交互式和批处理。

交互式（Interactive）：用户每输入一条命令就立即执行。

批处理（Batch）：由用户事先编写好一个完整的Shell脚本，Shell会一次性执行脚本中诸多的命令。

Linux系统中的Shell脚本语言早就考虑到了这些，已经内设了用于接收参数的变量，变量之间可以使用空格间隔。例如$0对应的是当前Shell脚本程序的名称，$#对应的是总共有几个参数，$\*对应的是所有位置的参数值，$?对应的是显示上一次命令的执行返回值，而$1、$2、$3……则分别对应着第N个位置的参数值。

切记，条件表达式两边均应有一个空格。

按照测试对象来划分，条件测试语句可以分为4种：

文件测试语句；

逻辑测试语句；

整数值比较语句；

字符串比较语句。

|  |  |
| --- | --- |
| -d | 测试文件是否为目录类型 |
| -e | 测试文件是否存在 |
| -f | 判断是否为一般文件 |
| -r | 测试当前用户是否有权限读取 |
| -w | 测试当前用户是否有权限写入 |
| -x | 测试当前用户是否有权限执行 |

下面使用文件测试语句来判断/etc/fstab是否为一个目录类型的文件，然后通过Shell解释器的内设$?变量显示上一条命令执行后的返回值。如果返回值为0，则目录存在；如果返回值为非零的值，则意味着目录不存在：

[root@linuxprobe ~]# [ -d /etc/fstab ]

[root@linuxprobe ~]# echo $?

1

|  |  |
| --- | --- |
| -eq | 是否等于 |
| -ne | 是否不等于 |
| -gt | 是否大于 |
| -lt | 是否小于 |
| -le | 是否等于或小于 |
| -ge | 是否大于或等于 |

具体if 等语句https://www.linuxprobe.com/chapter-04.html

### 4.4 自动化任务

计划任务服务程序

at 命令

[root@linuxprobe ~]# at 23:30

at > systemctl restart httpd

at > 此处请同时按下Ctrl+d来结束编写计划任务

job 3 at Mon Apr 27 23:30:00 2015

[root@linuxprobe ~]# at -l

3 Mon Apr 27 23:30:00 2016 a root

如果我们希望Linux系统能够周期性地、有规律地执行某些具体的任务，那么Linux系统中默认启用的crond服务简直再适合不过了。创建、编辑计划任务的命令为“crontab -e”，查看当前计划任务的命令为“crontab -l”，删除某条计划任务的命令为“crontab -r”。另外，如果您是以管理员的身份登录的系统，还可以在crontab命令中加上-u参数来编辑他人的计划任务。

### 4.5 用户身份和文件权限

Linux系统的管理员之所以是root，并不是因为它的名字叫root，而是因为该用户的身份号码即UID（User IDentification）的数值为0。在Linux系统中，UID就相当于我们的身份证号码一样具有唯一性，因此可通过用户的UID值来判断用户身份。

管理员UID为0：系统的管理员用户。

系统用户UID为1～999： Linux系统为了避免因某个服务程序出现漏洞而被黑客提权至整台服务器，默认服务程序会有独立的系统用户负责运行，进而有效控制被破坏范围。

普通用户UID从1000开始：是由管理员创建的用于日常工作的用户。

需要注意的是，UID是不能冲突的，而且管理员创建的普通用户的UID默认是从1000开始的（即使前面有闲置的号码）。

为了方便管理属于同一组的用户，Linux系统中还引入了用户组的概念。通过使用用户组号码（GID，Group IDentification），我们可以把多个用户加入到同一个组中，从而方便为组中的用户统一规划权限或指定任务。

在Linux系统中创建每个用户时，将自动创建一个与其同名的基本用户组，而且这个基本用户组只有该用户一个人。如果该用户以后被归纳入其他用户组，则这个其他用户组称之为扩展用户组。一个用户只有一个基本用户组，但是可以有多个扩展用户组，从而满足日常的工作需要。

#### 4.5.1 useradd命令

useradd命令用于创建新的用户，格式为“useradd [选项] 用户名”。

可以使用useradd命令创建用户账户。使用该命令创建用户账户时，默认的用户家目录会被存放在/home目录中，默认的Shell解释器为/bin/bash，而且默认会创建一个与该用户同名的基本用户组。这些默认设置可以根据表5-1中的useradd命令参数自行修改。

|  |  |
| --- | --- |
| -d | 指定用户的家目录（默认为/home/username） |
| -e | 账户的到期时间，格式为YYYY-MM-DD. |
| -u | 指定该用户的默认UID |
| -g | 指定一个初始的用户基本组（必须已存在） |
| -G | 指定一个或多个扩展用户组 |
| -N | 不创建与用户同名的基本用户组 |
| -s | 指定该用户的默认Shell解释器 |

[root@linuxprobe ~]# useradd -d /home/linux -u 8888 -s /sbin/nologin linuxprobe

[root@linuxprobe ~]# id linuxprobe

uid=8888(linuxprobe) gid=8888(linuxprobe) groups=8888(linuxprobe)

#### 4.5.2 groupadd命令

groupadd命令用于创建用户组，格式为“groupadd [选项] 群组名”。

#### 4.5.3 usermod命令

usermod命令用于修改用户的属性，格式为“usermod [选项] 用户名”。Linux系统中的一切都是文件，因此在系统中创建用户也就是修改配置文件的过程。用户的信息保存在/etc/passwd文件中，可以直接用文本编辑器来修改其中的用户参数项目，也可以用usermod命令修改已经创建的用户信息，诸如用户的UID、基本/扩展用户组、默认终端等。

|  |  |
| --- | --- |
| -c | 填写用户账户的备注信息 |
| -d -m | 参数-m与参数-d连用，可重新指定用户的家目录并自动把旧的数据转移过去 |
| -e | 账户的到期时间，格式为YYYY-MM-DD |
| -g | 变更所属用户组 |
| -G | 变更扩展用户组 |
| -L | 锁定用户禁止其登录系统 |
| -U | 解锁用户，允许其登录系统 |
| -s | 变更默认终端 |
| -u | 修改用户的UID |

#### 4.5.4 passwd命令

passwd命令用于修改用户密码、过期时间、认证信息等，格式为“passwd [选项] [用户名]”。普通用户只能使用passwd命令修改自身的系统密码，而root管理员则有权限修改其他所有人的密码。更酷的是，root管理员在Linux系统中修改自己或他人的密码时不需要验证旧密码，这一点特别方便。既然root管理员可以修改其他用户的密码，就表示完全拥有该用户的管理权限。

|  |  |
| --- | --- |
| 参数 | 作用 |
| -l | 锁定用户，禁止其登录 |
| -u | 解除锁定，允许用户登录 |
| --stdin | 允许通过标准输入修改用户密码，如echo "NewPassWord" | passwd --stdin Username |
| -d | 使该用户可用空密码登录系统 |
| -e | 强制用户在下次登录时修改密码 |
| -S | 显示用户的密码是否被锁定，以及密码所采用的加密算法名称 |

#### 4.5.5 userdel命令

userdel命令用于删除用户，格式为“userdel [选项] 用户名”。如果我们确认某位用户后续不再会登录到系统中，则可以通过userdel命令删除该用户的所有信息。在执行删除操作时，该用户的家目录默认会保留下来，此时可以使用-r参数将其删除。

|  |  |
| --- | --- |
| -f | 强制删除用户 |
| -r | 同时删除用户及用户家目录 |

#### 4.5.6 文件权限与归属

尽管在Linux系统中一切都是文件，但是每个文件的类型不尽相同，因此Linux系统使用了不同的字符来加以区分，常见的字符如下所示。

-：普通文件。

d：目录文件。

l：链接文件。

b：块设备文件。

c：字符设备文件。

p：管道文件。

SUID、SGID与SBIT的特殊权限位

SUID是一种对二进制程序进行设置的特殊权限，可以让二进制程序的执行者临时拥有属主的权限（仅对拥有执行权限的二进制程序有效）。例如，所有用户都可以执行passwd命令来修改自己的用户密码，而用户密码保存在/etc/shadow文件中。仔细查看这个文件就会发现它的默认权限是000，也就是说除了root管理员以外，所有用户都没有查看或编辑该文件的权限。但是，在使用passwd命令时如果加上SUID特殊权限位，就可让普通用户临时获得程序所有者的身份，把变更的密码信息写入到shadow文件中。这很像我们在古装剧中见到的手持尚方宝剑的钦差大臣，他手持的尚方宝剑代表的是皇上的权威，因此可以惩戒贪官，但这并不意味着他永久成为了皇上。因此这只是一种有条件的、临时的特殊权限授权方法。查看passwd命令属性时发现所有者的权限由rwx变成了rws，其中x改变成s就意味着该文件被赋予了SUID权限。另外有读者会好奇，那么如果原本的权限是rw-呢？如果原先权限位上没有x执行权限，那么被赋予特殊权限后将变成大写的S。

SGID主要实现如下两种功能：

让执行者临时拥有属组的权限（对拥有执行权限的二进制程序进行设置）；

在某个目录中创建的文件自动继承该目录的用户组（只可以对目录进行设置）。SGID的第一种功能是参考SUID而设计的，不同点在于执行程序的用户获取的不再是文件所有者的临时权限，而是获取到文件所属组的权限.除了上面提到的SGID的这两个功能，我们再介绍两个与本小节内容相关的命令：chmod和chown。

现在，大学里的很多老师都要求学生将作业上传到服务器的特定共享目录中，但总是有几个“破坏分子”喜欢删除其他同学的作业，这时就要设置SBIT（Sticky Bit）特殊权限位了（也可以称之为特殊权限位之粘滞位）。SBIT特殊权限位可确保用户只能删除自己的文件，而不能删除其他用户的文件。换句话说，当对某个目录设置了SBIT粘滞位权限后，那么该目录中的文件就只能被其所有者执行删除操作了。当然，要是也想对其他目录来设置SBIT特殊权限位，用chmod命令就可以了。对应的参数o+t代表设置SBIT粘滞位权限：

chattr命令用于设置文件的隐藏权限，格式为“chattr [参数] 文件”。如果想要把某个隐藏功能添加到文件上，则需要在命令后面追加“+参数”，如果想要把某个隐藏功能移出文件，则需要追加“-参数”。

|  |  |
| --- | --- |
| i | 无法对文件进行修改；若对目录设置了该参数，则仅能修改其中的子文件内容而不能新建或删除文件 |
| a | 仅允许补充（追加）内容，无法覆盖/删除内容（Append Only） |
| S | 文件内容在变更后立即同步到硬盘（sync） |
| s | 彻底从硬盘中删除，不可恢复（用0填充原文件所在硬盘区域） |
| A | 不再修改这个文件或目录的最后访问时间（atime） |
| b | 不再修改文件或目录的存取时间 |
| D | 检查压缩文件中的错误 |
| d | 使用dump命令备份时忽略本文件/目录 |
| c | 默认将文件或目录进行压缩 |
| u | 当删除该文件后依然保留其在硬盘中的数据，方便日后恢复 |
| t | 让文件系统支持尾部合并（tail-merging） |
| x | 可以直接访问压缩文件中的内容 |

lsattr命令用于显示文件的隐藏权限，格式为“lsattr [参数] 文件”。

如果希望对某个指定的用户进行单独的权限控制，就需要用到文件的访问控制列表（ACL）了。通俗来讲，基于普通文件或目录设置ACL其实就是针对指定的用户或用户组设置文件或目录的操作权限。另外，如果针对某个目录设置了ACL，则目录中的文件会继承其ACL；若针对文件设置了ACL，则文件不再继承其所在目录的ACL。

setfacl命令用于管理文件的ACL规则，格式为“setfacl [参数] 文件名称”。文件的ACL提供的是在所有者、所属组、其他人的读/写/执行权限之外的特殊权限控制，使用setfacl命令可以针对单一用户或用户组、单一文件或目录来进行读/写/执行权限的控制。其中，针对目录文件需要使用-R递归参数；针对普通文件则使用-m参数；如果想要删除某个文件的ACL，则可以使用-b参数。下面来设置用户在/root目录上的权限：

[root@linuxprobe ~]# setfacl -Rm u:linuxprobe:rwx /root

[root@linuxprobe ~]# su - linuxprobe

Last login: Sat Mar 21 15:45:03 CST 2017 on pts/1

[linuxprobe@linuxprobe ~]$ cd /root

[linuxprobe@linuxprobe root]$ ls

anaconda-ks.cfg Downloads Pictures Public

[linuxprobe@linuxprobe root]$ cat anaconda-ks.cfg

[linuxprobe@linuxprobe root]$ exit

getfacl命令用于显示文件上设置的ACL信息，格式为“getfacl 文件名称”。Linux系统中的命令就是这么又可爱又好记。想要设置ACL，用的是setfacl命令；要想查看ACL，则用的是getfacl命令。

sudo命令具有如下功能：

限制用户执行指定的命令：

记录用户执行的每一条命令；

配置文件（/etc/sudoers）提供集中的用户管理、权限与主机等参数；

验证密码的后5分钟内（默认值）无须再让用户再次验证密码。

|  |  |
| --- | --- |
| -h | 列出帮助信息 |
| -l | 列出当前用户可执行的命令 |
| -u 用户名或UID值 | 以指定的用户身份执行命令 |
| -k | 清空密码的有效时间，下次执行sudo时需要再次进行密码验证 |
| -b | 在后台执行指定的命令 |
| -p | 更改询问密码的提示语 |

### 4.6 存储和磁盘结构

|  |  |
| --- | --- |
| /boot | 开机所需文件—内核、开机菜单以及所需配置文件等 |
| /dev | 以文件形式存放任何设备与接口 |
| /etc | 配置文件 |
| /home | 用户主目录 |
| /bin | 存放单用户模式下还可以操作的[命令](https://www.linuxcool.com/" \o "命令" \t "https://www.linuxprobe.com/_blank) |
| /lib | 开机时用到的函数库，以及/bin与/sbin下面的命令要调用的函数 |
| /sbin | 开机过程中需要的命令 |
| /media | 用于挂载设备文件的目录 |
| /opt | 放置第三方的软件 |
| /root | 系统管理员的家目录 |
| /srv | 一些网络服务的数据文件目录 |
| /tmp | 任何人均可使用的“共享”临时目录 |
| /proc | 虚拟文件系统，例如系统内核、进程、外部设备及网络状态等 |
| /usr/local | 用户自行安装的软件 |
| /usr/sbin | Linux系统开机时不会使用到的软件/命令/[脚本](https://www.linuxcool.com/" \o "脚本" \t "https://www.linuxprobe.com/_blank) |
| /usr/share | 帮助与说明文件，也可放置共享文件 |
| /var | 主要存放经常变化的文件，如日志 |
| /lost+found | 当文件系统发生错误时，将一些丢失的文件片段存放在这里 |

|  |  |
| --- | --- |
| 硬件设备 | 文件名称 |
| IDE设备 | /dev/hd[a-d] |
| SCSI/SATA/U盘 | /dev/sd[a-p] |
| 软驱 | /dev/fd[0-1] |
| 打印机 | /dev/lp[0-15] |
| 光驱 | /dev/cdrom |
| 鼠标 | /dev/mouse |
| 磁带机 | /dev/st0或/dev/ht0 |

主分区或扩展分区的编号从1开始，到4结束；

逻辑分区从编号5开始。



硬盘设备是由大量的扇区组成的，每个扇区的容量为512字节。其中第一个扇区最重要，它里面保存着主引导记录与分区表信息。就第一个扇区来讲，主引导记录需要占用446字节，分区表为64字节，结束符占用2字节；其中分区表中每记录一个分区信息就需要16字节，这样一来最多只有4个分区信息可以写到第一个扇区中，这4个分区就是4个主分区。

Ext3：是一款日志文件系统，能够在系统异常宕机时避免文件系统资料丢失，并能自动修复数据的不一致与错误。然而，当硬盘容量较大时，所需的修复时间也会很长，而且也不能百分之百地保证资料不会丢失。它会把整个磁盘的每个写入动作的细节都预先记录下来，以便在发生异常宕机后能回溯追踪到被中断的部分，然后尝试进行修复。

Ext4：Ext3的改进版本，作为RHEL 6系统中的默认文件管理系统，它支持的存储容量高达1EB（1EB=1,073,741,824GB），且能够有无限多的子目录。另外，Ext4文件系统能够批量分配block块，从而极大地提高了读写效率。

XFS：是一种高性能的日志文件系统，而且是RHEL 7中默认的文件管理系统，它的优势在发生意外宕机后尤其明显，即可以快速地恢复可能被破坏的文件，而且强大的日志功能只用花费极低的计算和存储性能。并且它最大可支持的存储容量为18EB，这几乎满足了所有需求。

mount命令用于挂载文件系统，格式为“mount 文件系统 挂载目录”。只需使用mount命令把硬盘设备或分区与一个目录文件进行关联，然后就能在这个目录中看到硬件设备中的数据了。对于比较新的Linux系统来讲，一般不需要使用-t参数来指定文件系统的类型，Linux系统会自动进行判断。而mount 中的-a参数则厉害了，它会在执行后自动检查/etc/fstab文件中有无疏漏被挂载的设备文件，如果有，则进行自动挂载操作。

|  |  |
| --- | --- |
| -a | 挂载所有在/etc/fstab中定义的文件系统 |
| -t | 指定文件系统的类型 |

虽然按照上面的方法执行mount命令后就能立即使用文件系统了，但系统在重启后挂载就会失效，也就是说我们需要每次开机后都手动挂载一下。这肯定不是我们想要的效果，如果想让硬件设备和目录永久地进行自动关联，就必须把挂载信息按照指定的填写格式“设备文件 挂载目录 格式类型 权限选项 是否备份 是否自检”（各字段的意义见表6-4）写入到/etc/fstab文件中。这个文件中包含着挂载所需的诸多信息项目，一旦配置好之后就能一劳永逸了。

|  |  |
| --- | --- |
| 设备文件 | 一般为设备的路径+设备名称，也可以写唯一识别码（UUID，Universally Unique Identifier） |
| 挂载目录 | 指定要挂载到的目录，需在挂载前创建好 |
| 格式类型 | 指定文件系统的格式，比如Ext3、Ext4、XFS、SWAP、iso9660（此为光盘设备）等 |
| 权限选项 | 若设置为defaults，则默认权限为：rw, suid, dev, exec, auto, nouser, async |
| 是否备份 | 若为1则开机后使用dump进行磁盘备份，为0则不备份 |
| 是否自检 | 若为1则开机后自动进行磁盘自检，为0则不自检 |

如果想将文件系统为ext4的硬件设备/dev/sdb2在开机后自动挂载到/backup目录上，并保持默认权限且无需开机自检，就需要在/etc/fstab文件中写入下面的信息，这样在系统重启后也会成功挂载。

umount命令用于撤销已经挂载的设备文件，格式为“umount [挂载点/设备文件]”。

在Linux系统中，管理硬盘设备最常用的方法就当属fdisk命令了。fdisk命令用于管理磁盘分区，格式为“fdisk [磁盘名称]”，它提供了集添加、删除、转换分区等功能于一身的“一站式分区服务”。

|  |  |
| --- | --- |
| m | 查看全部可用的参数 |
| n | 添加新的分区 |
| d | 删除某个分区信息 |
| l | 列出所有可用的分区类型 |
| t | 改变某个分区的类型 |
| p | 查看分区表信息 |
| w | 保存并退出 |
| q | 不保存直接退出 |

如果硬件存储设备没有进行格式化，则Linux系统无法得知怎么在其上写入数据。因此，在对存储设备进行分区后还需要进行格式化操作。在Linux系统中用于格式化操作的命令是mkfs。这条命令很有意思，因为在Shell终端中输入mkfs名后再敲击两下用于补齐命令的Tab键，会有如下所示的效果。

查看文件数据占用量的du命令，其格式为“du [选项] [文件]”。简单来说，该命令就是用来查看一个或多个文件占用了多大的硬盘空间。我们还可以使用du -sh /\*命令来查看在Linux系统根目录下所有一级目录分别占用的空间大小。

SWAP（交换）分区是一种通过在硬盘中预先划分一定的空间，然后将把内存中暂时不常用的数据临时存放到硬盘中，以便腾出物理内存空间让更活跃的程序服务来使用的技术，其设计目的是为了解决真实物理内存不足的问题。但由于交换分区毕竟是通过硬盘设备读写数据的，速度肯定要比物理内存慢，所以只有当真实的物理内存耗尽后才会调用交换分区的资源。

使用SWAP分区专用的格式化命令mkswap，对新建的主分区进行格式化操作。

针对这种情况，root管理员就需要使用磁盘容量配额服务来限制某位用户或某个用户组针对特定文件夹可以使用的最大硬盘空间或最大文件个数，一旦达到这个最大值就不再允许继续使用。可以使用quota命令进行磁盘容量配额管理，从而限制用户的硬盘可用容量或所能创建的最大文件个数。quota命令还有软限制和硬限制的功能。

xfs\_quota命令是一个专门针对XFS文件系统来管理quota磁盘容量配额服务而设计的命令，格式为“xfs\_quota [参数] 配额 文件系统”。

edquota命令用于编辑用户的quota配额限制，格式为“edquota [参数] [用户] ”。在为用户设置了quota磁盘容量配额限制后，可以使用edquota命令按需修改限额的数值。其中，-u参数表示要针对哪个用户进行设置；-g参数表示要针对哪个用户组进行设置。

在Linux系统中存在硬链接和软连接两种文件。

硬链接（hard link）：可以将它理解为一个“指向原始文件inode的指针”，系统不为它分配独立的inode和文件。所以，硬链接文件与原始文件其实是同一个文件，只是名字不同。我们每添加一个硬链接，该文件的inode连接数就会增加1；而且只有当该文件的inode连接数为0时，才算彻底将它删除。换言之，由于硬链接实际上是指向原文件inode的指针，因此即便原始文件被删除，依然可以通过硬链接文件来访问。需要注意的是，由于技术的局限性，我们不能跨分区对目录文件进行链接。

