第一阶段学习报告

目录

一、 Python3

1、基本数据类型

1.1、整型 int

理论上长度不受限制。

1.2、浮点型 float

含有小数，运算可能会出现四舍五入。

1.3、布尔型 bool

True 和 False，其中 0，0.0，0j，None，空字符串，空列表，空元组，空字典等转换为布尔型，其值都为 False，而其它值都转换为 True。

代码：

```
# 基本数据类型测试
a = bool(0)
b = bool(0.0)
c = bool(0j)
d = bool(None)
e = bool("")
f = bool([])
g = bool(())
h = bool({})
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
print(h)
```

结果：

```
False
False
False
False
False
False
False
False
```

1.4、字符串 str

单引号和双引号均可以表示字符串。
常用操作函数：
代码：

```
# 字符串常用内建函数

#strip(object) 参数默认为空格
s = "!!hello,world!!"
s1 = s.strip('!')
print(s1)

# string.join(seq) 连接字符串
str = "-"
seq = ("a", "b", "c")
print(str.join(seq))

# str.replace(old,new[,max]) 把字符串中的 old（旧字符串）
# 替换成 new(新字符串)，如果指定第三个参数max，则替换不超过 max 次。
str = "hello,world"
print(str.replace("l","a",1))

# str.split(str[,num])通过指定分隔符对字符串进行切片，
# 如果参数num 有指定值，则仅分隔 num 个子字符串
str  = "a,b,c,d"
print(str.split(",",1))

# str.count(sub, start= 0,end=len(string))统计字符串里某
# 个字符出现的次数。可选参数为在字符串搜索的开始与结束位置。
str  = "you are beautiful!"
print(str.count("u",0,len(str)))
```

结果：

```
hello,world
a-b-c
healo,world
['a', 'b,c,d']
3
```

1.5、基本数据结构

1.5.1、列表

代码：

```
list = ['we',"us",2,3,4]
# 增加元素
list.append('Google')
print("增加元素之后: ",list)

# 删除元素
del list[1]
print("删除元素之后: ",list)

# 查找元素  当索引为负数时，表示从末尾开始查找
print(list[1:3])
print(list[1:])
print(list[-2])
```

结果：

```
增加元素之后:  ['we', 'us', 2, 3, 4, 'Google']
删除元素之后:  ['we', 2, 3, 4, 'Google']
[2, 3]
[2, 3, 4, 'Google']
4
```

列表计算表达式

表达式	结果	描述
Len([1,2,3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表内
for x in [1, 2, 3]: print(x)	1 2 3	迭代

1.5.2、元组

Python 的元组与列表类似，不同之处在于元组的元素不能修改。但元组中可以嵌套列表，元组中的列表可以修改。使用小括号。元组中只包含一个元素时，需要在元素后面添加逗号。

代码：

```
# 元组
tup1 = ('physics','chemistry',1992,1996)
# 创建一个只含有一个元素的元组时，要在元素后面添加逗号
tup2 = (3,)
# 创建一个含有一个列表元素的元组
tup3 = (["lanlan",333],'lvlv')
tup3[0].append(222)
print(tup1)
print(tup2)
print(tup3)
```

结果：

```
('physics', 'chemistry', 1992, 1996)
(3,)
(['lanlan', 333, 222], 'lvlv')
```

任意无符号的对象，以逗号隔开，默认为元组。
代码：

```
tup = 2,3,4
print(type(tup))
```

结果：

```
<class 'tuple'>
```

1.5.3、字典

字典的每个键值 key=>value 对用冒号 : 分割，每个键值对之间用逗号 , 分割，整个字典包括在花括号 {} 中。

字典值可以没有限制地取任何 python 对象，既可以是标准的对象，也可以是用户定义的，但键不行。键必须不可变，所以可以用数字，字符串或元组充当，所以用列表就不行。

不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住。

代码：

```
# 字典

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
print(dict)
dict['Name'] = 'Lanlan'
print(dict)
del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
```

结果：

```
{'Name': 'Zara', 'Age': 7, 'Class': 'First'}
{'Name': 'Lanlan', 'Age': 7, 'Class': 'First'}
```