软链接（也称为符号链接[symbolic link]）：仅仅包含所链接文件的路径名，因此能链接目录文件，也可以跨越文件系统进行链接。但是，当原始文件被删除后，链接文件也将失效，从这一点上来说与Windows系统中的“快捷方式”具有一样的性质。

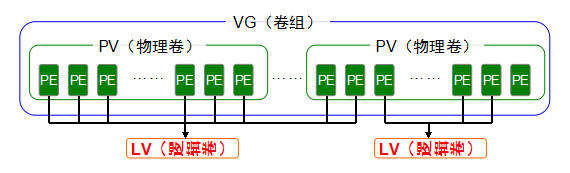
ln命令用于创建链接文件，格式为“ln [选项] 目标”，其可用的参数以及作用如表6-6所示。在使用ln命令时，是否添加-s参数，将创建出性质不同的两种“快捷方式”。

|  |  |
| --- | --- |
| -s | 创建“符号链接”（如果不带-s参数，则默认创建硬链接） |
| -f | 强制创建文件或目录的链接 |
| -i | 覆盖前先询问 |
| -v | 显示创建链接的过程 |

### 4.7 使用RAID与LVM磁盘阵列技术

RAID技术通过把多个硬盘设备组合成一个容量更大、安全性更好的磁盘阵列，并把数据切割成多个区段后分别存放在各个不同的物理硬盘设备上，然后利用分散读写技术来提升磁盘阵列整体的性能，同时把多个重要数据的副本同步到不同的物理硬盘设备上，从而起到了非常好的数据冗余备份效果。mdadm命令用于管理Linux系统中的软件RAID硬盘阵列，格式为“mdadm [模式] <RAID设备名称> [选项] [成员设备名称]”。

逻辑卷管理器是Linux系统用于对硬盘分区进行管理的一种机制，理论性较强，其创建初衷是为了解决硬盘设备在创建分区后不易修改分区大小的缺陷。尽管对传统的硬盘分区进行强制扩容或缩容从理论上来讲是可行的，但是却可能造成数据的丢失。而LVM技术是在硬盘分区和文件系统之间添加了一个逻辑层，它提供了一个抽象的卷组，可以把多块硬盘进行卷组合并。这样一来，用户不必关心物理硬盘设备的底层架构和布局，就可以实现对硬盘分区的动态调整。LVM的技术架构如图7-7所示。



为了帮助大家理解，刘遄老师来举一个吃货的例子。比如小明家里想吃馒头但是面粉不够了，于是妈妈从隔壁老王家、老李家、老张家分别借来一些面粉，准备蒸馒头吃。首先需要把这些面粉（物理卷[PV，Physical Volume]）揉成一个大面团（卷组[VG，Volume Group]），然后再把这个大面团分割成一个个小馒头（逻辑卷[LV，Logical Volume]），而且每个小馒头的重量必须是每勺面粉（基本单元[PE，Physical Extent]）的倍数。

表7-2                                                   常用的LVM部署命令

|  |  |  |  |
| --- | --- | --- | --- |
| 功能/命令 | 物理卷管理 | 卷组管理 | 逻辑卷管理 |
| 扫描 | pvscan | vgscan | lvscan |
| 建立 | pvcreate | vgcreate | lvcreate |
| 显示 | pvdisplay | vgdisplay | lvdisplay |
| 删除 | pvremove | vgremove | lvremove |
| 扩展 |  | vgextend | lvextend |
| 缩小 |  | vgreduce | lvreduce |

### 4.8 防火墙

iptables与firewalld都不是真正的防火墙，它们都只是用来定义防火墙策略的防火墙管理工具而已，或者说，它们只是一种服务。iptables服务会把配置好的防火墙策略交由内核层面的netfilter网络过滤器来处理，而firewalld服务则是把配置好的防火墙策略交由内核层面的nftables包过滤框架来处理。换句话说，当前在Linux系统中其实存在多个防火墙管理工具，旨在方便运维人员管理Linux系统中的防火墙策略，我们只需要配置妥当其中的一个就足够了。

iptables服务把用于处理或过滤流量的策略条目称之为规则，多条规则可以组成一个规则链，而规则链则依据数据包处理位置的不同进行分类，具体如下：

在进行路由选择前处理数据包（PREROUTING）；

处理流入的数据包（INPUT）；

处理流出的数据包（OUTPUT）；

处理转发的数据包（FORWARD）；

在进行路由选择后处理数据包（POSTROUTING）。

这些动作对应到iptables服务的术语中分别是ACCEPT（允许流量通过）、REJECT（拒绝流量通过）、LOG（记录日志信息）、DROP（拒绝流量通过）。“允许流量通过”和“记录日志信息”都比较好理解，这里需要着重讲解的是REJECT和DROP的不同点。就DROP来说，它是直接将流量丢弃而且不响应；REJECT则会在拒绝流量后再回复一条“您的信息已经收到，但是被扔掉了”信息，从而让流量发送方清晰地看到数据被拒绝的响应信息。

iptables中常用的参数以及作用

|  |  |
| --- | --- |
| 参数 | 作用 |
| -P | 设置默认策略 |
| -F | 清空规则链 |
| -L | 查看规则链 |
| -A | 在规则链的末尾加入新规则 |
| -I num | 在规则链的头部加入新规则 |
| -D num | 删除某一条规则 |
| -s | 匹配来源地址IP/MASK，加叹号“!”表示除这个IP外 |
| -d | 匹配目标地址 |
| -i 网卡名称 | 匹配从这块网卡流入的数据 |
| -o 网卡名称 | 匹配从这块网卡流出的数据 |
| -p | 匹配协议，如TCP、UDP、ICMP |
| --dport num | 匹配目标端口号 |
| --sport num | 匹配来源端口号 |

把INPUT规则链的默认策略设置为拒绝：

iptables -P INPUT DROP

向INPUT规则链中添加拒绝所有主机访问本机1000～1024端口的策略规则：

iptables -A INPUT -p tcp --dport 1000:1024 -j REJECT

firewalld中常用的区域名称及策略规则

|  |  |
| --- | --- |
| trusted | 允许所有的数据包 |
| home | 拒绝流入的流量，除非与流出的流量相关；而如果流量与ssh、mdns、ipp-client、amba-client与dhcpv6-client服务相关，则允许流量 |
| internal | 等同于home区域 |
| work | 拒绝流入的流量，除非与流出的流量相关；而如果流量与ssh、ipp-client与dhcpv6-client服务相关，则允许流量 |
| public | 拒绝流入的流量，除非与流出的流量相关；而如果流量与ssh、dhcpv6-client服务相关，则允许流量 |
| external | 拒绝流入的流量，除非与流出的流量相关；而如果流量与ssh服务相关，则允许流量 |
| dmz | 拒绝流入的流量，除非与流出的流量相关；而如果流量与ssh服务相关，则允许流量 |
| block | 拒绝流入的流量，除非与流出的流量相关 |
| drop | 拒绝流入的流量，除非与流出的流量相关 |

firewall-cmd命令中使用的参数以及作用

|  |  |
| --- | --- |
| 参数 | 作用 |
| --get-default-zone | 查询默认的区域名称 |
| --set-default-zone=<区域名称> | 设置默认的区域，使其永久生效 |
| --get-zones | 显示可用的区域 |
| --get-services | 显示预先定义的服务 |
| --get-active-zones | 显示当前正在使用的区域与网卡名称 |
| --add-source= | 将源自此IP或子网的流量导向指定的区域 |
| --remove-source= | 不再将源自此IP或子网的流量导向某个指定区域 |
| --add-interface=<网卡名称> | 将源自该网卡的所有流量都导向某个指定区域 |
| --change-interface=<网卡名称> | 将某个网卡与区域进行关联 |
| --list-all | 显示当前区域的网卡配置参数、资源、端口以及服务等信息 |
| --list-all-zones | 显示所有区域的网卡配置参数、资源、端口以及服务等信息 |
| --add-service=<服务名> | 设置默认区域允许该服务的流量 |
| --add-port=<端口号/协议> | 设置默认区域允许该端口的流量 |
| --remove-service=<服务名> | 设置默认区域不再允许该服务的流量 |
| --remove-port=<端口号/协议> | 设置默认区域不再允许该端口的流量 |
| --reload | 让“永久生效”的配置规则立即生效，并覆盖当前的配置规则 |
| --panic-on | 开启应急状况模式 |
| --panic-off | 关闭应急状况模式 |

TCP Wrappers服务的防火墙策略由两个控制列表文件所控制，用户可以编辑允许控制列表文件来放行对服务的请求流量，也可以编辑拒绝控制列表文件来阻止对服务的请求流量。控制列表文件修改后会立即生效，系统将会先检查允许控制列表文件（/etc/hosts.allow），如果匹配到相应的允许策略则放行流量；如果没有匹配，则去进一步匹配拒绝控制列表文件（/etc/hosts.deny），若找到匹配项则拒绝该流量。如果这两个文件全都没有匹配到，则默认放行流量。

### 4.9 linux管理

RHEL和CentOS系统默认使用NetworkManager来提供网络服务，这是一种动态管理网络配置的守护进程，能够让网络设备保持连接状态。可以使用nmcli命令来管理Network Manager服务。nmcli是一款基于命令行的网络配置工具，功能丰富，参数众多。它可以轻松地查看网络信息或网络状态。

SSH（Secure Shell）是一种能够以安全的方式提供远程登录的协议，也是目前远程管理Linux系统的首选方式。在此之前，一般使用FTP或Telnet来进行远程登录。但是因为它们以明文的形式在网络中传输账户密码和数据信息，因此很不安全，很容易受到黑客发起的中间人攻击，这轻则篡改传输的数据信息，重则直接抓取服务器的账户密码。想要使用SSH协议来远程管理Linux系统，则需要部署配置sshd服务程序。sshd是基于SSH协议开发的一款远程管理服务程序，不仅使用起来方便快捷，而且能够提供两种安全验证的方法。sshd服务的配置信息保存在/etc/ssh/sshd\_config文件中。

|  |  |
| --- | --- |
| Port 22 | 默认的sshd服务端口 |
| ListenAddress 0.0.0.0 | 设定sshd服务器监听的IP地址 |
| Protocol 2 | SSH协议的版本号 |
| HostKey /tc/ssh/ssh\_host\_key | SSH协议版本为1时，DES私钥存放的位置 |
| HostKey /etc/ssh/ssh\_host\_rsa\_key | SSH协议版本为2时，RSA私钥存放的位置 |
| HostKey /etc/ssh/ssh\_host\_dsa\_key | SSH协议版本为2时，DSA私钥存放的位置 |
| PermitRootLogin yes | 设定是否允许root管理员直接登录 |
| StrictModes yes | 当远程用户的私钥改变时直接拒绝连接 |
| MaxAuthTries 6 | 最大密码尝试次数 |
| MaxSessions 10 | 最大终端数 |
| PasswordAuthentication yes | 是否允许密码验证 |
| PermitEmptyPasswords no | 是否允许空密码登录（很不安全） |

一般的服务程序并不会在配置文件修改之后立即获得最新的参数。如果想让新配置文件生效，则需要手动重启相应的服务程序。最好也将这个服务程序加入到开机启动项中，这样系统在下一次启动时，该服务程序便会自动运行，继续为用户提供服务。

[root@linuxprobe ~]# systemctl restart sshd

[root@linuxprobe ~]# systemctl enable sshd

安全密钥验证，ssh-keygen生成密钥对。scp（secure copy）是一个基于SSH协议在网络之间进行安全传输的命令，其格式为“scp [参数] 本地文件 远程帐户@远程IP地址:远程目录”。

Selinux会对资源进行限制。getsebool

FTP是一种在互联网中进行文件传输的协议，基于客户端/服务器模式，默认使用20、21号端口，其中端口20（数据端口）用于进行数据传输，端口21（[命令](https://www.linuxcool.com/" \o "命令" \t "https://www.linuxprobe.com/_blank)端口）用于接受客户端发出的相关FTP[命令](https://www.linuxcool.com/" \o "命令" \t "https://www.linuxprobe.com/_blank)与参数。vsftpd（very secure ftp daemon，非常安全的FTP守护进程）是一款运行在Linux操作系统上的FTP服务程序，不仅完全开源而且免费，此外，还具有很高的安全性、传输速度，以及支持虚拟用户验证等其他FTP服务程序不具备的特点。

dhcpd是Linux系统中用于提供DHCP协议的服务程序。查看dhcpd服务程序的配置文件内容。/etc/dhcp/dhcpd.conf。

[root@linuxprobe ~]# systemctl start dhcpd

[root@linuxprobe ~]# systemctl enable dhcpd

Systemctl restart network

systemctl restart dhcpd

Squid是Linux系统中最为流行的一款高性能代理服务软件，通常用作Web网站的前置缓存服务，能够代替用户向网站服务器请求页面数据并进行缓存。简单来说，Squid服务程序会按照收到的用户请求向网站源服务器请求页面、图片等所需的数据，并将服务器返回的数据存储在运行Squid服务程序的服务器上。当有用户再请求相同的数据时，则可以直接将存储服务器本地的数据交付给用户，这样不仅减少了用户的等待时间，还缓解了网站服务器的负载压力。

Squid服务程序具有配置简单、效率高、功能丰富等特点，它能支持HTTP、FTP、SSL等多种协议的数据缓存，可以基于访问控制列表（ACL）和访问权限列表（ARL）执行内容过滤与权限管理功能，还可以基于多种条件禁止用户访问存在威胁或不适宜的网站资源，因此可以保护企业内网的安全，提升用户的网络体验，帮助节省网络带宽。

iSCSI技术实现了物理硬盘设备与TCP/IP网络协议的相互结合，使得用户可以通过互联网方便地访问远程机房提供的共享存储资源。

IDE是一种成熟稳定、价格便宜的并行传输接口。

SATA是一种传输速度更快、数据校验更完整的串行传输接口。

SCSI是一种用于计算机和硬盘、光驱等设备之间系统级接口的通用标准，具有系统资源占用率低、转速高、传输速度快等优点。

# 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()

### 3.1.4 cmake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

### 3.1.5 cmake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。

**add\_definitions(-DENABLE\_HLS)：**添加宏，设ENABLE\_HLS可以在代码中使用。

**find\_package(OpenSSL QUIET)**：find\_package()命令首先会在模块路径中寻找Find.cmake，这是查找库的一个典型方式。具体查找路径依次为CMake：变量${CMAKE\_MODULE\_PATH}中的所有目录。如果没有，然后再查看它自己的模块目录/share/cmake-x.y/Modules/（$CMAKE\_ROOT的具体值可以通过CMake中message命令输出）。这称为模块模式。为了能支持各种常见的库和包，CMake自带了很多模块。可以通过命令 cmake –help-module-list （输入cmake –help，然后双击Tab会有命令提示）得到你的CMake支持的模块的列表： 直接查看模块路径。比如Ubuntu linux上，模块的路径是 ls /usr/share/cmake/Modules/：

**list(APPEND <list><element> [<element> ...])**

LENGTH　　　　 　　　　 返回list的长度

GET　　　　　　 　　　　返回list中index的element到value中

APPEND　　　　 　　　　添加新element到list中

FIND　　　　　　 　　　　返回list中element的index，没有找到返回-1

INSERT 　　　　　　　　　 将新element插入到list中index的位置

REMOVE\_ITEM　　　　　　从list中删除某个element

REMOVE\_AT　　　　　　　从list中删除指定index的element

REMOVE\_DUPLICATES 从list中删除重复的element

REVERSE 　　　　　　　　将list的内容反转

SORT 　　　　　　　　　　将list按字母顺序排序

LIST与SET命令类似，即使列表本身是在父域中定义的，LIST命令也只会在当前域创建新的变量，要想将这些操作的结果向上传递，需要通过SET　PARENT\_SCOPE， SET CACHE INTERNAL或运用其他值域扩展的方法。

### 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

#### 4.1.1 回调函数

1.定义函数指针：typedef void (\*callback\_t)(void \*);

2.void repeat\_three\_times(callback\_t f, void \*para)

{

f(para);

}

repeat\_three\_times(count\_numbers, (void \*)8);

给函数指针传参数和调用方式

3.// 定义结构体数组（函数表）并初始化

typedef struct CmdEntry

{

PFUNCMD pfuncmd; // 定义函数指针，用于接收函数的入口地址

char cHelp[HELP\_LEN];

}CmdEntry;

static CmdEntry cmdArray[TABLE\_LEN] =

{

{CreateFile,"CreateFile HELP"}, // 取CreatFile()函数地址，帮助信息

{OpenFile,"OpenFile HELP"}, // 取OpenFile()函数地址，帮助信息

{&SaveFile,"SaveFile HELP"}, // 取SaveFile()函数地址，帮助信息

{0,0} // 退出

};

可以给cmdArray数组添加一个结构体cmdArray[i] = cmdentry;

1. 可以利用多线程传入函数指针

5. A\_SetTestCb(B\_UserTestCbDetail);

A\_CallTestCb();

首先A模块里面有全局函数指针，然后模块A对外函提供给模块B，模块A内部定义的全局callback变量与外部模块注册的回调函数绑定起来,用于后续模块A,满足触发条件后,触发回调。A\_CallTestCb()调用触发回调函数。

6.结构体赋值函数值方法

A.callback = a\_callback;

B.callback = &b\_callback;

给结构体数组赋值结构体

struct operation ops[2];

ops[0] = A;

ops[1] = B;

1. 不用typedef函数指针

int call2(int(\*ptr)(),char \*p)

int call1(CallBackFun pCallBack,char \*p)

typedef int (\*CallBackFun)(char \*p);

两个等价

1. 函数传入常值

func((struct argv){"hello ZiMianLiang",24});

int func(struct argv a)

struct argv

{

char a[64];

int b;

};

## 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

## 4.5 STL相关使用介绍

组成

容器（containers）

算法（algorithms）

迭代器（iterators）

仿函数（functors）

配接器（adapters）

空间配置器（allocator）

### 4.5.1 vector（数组）

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。特性如下：

1.顺序序列

顺序容器中的元素按照严格的线性顺序排序。可以通过元素在序列中的位置访问对应的元素。

2.动态数组

支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算述进行该操作。操供了在序列末尾相对快速地添加/删除元素的操作。

3.能够感知内存分配器的（Allocator-aware）

容器使用一个内存分配器对象来动态地处理它的存储需求。

**基本函数实现**

**构造函数**

vector():创建一个空vector

vector(int nSize):创建一个vector,元素个数为nSize

vector(int nSize,const t& t):创建一个vector，元素个数为nSize,且值均为t

vector(const vector&):复制构造函数

vector(begin,end):复制[begin,end)区间内另一个数组的元素到vector中

**增加函数**

void push\_back(const T& x):向量尾部增加一个元素X

iterator insert(iterator it,const T& x):向量中迭代器指向元素前增加一个元素x

iterator insert(iterator it,int n,const T& x):向量中迭代器指向元素前增加n个相同的元素x

iterator insert(iterator it,const\_iterator first,const\_iterator last):向量中迭代器指向元素前插入另一个相同类型向量的[first,last)间的数据

**删除函数**

iterator erase(iterator it):删除向量中迭代器指向元素

iterator erase(iterator first,iterator last):删除向量中[first,last)中元素

void pop\_back():删除向量中最后一个元素

void clear():清空向量中所有元素

**遍历函数**

reference at(int pos):返回pos位置元素的引用

reference front():返回首元素的引用

reference back():返回尾元素的引用

iterator begin():返回向量头指针，指向第一个元素

iterator end():返回向量尾指针，指向向量最后一个元素的下一个位置

reverse\_iterator rbegin():反向迭代器，指向最后一个元素

reverse\_iterator rend():反向迭代器，指向第一个元素之前的位置

**判断函数**

bool empty() const:判断向量是否为空，若为空，则向量中无元素

**大小函数**

int size() const:返回向量中元素的个数

int capacity() const:返回当前向量张红所能容纳的最大元素值

int max\_size() const:返回最大可允许的vector元素数量值

**其他函数**

void swap(vector&):交换两个同类型向量的数据

void assign(int n,const T& x):设置向量中第n个元素的值为x

void assign(const\_iterator first,const\_iterator last):向量中[first,last)中元素设置成当前向量元素

**汇总**

*1.push\_back 在数组的最后添加一个数据*

*2.pop\_back 去掉数组的最后一个数据*

*3.at 得到编号位置的数据*

*4.begin 得到数组头的指针*

*5.end 得到数组的最后一个单元+1的指针*

*6.front 得到数组头的引用*

*7.back 得到数组的最后一个单元的引用*

*8.max\_size 得到vector最大可以是多大*

*9.capacity 当前vector分配的大小*

*10.size 当前使用数据的大小*

*11.resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值*

*12.reserve 改变当前vecotr所分配空间的大小*

*13.erase 删除指针指向的数据项*

*14.clear 清空当前的vector*

*15.rbegin 将vector反转后的开始指针返回(其实就是原来的end-1)*

*16.rend 将vector反转构的结束指针返回(其实就是原来的begin-1)*

*17.empty 判断vector是否为空*

*18.swap 与另一个vector交换数据*

### 4.5.2 array（数组）

array也位于名称空间std中,与数组一样,array对象的长度也是固定的,也使用栈(静态内存分配),而不是自由存储区,因此其效率与数组相同,但更方便,更安全。

**创建**

array<int, 5> myarray = {2, 16, 77,34, 50};创建

**访问**

array::begin：返回指向数组容器中第一个元素的迭代器。

array::end：返回指向数组容器中最后一个元素之后的理论元素的迭代器。

array::rbegin：返回指向数组容器中最后一个元素的反向迭代器。

array::rend：返回一个反向迭代器，指向数组中第一个元素之前的理论元素（这被认为是它的反向结束）。

array::cbegin：返回指向数组容器中第一个元素的常量迭代器（const\_iterator）；这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

array::cend：返回指向数组容器中最后一个元素之后的理论元素的常量迭代器（const\_iterator）。这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

array::crbegin：返回指向数组容器中最后一个元素的常量反向迭代器（const\_reverse\_iterator）

array::crend：返回指向数组中第一个元素之前的理论元素的常量反向迭代器（const\_reverse\_iterator），它被认为是其反向结束。

**操作函数**

array::size：返回数组容器中元素的数量。

array::max\_size：返回数组容器可容纳的最大元素数。数组对象的max\_size与其size一样，始终等于用于实例化数组模板类的第二个模板参数。

array::empty：返回一个布尔值，指示数组容器是否为空，即它的size()是否为0。

array::operator[]：返回数组中第n个位置的元素的引用。与array::at相似，但array::at会检查数组边界并通过抛出一个out\_of\_range异常来判断n是否超出范围，而array::operator[]不检查边界。使用Array[1]访问。

array::at：返回数组中第n个位置的元素的引用。与array::operator[]相似，但array::at会检查数组边界并通过抛出一个out\_of\_range异常来判断n是否超出范围，而array::operator[]不检查边界。

array::front：返回对数组容器中第一个元素的引用。array::begin返回的是迭代器，array::front返回的是直接引用。在空容器上调用此函数会导致未定义的行为。

array::back：返回对数组容器中最后一个元素的引用。array::end返回的是迭代器，array::back返回的是直接引用。在空容器上调用此函数会导致未定义的行为。

array::data：返回指向数组对象中第一个元素的指针。由于数组中的元素存储在连续的存储位置，所以检索到的指针可以偏移以访问数组中的任何元素。

array::fill：用val填充数组所有元素，将val设置为数组对象中所有元素的值。

array::swap：通过x的内容交换数组的内容，这是另一个相同类型的数组对象（包括相同的大小）。与其他容器的交换成员函数不同，此成员函数通过在各个元素之间执行与其大小相同的单独交换操作，以线性时间运行。

get（array）：形如：std::get<0>(myarray)；传入一个数组容器，返回指定位置元素的引用。

### 4.5.3 deque（队列）

双端队列，底层是一个双向链表。常用的有队列的尾部入队、首部出队。

普通队列：queue

queue 模板类的定义在<queue>头文件中。与stack 模板类很相似，queue 模板类也需要两个模板参数，一个是元素类型，一个容器类型，元素类型是必要的，容器类型是可选的，默认为deque 类型。

定义queue 对象的示例代码如下：

queue<int> q1;

queue<double> q2;

queue 的基本操作有：

入队，如例：q.push(x); 将x 接到队列的末端。

出队，如例：q.pop(); 弹出队列的第一个元素，注意，并不会返回被弹出元素的值。

访问队首元素，如例：q.front()，即最早被压入队列的元素。

访问队尾元素，如例：q.back()，即最后被压入队列的元素。

判断队列空，如例：q.empty()，当队列空时，返回true。

访问队列中的元素个数，如例：q.size()

**1.构造**

无参构造：

deque<T> a;　　//<>内自定义数据类型；

带参构造：

deque(beg,end); //构造函数将[beg, end)区间中的元素拷贝给本身。注意该区间是左闭右开的区间。

deque(n,elem); //构造函数将n个elem拷贝给本身。

deque(const deque &deq); //拷贝构造函数。

**2.头部&尾部的添加和删除**

deque.push\_back(elem); //在容器尾部添加一个数据

deque.push\_front(elem); //在容器头部插入一个数据

deque.pop\_back(); //删除容器最后一个数据

deque.pop\_front(); //删除容器第一个数据

**3.中间数据存取**

deque.at(idx); //返回索引idx所指的数据，如果idx越界，抛出out\_of\_range。

deque[idx]; //返回索引idx所指的数据，如果idx越界，不抛出异常，直接出错。

deque.front(); //返回第一个数据。

deque.back(); //返回最后一个数据

**4.元素插入**

deque.insert(pos,elem); //在pos位置插入一个elem元素的拷贝，返回新数据的位置。

deque.insert(pos,n,elem); //在pos位置插入n个elem数据，无返回值。

deque.insert(pos,beg,end); //在pos位置插入[beg,end)区间的数据，无返回值。

**5.数据删除**

deque.clear(); //移除容器的所有数据

deque.erase(beg,end); //删除[beg,end)区间的数据，返回下一个数据的位置。

deque.erase(pos); //删除pos位置的数据，返回下一个数据的位置。

**6.迭代指针**

deque.begin(); //返回容器中第一个元素的迭代器。

deque.end(); //返回容器中最后一个元素之后的迭代器。

deque.rbegin(); //返回容器中倒数第一个元素的迭代器。

deque.rend(); //返回容器中倒数最后一个元素之后的迭代器。

deque<int>::iterator it = dequeA.begin();

**7.赋值&拷贝**

deque.assign(beg,end); //将[beg, end)区间中的数据拷贝赋值给本身。注意该区间是左闭右开的区间。

deque.assign(n,elem); //将n个elem拷贝赋值给本身。

deque& operator=(const deque &deq); //重载等号操作符

deque.swap(deq); // 将vec与本身的元素互换

**8.大小&判断非空**

复制代码

deque.size(); //返回容器中元素的个数

deque.empty(); //判断容器是否为空

deque.resize(num); //重新指定容器的长度为num，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

deque.resize(num, elem); //重新指定容器的长度为num，若容器变长，则以elem值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。

### 4.5.4 list（双向链表）

list 由双向链表（doubly linked list）实现而成，元素也存放在堆中，每个元素都是放在一块内存中，他的内存空间可以是不连续的，通过指针来进行数据的访问，这个特点使得它的随机存取变得非常没有效率，因此它没有提供 [] 操作符的重载。但是由于链表的特点，它可以很有效率的支持任意地方的插入和删除操作。

**1.构造**

list<int> a; // 定义一个int类型的列表a

list<int> a(10); // 定义一个int类型的列表a，并设置初始大小为10

list<int> a(10, 1); // 定义一个int类型的列表a，并设置初始大小为10且初始值都为1

list<int> b(a); // 定义并用列表a初始化列表b

deque<int> b(a.begin(), ++a.end()); // 将列表a中的第1个元素作为列表b的初始值

1. **容量函数**

容器大小：lst.size();

容器最大容量：lst.max\_size();

更改容器大小：lst.resize();

容器判空：lst.empty();

1. **添加函数&删除函数**

头部添加元素：lst.push\_front(const T& x);

末尾添加元素：lst.push\_back(const T& x);

任意位置插入一个元素：lst.insert(iterator it, const T& x);

任意位置插入 n 个相同元素：lst.insert(iterator it, int n, const T& x);

插入另一个向量的 [forst,last] 间的数据：lst.insert(iterator it, iterator first, iterator last);

头部删除元素：lst.pop\_front();

末尾删除元素：lst.pop\_back();

任意位置删除一个元素：lst.erase(iterator it);

删除 [first,last] 之间的元素：lst.erase(iterator first, iterator last);

清空所有元素：lst.clear();

1. **访问函数**

访问第一个元素：lst.front();

访问最后一个元素：lst.back();

1. **其他函数**

多个元素赋值：lst.assign(int nSize, const T& x); // 类似于初始化时用数组进行赋值

交换两个同类型容器的元素：swap(list&, list&); 或 lst.swap(list&);

合并两个列表的元素（默认升序排列）：lst.merge();

在任意位置拼接入另一个list：lst.splice(iterator it, list&);

删除容器中相邻的重复元素：lst.unique();

remove\_if()函数括号中可以传入，回调函数，创建用于比较的类，传入类名及初始化参数

Revese()list翻转

1. **迭代器**

开始迭代器指针：lst.begin();

末尾迭代器指针：lst.end(); // 指向最后一个元素的下一个位置

指向常量的开始迭代器指针：lst.cbegin(); // 意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。

指向常量的末尾迭代器指针：lst.cend();

反向迭代器指针，指向最后一个元素：lst.rbegin();

反向迭代器指针，指向第一个元素的前一个元素：lst.rend();

可以看到，list 与 vector、deque 的用法基本一致，除了以下几处不同：

list 为双向迭代器，故不支持it+=i；

list 不支持下标访问和at方法访问

### 4.5.5 forward\_list（单向链表）

forward\_list（单向链表）是序列容器，允许在序列中的任何地方进行恒定的时间插入和擦除操作。forward\_list（单向链表）被实现为单链表; 单链表可以将它们包含的每个元素存储在不同和不相关的存储位置中。通过关联到序列中下一个元素的链接的每个元素来保留排序。forward\_list容器和列表之间的主要设计区别容器是第一个内部只保留一个到下一个元素的链接，而后者每个元素保留两个链接：一个指向下一个元素，一个指向前一个元素，允许在两个方向上有效的迭代，但是每个元素消耗额外的存储空间并且插入和移除元件的时间开销略高。因此，forward\_list对象比列表对象更有效率，尽管它们只能向前迭代。

与其他基本的标准序列容器（array，vector和deque），forward\_list通常在插入，提取和移动容器内任何位置的元素方面效果更好，因此也适用于密集使用这些元素的算法，如排序算法。



list的特点

不支持随机访问；

在任何位置插入或删除非常迅速；

forward\_list的特点

forward\_list只提供前向迭代器，因此不支持反向迭代器，比如rbegin()等成员函数。

forward\_list不提供size（）成员函数。

forward\_list没有指向最末元素的锚点，因此不提供back（）、push\_back（）和pop\_back

forward\_list不提供随机访问，这一点跟list相同。

插入和删除元素不会造成“指向至其他元素”的指针，引用和迭代器失效。

### 4.5.6 [stack](https://github.com/huihut/interview/tree/master/STL" \l "stack)（堆栈）

stack（堆栈） 是一个容器类的改编，为程序员提供了堆栈的全部功能，——也就是说实现了一个先进后出（FILO）的数据结构。特性总结: 栈不能遍历,不支持随机存取，只能通过 top 从栈顶获取和删除元素.

c++ stl栈stack的成员函数介绍

操作 比较和分配堆栈

stack 构造函数

stack<T> stkT;//stack 采用模板类实现， stack 对象的默认构造形式：

stack(const stack &stk);//拷贝构造函数

stack 赋值操作

stack& operator=(const stack &stk);//重载等号操作符

stack 数据存取操作

push(elem);//向栈顶添加元素

pop();//从栈顶移除第一个元素

top();//返回栈顶元素

stack 大小操作

empty();//判断堆栈是否为空

size();//返回堆栈的大小

### 4.5.7 [queue](https://github.com/huihut/interview/tree/master/STL" \l "queue)(队列)

queue翻译为队列，在STL中主要则是实现了一个先进先出的容器。

1.初始化

需要头文件<queue>

queue<int>que;

2.成员函数

C++队列Queue类成员函数如下:

back()返回最后一个元素

empty()如果队列空则返回真

front()返回第一个元素

pop()删除第一个元素

push()在末尾加入一个元素

size()返回队列中元素的个数

3.queue 的基本操作举例如下：

queue入队，如例：q.push(x); 将x 接到队列的末端。

queue出队，如例：q.pop(); 弹出队列的第一个元素，注意，并不会返回被弹出元素的值。

访问queue队首元素，如例：q.front()，即最早被压入队列的元素。

访问queue队尾元素，如例：q.back()，即最后被压入队列的元素。

判断queue队列空，如例：q.empty()，当队列空时，返回true。

访问队列中的元素个数，如例：q.size()

### 4.5.8 [priority\_queue](https://github.com/huihut/interview/tree/master/STL" \l "priority_queue)（[优先队列](https://www.cnblogs.com/huashanqingzhu/p/11040390.html)）

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出 （first in, largest out）的行为特征。要包含头文件#include<queue>, 他和queue不同的就在于我们可以自定义其中数据的优先级, 让优先级高的排在队列前面,优先出队。优先队列具有队列的所有特性，包括队列的基本操作，只是在这基础上添加了内部的一个排序，它本质是一个堆实现的。

和队列基本操作相同:

top 访问队头元素

empty 队列是否为空

size 返回队列内元素个数

push 插入元素到队尾 (并排序)

emplace 原地构造一个元素并插入队列

pop 弹出队头元素

swap 交换内容

priority\_queue 优先队列，其底层是用堆来实现的。在优先队列中，队首元素一定是当前队列中优先级最高的那一个。在优先队列中，没有 front() 函数与 back() 函数，而只能通过 top() 函数来访问队首元素（也可称为堆顶元素），也就是优先级最高的元素。

**基本数据类型的优先级设置**

此处指的基本数据类型就是 int 型，double 型，char 型等可以直接使用的数据类型，优先队列对他们的优先级设置一般是数字大的优先级高，因此队首元素就是优先队列内元素最大的那个（如果是 char 型，则是字典序最大的）。

下面两种优先队列的定义是等价的

priority\_queue<int> q; //默认大在前

priority\_queue<int,vector<int>,less<int> >;//后面有一个空格，大在前

　　其中第二个参数( vector )，是来承载底层数据结构堆的容器，第三个参数( less )，则是一个比较类，less 表示数字大的优先级高，而 greater 表示数字小的优先级高。如果想让优先队列总是把最小的元素放在队首，只需进行如下的定义：

priority\_queue<int,vector<int>,greater<int> >q；//小在前

**和结构体结合使用**

struct student{

int grade;

string name;

//重载运算符，grade 值高的优先级大

friend operator < (student s1,student s2){

return s1.grade < s2.grade;

}

};

priority\_queue<student> q;

student s1,s2,s3;

s1.grade = 90;

s1.name = "Tom";

s2.grade = 80;

s2.name = "Jerry";

s3.grade = 100;

s3.name = "Kevin";

q.push(s1);

q.push(s2);

q.push(s3);

while(!q.empty()){

cout<<q.top().name<<":"<<q.top().grade<<endl;

q.pop();

}

定义：priority\_queue<Type, Container, Functional>

Type 就是数据类型，Container 就是容器类型（Container必须是用数组实现的容器，比如vector,deque等等，但不能用 list。STL里面默认用的是vector），Functional 就是比较的方式，当需要用自定义的数据类型时才需要传入这三个参数，使用基本数据类型时，只需要传入数据类型，默认是大顶堆

### 4.5.9 [set](https://github.com/huihut/interview/tree/master/STL" \l "set)

关于set，必须说明的是set关联式容器。set作为一个容器也是用来存储同一数据类型的数据类型，并且能从一个数据集合中取出数据，在set中每个元素的值都唯一，而且系统能根据元素的值自动进行排序。应该注意的是set中数元素的值不能直接被改变。C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树：红黑树，也成为RB树(Red-Black Tree)。RB树的统计性能要好于一般平衡二叉树，所以被STL选择作为了关联容器的内部结构。

set使用方法：

begin()     　　 ,返回set容器的第一个迭代器

end() 　　　　 ,返回set容器的最后一个迭代器

clear()   　　     ,删除set容器中的所有的元素

empty() 　　　,判断set容器是否为空

max\_size() 　 ,返回set容器可能包含的元素最大个数

size() 　　　　 ,返回当前set容器中的元素个数

rbegin　　　　 ,返回的值和end()相同

rend()　　　　 ,返回的值和rbegin()相同

count() 用来查找set中某个某个键值出现的次数。这个函数在set并不是很实用，因为一个键值在set只可能出现0或1次，这样就变成了判断某一键值是否在set出现过了。

erase(iterator) ,删除定位器iterator指向的值

erase(first,second),删除定位器first和second之间的值

erase(key\_value),删除键值key\_value的值

insert(key\_value); 将key\_value插入到set中 ，返回值是pair<set<int>::iterator,bool>，bool标志着插入是否成功，而iterator代表插入的位置，若key\_value已经在set中，则iterator表示的key\_value在set中的位置。

lower\_bound(key\_value) ，返回第一个大于等于key\_value的定位器

upper\_bound(key\_value)，返回最后一个大于等于key\_value的定位器

小结：set中的删除操作是不进行任何的错误检查的，比如定位器的是否合法等等，所以用的时候自己一定要注意。find() ，返回给定值值得定位器，如果没找到则返回end()。

### 4.5.10 [multiset](https://github.com/huihut/interview/tree/master/STL" \l "multiset)

set和multiset会根据特定的排序原则将元素排序。两者不同之处在于，multisets允许元素重复，而set不允许重复。

multiset mc:创建空集合,不包含任何元素

multiset mc(op):以op为排序准则，产生一个空的set

multiset c1(c2):复制c2中的元素到c1中

multiset c(const value\_type \*first, const value\_type\* last):复制[first, last)之间元素构成新集合

multiset c(const value\_type \*first, const value\_type\* last,op):以op为排序准则，复制[first, last)之间元素构成新集合。

Set mulitiset函数都差不多。

### 4.5.11 hash\_set & hash\_multiset

hash\_set与hash\_multiset：实值就是键值。以hashtable为底层机制，所以几乎所有的hash\_set行为，都是转调hashtable的操作行为。与set、multiset相比，hash\_set、hash\_multiset没有自动排序功能。hash\_set与set一样，实值就是键值。使用方式与set完全相同，插入元素操作采用底层机制hashtable的insert\_unique()。hash\_multiset与multiset一样，键值可以重复，使用方式和multiset相同。插入元素操作采用底层机制hashtable的insert\_equal()。

### 4.5.12 [map](https://github.com/huihut/interview/tree/master/STL" \l "map)

map 是关联容器，按照特定顺序存储由 key value (键值) 和 mapped value (映射值) 组合形成的元素。在内部，映射中的元素总是按照由其内部比较对象（比较类型）指示的特定的严格弱排序标准按键排序。映射容器通常比unordered\_map容器慢，以通过它们的键来访问各个元素，但是它们允许基于它们的顺序对子集进行直接迭代。 在该映射值地图可以直接通过使用其相应的键来访问括号运算符（（操作符[] ）。

它的特点是增加和删除节点对迭代器的影响很小，除了那个操作节点，对其他的节点都没有什么影响。对于迭代器来说，可以修改实值，而不能修改key。

**1.构造函数**

构造一个映射容器对象，根据所使用的构造器版本初始化其内容：

（1）空容器构造函数（默认构造函数）

构造一个空的容器，没有元素。

（2）范围构造函数

构造具有一样多的元素的范围内的容器[第一，最后一个），其中每个元件布设构造的从在该范围内它的相应的元件。

（3）复制构造函数（并用分配器复制）

使用x中的每个元素的副本构造一个容器。

（4）移动构造函数（并与分配器一起移动）

构造一个获取x元素的容器。 如果指定了alloc并且与x的分配器不同，那么元素将被移动。否则，没有构建元素（他们的所有权直接转移）。 x保持未指定但有效的状态。

（5）初始化列表构造函数

用il中的每个元素的副本构造一个容器。

map<char,int> first;

**2.迭代器**

map::begin end

返回引用map容器中第一个元素的迭代器。由于map容器始终保持其元素的顺序，所以开始指向遵循容器排序标准的元素。如果容器是空的，则返回的迭代器值不应被解除引用。

map::key\_comp

返回容器用于比较键的比较对象的副本。

map::value\_comp

返回可用于比较两个元素的比较对象，以获取第一个元素的键是否在第二个元素之前。

map::find

在容器中搜索具有等于k的键的元素，如果找到则返回一个迭代器，否则返回map::end的迭代器。如果容器的比较对象自反地返回假（即，不管元素作为参数传递的顺序），则两个key被认为是等同的。另一个成员函数map::count可以用来检查一个特定的键是否存在。

map::count

在容器中搜索具有等于k的键的元素，并返回匹配的数量。由于地图容器中的所有元素都是唯一的，因此该函数只能返回1（如果找到该元素）或返回零（否则）。如果容器的比较对象自反地返回错误（即，不管按键作为参数传递的顺序），则两个键被认为是等同的。

map::lower\_bound

将迭代器返回到下限。返回指向容器中第一个元素的迭代器，该元素的键不会在k之前出现（即，它是等价的或者在其后）。返回上一个。

map::upper\_bound

将迭代器返回到上限。返回一个指向容器中第一个元素的迭代器，它的关键字被认为是在k之后。返回下一个

Size 返回个数，用下标访问。

erase(1)：删除可以用下标删除，也可以用迭代器范围删除。

map的基本操作函数：

C++ maps是一种关联式容器，包含“关键字/值”对

begin() 返回指向map头部的迭代器

clear(） 删除所有元素

count() 返回指定元素出现的次数

empty() 如果map为空则返回true

end() 返回指向map末尾的迭代器

equal\_range() 返回特殊条目的迭代器对

erase() 删除一个元素

find() 查找一个元素

get\_allocator() 返回map的配置器

insert() 插入元素

key\_comp() 返回比较元素key的函数

lower\_bound() 返回键值>=给定元素的第一个位置

max\_size() 返回可以容纳的最大元素个数

rbegin() 返回一个指向map尾部的逆向迭代器

rend() 返回一个指向map头部的逆向迭代器

size() 返回map中元素的个数

swap() 交换两个map

upper\_bound() 返回键值>给定元素的第一个位置

value\_comp() 返回比较元素value的函数

### 4.5.13 [multimap](https://github.com/huihut/interview/tree/master/STL" \l "multimap)

无序容器（Unordered Container）：unordered\_set、unordered\_multiset、unordered\_map、unordered\_multimap。multimap 容器保存的是有序的键/值对，但它可以保存重复的元素。multimap 中会出现具有相同键的元素序列，它们会被添加到容器中。multimap 和 map 有相同范围的构造函数，默认的比较键的函数是 less<K>()。multimap 大部分成员函数的使用方式和 map 相同。因为重复键的原因，multimap 有一些函数的使用方式和 map 有一些区别。