2、流程控制语句

Python 的循环控制语句和其他语言的大致差不多，只是 python 使用缩进来代替大括号。支持 foreach。

3、函数

Python 的返回值可以有多个，但其实只是一个元组、列表或者是字典。

3.1、函数参数

3.1.1、默认参数

要求放在参数列表最后。
代码：

```
# 函数

# 参数
def stu(name,age = 22,city = "chongqing"):
    print(name,age,city)

stu("lanlan")
```

结果：

```
lanlan 22 chongqing
```

3.1.2、不定长参数

代码：

```
# 不定长参数
def functionname(*var_args_tuple ):
    for var in var_args_tuple:
        print(var)

functionname(1,5,9)
```

结果：

```
1
5
9
```

3.2、匿名函数 lambda

函数的简略写法。
代码：

```
# 匿名函数
s = lambda x:x*9
print(s(2))
```

结果：

```
18
```

分析：虽然简化了函数的写法，但只适合简短的简单处理函数，如果内容过多影响阅读性。

3.3、高阶函数

3.3.1、map

map()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。
代码：

```
# map
s = [1,2,3,4,5]
f = lambda x:x*2
r = map(f,s)
print(type(r))
print(list(r))
```

结果：

```
<class 'map'>
[2, 4, 6, 8, 10]
```

3.3.2、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。

代码：

```
# reduce
from functools import reduce
s = [1,2,3,4,5]
def add(x,y):
    return x+y
print(reduce(add,s))
def calc(x,y):
    return 10*x+y

print(reduce(calc, s))
```

结果：

```
15
12345
```

3.3.3、filter

Python 内建的 filter()函数用于过滤序列。和 map()类似，filter()也接收一个函数和一个序列。和 map()不同的是，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。

Yield 用法，yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 _odd_iter()不会执行 _odd_iter 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 _odd_iter 函数内部的代码，执行到 yield b 时，_odd_iter 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

代码：

```
# filter
# 在一个list中，删掉偶数，只保留奇数，可以这么写：

def is_odd(n):
    return n % 2 == 1

print(list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15])))

from inspect import isgeneratorfunction
# 建立一个生成器
def _odd_iter():
    n = 1
    while True:
        n = n + 2
        yield n

it = _odd_iter()

print(isgeneratorfunction(_odd_iter))
print(it.__next__())
print(it.__next__())
print(it.__next__())
```

结果：

```
[1, 5, 9, 15]
True
3
5
7
```

3.3.4、sorted

代码：

```
# list
# Python内置的sorted()函数就可以对list进行排序:
list = [2, 1, 4, 3, 0]
print(sorted(list))

# 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:

list2 = sorted([36, 5, -12, 9, -21], key=abs)
print(list2)
```

结果：

```
[0, 1, 2, 3, 4]
[5, 9, -12, -21, 36]
```

4、列表推导式

代码：

```
# 列表推导式

q = [ 'the %s' % x for x in range(1,10)]
print(q)
```

结果：

```
['the 1', 'the 2', 'the 3', 'the 4', 'the 5', 'the 6', 'the 7', 'the 8', 'the 9']
```

小结：

① python2 和 3 有些区别，比如利用 yeild 编写生成器，在进行获取元素是 python3 使用的是__next__(),python2 使用的是 next()。

② 调用 map、filter 函数，返回的都是 Iterator，也就是一个惰性序列，所以要强迫 map()、filter()完成计算结果，需要用 list()函数获得所有结果并返回 list。

5、日期和时间

时间戳单位最适于做日期运算。但是 1970 年之前的日期就无法以此表示了。太遥远的日期也不行，UNIX 和 Windows 只支持到 2038 年。

Struct_time 元组

序号	属性	值
0	tm_year	2018
1	tm_mon	1~12
2	tm_mday	1~31
3	tm_hour	0~23
4	tm_min	0~59
5	tm_sec	0~61(60 或 61 是闰秒)
6	tm_wday	0~6(0 是周一)
7	tm_yday	1 到 366(儒略历)
8	tm_isdst	-1, 0, 1, -1 是决定是否为夏令时的旗帜

5.1、time 模块

代码：

```
import time;

# 获取当前时间
localtime = time.localtime(time.time())
print("本地时间为: ",localtime)
# 获取时间戳
print(time.time())

#以struct_time元组为基础，获取格式化时间
print("本地时间为: ",time.asctime(time.localtime()))

# 使用time.strftime(format[, t])函数格式化日期

# 格式化成2016-03-20 11:45:39形式
print(time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) )
print(time.strftime("%Y-%m-%d %H:%M:%S"))#默认当地时间

# 格式化成Sat Mar 28 22:24:24 2016形式
print(time.strftime("%a %b %d %H:%M:%S %Y", time.localtime()))

# 将格式字符串转换为时间戳
a = "Sat Mar 28 22:24:24 2016"
print(time.mktime(time.strptime(a,"%a %b %d %H:%M:%S %Y")))
```

结果：

```
本地时间为：  time.struct_time(tm_year=2018, tm_mon=4, tm_mday=4, tm_hour=9, tm_m
in=32, tm_sec=34, tm_wday=2, tm_yday=94, tm_isdst=0)
1522805554.5199227
本地时间为：  Wed Apr  4 09:32:34 2018
2018-04-04 09:32:34
2018-04-04 09:32:34
Wed Apr 04 09:32:34 2018
1459175064.0
```

5.2、Calendar 模块

Calendar 内置函数

序号	函数及描述
1	calendar.calendar(year,w=2,l=1,c=6) 返回一个多行字符串格式的 year 年年历，3 个月一行，间隔距离为 c。 每日宽度间隔为 w 字符。每行长度为 21* W+18+2* C。l 是每星期行数。
2	calendar.firstweekday( ) 返回当前每周起始日期的设置。默认情况下，首次载入 caendar 模块时返回 0，即星期一。
3	calendar.isleap(year) 是闰年返回 True，否则为 false。
4	calendar.month(year,month,w=2,l=1) 返回一个多行字符串格式的 year 年 month 月日历，两行标题，一周一行。每日宽度间隔为 w 字符。每行的长度为 7* w+6。l 是每星期的行数。
5	calendar.monthcalendar(year,month) 返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。Year 年 month 月外的日期都设为 0;范围内的日子都由该月第几日表示，从 1 开始。
6	calendar.timegm(tupletime) 和 time.gmtime 相反：接受一个时间元组形式，返回该时刻的时间戳（1970 纪元后经过的浮点秒数）。
7	calendar.weekday(year,month,day) 返回给定日期的日期码。0（星期一）到 6（星期日）。月份为 1（一月） 到 12（12 月）。

代码：

```
import calendar

cal = calendar.month(2018, 4)
print("以下输出2018年4月份的日历:")
print(cal)
```

结果：

```
以下输出2018年4月份的日历:
     April 2018
Mo Tu We Th Fr Sa Su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30


请按任意键继续. . .
```

小结：python 提供的关于时间和日期的内置函数十分的方便，使用时只需要来查询各个函数的作用即可。

6、多线程

6.1、线程的创建

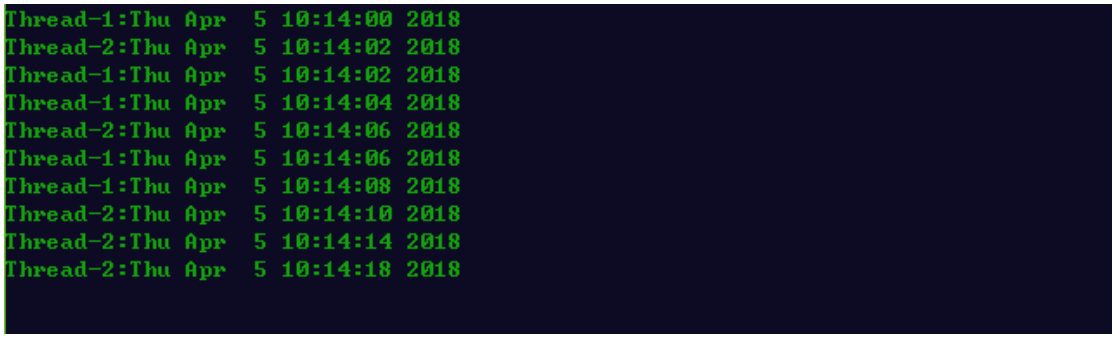
方法一：使用_thread 函数创建线程。
代码：

```
import _thread
import time
def print_time( threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count +=1
        print("%s:%s"%(threadName,time.ctime(time.time())))


try:
    _thread.start_new_thread(print_time,("Thread-1",2,))
    _thread.start_new_thread(print_time,("Thread-2",4,))
except:
    print("Error: unable to start thread")

while 1:
    pass
```