### 4.5.14 hash\_map&hash\_multimap

hash\_map与hash\_multmap：每一个元素同时拥有实值和键值。以hashtable为底层机制，所以几乎所有的hash\_map行为，都是转调hashtable的操作行为。与map、multimap相比，hash\_map、hash\_multimap没有自动排序功能。hash\_map与map一样，实值就是键值。使用方式与set完全相同插入元素操作采用底层机制hashtable的insert\_unique()。hash\_multimap与multimap一样，键值可以重复，pair<key, value>插入元素操作采用底层机制hashtable的insert\_equal()。

### 4.5.15 tuple(元组)

元组是一个能够容纳元素集合的对象。每个元素可以是不同的类型。

tuple::tuple

**1.构建一个 tuple（元组）对象。**

这涉及单独构建其元素，初始化取决于调用的构造函数形式：

（1）默认的构造函数

构建一个 元组对象的元素值初始化。

（2）复制/移动构造函数

该对象使用tpl的内容进行初始化 元组目的。tpl 的相应元素被传递给每个元素的构造函数。

（3）隐式转换构造函数

同上。tpl中的 所有类型都可以隐含地转换为构造中它们各自元素的类型元组 目的。

（4）初始化构造函数 用elems中的相应元素初始化每个元素。elems 的相应元素被传递给每个元素的构造函数。

（5）对转换构造函数

该对象有两个对应于pr.first和的元素pr.second。PR中的所有类型都应该隐含地转换为其中各自元素的类型元组 目的。

（6）分配器版本

和上面的版本一样，除了每个元素都是使用allocator alloc构造的。

创建并初始化tuple

auto tup = std::make\_tuple("liu","yi","jiang","is",6,1,9);

**2.操作函数**

Tie：分而取之，获取tuple中的单个元素

auto tup = std::make\_tuple('l',6,2.33);

char a;

int b;

double c;

std:tie(a,b,c) = tup;

forward\_as\_tuple：用于接收右值引用数据生成tuple

auto tup = std::forward\_as\_tuple(1,"csdn")

tuple\_cat：用于连接tuple

std::tuple<float, string> tup1(3.14, "pi");

std::tuple<int, char> tup2(10, 'a');

auto tup3 = tuple\_cat(tup1, tup2);

get < i >:获取第i个元素的值

std::tuple<float, string> tup(666, "emmmm");

cout << get<0>(tup);

输入第一个元素666.

tuple\_element：获取tuple中特定元素数据结构

std::tuple\_element<0, decltype(tup1)>::type

Size：获取tuple中的元素个数

std::tuple<float, string> tup(666, "emmmm");

cout << tuple\_size<decltype(tup1)>::value;

### 4.5.16 pair

这个类把一对值（values）结合在一起，这些值可能是不同的类型（T1 和 T2）。每个值可以被公有的成员变量first、second访问。pair是tuple（元组）的一个特例。:pair主要的作用是将两个数据组合成一个数据，两个数据可以是同一类型或者不同类型。

pair::pair

构建一个pair对象。

这涉及到单独构建它的两个组件对象，初始化依赖于调用的构造器形式：

（1）默认的构造函数

构建一个 对对象的元素值初始化。

（2）复制/移动构造函数（和隐式转换）

该对象被初始化为pr的内容 对目的。pr 的相应成员被传递给每个成员的构造函数。

（3）初始化构造函数

会员 第一是由一个和成员构建的第二与b。

（4）分段构造

构造成员 first 和 second 到位，传递元素first\_args 作为参数的构造函数 first，和元素 second\_args 到的构造函数 second 。

（1）利用make\_pair：

pair<int, double> p1;

p1 = make\_pair(1, 1.2);

（2）变量间赋值：

pair<int, double> p1(1, 1.2);

pair<int, double> p2 = p1;

First / second：访问数据

## 4.6 C++设计模式

### 4.6.1 工厂模式（衣服工厂）

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。工厂模式作为一种创建模式，一般在创建复杂对象时，考虑使用；在创建简单对象时，建议直接new完成一个实例对象的创建。

由产品构成工厂，即创建一个虚基类产品，不同的工厂构成不同产品，如果需要新的产品可能需要新的基类，直接创建create对应工厂就行，这样不暴露产品特性。缺点就是产品类数据较多时，需要实现大量的工厂类，这无疑增加了代码量。

### 4.6.2 策略模式（apc adc）

策略模式是指定义一系列的算法，把它们单独封装起来，并且使它们可以互相替换，使得算法可以独立于使用它的客户端而变化，也是说这些算法所完成的功能类型是一样的，对外接口也是一样的，只是不同的策略为引起环境角色环境角色表现出不同的行为。相比于使用大量的if...else，使用策略模式可以降低复杂度，使得代码更容易维护。

缺点：可能需要定义大量的策略类，并且这些策略类都要提供给客户端。

[环境角色] 持有一个策略类的引用，最终给客户端调用。

比如一个hurt(type):根据type不同new一个adc或者apc对象。然后用模板类创建不同的对象，对象根据type new adc apc对象。

也可以用继承的方式来写，创建不同的类。

也可以用函数指针进行实现。

Archer<ApcHurt>\* arc = new Archer<ApcHurt>;110 arc->attack();

template<typename T>

class Archer

{

public:

void attack()

{

m\_hurt.blood();

}

private:

T m\_hurt;

};

### 4.6.3 适配器模式（队列）

适配器模式可以将一个类的接口转换成客户端希望的另一个接口，使得原来由于接口不兼容而不能在一起工作的那些类可以在一起工作。通俗的讲就是当我们已经有了一些类，而这些类不能满足新的需求，此时就可以考虑是否能将现有的类适配成可以满足新需求的类。适配器类需要继承或依赖已有的类，实现想要的目标接口。

缺点：过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

有以下两种方式：

1.使用复合实现适配器模式

有一个双端队列Deque（被适配类），创建一个虚类顺序类抽象目标类Sequence，创建一个栈适配类，对象是双端队列，重新写方法。

1. 使用继承实现适配器模式

前面和第一个差不多，不过不在适配类中创建对象，而是直接在适配类继承抽象和被适配类。

### 4.6.4 单例模式（instance）

单例模式顾名思义，保证一个类仅可以有一个实例化对象，并且提供一个可以访问它的全局接口。实现单例模式必须注意一下几点：

单例类只能由一个实例化对象。

单例类必须自己提供一个实例化对象。

单例类必须提供一个可以访问唯一实例化对象的接口。可以用delete阻止复制拷贝或者nocopy类。声明delet函数，删除函数，可以声明但是不能使用，阻止类的拷贝赋值，但也可以指定给非拷贝控制成员。

单例模式分为懒汉和饿汉两种实现方式。

懒汉单例模式

懒汉：故名思义，不到万不得已就不会去实例化类，也就是说在第一次用到类实例的时候才会去实例化一个对象。在访问量较小，甚至可能不会去访问的情况下，采用懒汉实现，这是以时间换空间。即在类中getinstance中new自己对象，如果是线程安全的懒汉单例模式，需要在new中lock一下。或者在geinstace中创建一个static对象返回一个reference指向local static对象。

饿汉单例模式

饿汉：饿了肯定要饥不择食。所以在单例类定义的时候就进行实例化。在访问量比较大，或者可能访问的线程比较多时，采用饿汉实现，可以实现更好的性能。这是以空间换时间。

Singleton\* Singleton::m\_pSingleton = new Singleton();即在外面进行定义。

### 4.6.5 原型模式（clone修改）

原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。通俗的讲就是当需要创建一个新的实例化对象时，我们刚好有一个实例化对象，但是已经存在的实例化对象又不能直接使用。这种情况下拷贝一个现有的实例化对象来用，可能会更方便。

以下情形可以考虑使用原型模式：

当new一个对象，非常繁琐复杂时，可以使用原型模式来进行复制一个对象。比如创建对象时，构造函数的参数很多，而自己又不完全的知道每个参数的意义，就可以使用原型模式来创建一个新的对象，不必去理会创建的过程。

当需要new一个新的对象，这个对象和现有的对象区别不大，我们就可以直接复制一个已有的对象，然后稍加修改。

当需要一个对象副本时，比如需要提供对象的数据，同时又需要避免外部对数据对象进行修改，那就拷贝一个对象副本供外部使用。

//提供一个抽象克隆基类Clone。

/具体的实现类 class Sheep:public Clone

### 4.6.6 模板模式（computer cpu）

模板模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

当多个类有相同的方法，并且逻辑相同，只是细节上有差异时，可以考虑使用模板模式。具体的实现上可以将相同的核心算法设计为模板方法，具体的实现细节有子类实现。

缺点:每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。以生产电脑为例，电脑生产的过程都是一样的，只是一些装配的器件可能不同而已。

代码就是computer 有虚函数，然后用computerA继承基类虚函数overwrite。分别由电脑abc不同内容。

### 4.6.7 建造者模式（建筑builder）

建造者模式：将复杂对象的构建和其表示分离，使得相同的构建过程可以产生不同的表示。以下情形可以考虑使用建造者模式：

对象的创建复杂，但是其各个部分的子对象创建算法一定。需求变化大，构造复杂对象的子对象经常变化，但将其组合在一起的算法相对稳定。

建造者模式的优点：

将对象的创建和表示分离，客户端不需要了解具体的构建细节。增加新的产品对象时，只需要增加其具体的建造类即可，不需要修改原来的代码，扩展方便。产品之间差异性大，内部变化较大、较复杂时不建议使用建造者模式。

建造者类：创建和提供实例； Director类：管理建造出来的实例的依赖关系。

Order具体的产品类 OrderBuilder抽象建造类，提供建造接口。

具体的建造类class VegetarianOrderBuilder : public OrderBuilder

具体的建造类class MeatOrderBuilder : public OrderBuilder

Director类，负责管理实例创建的依赖关系，指挥构建者类创建实例class Director 设置具体的建造类builder

### 4.6.8 外观模式（computer cpu start）

外观模式：为子系统中的一组接口定义一个一致的界面；外观模式提供一个高层的接口，这个接口使得这一子系统更加容易被使用；对于复杂的系统，系统为客户端提供一个简单的接口，把负责的实现过程封装起来，客户端不需要连接系统内部的细节。

以下情形建议考虑外观模式：

设计初期阶段，应有意识的将不同层分离，层与层之间建立外观模式。开发阶段，子系统越来越复杂，使用外观模式提供一个简单的调用接口。一个系统可能已经非常难易维护和扩展，但又包含了非常重要的功能，可以为其开发一个外观类，使得新系统可以方便的与其交互。

优点：

实现了子系统与客户端之间的松耦合关系。客户端屏蔽了子系统组件，减少了客户端所需要处理的对象数据，使得子系统使用起来更方便容易。更好的划分了设计层次，对于后期维护更加的容易。

外观模式其实和模板差不多，以电脑的启动过程为例，客户端只关心电脑开机的、关机的过程，并不需要了解电脑内部子系统的启动过程。电脑start里面由子控件显示屏start，子控件外部设备start等组合而成。

### 4.6.9 组合模式（root add leaf）

组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构，组合模式使得客户端对单个对象和组合对象的使用具有一直性。

既然讲到以树形结构表示“部分-整体”，那可以将组合模式想象成一根大树，将大树分成树枝和树叶两部分，树枝上可以再长树枝，也可以长树叶，树叶上则不能再长出别的东西。

以下情况可以考虑使用组合模式：希望表示对象的部分-整体层次结构。希望客户端忽略组合对象与单个对象的不同，客户端将统一的使用组合结构中的所有对象。

HeadCompany\* headRoot = new HeadCompany("Head Root Company");

155

156 HeadCompany\* childRoot1 = new HeadCompany("Child Company A");

157 ResearchCompany\* r1 = new ResearchCompany("Research Company A");

158 SalesCompany\* s1 = new SalesCompany("Sales Company A");

159 SalesCompany\* s2 = new SalesCompany("Sales Company B");

160 FinanceCompany\* f1 = new FinanceCompany("FinanceCompany A");

161

162 childRoot1->add(r1);

163 childRoot1->add(s1);

164 childRoot1->add(s2);

165 childRoot1->add(f1);

166

167 HeadCompany\* childRoot2 = new HeadCompany("Child Company B");

168 ResearchCompany\* r2 = new ResearchCompany("Research Company B");

169 SalesCompany\* s3 = new SalesCompany("Sales Company C");

170 SalesCompany\* s4 = new SalesCompany("Sales Company D");

171 FinanceCompany\* f2 = new FinanceCompany("FinanceCompany B");

172

173 childRoot2->add(r2);

174 childRoot2->add(s3);

175 childRoot2->add(s4);

176 childRoot2->add(f2);

177

178 headRoot->add(childRoot1);

179 headRoot->add(childRoot2);

180 headRoot->display(1);

就是root添加不同的树叶。

### 4.6.10 代理模式（代理）

代理模式：为其它对象提供一种代理以控制这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介作用。

优点：

职责清晰。真实的角色只负责实现实际的业务逻辑，不用关心其它非本职责的事务，通过后期的代理完成具体的任务。这样代码会简洁清晰。

代理对象可以在客户端和目标对象之间起到中介的作用，这样就保护了目标对象。扩展性好。

类-》虚类-》类的指针指向目标-》执行。

class ManProxy : public Profession

41 {

42 public:

43 ManProxy(const Gril& gril):m\_pMan(new YoungMan(gril)){}

44 ~ManProxy()

45 {

46 delete m\_pMan;

47 m\_pMan = nullptr;

48 }

49 void profess()

50 {

51 m\_pMan->profess();

52 }

53 private:

54 YoungMan\* m\_pMan;

55 };

### 4.6.11 享元模式（外部+内部map结合）

享元模式：运用共享技术有效地支持大量细粒度的对象。在有大量对象时，把其中共同的部分抽象出来，如果有相同的业务请求，直接返回内存中已有的对象，避免重新创建。

以下情况可以考虑使用享元模式：系统中有大量的对象，这些对象消耗大量的内存，且这些对象的状态可以被外部化。对于享元模式，需要将对象的信息分为两个部分：内部状态和外部状态。内部状态是指被共享出来的信息，储存在享元对象内部且不随环境变化而改变；外部状态是不可以共享的，它随环境改变而改变，是由客户端控制的。

即将将内部状态作为标识，进行共享。抽象享元类，提供享元类外部接口AbstractConsumer。具体的享元类 class Consumer : public AbstractConsumer。

享元工厂类 class Trusteeship 里面map<string, shared\_ptr<Consumer>> m\_consumerMap;

### 4.6. 12 桥接模式（mobile app）

桥接模式：将抽象部分与实现部分分离，使它们都可以独立变换。

以下情形考虑使用桥接模式：

当一个对象有多个变化因素的时候，考虑依赖于抽象的实现，而不是具体的实现。

当多个变化因素在多个对象间共享时，考虑将这部分变化的部分抽象出来再聚合/合成进来。

当一个对象的多个变化因素可以动态变化的时候。

优点：

将实现抽离出来，再实现抽象，使得对象的具体实现依赖于抽象，满足了依赖倒转原则。

更好的可扩展性。可动态的切换实现。桥接模式实现了抽象和实现的分离，在实现桥接模式时，就可以实现动态的选择具体的实现。

具体的App实现类class GameApp:public App

具体的App实现类 class TranslateApp:public App

抽象手机类，提供接口 class MobilePhone appRun(App\* app) = 0; //实现App与手机的桥接

具体的手机实现类 class HuaWei:public MobilePhone

### 4.6. 13 装饰模式（构件依次添加装饰）

装饰模式：动态地给一个对象添加一些额外的功能，它是通过创建一个包装对象，也就是装饰来包裹真实的对象。新增加功能来说，装饰器模式比生产子类更加灵活。

以下情形考虑使用装饰模式：

需要扩展一个类的功能，或给一个类添加附加职责。

需要动态的给一个对象添加功能，这些功能可以再动态的撤销。

需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变的不现实。

当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

//抽象构件（Component）角色：给出一个抽象接口，以规范准备接收附加责任的对象。

//具体构件（Concrete Component）角色：定义一个将要接收附加责任的类。

//装饰（Decorator）角色：持有一个构件（Component）对象的实例，并实现一个与抽象构件接口一致的接口。

//具体装饰（Concrete Decorator）角色：负责给构件对象添加上附加的责任。

### 4.6. 14 备忘录模式（备忘录记录内部状态）

备忘录模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可以将该对象恢复到原来保存的状态。

备忘录模式中需要定义的角色类：

Originator(发起人)：负责创建一个备忘录Memento，用以记录当前时刻自身的内部状态，并可使用备忘录恢复内部状态。Originator可以根据需要决定Memento存储自己的哪些内部状态。

Memento(备忘录)：负责存储Originator对象的内部状态，并可以防止Originator以外的其他对象访问备忘录。备忘录有两个接口：Caretaker只能看到备忘录的窄接口，他只能将备忘录传递给其他对象。Originator却可看到备忘录的宽接口，允许它访问返回到先前状态所需要的所有数据。

Caretaker(管理者):负责备忘录Memento，不能对Memento的内容进行访问或者操作。

84 int main()

85 {

86 GameValue v1 = {0, "Ak", "3K"};

87 Game game(v1); //初始值

88 game.addGrade();

89 game.showValue();

90 cout << "----------" << endl;

91 Caretake care;

92 care.save(game.saveValue()); //保存当前值

93 game.addGrade(); //修改当前值

94 game.replaceArm("M16");

95 game.replaceCorps("123");

96 game.showValue();

97 cout << "----------" << endl;

98 game.load(care.load()); //恢复初始值

99 game.showValue();

100 return 0;

101 }

### 4.6. 15 中介者模式（中介添加）

中介者模式：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显示地相互引用，从而使其耦合松散，而且可以独立地改变它们之前的交互。

如果对象与对象之前存在大量的关联关系，若一个对象改变，常常需要跟踪与之关联的对象，并做出相应的处理，这样势必会造成系统变得复杂，遇到这种情形可以考虑使用中介者模式。当多个对象存在关联关系时，为它们设计一个中介对象，当一个对象改变时，只需要通知它的中介对象，再由它的中介对象通知每个与它相关的对象。

108 int main()

109 {

110 HouseMediator\* hMediator = new HouseMediator;

111 Buyer\* buyer = new Buyer(hMediator);

112 Seller\* seller = new Seller(hMediator);

113 ​

114 hMediator->setBuyer(buyer);

115 hMediator->setSeller(seller);

116 ​

117 buyer->sendMessage("Sell not to sell?");

118 seller->sendMessage("Of course selling!");

119 ​

120 delete buyer;

121 buyer = nullptr;

123 delete seller;

124 seller = nullptr;

126 delete hMediator;

127 hMediator = nullptr;

130 return 0;

131 }

### 4.6. 16 职责链模式（依次职责添加）

职责链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之前的耦合关系，将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。

职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无需关心请求的处理细节和请求的传递，所有职责链将请求的发送者和请求的处理者解耦了。

74 int main()

75 {

76 Leader\* general = new General(nullptr);

77 Leader\* captain = new Captain(general);

78 Leader\* monitor = new Monitor(captain);

79 monitor->handleRequest(Level\_One);

80 ​

81 delete monitor;

82 monitor = nullptr;

83 delete captain;

84 captain = nullptr;

85 delete general;

86 general = nullptr;

87 return 0;

88 }

### 4.6. 17 观察者模式（注册观测者与被观测者一样变化）

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都要得到通知并自动更新。

观察者模式从根本上讲必须包含两个角色：观察者和被观察对象。

被观察对象自身应该包含一个容器来存放观察者对象，当被观察者自身发生改变时通知容器内所有的观察者对象自动更新。

观察者对象可以注册到被观察者的中，完成注册后可以检测被观察者的变化，接收被观察者的通知。当然观察者也可以被注销掉，停止对被观察者的监控。

/\*

\* 关键代码：在目标类中增加一个ArrayList来存放观察者们。

\*/

#include <iostream>

#include <list>

#include <memory>

​

using namespace std;

​

class View;

​

//被观察者抽象类 数据模型

class DataModel

{

public:

virtual ~DataModel(){}

virtual void addView(View\* view) = 0;

virtual void removeView(View\* view) = 0;

virtual void notify() = 0; //通知函数

};

​

//观察者抽象类 视图

class View

{

public:

virtual ~View(){ cout << "~View()" << endl; }

virtual void update() = 0;

virtual void setViewName(const string& name) = 0;

virtual const string& name() = 0;

};

​

//具体的被观察类， 整数模型

class IntDataModel:public DataModel

{

public:

~IntDataModel()

{

m\_pViewList.clear();

}

​

virtual void addView(View\* view) override

{

shared\_ptr<View> temp(view);

auto iter = find(m\_pViewList.begin(), m\_pViewList.end(), temp);

if(iter == m\_pViewList.end())

{

m\_pViewList.push\_front(temp);

}

else

{

cout << "View already exists" << endl;

}

}

​

void removeView(View\* view) override

{

auto iter = m\_pViewList.begin();

for(; iter != m\_pViewList.end(); iter++)

{

if((\*iter).get() == view)

{

m\_pViewList.erase(iter);

cout << "remove view" << endl;

return;

}

}

}

​

virtual void notify() override

{

auto iter = m\_pViewList.begin();

for(; iter != m\_pViewList.end(); iter++)

{

(\*iter).get()->update();

}

}

​

private:

list<shared\_ptr<View>> m\_pViewList;

};

​

//具体的观察者类 表视图

class TableView : public View

{

public:

TableView() : m\_name("unknow"){}

TableView(const string& name) : m\_name(name){}

~TableView(){ cout << "~TableView(): " << m\_name.data() << endl; }

​

void setViewName(const string& name)

{

m\_name = name;

}

​

const string& name()