结果：



方法二：使用 threading 模块创建线程。

Python3 通过两个标准库 _thread 和 threading 提供对线程的支持。_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 _thread 模块中的所有方法外，还提供的其他方法：

①threading.currentThread(): 返回当前的线程变量。

②threading.enumerate(): 返回一个包含正在运行的线程的 list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。

③ threading.activeCount(): 返回正在运行的线程数量，与 len(threading.enumerate())有相同的结果。

除了使用方法外，线程模块同样提供了 Thread 类来处理线程，Thread 类提供了以下方法:

①run(): 用以表示线程活动的方法。

②start():启动线程活动。

③join([time]): 等待至线程中止。这阻塞调用线程直至线程的 join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。

④isAlive(): 返回线程是否活动的。

⑤getName(): 返回线程名。

⑥setName(): 设置线程名。

代码：

```
import threading
import time

exitFlag = 0
class myThread(threading.Thread):
    def __init__(self,threadId,threadName,counter):
        threading.Thread.__init__(self)
        self.threadId = threadId
        self.threadName = threadName
        self.counter = counter

    def run(self):
        print("开始线程:"+self.threadName)
        print_time(self.threadName,self.counter,5)
        print("退出线程:"+self.threadName)

def print_time(threadName,delay,counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print(threadName+time.asctime(time.localtime()))
        counter -=1

thread1 = myThread(1,"Thread-1",1)
thread2 = myThread(2,"Thread-2",1.5)

thread1.start()
thread2.start()
thread1.join()
thread2.join()
print("退出主线程")
```

结果：

```
开始线程:Thread-1
开始线程:Thread-2
Thread-1Thu Apr  5 11:40:25 2018
Thread-2Thu Apr  5 11:40:26 2018
Thread-1Thu Apr  5 11:40:26 2018
Thread-2Thu Apr  5 11:40:27 2018
Thread-1Thu Apr  5 11:40:27 2018
Thread-1Thu Apr  5 11:40:28 2018
Thread-2Thu Apr  5 11:40:29 2018
Thread-1Thu Apr  5 11:40:29 2018
退出线程:Thread-1
Thread-2Thu Apr  5 11:40:30 2018
Thread-2Thu Apr  5 11:40:32 2018
退出线程:Thread-2
退出主线程
```

关于 start 方法和 join 方法的分析：

start 函数，顾名思义，及开始一个线程，而 jion 则是原来阻塞主线程函数，该函数可以跟参数，即为阻塞主线程多少秒，常用来线程同步。

6.2、线程的同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。如下：

加了同步锁的代码：

```
import threading
import time

list = []

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("开启线程：  " + self.name)
        # 获取锁，用于线程同步
        threadLock.acquire()
        add_counter(self.name,self.threadID,self.counter)
        # 释放锁，开启下一个线程
        threadLock.release()

def add_counter(threadName,num,counter):
    while counter:
        list.append(num)
        print (threadName)
        print(list)
        counter -=1
```

```
threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 4)
thread2 = myThread(2, "Thread-2", 4)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

结果：

```
开启线程：  Thread-1
Thread-1
开启线程：  Thread-2
[1]
Thread-2
Thread-1
[1, 2, 1]
[1, 2, 1]
Thread-1
Thread-2
[1, 2, 1, 2, 1]
[1, 2, 1, 2, 1]
Thread-1
Thread-2
[1, 2, 1, 2, 1, 1, 2]
[1, 2, 1, 2, 1, 1, 2]
Thread-2
[1, 2, 1, 2, 1, 1, 2, 2]
退出主线程
```

未加同步锁结果：

```
开启线程：  Thread-1
开启线程：  Thread-2
Thread-1
[1]
Thread-1
[1, 1]
Thread-1
[1, 1, 1]
Thread-1
[1, 1, 1, 1]
Thread-2
[1, 1, 1, 1, 2]
Thread-2
[1, 1, 1, 1, 2, 2]
Thread-2
[1, 1, 1, 1, 2, 2, 2]
Thread-2
[1, 1, 1, 1, 2, 2, 2, 2]
退出主线程
```

分析：未加锁时，两个线程均可以同时对 list 进行操作，而加了锁之后，必须等一个线程执行结束释放锁之后，后一个线程才能对 list 进行操作，从而保证了线程的同步性。

6.3、线程优先级队列(Queue)

介绍：Python 的 Queue 模块中提供了同步的、线程安全的队列类，包括 FIFO（先入先出)队列 Queue，LIFO（后入先出）队列 LifoQueue，和优先级队列 PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步。

代码：

```python
import queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print ("开启线程：" + self.name)
        process_data(self.name, self.q)
        print ("退出线程：" + self.name)

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
        time.sleep(1)
```

```python
# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    # print(tName+'开启')
# print('循环结束')
    threads.append(thread)
    threadID += 1

# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

结果：

分析：
先开启四个线程，随后将他们入队，最后再出队。根据出队的顺序来改变线程的执行顺序。

二、SQL

练习所使用是数据库表如下：

Student

Sno	Sname	Ssex	Sage	Sdept
201215121	李勇	男	20	CS
201215122	刘晨	女	19	CS
201215123	王敏	女	18	MA
201215125	张莉	男	19	IS

Course

Cno	Cname	Cpno	Ccredit
1	数据库	5	4
2	数学	(Null)	2
3	信息系统	1	4
4	操作系统	6	3
5	数据结构	7	4
6	数据处理	(Null)	2
7	PASCAL语言	6	4

SC

Sno	Cno	Grade
201215121	1	92
201215121	2	85
201215121	3	88
201215122	2	90
201215122	3	80

1、表连接

1.1、内连接

内连接，是取两个表或者多个表的交集，从而得到自己想要的结果。
指令：

```
mysql> SELECT Student.Sname,Student.Sdept,Course.Cname, Sc.Grade
    -> from student
    -> inner join
    -> (course inner join SC on course.Cno=sc.Cno) on student.Sno=SC.Sno;
```

结果：

Sname	Sdept	Cname	Grade
李勇	CS	数据库	92
李勇	CS	数学	85
李勇	CS	信息系统	88
刘晨	CS	数学	90
刘晨	CS	信息系统	80

1.2、外连接

外联接可以是左向外联接、右向外联接或完整外部联接。
在 FROM 子句中指定外联接时，可以由下列几组关键字中的一组指定：

1.2.1、左外连接

LEFT JOIN 或 LEFT OUTER JOIN，左向外联接的结果集包括 LEFT OUTER 子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。
指令：

```
mysql> SELECT  student.Sno,student.Sname,student.Ssex,student.Sage,sc.Cno,Sc.Grade
    -> from student left join sc
    -> on student.Sno=sc.Sno;
```

结果：

Sno	Sname	Ssex	Sage	Cno	Grade
201215121	李勇	男	20	1	92
201215121	李勇	男	20	2	85
201215121	李勇	男	20	3	88
201215122	刘晨	女	19	2	90
201215122	刘晨	女	19	3	80
201215123	王敏	女	18	NULL	NULL
201215125	张莉	男	19	NULL	NULL

1.2.2、右外连接

RIGHT JOIN 或 RIGHT OUTER JOIN，右向外联接是左向外联接的反向联接。将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。