{

return m\_name;

}

​

void update() override

{

cout << m\_name.data() << " update" << endl;

}

​

private:

string m\_name;

};

​

int main()

{

/\*

\* 这里需要补充说明的是在此示例代码中，View一旦被注册到DataModel类之后，DataModel解析时会自动解析掉 \* 内部容器中存储的View对象，因此注册后的View对象不需要在手动去delete，再去delete View对象会出错。

\*/

View\* v1 = new TableView("TableView1");

View\* v2 = new TableView("TableView2");

View\* v3 = new TableView("TableView3");

View\* v4 = new TableView("TableView4");

​

IntDataModel\* model = new IntDataModel;

model->addView(v1);

model->addView(v2);

model->addView(v3);

model->addView(v4);

​

model->notify();

​

cout << "-------------\n" << endl;

​

model->removeView(v1);

​

model->notify();

​

delete model;

model = nullptr;

​

return 0;

}

​

参考连接：

<https://www.cnblogs.com/chengjundu/p/8473564.html>

# 数据结构与算法

## 5.1数据结构与算法

数据结构

1.线性

栈

队列 双端队列 循环队列 优先队列

散列表 哈希表

2非线性结构

二叉树 满二叉树 完全二叉树 二叉树自平衡(红黑树，AVL树，树堆)

深度优先遍历：前序 中序 后序 广度优先遍历：层序

二叉堆(最小堆最小在最上面，最大堆最大在最上面。利用完全二叉树交换最后面一个)----可实现优先队列

算法

1排序算法

冒泡 选择 插入 希尔O(n2)

快速 并归 堆 O(nlogn)

计数 桶 基数 复杂度线性

冒泡：优化可能后半有序，可设置标志位推出，保存最后一次值下次跳过 起始值跳过。

鸡尾酒排序：比较交换双向，类似钟摆，先是左到右，后是右到左。减少排序回合，适用大部分有序。

插入排序：有序和无序，将无序插入有序，进行移位操作。

选择：选择最大最右，然后最大的放在右边，依次直到排序完成。

希尔排序：选择上修改，10数增量5-2-1，每5增量形成一组，每组按大小重新挑选成一个数组，然后再按2-1增量排序。

快速排序：分治冒泡，交换排序，选择基准把数分两边，直到不能分。交换有双边和单边循环。双边除基数，头尾两个指针指向数和基数比较，符合分组移动指针，交换值，直到排序完成。单边循环有一个指针进行分组。

堆排序：在二叉堆上进行插入，最后输出。

计数排序：创建最大-最小范围数组，依次找数在对应下标下加1，最后根据数组大小排序。

桶排序：创建几个分组桶。(最大-最小)/桶数，分组拍寻。

面试算法

1. 判断是否有环：依次遍历比较之前是否有相同值，或者依次放入hash中看重复值，两个指针一个+1一个+2，比较是否相同，类似速度追寻比较。
2. 计算环长：环头依次找到和环头相等值，或者两个指针相差乘以步数相等即环长。

## 5.2 算法数据结构实现

### 5.2.1 二叉查找树插入和显示

二叉树基本结构：

struct TreeList  
{  
 int data;  
 struct TreeList \* p\_leftleaf;  
 struct TreeList \* p\_rightleaf;  
};

二叉查找树：左边小于根节点，右边大于根节点

插入数据方法：

1. 通过调用递归函数方法插入数据，一个个找数据，大于右边或者小于左边并为0则插入左边，否则不为0则递归调用
2. 通过一个中间指针循环找出适合插入得根结点，最后插入

显示数据方法：左 中 右 也是递归方法

printf("\n data = %d", p\_leaf->data);  
ShowTreeDataPre(p\_leaf->p\_leftleaf);  
ShowTreeDataPre(p\_leaf->p\_rightleaf);

### 5.2.2 单向链表基本操作

单向链表基础结构：

truct NodeList  
{  
 int data;  
 struct NodeList \* p\_next;  
};

链表插入数据：找到next尾部为0，在next指向插入malloc数据。

在指定位置插入数据：一次查找到指定位置，p\_temp = p\_ptr->p\_next;p\_ptr = p\_ptr->p\_next;插入数据。

查找数据返回位置：指针一次查找，返回位置

升序或降序排序：两个指针指向，依次指向为非0，比较大小冒泡排序

显示数据：指针依次找为非0，依次打印

删除：指针得指针

while(0 != \*p\_head)  
{  
 p\_ptr = \*p\_head;  
 \*p\_head = (\*p\_head)->p\_next;  
 free(p\_ptr);  
}

## 5.3 leetcode

### 5.3.1 求数组中和为目标值得

1.暴力法，是依次找和为目标值

2从开头，找目标值-第一个值

1. 用key value
2. 用这样方式

while (low < high){  
 if (numbers[low] + numbers[high] == target){  
 result.push\_back(low+1);  
 result.push\_back(high+1);  
 return result;  
 }else{  
 numbers[low] + numbers[high] > target ? high-- : low++;  
 }  
}

如果是求三个数和，可以安装两个和。或者先排序再计算和。

### 5.3.2 求数组中最长不含重复字符长度

比如aijhgkab，就是a到a。从第一位开始，计算到末尾出现重复长度，保存最长长度。

### 5.3.3 求整形数据反转

比如198 891。198/10 =8，sum = 8\*10,依次整除直到为0.while（x!=0）

### 5.3.4 求两个有序数组中位数

首先判断两个数组为0情况，计算A数组中间值在B组得位置，如果位置值等于两个数组得和一半，即为中间值。如果不等于一半，需要减少一个值计算中间值。如果位置值大于两个数组和，需要递归计算到正确得中间值。如果位置值小于两个数组之和，需要递归计算到正确中间值。

### 5.3.5 最长回文子串

求一个字符串最长回文子串，1232134即为12321长度。方法1从1开始遍历找到下一个相同得。方法2字符串反转再求公共字符串。方法3中心扩展，即一个数前后比较。方法4假设回文长度为1~n进行查找比较是否存在。方法5动态规划法即123321是 2332也是依次。

### 5.3.6 字符串进行反转

字符串反转，asdfg变成gfdsa，方法1首尾交换，依次交换即可反转。

### 5.3.7正则表达式匹配

匹配正则表达式和是字符串是否相同，.\*用指针比较，是否相等。叠加比较。

### 5.3.8 计算一系列线段包围最大面积

盛最多水的容器，暴力法从1开始计算面积。

### 5.3.9 括号生成

方法1暴力法检查是否有效。方法2递归

void generator(vector<string>& result, int left, int right, string s){  
 if (left==0 && right==0){  
 result.push\_back(s);  
 return;  
 }  
 if (left>0){  
 generator(result, left-1, right, s+'(');  
 }  
 if (right>0 && right>left){  
 generator(result, left, right-1, s+')');  
 }  
}

### 5.3.10 **[移除相同元素](https://leetcode-cn.com/problems/remove-element/)**

排序后进行查找，

int removeElement(vector<int>& nums, int val) {

int pos = 0;

for (int i=0; i<nums.size(); i++){

if (nums[i] != val){

nums[pos++] = nums[i];

}

}

return pos;

}

int removeElement(int A[], int n, int elem) {

int tail = n-1;

int i = 0;

while ( i<=tail ){

if (A[i]==elem){

A[i] = A[tail--];

continue;

}

i++;

}

return tail+1;

}

将后面的排在后面或者将尾部排在后面

### 5.3.11 下一个数

1234 1243

方法1 暴力法

方法2 一遍扫描



1往后找到两是下降的4比7小。再往找比4大最接近5交换

### 5.3.12 [组合总和](https://leetcode-cn.com/problems/combination-sum/)

计算一个数组中和为目标的总和

递归法

### 5.3.13 [缺失的第一个正数](https://leetcode-cn.com/problems/first-missing-positive/)

给定一个未排序的整数数组，找出其中没有出现的最小的正整数。方法1 排序之后再查找一下

方法2.桶排序，比较索引上得值，放在合适位置上，到时候比较位置上值和索引值，不合适得即为

方法3 如果 a ^ b = c ，那么 a ^ c = b 与 b ^ c = a 同时成立，利用这一条，可以用于交换两个变量的值。

### 5.3.14 [接雨水](https://leetcode-cn.com/problems/trapping-rain-water/)

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。输入: [0,1,0,2,1,0,1,3,2,1,2,1]

输出: 6



方法1 按行求

整个思路就是，求第 i 层的水，遍历每个位置，如果当前的高度小于 i，并且两边有高度大于等于 i 的，说明这个地方一定有水，水就可以加 11。

方法2 按列求

初始化 ans=0ans=0

从左向右扫描数组：

初始化 {max\_left}=0max\_left=0 和 {max\_right}=0max\_right=0

从当前元素向左扫描并更新：

{max\_left}=\max({max\_left},{height}[j])max\_left=max(max\_left,height[j])

从当前元素向右扫描并更新：

{max\_right}=\max({max\_right},{height}[j])max\_right=max(max\_right,height[j])

将\min({max\_left},{max\_right}) - {height}[i]min(max\_left,max\_right)−height[i] 累加到 {ans}ans

方法3

找到数组中从下标 i 到最左端最高的条形块高度 {left\\_max}left\_max。

找到数组中从下标 i 到最右端最高的条形块高度 {right\\_max}right\_max。

扫描数组 {height}height 并更新答案：

累加 \min({max\\_left}[i],{max\\_right}[i]) - {height}[i]min(max\_left[i],max\_right[i])−height[i] 到 ansans 上

### 5.3.15 [跳跃游戏](https://leetcode-cn.com/problems/jump-game-ii/)

给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。你的目标是使用最少的跳跃次数到达数组的最后一个位置。

方法1 计算每一步内能达到最长长度 ，然后下一步依次到最后一步

方法2 计算每一步到最后一步最短长度，暴力法

### 5.3.15 图像数组旋转90度

方法1 单次循环中旋转 4 个矩形

for( int i=0; i<n/2; i++ ){

int low=i, high=n-i-1;

for (int j=low; j<high; j++){

int tmp;

tmp = matrix[i][j];

// left to top

matrix[i][j] = matrix[n-j-1][i];

// bottom to left

matrix[n-j-1][i] = matrix[n-i-1][n-j-1];

// right to bottom

matrix[n-i-1][n-j-1] = matrix[j][n-i-1];

// top to right

matrix[j][n-i-1] = tmp;

}

}

### 5.3.16 N皇后问题

n 皇后问题研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

方法1 依次排除前后下上

方法2 约束编程 它的基本含义是在放置每个皇后以后增加限制。当在棋盘上放置了一个皇后后，立即排除当前行，列和对应的两个对角线。该过程传递了 约束 从而有助于减少需要考虑情况数。

### 5.3.17求数组合并区间

方法1 数组根据start数进行排序进行合并

### 5.3.18 旋转链表

输入: 1->2->3->4->5->NULL, k = 2

输出: 4->5->1->2->3->NULL

方法1：找到断点k = k%len，然后后面的作为head 前面移到末尾

### 5.3.19 不同路径动态规划

个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。问总共有多少条不同的路径？

方法1 动态规划

vector< vector <int> > dp (n, vector<int>(m, 1));

for (int row=1; row<n; row++) {

for (int col=1; col<m; col++) {

dp[row][col] = dp[row-1][col] + dp[row][col-1];

}

}

return dp[n-1][m-1];

障碍物

如果第一个格点 obstacleGrid[0,0] 是 1，说明有障碍物，那么机器人不能做任何移动，我们返回结果 0。

否则，如果 obstacleGrid[0,0] 是 0，我们初始化这个值为 1 然后继续算法。

遍历第一行，如果有一个格点初始值为 1 ，说明当前节点有障碍物，没有路径可以通过，设值为 0 ；否则设这个值是前一个节点的值 obstacleGrid[i,j] = obstacleGrid[i,j-1]。

遍历第一列，如果有一个格点初始值为 1 ，说明当前节点有障碍物，没有路径可以通过，设值为 0 ；否则设这个值是前一个节点的值 obstacleGrid[i,j] = obstacleGrid[i-1,j]。

现在，从 obstacleGrid[1,1] 开始遍历整个数组，如果某个格点初始不包含任何障碍物，就把值赋为上方和左侧两个格点方案数之和 obstacleGrid[i,j] = obstacleGrid[i-1,j] + obstacleGrid[i,j-1]。

如果这个点有障碍物，设值为 0 ，这可以保证不会对后面的路径产生贡献。

#### [最小路径和](https://leetcode-cn.com/problems/minimum-path-sum/)

暴力



#### **二维动态规划**



### 5.3.20 求平方根

res = (res + x / res) / 2多执行几次既可以

( 4 + 2/ 4 ) / 2 = 2.25

( 2.25 + 2/ 2.25 ) / 2 = 1.56944..

( 1.56944..+ 2/1.56944..) / 2 = 1.42189..

( 1.42189..+ 2/1.42189..) / 2 = 1.41423..

### 5.3.21 求路径

/a/./b/../../c/

利用堆栈的原理去解决

### 5.3.22 编辑距离

给定两个单词 word1 和 word2，计算出将 word1 转换成 word2 所使用的最少操作数 。

动态规划

//Dynamic Programming

int row, col;

for (row=1; row<m.size(); row++) {

for(col=1; col<m[row].size(); col++){

if (word1[row-1] == word2[col-1] ){

m[row][col] = m[row-1][col-1];

}else{

int minValue = min(m[row-1][col-1], m[row-1][col], m[row][col-1]);

m[row][col] = minValue + 1;

}

}

}



### 5.3.23 [最小覆盖子串](https://leetcode-cn.com/problems/minimum-window-substring/)

给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

方法1采用窗口得方法，即用两个指针依次排查，一前一后。

初始，leftleft指针和rightright指针都指向SS的第一个元素.

将 rightright 指针右移，扩张窗口，直到得到一个可行窗口，亦即包含TT的全部字母的窗口。得到可行的窗口后，将lefttleftt指针逐个右移，若得到的窗口依然可行，则更新最小窗口大小。若窗口不再可行，则跳转至 22。

### 5.3.24 组合

给定两个整数 n 和 k，返回 1 ... n 中所有可能的 k 个数的组合。

方法1 依次排列出来

方法2 利用字典 00011 对应所有数字

### 5.3.25 柱状图中最大的矩形

方法1 暴力法求解

放法2 分治

算法

通过观察，可以发现，最大面积矩形存在于以下几种情况：

确定了最矮柱子以后，矩形的宽尽可能往两边延伸。

在最矮柱子左边的最大面积矩形（子问题）。

在最矮柱子右边的最大面积矩形（子问题）。

举个例子：

[6, 4, 5, 2, 4, 3, 9]

这里最矮柱子高度为 2 。以 2 为高的最大子矩阵面积是 2x7=14 。现在，我们考虑上面提到的第二种和第三种情况。我们对高度为 2 柱子的左边和右边采用同样的过程。在 2 的左边， 4 是最小的，形成区域为 4x3=12 。将左边区域再继续分，矩形的面积分别为 6x1=6 和 5x1=5 。同样的，我们可以求出右边区域的面积为 3x3=9, 4x1=4 和 9x1=9 。因此，我们得到最大面积是 16 。

方法3 栈

在这种方法中，我们维护一个栈。一开始，我们把 -1 放进栈的顶部来表示开始。初始化时，按照从左到右的顺序，我们不断将柱子的序号放进栈中，直到遇到相邻柱子呈下降关系，也就是 a[i-1] > a[i]a[i−1]>a[i] 。现在，我们开始将栈中的序号弹出，直到遇到 stack[j]stack[j] 满足a\big[stack[j]\big] \leq a[i]a[stack[j]]≤a[i] 。每次我们弹出下标时，我们用弹出元素作为高形成的最大面积矩形的宽是当前元素与 stack[top-1]stack[top−1] 之间的那些柱子。也就是当我们弹出 stack[top]stack[top] 时，记当前元素在原数组中的下标为 i ，当前弹出元素为高的最大矩形面积为：

(i-stack[top-1]-1) \times a\big[stack[top]\big].

(i−stack[top−1]−1)×a[stack[top]].

更进一步，当我们到达数组的尾部时，我们将栈中剩余元素全部弹出栈。在弹出每一个元素是，我们用下面的式子来求面积： (stack[top]-stack[top-1]) \times a\big[stack[top]\big](stack[top]−stack[top−1])×a[stack[top]]，其中，stack[top]stack[top]表示刚刚被弹出的元素。因此，我们可以通过每次比较新计算的矩形面积来获得最大的矩形面积。

### 5.3.26 格雷编码

格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个位数的差异。

给定一个代表编码总位数的非负整数 n，打印其格雷编码序列。格雷编码序列必须以 0 开头



就是在前面依次加 0 1 叠加就可以

### 5.3.27 [交错字符串](https://leetcode-cn.com/problems/interleaving-string/)

给定三个字符串 s1, s2, s3, 验证 s3 是否是由 s1 和 s2 交错组成的。输入: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"

输出: true

方法1 暴力法列举所有得可能性

方法2 利用队列得方法，有相同得取出即可，都没有就报错。

方法3 动态规划

### 5.3.28 [二叉树展开为链表](https://leetcode-cn.com/problems/flatten-binary-tree-to-linked-list/)

给定一个二叉树，原地将它展开为链表。

\* 1

\* / \

\* 2 5

\* / \ \

\* 3 4 6

\* 1

\* \

\* 2

\* \

\* 3

\* \

\* 4

\* \

\* 5

\* \

\* 6

方法1 递归依次计算

方法2

将左子树插入到右子树的地方

将原来的右子树接到左子树的最右边节点

考虑新的右子树的根节点，一直重复上边的过程，直到新的右子树为 null

### 5.3.29 [不同的子序列](https://leetcode-cn.com/problems/distinct-subsequences/)

给定一个字符串 S 和一个字符串 T，计算在 S 的子序列中 T 出现的个数。

一个字符串的一个子序列是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，"ACE" 是 "ABCDE" 的一个子序列，而 "AEC" 不是）

方法1 暴力法

方法2 动态规划

从0到字符串的长度，即需要一个  dp[ t.size()+1 ] [ s.size()+1 ]  的矩阵。其中，dp[ i ][ j ] 代表当字符串 s 走到 j 位置，字符串 t 走到 i 位置， 有多少次匹配。

其中暴力递归中

dfs(res, s, t, i, j+1)  对应  dp[ i ][ j ] = dp[ i ][ j-1 ]，

dfs(res, s, t, i+1, j+1)  对应  dp[ i ][ j ] += dp[ i-1 ][ j-1 ]。

初始化第一行的意义是字符串 s 匹配空字符串的结果， 结果为 1 。

S = "babgbag", T = "bag" 时的 dp 矩阵



即遇到相同得计算左边和左边上面和之值，不相同得话保持相同值。

### 5.3.30 单词接龙

给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。转换需遵循如下规则：

每次转换只能改变一个字母。

转换过程中的中间单词必须是字典中的单词。

beginWord = "hit",

endWord = "cog",

wordList = ["hot","dot","dog","lot","log","cog"]

输出

[

["hit","hot","dot","dog","cog"],

  ["hit","hot","lot","log","cog"]

]

利用map或者堆栈进行记录所有路线

### 5.3.31 分发糖果

老师想给孩子们分发糖果，有 N 个孩子站成了一条直线，老师会根据每个孩子的表现，预先给他们评分。你需要按照以下要求，帮助老师给这些孩子分发糖果：

每个孩子至少分配到 1 个糖果。相邻的孩子中，评分高的孩子必须获得更多的糖果。

方法1暴力法，先都赋值1 依次比较 增加

方法2 用两个数组，一个从左，一个从右比较。大于就加1 ，然后两个数组比较，取大的。即满足要求

### 5.3.32 [LRU缓存机制](https://leetcode-cn.com/problems/lru-cache/)

LRU是Least Recently Used的缩写，即最近最少使用，是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 t，当须淘汰一个页面时，选择现有页面中其 t 值最大的，即最近最少使用的页面予以淘汰。

支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

就是双向链表 key-vlaue 去除最后一个

### 5.3.33 直线上最多的点数

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

方法 ：暴力查找法

### 5.3.34 逆波兰表达式求值

根据逆波兰表示法，求表达式的值。

有效的运算符包括 +, -, \*, / 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

### 5.3.35 股票买卖

方法1 暴力穷举法

方法2 现在，我们完成了「状态」的穷举，我们开始思考每种「状态」有哪些「选择」，应该如何更新「状态」。只看「持有状态」，可以画个状态转移图。



通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来写一下状态转移方程：

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

max( 选择 rest , 选择 sell )

解释：今天我没有持有股票，有两种可能：

要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；

要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

max( 选择 rest , 选择 buy )

解释：今天我持有着股票，有两种可能：

要么我昨天就持有着股票，然后今天选择 rest，所以我今天还持有着股票；

要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。

这个解释应该很清楚了，如果 buy，就要从利润中减去 prices[i]，如果 sell，就要给利润增加 prices[i]。今天的最大利润就是这两种可能选择中较大的那个。而且注意 k 的限制，我们在选择 buy 的时候，把 k 减小了 1，很好理解吧，当然你也可以在 sell 的时候减 1，一样的。

现在，我们已经完成了动态规划中最困难的一步：状态转移方程。如果之前的内容你都可以理解，那么你已经可以秒杀所有问题了，只要套这个框架就行了。不过还差最后一点点，就是定义 base case，即最简单的情况。

dp[-1][k][0] = 0

解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0 。

dp[-1][k][1] = -infinity

解释：还没开始的时候，是不可能持有股票的，用负无穷表示这种不可能。

dp[i][0][0] = 0

解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0 。

dp[i][0][1] = -infinity

解释：不允许交易的情况下，是不可能持有股票的，用负无穷表示这种不可能。

把上面的状态转移方程总结一下：

base case：

dp[-1][k][0] = dp[i][0][0] = 0

dp[-1][k][1] = dp[i][0][1] = -infinity

状态转移方程：

dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])