指令：

```
mysql> SELECT  student.Sno,student.Sname,student.Ssex,student.Sage,sc.Cno,Sc.Grade
    -> from student right join sc
    -> on student.Sno=sc.Sno;
```

结果：

```
+-----------+-------+------+------+------+-------+
| Sno       | Sname | Ssex | Sage | Cno  | Grade |
+-----------+-------+------+------+------+-------+
| 201215121 | 李勇  | 男   |   20 |    1 |    92 |
| 201215121 | 李勇  | 男   |   20 |    2 |    85 |
| 201215121 | 李勇  | 男   |   20 |    3 |    88 |
| 201215122 | 刘晨  | 女   |   19 |    2 |    90 |
| 201215122 | 刘晨  | 女   |   19 |    3 |    80 |
+-----------+-------+------+------+------+-------+
5 rows in set (0.00 sec)
```

1.2.3、完全外连接

FULL JOIN 或 FULL OUTER JOIN，完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。由于 MySQL 不支持外连接，所以这里使用左外连接+右外连接实现完全外连接。

指令：

```
mysql> SELECT  student.Sno,student.Sname,student.Ssex,student.Sage,sc.Cno,Sc.Grade
    -> from student left join sc
    -> on student.Sno=sc.Sno
    -> union
    -> SELECT  student.Sno,student.Sname,student.Ssex,student.Sage,sc.Cno,Sc.Grade
    -> from student right join sc
    -> on student.Sno=sc.Sno;
```

结果：

```
+-----------+-------+------+------+------+-------+
| Sno       | Sname | Ssex | Sage | Cno  | Grade |
+-----------+-------+------+------+------+-------+
| 201215121 | 李勇  | 男   |   20 |    1 |    92 |
| 201215121 | 李勇  | 男   |   20 |    2 |    85 |
| 201215121 | 李勇  | 男   |   20 |    3 |    88 |
| 201215122 | 刘晨  | 女   |   19 |    2 |    90 |
| 201215122 | 刘晨  | 女   |   19 |    3 |    80 |
| 201215123 | 王敏  | 女   |   18 | NULL |  NULL |
| 201215125 | 张莉  | 男   |   19 | NULL |  NULL |
+-----------+-------+------+------+------+-------+
7 rows in set (0.01 sec)
```

1.3、交叉连接

交叉连接（CROSS JOIN）：有两种，显式的和隐式的，不带 ON 子句，返回的是两表的乘积，也叫笛卡尔积。
显式：

```
mysql> SELECT  student.Sno,student.Sname,student.Ssex,student.Sage,sc.Cno,Sc.Grade
    -> from student cross join sc;
```

隐式：

```
mysql> SELECT  student.Sno,student.Sname,student.Ssex,student.Sage,sc.Cno,Sc.Grade
    -> from student , sc;
```

结果：

Sno	Sname	Ssex	Sage	Cno	Grade
201215121	李勇	男	20	1	92
201215122	刘晨	女	19	1	92
201215123	王敏	女	18	1	92
201215125	张莉	男	19	1	92
201215121	李勇	男	20	2	85
201215122	刘晨	女	19	2	85
201215123	王敏	女	18	2	85
201215125	张莉	男	19	2	85
201215121	李勇	男	20	3	88
201215122	刘晨	女	19	3	88
201215123	王敏	女	18	3	88
201215125	张莉	男	19	3	88
201215121	李勇	男	20	2	90
201215122	刘晨	女	19	2	90
201215123	王敏	女	18	2	90
201215125	张莉	男	19	2	90
201215121	李勇	男	20	3	80
201215122	刘晨	女	19	3	80
201215123	王敏	女	18	3	80
201215125	张莉	男	19	3	80

2、聚合函数

SQL 基本函数，聚合函数对一组值执行计算，并返回单个值。除了 COUNT 以外，聚合函数都会忽略空值。 聚合函数经常与 SELECT 语句的 GROUP BY 子句一起使用。

常见聚合函数

SUM()	返回总和
COUNT()	返回列中项目数量
MAX()	返回最大值
MIN()	返回最小值
AVG()	返回平均值

2.1、SUM

查询某一个学生的总分。参数可为表达式，语句如下：

```
mysql> select sum(sc.Grade) as 总分
    -> from Sc
    -> where sc.Sno=201215121;
```