### 5.3.36 最短回文串

给定一个字符串 s，你可以通过在字符串前面添加字符将其转换为回文串。找到并返回可以用这种方式转换的最短回文串。

示例 1:

输入: "aacecaaa"

输出: "aaacecaaa"

方法1 暴力法找到最短回文字符串然后反转添加头或者尾

方法2 双指针递归方法

方法3 kmp Knuth-Morris-Pratt 字符串查找算法，简称为 “KMP算法”，常用于在一个文本串S内查找一个模式串P 的出现位置，这个算法由Donald Knuth、Vaughan Pratt、James H. Morris三人于1977年联合发表，故取这3人的姓氏命名此算法。即用前缀和后缀进行比较

或者进行反转进行比较

### 5.3.37 [寻找重复数](https://leetcode-cn.com/problems/find-the-duplicate-number/)

给定一个包含 n + 1 个整数的数组 nums，其数字都在 1 到 n 之间（包括 1 和 n），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

方法1 排完序之后再找

方法2 利用key vvlue

方法3 弗洛伊德的乌龟和兔子（循环检测）



双指针，两个一个指向index，慢慢比较两个值

### 5.3.38 Nim 游戏

你和你的朋友，两个人一起玩 Nim 游戏：桌子上有一堆石头，每次你们轮流拿掉 1 - 3 块石头。 拿掉最后一块石头的人就是获胜者。你作为先手。

你们是聪明人，每一步都是最优解。 编写一个函数，来判断你是否可以在给定石头数量的情况下赢得游戏。

如果堆中石头的数量 nn 不能被 44 整除，那么你总是可以赢得 Nim 游戏的胜利。

推理

让我们考虑一些小例子。显而易见的是，如果石头堆中只有一块、两块、或是三块石头，那么在你的回合，你就可以把全部石子拿走，从而在游戏中取胜。而如果就像题目描述那样，堆中恰好有四块石头，你就会失败。因为在这种情况下不管你取走多少石头，总会为你的对手留下几块，使得他可以在游戏中打败你。因此，要想获胜，在你的回合中，必须避免石头堆中的石子数为 4 的情况。

同样地，如果有五块、六块、或是七块石头，你可以控制自己拿取的石头数，总是恰好给你的对手留下四块石头，使他输掉这场比赛。但是如果石头堆里有八块石头，你就不可避免地会输掉，因为不管你从一堆石头中挑出一块、两块还是三块，你的对手都可以选择三块、两块或一块，以确保在再一次轮到你的时候，你会面对四块石头。

显然，它以相同的模式不断重复 n=4,8,12,16,\dotsn=4,8,12,16,…，基本可以看出是 44 的倍数。

### 5.3.39**[数据流的中位数](https://leetcode-cn.com/problems/find-median-from-data-stream/solution/shu-ju-liu-de-zhong-wei-shu-by-leetcode/)**

方法1 将数字存储在可调整大小的容器中。每次需要输出中间值时，对容器进行排序并输出中间值。

方法2 插入排序 保持容易排序

方法3 用两个堆，保持两个堆平衡

### 5.3.40 判断是否是4的倍数

与01010101比较

### 5.3.41 最大单词长度乘积

给定一个字符串数组 words，找到 length(word[i]) \* length(word[j]) 的最大值，并且这两个单词不含有公共字母。你可以认为每个单词只包含小写字母。如果不存在这样的两个单词，返回 0。

示例 1:

输入: ["abcw","baz","foo","bar","xtfn","abcdef"]

输出: 16

解释: 这两个单词为 "abcw", "xtfn"。

方法1 用00000000000 代表每一个字母 然后或一下

### 5.3.42 拼接最大数

给定长度分别为 m 和 n 的两个数组，其元素由 0-9 构成，表示两个自然数各位上的数字。现在从这两个数组中选出 k (k <= m + n) 个数字拼接成一个新的数，要求从同一个数组中取出的数字保持其在原数组中的相对顺序。

求满足该条件的最大数。结果返回一个表示该最大数的长度为 k 的数组。

示例 1:

输入:

nums1 = [3, 4, 6, 5]

nums2 = [9, 1, 2, 5, 8, 3]

k = 5

输出:

[9, 8, 6, 5, 3]

方法1 依次从两个数组中找到k k-i个数最大数 用一个数组循环从1 到k进行查找比较

### 5.3.43 求一个二叉树和

public int pathSum(TreeNode root, int sum) {

if(root == null) return 0;

return helper(root, sum) + pathSum(root.left, sum) + pathSum(root.right, sum);

}

int helper(TreeNode root, int sum){

if(root == null) return 0;

sum -= root.val;

return (sum == 0 ? 1 : 0) + helper(root.left, sum) + helper(root.right, sum);

}

### 5.3.44 求一个字符串中匹配子符串

思路1：穷举法，依次比较比较值是否相等。

思路2：计算每个字母对应得hashcode值，然后比较hash值。可能存在重复值可以用8421位代替，计算值比较大。

# JSON

## 6.1 介绍

JSON是JavaScript Object Notation的缩写，它是一种数据交换格式。

在JSON出现之前，大家一直用XML来传递数据。因为XML是一种纯文本格式，所以 它适合在网络上交换数据。XML本身不算复杂，但是，加上DTD、XSD、XPath、XSLT等一大堆复杂的规范以后，任何正常的软件开发人员碰到XML都会感觉头大了，最后大家发现，即使你努力钻研几个月，也未必搞得清楚XML的规范。

终于，在2002年的一天，道格拉斯·克罗克福特（Douglas Crockford）同学为了拯救深陷水深火热同时又被某几个巨型软件企业长期愚弄的软件工程师，发明了JSON这种超轻量级的数据交换格式。

道格拉斯同学长期担任雅虎的高级架构师，自然钟情于JavaScript。他设计的JSON实际上是JavaScript的一个子集。在JSON中，一共就这么几种数据类型：

number：和JavaScript的number完全一致；

boolean：就是JavaScript的true或false；

string：就是JavaScript的string；

null：就是JavaScript的null；

array：就是JavaScript的Array表示方式——[]；

object：就是JavaScript的{ ... }表示方式。

以及上面的任意组合。

并且，JSON还定死了字符集必须是UTF-8，表示多语言就没有问题了。为了统一解析，JSON的字符串规定必须用双引号""，Object的键也必须用双引号""。

由于JSON非常简单，很快就风靡Web世界，并且成为ECMA标准。几乎所有编程语言都有解析JSON的库，而在JavaScript中，我们可以直接使用JSON，因为JavaScript内置了JSON的解析。

把任何JavaScript对象变成JSON，就是把这个对象序列化成一个JSON格式的字符串，这样才能够通过网络传递给其他计算机。

如果我们收到一个JSON格式的字符串，只需要把它反序列化成一个JavaScript对象，就可以在JavaScript中直接使用这个对象了。

让我们先把小明这个对象序列化成JSON格式的字符串：

'use strict';

var xiaoming = {

name: '小明',

age: 14,

gender: true,

height: 1.65,

grade: null,

'middle-school': '\"W3C\" Middle School',

skills: ['JavaScript', 'Java', 'Python', 'Lisp']

};

var s = JSON.stringify(xiaoming);

要输出得好看一些，可以加上参数，按缩进输出：JSON.stringify(xiaoming, null, ' ');第二个参数用于控制如何筛选对象的键值，

如果我们只想输出指定的属性，可以传入Array：JSON.stringify(xiaoming, ['name', 'skills'], ' ');

JSON.stringify(xiaoming, convert, ' ');function convert(key, value) {

if (typeof value === 'string') {

return value.toUpperCase();

}

return value;

}

如果我们还想要精确控制如何序列化小明，可以给xiaoming定义一个toJSON()的方法，直接返回JSON应该序列化的数据：

反序列化

拿到一个JSON格式的字符串，我们直接用JSON.parse()把它变成一个JavaScript对象：

JSON.parse()还可以接收一个函数，用来转换解析出的属性：

var obj = JSON.parse('{"name":"小明","age":14}', function (key, value) {

if (key === 'name') {

return value + '同学';

}

return value;

});

console.log(JSON.stringify(obj)); // {name: '小明同学', age: 14}

参考连接：<https://www.liaoxuefeng.com/wiki/1022910821149312/1023021554858080>

# 开源软件学习

## Sylar

### 参考链接

<https://www.bilibili.com/video/av53602631/?from=www.sylar.top>

### 基础学习

#### 协程

协程，又称微线程，纤程。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。

协程可讲异步封装成同步。

#### 日志系统

类 log ：debug append等成员，添加append

类event：成员输出行号文件命等信息

类format:格式子类。

类append：输出器。成员format格式类

基本就是在format格式类中初始化格式字符串，分析%s不同的item类对应输出不同的中得信息，利用tuple(str format type)分析格式串绑定%s对应item类，利用list绑定%s对应item。输出debug日志，调用log类中每个append，再从append调用log成员函数，再调用format函数依次从格式串输出item得函数值（里面得值都保存在event类中）。

Log->append format->items

# 八．常见开源库

## 8.1 yaml

参考链接：

<https://www.runoob.com/w3cnote/yaml-intro.html>

YAML 的意思其实是："Yet Another Markup Language"（仍是一种标记语言）。YAML 的语法和其他高级语言类似，并且可以简单表达清单、散列表，标量等数据形态。它使用空白符号缩进和大量依赖外观的特色，特别适合用来表达或编辑数据结构、各种配置文件、倾印调试内容、文件大纲（例如：许多电子邮件标题格式和YAML非常接近）。

YAML 的配置文件后缀为 .yml，如：runoob.yml 。

基本语法

1. 大小写敏感
2. 使用缩进表示层级关系
3. 缩进不允许使用tab，只允许空格
4. 缩进的空格数不重要，只要相同层级的元素左对齐即可
5. '#'表示注释

YAML 支持以下几种数据类型：

1. 对象：键值对的集合，又称为映射（mapping）/ 哈希（hashes） / 字典（dictionary）
2. 数组：一组按次序排列的值，又称为序列（sequence） / 列表（list）
3. 纯量（scalars）：单个的、不可再分的值

**YAML 对象**

对象键值对使用冒号结构表示 key: value，冒号后面要加一个空格。

也可以使用 key:{key1: value1, key2: value2, ...}

还可以使用缩进表示层级关系；

key:

child-key: value

child-key2: value2

较为复杂的对象格式，可以使用问号加一个空格代表一个复杂的 key，配合一个冒号加一个空格代表一个 value：

?

- complexkey1

- complexkey2

:

- complexvalue1

- complexvalue2

意思即对象的属性是一个数组 [complexkey1,complexkey2]，对应的值也是一个数组 [complexvalue1,complexvalue2] 对象是数组

**YAML 数组**

以 - 开头的行表示构成一个数组：

- A

- B

- C

YAML 支持多维数组，可以使用行内表示：

key: [value1, value2, ...]

数据结构的子成员是一个数组，则可以在该项下面缩进一个空格。

-

- A

- B

- C

一个相对复杂的例子：

companies:

-

id: 1

name: company1

price: 200W

-

id: 2

name: company2

price: 500W

意思是 companies 属性是一个数组，每一个数组元素又是由 id、name、price 三个属性构成。

数组也可以使用流式(flow)的方式表示：

companies: [{id: 1,name: company1,price: 200W},{id: 2,name: company2,price: 500W}]

**复合结构**

数组和对象可以构成复合结构，例：

languages:

- Ruby

- Perl

- Python

websites:

YAML: yaml.org

Ruby: ruby-lang.org

Python: python.org

Perl: use.perl.org

转换为 json 为：

{

languages: [ 'Ruby', 'Perl', 'Python'],

websites: {

YAML: 'yaml.org',

Ruby: 'ruby-lang.org',

Python: 'python.org',

Perl: 'use.perl.org'

}

}

**纯量**

纯量是最基本的，不可再分的值，包括：

字符串

布尔值

整数

浮点数

Null

时间

日期

使用一个例子来快速了解纯量的基本使用：

boolean:

- TRUE #true,True都可以

- FALSE #false，False都可以

float:

- 3.14

- 6.8523015e+5 #可以使用科学计数法

int:

- 123

- 0b1010\_0111\_0100\_1010\_1110 #二进制表示

null:

nodeName: 'node'

parent: ~ #使用~表示null

string:

- 哈哈

- 'Hello world' #可以使用双引号或者单引号包裹特殊字符

- newline

newline2 #字符串可以拆成多行，每一行会被转化成一个空格

date:

- 2018-02-17 #日期必须使用ISO 8601格式，即yyyy-MM-dd

datetime:

- 2018-02-17T15:02:31+08:00 #时间使用ISO 8601格式，时间和日期之间使用T连接，最后使用+代表时区

**引用**

& 锚点和 \* 别名，可以用来引用:

defaults: &defaults

adapter: postgres

host: localhost

development:

database: myapp\_development

<<: \*defaults

test:

database: myapp\_test

<<: \*defaults

相当于:

defaults:

adapter: postgres

host: localhost

development:

database: myapp\_development

adapter: postgres

host: localhost

test:

database: myapp\_test

adapter: postgres

host: localhost

& 用来建立锚点（defaults），<< 表示合并到当前数据，\* 用来引用锚点。

下面是另一个例子:

- &showell Steve

- Clark

- Brian

- Oren

- \*showell

转为 JavaScript 代码如下:

[ 'Steve', 'Clark', 'Brian', 'Oren', 'Steve' ]

# go语言

## 9.1 参考链接

1.runoob

<https://www.runoob.com/go/go-tutorial.html>

2.学习教程

<http://c.biancheng.net/view/9.html>

## 9.1 基础操作

要执行 Go 语言代码可以使用 go run 命令。

使用 go build 命令来生成二进制文件。

需要注意的是 { 不能单独放在一行，所以以下代码在运行时会产生错误.

## 9.2基本语法

### 9.2.1 Go 语言结构

Go 语言的基础组成有以下几个部分：

1. 包声明
2. 引入包
3. 函数
4. 变量
5. 语句 & 表达式
6. 注释

**package** main  
**import** "fmt"  
func main() {  
   */\* 这是我的第一个简单的程序 \*/*  
   fmt.Println("Hello, World!")  
}

让我们来看下以上程序的各个部分：

第一行代码 package main 定义了包名。你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。

下一行 import "fmt" 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。

下一行 func main() 是程序开始执行的函数。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）。

下一行 /\*...\*/ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 /\* 开头，并以 \*/ 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。

下一行 fmt.Println(...) 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。

使用 fmt.Print("hello, world\n") 可以得到相同的结果。

Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。

当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 protected ）。

关于包，根据本地测试得出以下几点：

文件名与包名没有直接关系，不一定要将文件名与包名定成同一个。

文件夹名与包名没有直接关系，并非需要一致。

同一个文件夹下的文件只能有一个包名，否则编译报错。

### 9.2.2 语言基础语法

在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号 ; 结尾，因为这些工作都将由 Go 编译器自动完成。如果你打算将多个语句写在同一行，它们则必须使用 ; 人为区分，但在实际开发中我们并不鼓励这种做法。

字符串连接Go 语言的字符串可以通过 + 实现：

关键字

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| break | default | func | interface | select |
| case | defer | go | map | struct |
| chan | else | goto | package | switch |
| const | fallthrough | if | range | type |
| continue | for | import | return | var |

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| append | bool | byte | cap | close | complex | complex64 | complex128 | uint16 |
| copy | false | float32 | float64 | imag | int | int8 | int16 | uint32 |
| int32 | int64 | iota | len | make | new | nil | panic | uint64 |
| print | println | real | recover | string | true | uint | uint8 | uintptr |

程序一般由关键字、常量、变量、运算符、类型和函数组成。

程序中可能会使用到这些分隔符：括号 ()，中括号 [] 和大括号 {}。

程序中可能会使用到这些标点符号：.、,、;、: 和 …。

### 9.2.3 Go 语言数据类型

Go 语言按类别有以下几种数据类型：

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | 布尔型 布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。 |
| 2 | 数字类型 整型 int 和浮点型 float32、float64，Go 语言支持整型和浮点型数字，并且支持复数，其中位的运算采用补码。 |
| 3 | 字符串类型: 字符串就是一串固定长度的字符连接起来的字符序列。Go 的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用 UTF-8 编码标识 Unicode 文本。 |
| 4 | 派生类型: 包括：  (a) 指针类型（Pointer）  (b) 数组类型  (c) 结构化类型(struct)  (d) Channel 类型  (e) 函数类型  (f) 切片类型  (g) 接口类型（interface）  (h) Map 类型 |

数字类型

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | uint8 无符号 8 位整型 (0 到 255) |
| 2 | uint16 无符号 16 位整型 (0 到 65535) |
| 3 | uint32 无符号 32 位整型 (0 到 4294967295) |
| 4 | uint64 无符号 64 位整型 (0 到 18446744073709551615) |
| 5 | int8 有符号 8 位整型 (-128 到 127) |
| 6 | int16 有符号 16 位整型 (-32768 到 32767) |
| 7 | int32 有符号 32 位整型 (-2147483648 到 2147483647) |
| 8 | int64 有符号 64 位整型 (-9223372036854775808 到 9223372036854775807) |

浮点型

|  |  |
| --- | --- |
| 序号 | 类型和描述 |
| 1 | float32 IEEE-754 32位浮点型数 |
| 2 | float64 IEEE-754 64位浮点型数 |
| 3 | complex64 32 位实数和虚数 |
| 4 | complex128 64 位实数和虚数 |

其他数字类型

以下列出了其他更多的数字类型：

|  |  |
| --- | --- |
| **序号** | **类型和描述** |
| 1 | **byte** 类似 uint8 |
| 2 | **rune** 类似 int32 |
| 3 | **uint** 32 或 64 位 |
| 4 | **int** 与 uint 一样大小 |
| 5 | **uintptr** 无符号整型，用于存放一个指针 |

### 9.2.4 Go 语言变量

声明变量的一般形式是使用 var 关键字：

var identifier type

可以一次声明多个变量：

var identifier1, identifier2 type

var a string = "Runoob"

fmt.Println(a)

var b, c int = 1, 2

fmt.Println(b, c)

**变量声明**

第一种，指定变量类型，如果没有初始化，则变量默认为零值。

数值类型（包括complex64/128）为 0

布尔类型为 false

字符串为 ""（空字符串）

以下几种类型为 nil：

var a \*int

var a []int

var a map[string] int

var a chan int

var a func(string) int

var a error // error 是接口

第二种，根据值自行判定变量类型。

第三种，省略 var, 注意 := 左侧如果没有声明新的变量，就产生编译错误，格式：

v\_name := value

var intVal int

intVal :=1 // 这时候会产生编译错误

intVal,intVal1 := 1,2 // 此时不会产生编译错误，因为有声明新的变量，因为 := 是一个声明语句

// 这种因式分解关键字的写法一般用于声明全局变量

var (

vname1 v\_type1

vname2 v\_type2

)

简短形式，使用 := 赋值操作符.这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

你声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误。

### 9.2.5 Go 语言常量

常量的定义格式：

const identifier [type] = value

你可以省略类型说明符 [type]，因为编译器可以根据变量的值来推断其类型。

显式类型定义： const b string = "abc"

隐式类型定义： const b = "abc"

常量还可以用作枚举：

const (

Unknown = 0

Female = 1

Male = 2

)

常量可以用len(), cap(), unsafe.Sizeof()函数计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过.

iota，特殊常量，可以认为是一个可以被编译器修改的常量。iota 在 const关键字出现时将被重置为 0(const 内部的第一行之前)，const 中每新增一行常量声明将使 iota 计数一次(iota 可理解为 const 语句块中的行索引)。

const (

a = iota //0

b //1

c //2

d = "ha" //独立值，iota += 1

e //"ha" iota += 1

f = 100 //iota +=1

g //100 iota +=1

h = iota //7,恢复计数

i //8

)

### 9.2.6 Go 语言运算符

Go 语言内置的运算符有：

算术运算符

关系运算符

逻辑运算符

位运算符

赋值运算符

其他运算符

**运算符优先级**

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

|  |  |
| --- | --- |
| 优先级 | 运算符 |
| 5 | \* / % << >> & &^ |
| 4 | + - | ^ |
| 3 | == != < <= > >= |
| 2 | && |
| 1 | || |

指针变量 \* 和地址值 & 的区别：指针变量保存的是一个地址值，会分配独立的内存来存储一个整型数字。当变量前面有 \* 标识时，才等同于 & 的用法，否则会直接输出一个整型数字。

### 9.2.7 Go 语言条件语句

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

}

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

} else {

/\* 在布尔表达式为 false 时执行 \*/

}

switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，从上至下逐一测试，直到匹配为止。

switch 语句执行的过程从上至下，直到找到匹配项，匹配项后面也不需要再加 break。

switch 默认情况下 case 最后自带 break 语句，匹配成功后就不会执行其他 case，如果我们需要执行后面的 case，可以使用 fallthrough 。

select 是 Go 中的一个控制结构，类似于用于通信的 switch 语句。每个 case 必须是一个通信操作，要么是发送要么是接收。

select 随机执行一个可运行的 case。如果没有 case 可运行，它将阻塞，直到有 case 可运行。一个默认的子句应该总是可运行的。配合<-使用

### 9.2.8 Go 语言循环语句

和 C 语言的 for 一样：

for init; condition; post { }

和 C 的 while 一样：

for condition { }

和 C 的 for(;;) 一样：

for { }

for i := 0; i <= 10; i++ {

sum += i

}

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

For-each range 循环

这种格式的循环可以对字符串、数组、切片等进行迭代输出元素。

for key, value := range oldMap {

newMap[key] = value

}

GO 语言支持以下几种循环控制语句：

| 控制语句 | 描述 |
| --- | --- |
| [break 语句](https://www.runoob.com/go/go-break-statement.html) | 经常用于中断当前 for 循环或跳出 switch 语句 |
| [continue 语句](https://www.runoob.com/go/go-continue-statement.html) | 跳过当前循环的剩余语句，然后继续进行下一轮循环。 |
| [goto 语句](https://www.runoob.com/go/go-goto-statement.html) | 将控制转移到被标记的语句。 |