结果：

```
+------+
| 总分 |
+------+
|  265 |
+------+
```

2.2、COUNT

SQL 提供了 COUNT 函数来查询满足设定标准的记录的数量。我们可以使用单独 COUNT(*)语法来检索一个表内的行数。此外，还可以利用 WHERE 子句来设置计数条件，返回特定记录的条数。例如：统计 Student 表里的男生总数，语句如下：

```
mysql> SELECT COUNT(student.Ssex) AS 男生总数
    -> FROM student
    -> where student.Ssex='男';
```

结果：

```
+----------+
| 男生总数 |
+----------+
|        2 |
+----------+
```

2.3、MAX

我们可以给该函数一个字段名称来返回表中给定字段的最大值。还可以在 MAX()函数中使用表达式和 GROUP BY 从句来加强查找功能。查询学生成绩的最高分，语句如下：

```
mysql> SELECT MAX(SC.Grade) as '最高分'
    -> FROM SC;
```

结果：

```
+--------+
| 最高分 |
+--------+
|     92 |
+--------+
```

2.4、MIN

用法与 MAX 函数类似。这里就不举例子了。

2.5、AVG

计算并返回表达式的平均值。查询某学生平均成绩语句如下：

```
mysql> SELECT AVG(SC.Grade) as '平均分'
    -> from sc
    -> where Sc.Sno=201215121;
```

结果：

```
+----------+
| 平均分   |
+----------+
| 88.3333 |
+----------+
```

3、子查询

嵌套 SELECT 语句也叫子查询，一个 SELECT 语句的查询结果能够作为另一个语句的输入值。子查询不但能够出现在 Where 子句中，也能够出现在 from 子句中，作为一个临时表使用，也能够出现在 select list 中，作为一个字段值来返回。

3.1、单行子查询

单行子查询是指子查询的返回结果只有一行数据。当主查询语句的条件语句中引用子查询结果时可用单行比较符号（＝，＞，＜，＞=，＜=，＜＞）来进行比较。例如：

```
mysql> select sc.Grade from sc where(select student.Sno from student where student.Sage=20);
```

结果：

```
+-------+
| Grade |
+-------+
|    92 |
|    85 |
|    88 |
|    90 |
|    80 |
+-------+
```

3.2、多行子查询

多行子查询即是子查询的返回结果是多行数据。当主查询语句的条件语句中引用子查询结果时必须用多行比较符号（IN，ALL,ANY）来进行比较。其中，IN 的含义是匹配子查询结果中的任一个值即可（"IN" 操作符，能够测试某个值是否在一个列表中），ALL 则必须要符合子查询的所有值才可，ANY 要符合子查询结果的任何一个值即可。而且须注意 ALL 和 ANY 操作符不能单独使用，而只能与单行比较符（=、>结合使用。

```
mysql> select student.Sname from student
    -> where student.Sdept in
    -> (select student.Sdept from student where student.Sage=19);
```

使用 in 谓词查询 student 表中年龄为 19 的学生，语句如下：
结果：

```
+--------+
| Sname  |
+--------+
| 李勇   |
| 刘晨   |
| 张莉   |
+--------+
```

使用谓词 all 查询选修了名字中第一个是数的课程的学生名单，语句如下：

```
mysql> select student.Sname
    -> from student
    -> where student.Sno in (select sc.Sno from sc
    -> where sc.Cno=any (select course.Cno from course where course.Cname like '数%'));
```

结果：

```
+--------+
| Sname  |
+--------+
| 李勇   |
| 刘晨   |
+--------+
```

带有 exists 的子查询：
查询选修了 2 号课程的学生，语句如下：

```
mysql> select student.Sname
    -> from student
    -> where student.Sno in (select sc.Sno from sc
    -> where exists(select * from sc where sc.Sno=student.Sno and sc.Cno=2));
```

结果：

```
+--------+
| Sname  |
+--------+
| 李勇   |
| 刘晨   |
+--------+
```