### 9.2.9 Go 语言函数

Go 语言函数定义格式如下：

func function\_name( [parameter list] ) [return\_types] {

函数体

}

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

以下实例中，我们创建了函数 getSequence() ，返回另外一个函数。

func getSequence() func() int {

i:=0

return func() int {

i+=1

return i

}

}

/\* nextNumber 为一个函数，函数 i 为 0 \*/

nextNumber := getSequence()

/\* 调用 nextNumber 函数，i 变量自增 1 并返回 \*/

fmt.Println(nextNumber())

fmt.Println(nextNumber())

fmt.Println(nextNumber())

/\* 创建新的函数 nextNumber1，并查看结果 \*/

nextNumber1 := getSequence()

fmt.Println(nextNumber1())

fmt.Println(nextNumber1())

1

2

3

1

2

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

func (variable\_name variable\_data\_type) function\_name() [return\_type]{

/\* 函数体\*/

}

/\* 定义结构体 \*/

type Circle struct {

radius float64

}

func main() {

var c1 Circle

c1.radius = 10.00

fmt.Println("圆的面积 = ", c1.getArea())

}

//该 method 属于 Circle 类型对象中的方法

func (c Circle) getArea() float64 {

//c.radius 即为 Circle 类型对象中的属性

return 3.14 \* c.radius \* c.radius

}

### 9.2.10 Go 语言数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

var variable\_name [SIZE] variable\_type

var balance [10] float32

数组初始化：

var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}

var ptr [MAX]\*int;

### 9.2.11 Go 语言结构体

type struct\_variable\_type struct {

member definition

member definition

...

member definition

}

type Books struct {

title string

author string

subject string

book\_id int

}

variable\_name := structure\_variable\_type {value1, value2...valuen}或

variable\_name := structure\_variable\_type { key1: value1, key2: value2..., keyn: valuen}

// 创建一个新的结构体

fmt.Println(Books{"Go 语言", "www.runoob.com", "Go 语言教程", 6495407})

// 也可以使用 key => value 格式

fmt.Println(Books{title: "Go 语言", author: "www.runoob.com", subject: "Go 语言教程", book\_id: 6495407})

### 9.2.12Go 语言切片(Slice)

Go 语言切片是对数组的抽象。

Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

你可以声明一个未指定大小的数组来定义切片：

var identifier []type

切片不需要说明长度。

或使用make()函数来创建切片:

var slice1 []type = make([]type, len)

也可以简写为

slice1 := make([]type, len)

也可以指定容量，其中capacity为可选参数。

make([]T, length, capacity)

这里 len 是数组的长度并且也是切片的初始长度。

:=[] int {1,2,3 }

直接初始化切片，[]表示是切片类型，{1,2,3}初始化值依次是1,2,3.其cap=len=3

s := arr[:]

初始化切片s,是数组arr的引用

s := arr[startIndex:endIndex]

将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片

s := arr[startIndex:]

默认 endIndex 时将表示一直到arr的最后一个元素

s := arr[:endIndex]

默认 startIndex 时将表示从arr的第一个元素开始

s1 := s[startIndex:endIndex]

通过切片s初始化切片s1

s :=make([]int,len,cap)

通过内置函数make()初始化切片s,[]int 标识为其元素类型为int的切片

len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

append() 和 copy() 函数

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。

下面的代码描述了从拷贝切片的 copy 方法和向切片追加新元素的 append 方法。

### 9.2.13 Go 语言Map(集合)

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。

Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

/\* 声明变量，默认 map 是 nil \*/

var map\_variable map[key\_data\_type]value\_data\_type

/\* 使用 make 函数 \*/

map\_variable := make(map[key\_data\_type]value\_data\_type)

var countryCapitalMap map[string]string /\*创建集合 \*/

countryCapitalMap = make(map[string]string)

delete() 函数用于删除集合的元素, 参数为 map 和其对应的 key。

delete(countryCapitalMap, "France")

### 9.2.14 Go 语言接口

/\* 定义接口 \*/

type interface\_name interface {

method\_name1 [return\_type]

method\_name2 [return\_type]

method\_name3 [return\_type]

...

method\_namen [return\_type]

}

/\* 定义结构体 \*/

type struct\_name struct {

/\* variables \*/

}

/\* 实现接口方法 \*/

func (struct\_name\_variable struct\_name) method\_name1() [return\_type] {

/\* 方法实现 \*/

}

...

func (struct\_name\_variable struct\_name) method\_namen() [return\_type] {

/\* 方法实现\*/

}

### 9.2.15 Go 并发

**goroutine**

Go 语言支持并发，我们只需要通过 go 关键字来开启 goroutine 即可。

goroutine 是轻量级线程，goroutine 的调度是由 Golang 运行时进行管理的。

goroutine 语法格式：

go 函数名( 参数列表 )

例如：

go f(x, y, z)

Go 允许使用 go 语句开启一个新的运行期线程， 即 goroutine，以一个不同的、新创建的 goroutine 来执行一个函数。 同一个程序中的所有 goroutine 共享同一个地址空间。

**通道（channel）**

通道（channel）是用来传递数据的一个数据结构。

通道可用于两个 goroutine 之间通过传递一个指定类型的值来同步运行和通讯。操作符 <- 用于指定通道的方向，发送或接收。如果未指定方向，则为双向通道。

ch <- v // 把 v 发送到通道 ch

v := <-ch // 从 ch 接收数据

// 并把值赋给 v

声明一个通道很简单，我们使用chan关键字即可，通道在使用前必须先创建：

ch := make(chan int)

注意：默认情况下，通道是不带缓冲区的。发送端发送数据，同时必须有接收端相应的接收数据.

通道可以设置缓冲区，通过 make 的第二个参数指定缓冲区大小：

ch := make(chan int, 100)

带缓冲区的通道允许发送端的数据发送和接收端的数据获取处于异步状态，就是说发送端发送的数据可以放在缓冲区里面，可以等待接收端去获取数据，而不是立刻需要接收端去获取数据。

不过由于缓冲区的大小是有限的，所以还是必须有接收端来接收数据的，否则缓冲区一满，数据发送端就无法再发送数据了。

注意：如果通道不带缓冲，发送方会阻塞直到接收方从通道中接收了值。如果通道带缓冲，发送方则会阻塞直到发送的值被拷贝到缓冲区内；如果缓冲区已满，则意味着需要等待直到某个接收方获取到一个值。接收方在有值可以接收之前会一直阻塞。

Go 通过 range 关键字来实现遍历读取到的数据，类似于与数组或切片。格式如下：

v, ok := <-ch

如果通道接收不到数据后 ok 就为 false，这时通道就可以使用 close() 函数来关闭。

## 9.3 go基础补充

**变量定义**

定义三个类型都是"type"的变量,并且分别初始化为相应的值vname1为v1，vname2为v2，vname3为v3

var vname1, vname2, vname3 type= v1, v2, v3

定义三个变量，它们分别初始化为相应的值vname1为v1，vname2为v2，vname3为v3然后Go会根据其相应值的类型来帮你初始化它们

var vname1, vname2, vname3 = v1, v2, v3

定义三个变量，它们分别初始化为相应的值vname1为v1，vname2为v2，vname3为v3编译器会根据初始化的值自动推导出相应的类型

vname1, vname2, vname3 := v1, v2, v3

有一个限制，那就是它只能用在函数内部；在函数外部使用则会无法编译通过，所以一般用var方式来定义全局变量。

\_（下划线）是个特殊的变量名，任何赋予它的值都会被丢弃。

**错误类型**

err := errors.New("emit macho dwarf: elf header corrupted")

if err != nil {

fmt.Print(err)

}

**接口**

Go之所以会那么简洁，是因为它有一些默认的行为：

大写字母开头的变量是可导出的，也就是其它包可以读取的，是公有变量；小写字母开头的就是不可导出的，是私有变量。

大写字母开头的函数也是一样，相当于class中的带public关键词的公有函数；小写字母开头的就是有private关键词的私有函数。

**make、new操作**

make用于内建类型（map、slice 和channel）的内存分配。new用于各种类型的内存分配。

内建函数new本质上说跟其它语言中的同名函数功能一样：new(T)分配了零值填充的T类型的内存空间，并且返回其地址，即一个\*T类型的值。用Go的术语说，它返回了一个指针，指向新分配的类型T的零值。有一点非常重要：

new返回指针。

内建函数make(T, args)与new(T)有着不同的功能，make只能创建slice、map和channel，并且返回一个有初始值(非零)的T类型，而不是\*T。

**变参**

Go函数支持变参。接受变参的函数是有着不定数量的参数的。为了做到这点，首先需要定义函数使其接受变参：

func myfunc(arg ...int) {}

arg ...int告诉Go这个函数接受不定数量的参数。注意，这些参数的类型全部是int。在函数体中，变量arg是一个int的slice：

for \_, n := range arg {

fmt.Printf("And the number is: %d\n", n)

}

**defer**

Go语言中有种不错的设计，即延迟（defer）语句，你可以在函数中添加多个defer语句。当函数执行到最后时，这些defer语句会按照逆序执行，最后该函数返回。

如果有很多调用defer，那么defer是采用后进先出模式，所以如下代码会输出4 3 2 1 0

for i := 0; i < 5; i++ {

defer fmt.Printf("%d ", i)

}

**函数作为值、类型--函数指针**

在Go中函数也是一种变量，我们可以通过type来定义它，它的类型就是所有拥有相同的参数，相同的返回值的一种类型

type typeName func(input1 inputType1 , input2 inputType2 [, ...]) (result1 resultType1 [, ...])

**Panic和Recover**

Go没有像Java那样的异常机制，它不能抛出异常，而是使用了panic和recover机制。一定要记住，你应当把它作为最后的手段来使用，也就是说，你的代码中应当没有，或者很少有panic的东西。这是个强大的工具，请明智地使用它。那么，我们应该如何使用它呢？

Panic

是一个内建函数，可以中断原有的控制流程，进入一个panic状态中。当函数F调用panic，函数F的执行被中断，但是F中的延迟函数会正常执行，然后F返回到调用它的地方。在调用的地方，F的行为就像调用了panic。这一过程继续向上，直到发生panic的goroutine中所有调用的函数返回，此时程序退出。panic可以直接调用panic产生。也可以由运行时错误产生，例如访问越界的数组。

Recover

是一个内建的函数，可以让进入panic状态的goroutine恢复过来。recover仅在延迟函数中有效。在正常的执行过程中，调用recover会返回nil，并且没有其它任何效果。如果当前的goroutine陷入panic状态，调用recover可以捕获到panic的输入值，并且恢复正常的执行。

**面向对象**

method

method的语法如下：

func (r ReceiverType) funcName(parameters) (results)

type Rectangle struct {

width, height float64

}

func (r Rectangle) area() float64 {

return r.width\*r.height

}

也就是说：

如果一个method的receiver是\*T,你可以在一个T类型的实例变量V上面调用这个method，而不需要&V去调用这个method

类似的

如果一个method的receiver是T，你可以在一个\*T类型的变量P上面调用这个method，而不需要 \*P去调用这个method

**interface**

简单的说，interface是一组method签名的组合，我们通过interface来定义对象的一组行为。

这样Student实现了三个方法：SayHi、Sing、BorrowMoney；而Employee实现了SayHi、Sing、SpendSalary。

上面这些方法的组合称为interface(被对象Student和Employee实现)。例如Student和Employee都实现了interface：SayHi和Sing，也就是这两个对象是该interface类型。而Employee没有实现这个interface：SayHi、Sing和BorrowMoney，因为Employee没有实现BorrowMoney这个方法。interface类型定义了一组方法，如果某个对象实现了某个接口的所有方法，则此对象就实现了此接口。

// 定义interfacetype Men interface {

SayHi()

Sing(lyrics string)

Guzzle(beerStein string)

}

type YoungChap interface {

SayHi()

Sing(song string)

BorrowMoney(amount float32)

}

type ElderlyGent interface {

SayHi()

Sing(song string)

SpendSalary(amount float32)

}

通过上面的代码我们可以知道，interface可以被任意的对象实现。我们看到上面的Men interface被Human、Student和Employee实现。同理，一个对象可以实现任意多个interface，例如上面的Student实现了Men和YoungChap两个interface。

空interface(interface{})不包含任何的method，有点类似于C语言的void\*类型。interface的变量里面可以存储任意类型的数值(该类型实现了interface)。

value, ok = element.(T)，这里value就是变量的值，ok是一个bool类型，element是interface变量，T是断言的类型。

**goroutine**

goroutine是Go并行设计的核心。goroutine说到底其实就是协程，但是它比线程更小，十几个goroutine可能体现在底层就是五六个线程，Go语言内部帮你实现了这些goroutine之间的内存共享。执行goroutine只需极少的栈内存(大概是4~5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。goroutine比thread更易用、更高效、更轻便。

go hello(a, b, c)

通过关键字go就启动了一个goroutine。

runtime.Gosched()表示让CPU把时间片让给别人,下次某个时候继续恢复执行该goroutine。

**channels**

channel可以与Unix shell 中的双向管道做类比：可以通过它发送或者接收值。这些值只能是特定的类型：channel类型。定义一个channel时，也需要定义发送到channel的值的类型。注意，必须使用make 创建channel：

ci := make(chan int)

cs := make(chan string)

cf := make(chan interface{})

channel通过操作符<-来接收和发送数据

ch <- v // 发送v到channel ch.

v := <-ch // 从ch中接收数据，并赋值给v

func sum(a []int, c chan int) {

total := 0

for \_, v := range a {

total += v

}

c <- total // send total to c

}

func main() {

a := []int{7, 2, 8, -9, 4, 0}

c := make(chan int)

go sum(a[:len(a)/2], c)

go sum(a[len(a)/2:], c)

x, y := <-c, <-c // receive from c

fmt.Println(x, y, x + y)

}

默认情况下，channel接收和发送数据都是阻塞的，除非另一端已经准备好，这样就使得Goroutines同步变的更加的简单，而不需要显式的lock。所谓阻塞，也就是如果读取（value := <-ch）它将会被阻塞，直到有数据接收。其次，任何发送（ch<-5）将会被阻塞，直到数据被读出。无缓冲channel是在多个goroutine之间同步很棒的工具。

**Buffered Channels**

ch:= make(chan bool, 4)，创建了可以存储4个元素的bool 型channel。在这个channel 中，前4个元素可以无阻塞的写入。当写入第5个元素时，代码将会阻塞，直到其他goroutine从channel 中读取一些元素，腾出空间。

for i := range c能够不断的读取channel里面的数据，直到该channel被显式的关闭。上面代码我们看到可以显式的关闭channel，生产者通过内置函数close关闭channel。关闭channel之后就无法再发送任何数据了，在消费方可以通过语法v, ok := <-ch测试channel是否被关闭。如果ok返回false，那么说明channel已经没有任何数据并且已经被关闭。

用select来设置超时

通过select可以监听channel上的数据流动。

select默认是阻塞的，只有当监听的channel中有发送或接收可以进行时才会运行，当多个channel都准备好的时候，select是随机的选择一个执行的。

runtime包中有几个处理goroutine的函数：

Goexit

退出当前执行的goroutine，但是defer函数还会继续调用

Gosched

让出当前goroutine的执行权限，调度器安排其他等待的任务运行，并在下次某个时候从该位置恢复执行。

NumCPU

返回 CPU 核数量

NumGoroutine

返回正在执行和排队的任务总数

GOMAXPROCS

用来设置可以并行计算的CPU核数的最大值，并返回之前的值。

## 9.4 搭建web

### 9.4.1 http相关

Web服务器的工作原理可以简单地归纳为：

* 客户机通过TCP/IP协议建立到服务器的TCP连接
* 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档
* 服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端
* 客户机与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

一个简单的HTTP事务就是这样实现的，看起来很复杂，原理其实是挺简单的。需要注意的是客户机与服务器之间的通信是非持久连接的，也就是当服务器发送了应答后就与客户机断开连接，等待下一次请求。

URL(Uniform Resource Locator)是“统一资源定位符”的英文缩写，用于描述一个网络上的资源, 基本格式如下

scheme://host[:port#]/path/.../[?query-string][#anchor]

scheme 指定底层使用的协议(例如：http, https, ftp)

host HTTP服务器的IP地址或者域名

port# HTTP服务器的默认端口是80，这种情况下端口号可以省略。如果使用了别的端口，必须指明，例如 http://www.cnblogs.com:8080/

path 访问资源的路径

query-string 发送给http服务器的数据

anchor 锚

HTTP协议是无状态的，同一个客户端的这次请求和上次请求是没有对应关系的，对HTTP服务器来说，它并不知道这两个请求是否来自同一个客户端。为了解决这个问题， Web程序引入了Cookie机制来维护连接的可持续状态。

GET提交的数据会放在URL之后，以?分割URL和传输数据，参数之间以&相连，如EditPosts.aspx?name=test1&id=123456。POST方法是把提交的数据放在HTTP包的body中。

GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST方法提交的数据没有限制。

HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（面对无连接）。从HTTP/1.1起，默认都开启了Keep-Alive保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的TCP连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同服务器软件（如Apache）中设置这个时间。

### 9.4.2 go相关

fmt.Println("username:", template.HTMLEscapeString(r.Form.Get("username"))) //输出到服务器端

fmt.Println("password:", template.HTMLEscapeString(r.Form.Get("password")))

template.HTMLEscape(w, []byte(r.Form.Get("username"))) //输出到客户端

import "text/template"

...

t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)

err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwned')</script>")

输出

Hello, <script>alert('you have been pwned')</script>!

使用template.HTML类型

import "html/template"

...

t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)

err = t.ExecuteTemplate(out, "T", template.HTML("<script>alert('you have been pwned')</script>"))

输出

Hello, <script>alert('you have been pwned')</script>!

要使表单能够上传文件，首先第一步就是要添加form的enctype属性，enctype属性有如下三种情况:

application/x-www-form-urlencoded 表示在发送前编码所有字符（默认）

multipart/form-data 不对字符编码。在使用包含文件上传控件的表单时，必须使用该值。

text/plain 空格转换为 "+" 加号，但不对特殊字符编码。

### 9.4.3 数据库

Mysql

go get -u github.com/go-sql-driver/mysql

[username[:password]@][protocol[(address)]]/dbname[?param1=value1&...&paramN=valueN]

username:password@protocol(address)/dbname?param=value

Conn是一个数据库连接的接口定义，他定义了一系列方法，这个Conn只能应用在一个goroutine里面，不能使用在多个goroutine里面

Stmt是一种准备好的状态，和Conn相关联，而且只能应用于一个goroutine中，不能应用于多个goroutine。

打开数据库格式

user@unix(/path/to/socket)/dbname?charset=utf8

user:password@tcp(localhost:5555)/dbname?charset=utf8

user:password@/dbname

user:password@tcp([de:ad:be:ef::ca:fe]:80)/dbname

db, err := sql.Open("mysql", "astaxie:astaxie@/test?charset=utf8")

//插入数据

stmt, err := db.Prepare("INSERT INTO userinfo SET username=?,department=?,created=?")

res, err := stmt.Exec("astaxie", "研发部门", "2012-12-09")

id, err := res.LastInsertId()

stmt, err = db.Prepare("update userinfo set username=? where uid=?")

res, err = stmt.Exec("astaxieupdate", id)

affect, err := res.RowsAffected()

fmt.Println(affect)

//查询数据

rows, err := db.Query("SELECT \* FROM userinfo")

for rows.Next() {

var uid int

var username string

var department string

var created string

err = rows.Scan(&uid, &username, &department, &created)

checkErr(err)

fmt.Println(uid)

fmt.Println(username)

fmt.Println(department)

fmt.Println(created)

}

//删除数据

stmt, err = db.Prepare("delete from userinfo where uid=?")

res, err = stmt.Exec(id)

affect, err = res.RowsAffected()

fmt.Println(affect)

db.Close()

db.Prepare()函数用来返回准备要执行的sql操作，然后返回准备完毕的执行状态。

stmt.Exec()函数用来执行stmt准备好的SQL语句

db.Query()函数用来直接执行Sql返回Rows结果。

### 9.4.4 session和数据存储

cookie，简而言之就是在本地计算机保存一些用户操作的历史信息（当然包括登录信息），并在用户再次访问该站点时浏览器通过HTTP协议将本地cookie内容发送给服务器，从而完成验证，或继续上一步操作。

session，简而言之就是在服务器上保存用户操作的历史信息。服务器使用session id来标识session，session id由服务器负责产生，保证随机性与唯一性，相当于一个随机密钥，避免在握手或传输中暴露用户真实密码。但该方式下，仍然需要将发送请求的客户端与session进行对应，所以可以借助cookie机制来获取客户端的标识（即session id），也可以通过GET方式将id提交给服务器。

cookie是有时间限制的，根据生命期不同分成两种：会话cookie和持久cookie；如果不设置过期时间，则表示这个cookie的生命周期为从创建到浏览器关闭为止，只要关闭浏览器窗口，cookie就消失了。这种生命期为浏览会话期的cookie被称为会话cookie。会话cookie一般不保存在硬盘上而是保存在内存里。如果设置了过期时间(setMaxAge(606024))，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie依然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏览器进程间共享，比如两个IE窗口。而对于保存在内存的cookie，不同的浏览器有不同的处理方式。

如何发送这个session的唯一标识这一步上。考虑到HTTP协议的定义，数据无非可以放到请求行、头域或Body里，所以一般来说会有两种常用的方式：cookie和URL重写。

1. Cookie 服务端通过设置Set-cookie头就可以将session的标识符传送到客户端，而客户端此后的每一次请求都会带上这个标识符，另外一般包含session信息的cookie会将失效时间设置为0(会话cookie)，即浏览器进程有效时间。至于浏览器怎么处理这个0，每个浏览器都有自己的方案，但差别都不会太大(一般体现在新建浏览器窗口的时候)；
2. URL重写 所谓URL重写，就是在返回给用户的页面里的所有的URL后面追加session标识符，这样用户在收到响应之后，无论点击响应页面里的哪个链接或提交表单，都会自动带上session标识符，从而就实现了会话的保持。虽然这种做法比较麻烦，但是，如果客户端禁用了cookie的话，此种方案将会是首选。

对Session的处理基本就 设置值、读取值、删除值以及获取当前sessionID这四个操作，所以我们的Session接口也就实现这四个操作。

如何有效的防止session劫持呢？

其中一个解决方案就是sessionID的值只允许cookie设置，而不是通过URL重置方式设置，同时设置cookie的httponly为true,这个属性是设置是否可通过客户端脚本访问这个设置的cookie，第一这个可以防止这个cookie被XSS读取从而引起session劫持，第二cookie设置不会像URL重置方式那么容易获取sessionID。

第二步就是在每个请求里面加上token，实现类似前面章节里面讲的防止form重复递交类似的功能，我们在每个请求里面加上一个隐藏的token，然后每次验证这个token，从而保证用户的请求都是唯一性。

还有一个解决方案就是，我们给session额外设置一个创建时间的值，一旦过了一定的时间，我们销毁这个sessionID，重新生成新的session，这样可以一定程度上防止session劫持的问题。

### 9.4.5 文本处理

如果在我们不知道他的结构的情况下，我们把他解析到interface{}里面

var f interface{}err := json.Unmarshal(b, &f)

这个时候f里面存储了一个map类型，他们的key是string，值存储在空的interface{}里

f = map[string]interface{}{

"Name": "Wednesday",

"Age": 6,

"Parents": []interface{}{

"Gomez",

"Morticia",

},

}

那么如何来访问这些数据呢？通过断言的方式：

m := f.(map[string]interface{})

### 9.4.6 模板处理

我们使用template包来进行模板处理，使用类似Parse、ParseFile、Execute等方法从文件或者字符串加载模板

Go语言的模板通过{{}}来包含需要在渲染时被替换的字段，{{.}}表示当前的对象

模板中含有了未导出的字段，那么就会报错

# SQL数据库

## 10.1数据库概述

数据库按照数据结构来组织、存储和管理数据，实际上，数据库一共有三种模型：

层次模型

网状模型

关系模型

数据类型

对于一个关系表，除了定义每一列的名称外，还需要定义每一列的数据类型。关系数据库支持的标准数据类型包括数值、字符串、时间等：

| 名称 | 类型 | 说明 |
| --- | --- | --- |
| INT | 整型 | 4字节整数类型，范围约+/-21亿 |
| BIGINT | 长整型 | 8字节整数类型，范围约+/-922亿亿 |
| REAL | 浮点型 | 4字节浮点数，范围约+/-1038 |
| DOUBLE | 浮点型 | 8字节浮点数，范围约+/-10308 |
| DECIMAL(M,N) | 高精度小数 | 由用户指定精度的小数，例如，DECIMAL(20,10)表示一共20位，其中小数10位，通常用于财务计算 |
| CHAR(N) | 定长字符串 | 存储指定长度的字符串，例如，CHAR(100)总是存储100个字符的字符串 |
| VARCHAR(N) | 变长字符串 | 存储可变长度的字符串，例如，VARCHAR(100)可以存储0~100个字符的字符串 |
| BOOLEAN | 布尔类型 | 存储True或者False |
| DATE | 日期类型 | 存储日期，例如，2018-06-22 |
| TIME | 时间类型 | 存储时间，例如，12:20:59 |
| DATETIME | 日期和时间类型 | 存储日期+时间，例如，2018-06-22 12:20:59 |

主流关系数据库

目前，主流的关系数据库主要分为以下几类：

商用数据库，例如：Oracle，SQL Server，DB2等；

开源数据库，例如：MySQL，PostgreSQL等；

桌面数据库，以微软Access为代表，适合桌面应用程序使用；

嵌入式数据库，以Sqlite为代表，适合手机应用和桌面程序。

虽然SQL已经被ANSI组织定义为标准，不幸地是，各个不同的数据库对标准的SQL支持不太一致。并且，大部分数据库都在标准的SQL上做了扩展。也就是说，如果只使用标准SQL，理论上所有数据库都可以支持，但如果使用某个特定数据库的扩展SQL，换一个数据库就不能执行了。例如，Oracle把自己扩展的SQL称为PL/SQL，Microsoft把自己扩展的SQL称为T-SQL。

SQL语言关键字不区分大小写！！！但是，针对不同的数据库，对于表名和列名，有的数据库区分大小写，有的数据库不区分大小写。同一个数据库，有的在Linux上区分大小写，有的在Windows上不区分大小写。

所以，本教程约定：SQL关键字总是大写，以示突出，表名和列名均使用小写。

## 10.2数据库基础

关系数据库是建立在关系模型上的。而关系模型本质上就是若干个存储数据的二维表，可以把它们看作很多Excel表。

表的每一行称为记录（Record），记录是一个逻辑意义上的数据。

表的每一列称为字段（Column），同一个表的每一行记录都拥有相同的若干字段。

字段定义了数据类型（整型、浮点型、字符串、日期等），以及是否允许为NULL。注意NULL表示字段数据不存在。一个整型字段如果为NULL不表示它的值为0，同样的，一个字符串型字段为NULL也不表示它的值为空串''。

通常情况下，字段应该避免允许为NULL。不允许为NULL可以简化查询条件，加快查询速度，也利于应用程序读取数据后无需判断是否为NULL。

主键

在关系数据库中，关系是通过主键和外键来维护的。

对于关系表，有个很重要的约束，就是任意两条记录不能重复。不能重复不是指两条记录不完全相同，而是指能够通过某个字段唯一区分出不同的记录，这个字段被称为主键。

所以，选取主键的一个基本原则是：不使用任何业务相关的字段作为主键。

作为主键最好是完全业务无关的字段，我们一般把这个字段命名为id。常见的可作为id字段的类型有：

自增整数类型：数据库会在插入数据时自动为每一条记录分配一个自增整数，这样我们就完全不用担心主键重复，也不用自己预先生成主键；

全局唯一GUID类型：使用一种全局唯一的字符串作为主键，类似8f55d96b-8acc-4636-8cb8-76bf8abc2f57。GUID算法通过网卡MAC地址、时间戳和随机数保证任意计算机在任意时间生成的字符串都是不同的，大部分编程语言都内置了GUID算法，可以自己预算出主键。

联合主键

关系数据库实际上还允许通过多个字段唯一标识记录，即两个或更多的字段都设置为主键，这种主键被称为联合主键。

对于联合主键，允许一列有重复，只要不是所有主键列都重复即可：

没有必要的情况下，我们尽量不使用联合主键，因为它给关系表带来了复杂度的上升。

外键

由于一个班级可以有多个学生，在关系模型中，这两个表的关系可以称为“一对多”，即一个classes的记录可以对应多个students表的记录。

为了表达这种一对多的关系，我们需要在students表中加入一列class\_id，让它的值与classes表的某条记录相对应

在students表中，通过class\_id的字段，可以把数据与另一张表关联起来，这种列称为外键。

外键并不是通过列名实现的，而是通过定义外键约束实现的：

ALTER TABLE students

ADD CONSTRAINT fk\_class\_id

FOREIGN KEY (class\_id)

REFERENCES classes (id);

其中，外键约束的名称fk\_class\_id可以任意，FOREIGN KEY (class\_id)指定了class\_id作为外键，REFERENCES classes (id)指定了这个外键将关联到classes表的id列（即classes表的主键）。

通过定义外键约束，关系数据库可以保证无法插入无效的数据。即如果classes表不存在id=99的记录，students表就无法插入class\_id=99的记录。

由于外键约束会降低数据库的性能，大部分互联网应用程序为了追求速度，并不设置外键约束，而是仅靠应用程序自身来保证逻辑的正确性。这种情况下，class\_id仅仅是一个普通的列，只是它起到了外键的作用而已。

要删除一个外键约束，也是通过ALTER TABLE实现的：

ALTER TABLE students

DROP FOREIGN KEY fk\_class\_id;

注意：删除外键约束并没有删除外键这一列。删除列是通过DROP COLUMN ...实现的。

多对多关系实际上是通过两个一对多关系实现的，即通过一个中间表，关联两个一对多关系，就形成了多对多关系：即用一个 表结合两个表形成

索引

在关系数据库中，如果有上万甚至上亿条记录，在查找记录的时候，想要获得非常快的速度，就需要使用索引。

索引是关系数据库中对某一列或多个列的值进行预排序的数据结构。通过使用索引，可以让数据库系统不必扫描整个表，而是直接定位到符合条件的记录，这样就大大加快了查询速度。

如果要经常根据score列进行查询，就可以对score列创建索引：

ALTER TABLE students

ADD INDEX idx\_score (score);

使用ADD INDEX idx\_score (score)就创建了一个名称为idx\_score，使用列score的索引。索引名称是任意的，索引如果有多列，可以在括号里依次写上，例如：

ALTER TABLE students

ADD INDEX idx\_name\_score (name, score);

可以对一张表创建多个索引。索引的优点是提高了查询效率，缺点是在插入、更新和删除记录时，需要同时修改索引，因此，索引越多，插入、更新和删除记录的速度就越慢。

对于主键，关系数据库会自动对其创建主键索引。使用主键索引的效率是最高的，因为主键会保证绝对唯一。

唯一索引

ALTER TABLE students

ADD UNIQUE INDEX uni\_name (name);

通过UNIQUE关键字我们就添加了一个唯一索引。

也可以只对某一列添加一个唯一约束而不创建唯一索引：

ALTER TABLE students

ADD CONSTRAINT uni\_name UNIQUE (name);

这种情况下，name列没有索引，但仍然具有唯一性保证。

无论是否创建索引，对于用户和应用程序来说，使用关系数据库不会有任何区别。这里的意思是说，当我们在数据库中查询时，如果有相应的索引可用，数据库系统就会自动使用索引来提高查询效率，如果没有索引，查询也能正常执行，只是速度会变慢。因此，索引可以在使用数据库的过程中逐步优化。

## 10.3 SQL语句

### 10.3.1 查询语句

**条件查询**

使用SELECT查询的基本语句SELECT \* FROM <表名>可以查询一个表的所有行和所有列的数据。SELECT查询的结果是一个二维表。

SELECT 100+200;

虽然SELECT可以用作计算，但它并不是SQL的强项。但是，不带FROM子句的SELECT语句有一个有用的用途，就是用来判断当前到数据库的连接是否有效。许多检测工具会执行一条SELECT 1;来测试数据库连接。

SELECT语句可以通过WHERE条件来设定查询条件，查询结果是满足查询条件的记录。例如，要指定条件“分数在80分或以上的学生”，写成WHERE条件就是SELECT \* FROM students WHERE score >= 80。

条件查询的语法就是：

SELECT \* FROM <表名> WHERE <条件表达式>

条件表达式可以用<条件1> AND <条件2>表达满足条件1并且满足条件2

第二种条件是<条件1> OR <条件2>，表示满足条件1或者满足条件2。

第三种条件是NOT <条件>，表示“不符合该条件”的记录。

要组合三个或者更多的条件，就需要用小括号()表示如何进行条件运算。

如果不加括号，条件运算按照NOT、AND、OR的优先级进行。

**投影查询**

使用SELECT \* FROM <表名> WHERE <条件>可以选出表中的若干条记录。我们注意到返回的二维表结构和原表是相同的，即结果集的所有列与原表的所有列都一一对应。

如果我们只希望返回某些列的数据，而不是所有列的数据，我们可以用SELECT 列1, 列2, 列3 FROM ...，让结果集仅包含指定列。这种操作称为投影查询。

使用SELECT 列1, 列2, 列3 FROM ...时，还可以给每一列起个别名，这样，结果集的列名就可以与原表的列名不同。它的语法是SELECT 列1 别名1, 列2 别名2, 列3 别名3 FROM ...。

**排序**

我们使用SELECT查询时，细心的读者可能注意到，查询结果集通常是按照id排序的，也就是根据主键排序。这也是大部分数据库的做法。如果我们要根据其他条件排序怎么办？可以加上ORDER BY子句。例如按照成绩从低到高进行排序.如果要反过来，按照成绩从高到底排序，我们可以加上DESC表示“倒序”：

如果score列有相同的数据，要进一步排序，可以继续添加列名。例如，使用ORDER BY score DESC, gender表示先按score列倒序，如果有相同分数的，再按gender列排序

SELECT id, name, gender, score FROM students ORDER BY score DESC, gender;

如果有WHERE子句，那么ORDER BY子句要放到WHERE子句后面。

分页

使用SELECT查询时，如果结果集数据量很大，比如几万行数据，放在一个页面显示的话数据量太大，不如分页显示，每次显示100条。

因此，分页实际上就是从结果集中“截取”出第M~N条记录。这个查询可以通过LIMIT <M> OFFSET <N>子句实现。我们先把所有学生按照成绩从高到低进行排序：

把结果集分页，每页3条记录。要获取第1页的记录，可以使用LIMIT 3 OFFSET 0：

SELECT id, name, gender, score FROM students ORDER BY score DESC LIMIT 3 OFFSET 1;

LIMIT 3 OFFSET 0表示，对结果集从0号记录开始，最多取3条。注意SQL记录集的索引从0开始。如果要查询第2页，把OFFSET设定为3：查询第3页的时候，OFFSET应该设定为6:

OFFSET是可选的，如果只写LIMIT 15，那么相当于LIMIT 15 OFFSET 0。

在MySQL中，LIMIT 15 OFFSET 30还可以简写成LIMIT 30, 15。

聚合查询

如果我们要统计一张表的数据量，例如，想查询students表一共有多少条记录，难道必须用SELECT \* FROM students查出来然后再数一数有多少行吗？

这个方法当然可以，但是比较弱智。对于统计总数、平均数这类计算，SQL提供了专门的聚合函数，使用聚合函数进行查询，就是聚合查询，它可以快速获得结果。COUNT(\*)表示查询所有列的行数，要注意聚合的计算结果虽然是一个数字，但查询的结果仍然是一个二维表，只是这个二维表只有一行一列，并且列名是COUNT(\*)。

通常，使用聚合查询时，我们应该给列名设置一个别名，便于处理结果：

SELECT COUNT(\*) num FROM students;

SELECT COUNT(\*) boys FROM students WHERE gender = 'M';

除了COUNT()函数外，SQL还提供了如下聚合函数：

| 函数 | 说明 |
| --- | --- |
| SUM | 计算某一列的合计值，该列必须为数值类型 |
| AVG | 计算某一列的平均值，该列必须为数值类型 |
| MAX | 计算某一列的最大值 |
| MIN | 计算某一列的最小值 |

要特别注意：如果聚合查询的WHERE条件没有匹配到任何行，COUNT()会返回0，而SUM()、AVG()、MAX()和MIN()会返回NULL：

**分组**

对于聚合查询，SQL还提供了“分组聚合”的功能。我们观察下面的聚合查询。

SELECT COUNT(\*) num FROM students GROUP BY class\_id;

SELECT class\_id, COUNT(\*) num FROM students GROUP BY class\_id;

我们想统计各班的男生和女生人数：

SELECT class\_id, gender, COUNT(\*) num FROM students GROUP BY class\_id, gender;

**多表查询**

SELECT查询不但可以从一张表查询数据，还可以从多张表同时查询数据。查询多张表的语法是：SELECT \* FROM <表1> <表2>。

这种一次查询两个表的数据，查询的结果也是一个二维表，它是students表和classes表的“乘积”，即students表的每一行与classes表的每一行都两两拼在一起返回。结果集的列数是students表和classes表的列数之和，行数是students表和classes表的行数之积。

上述查询的结果集有两列id和两列name，两列id是因为其中一列是students表的id，而另一列是classes表的id，但是在结果集中，不好区分。两列name同理。

可以利用投影查询的“设置列的别名”来给两个表各自的id和name列起别名：

SELECT

students.id sid,

students.name,

students.gender,

students.score,

classes.id cid,

classes.name cname

FROM students, classes;

SQL还允许给表设置一个别名，让我们在投影查询中引用起来稍微简洁一点：

SELECT

s.id sid,

s.name,

s.gender,

s.score,

c.id cid,

c.name cname

FROM students s, classes c;

SELECT

s.id sid,

s.name,

s.gender,

s.score,

c.id cid,

c.name cname

FROM students s, classes c

WHERE s.gender = 'M' AND c.id = 1;

**连接查询**

连接查询是另一种类型的多表查询。连接查询对多个表进行JOIN运算，简单地说，就是先确定一个主表作为结果集，然后，把其他表的行有选择性地“连接”在主表结果集上。

SELECT s.id, s.name, s.class\_id, c.name class\_name, s.gender, s.score

FROM students s

INNER JOIN classes c

ON s.class\_id = c.id;

注意INNER JOIN查询的写法是：

先确定主表，仍然使用FROM <表1>的语法；

再确定需要连接的表，使用INNER JOIN <表2>的语法；

然后确定连接条件，使用ON <条件...>，这里的条件是s.class\_id = c.id，表示students表的class\_id列与classes表的id列相同的行需要连接；

可选：加上WHERE子句、ORDER BY等子句。

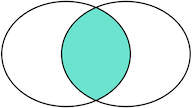
INNER JOIN只返回同时存在于两张表的行数据，由于students表的class\_id包含1，2，3，classes表的id包含1，2，3，4，所以，INNER JOIN根据条件s.class\_id = c.id返回的结果集仅包含1，2，3。

RIGHT OUTER JOIN返回右表都存在的行。如果某一行仅在右表存在，那么结果集就会以NULL填充剩下的字段。

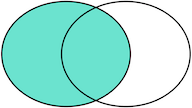
LEFT OUTER JOIN则返回左表都存在的行。如果我们给students表增加一行，并添加class\_id=5，由于classes表并不存在id=5的行，所以，LEFT OUTER JOIN的结果会增加一行，对应的class\_name是NULL：

最后，我们使用FULL OUTER JOIN，它会把两张表的所有记录全部选择出来，并且，自动把对方不存在的列填充为NULL：

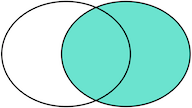
我们把tableA看作左表，把tableB看成右表，那么INNER JOIN是选出两张表都存在的记录：



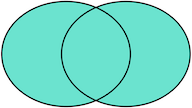
LEFT OUTER JOIN是选出左表存在的记录：



RIGHT OUTER JOIN是选出右表存在的记录：



FULL OUTER JOIN则是选出左右表都存在的记录：



### 10.3.2 修改数据

INSERT语句的基本语法是：

INSERT INTO <表名> (字段1, 字段2, ...) VALUES (值1, 值2, ...);

INSERT INTO students (class\_id, name, gender, score) VALUES

(1, '大宝', 'M', 87),

(2, '二宝', 'M', 81);

UPDATE语句的基本语法是：

UPDATE <表名> SET 字段1=值1, 字段2=值2, ... WHERE ...;

UPDATE students SET name='小牛', score=77 WHERE id>=5 AND id<=7;

如果WHERE条件没有匹配到任何记录，UPDATE语句不会报错，也不会有任何记录被更新

最后，要特别小心的是，UPDATE语句可以没有WHERE条件，例如：

UPDATE students SET score=60;

这时，整个表的所有记录都会被更新。所以，在执行UPDATE语句时要非常小心，最好先用SELECT语句来测试WHERE条件是否筛选出了期望的记录集，然后再用UPDATE更新。

DELETE语句的基本语法是：

DELETE FROM <表名> WHERE ...;

注意到DELETE语句的WHERE条件也是用来筛选需要删除的行，因此和UPDATE类似，DELETE语句也可以一次删除多条记录：

DELETE FROM students WHERE id>=5 AND id<=7;

最后，要特别小心的是，和UPDATE类似，不带WHERE条件的DELETE语句会删除整个表的数据：

DELETE FROM students;

这时，整个表的所有记录都会被删除。所以，在执行DELETE语句时也要非常小心，最好先用SELECT语句来测试WHERE条件是否筛选出了期望的记录集，然后再用DELETE删除。

## 10.4 mysql操作语句

创建一个新数据库，使用命令：

mysql> CREATE DATABASE test;

要删除一个数据库，使用命令：

mysql> DROP DATABASE test;

注意：删除一个数据库将导致该数据库的所有表全部被删除。

对一个数据库进行操作时，要首先将其切换为当前数据库：

mysql> USE test;

列出当前数据库的所有表，使用命令：

mysql> SHOW TABLES;

要查看一个表的结构，使用命令：

mysql> DESC students;

可以使用以下命令查看创建表的SQL语句：

mysql> SHOW CREATE TABLE students;

创建表使用CREATE TABLE语句，而删除表使用DROP TABLE语句：

mysql> DROP TABLE students;

修改表就比较复杂。如果要给students表新增一列birth，使用：

ALTER TABLE students ADD COLUMN birth VARCHAR(10) NOT NULL;

改表就比较复杂。如果要给students表新增一列birth，使用：

ALTER TABLE students ADD COLUMN birth VARCHAR(10) NOT NULL;

要修改birth列，例如把列名改为birthday，类型改为VARCHAR(20)：

ALTER TABLE students CHANGE COLUMN birth birhday VARCHAR(20) NOT NULL;

要删除列，使用：

ALTER TABLE students DROP COLUMN birthday;