

目录

01day	1
day02	19
day03	29
附录	41

multiprocessing [Process,Queue] queue.PriorityQueue

01day

1) 聊聊计算机程序是什么？进程的定义是什么？

程序：计算机程序是指以某些程序设计语言编写，运行于某种目标结构体系上。存储在电子设备上 0 与 1 有序组合； 类

进程：是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。操作系统结构的基础程序在操作系统(OS)中运行的一个抽象； 程序的实例对象
抽象是一种近似，一种简化，帮助我们认识事物和使用事物；

程序与进程的关系

一对一， 一对多；

多对一， 多对多；

以上都不是；

2) 如何对程序，进程和线程进行一种抽象类似的近似；

OS 操作系统： 公司

CPU, I/O: 公司的资源

进程： 公司的部门

 部门有自己的办公区，地点； 部门有自己的厕所，有自己的餐厅，

 健身房； 部门要有人，至少有一个人；

线程： 部门的人

 人就是干活的；

锁： 部门的厕所那把锁； 健身房的锁； 餐厅的锁；

 资源是有限的，所以我们要管理资源； 管理资源的方式就是加锁；

锁必然会有钥匙；

进程：程序在一个数据集上运行过程。进程是有生命周期的。程序可以同时形成多个运行副本，一个进程可以执行多个程序。每个进程享有独立的空间。进程是操作系统进行资源调度和分配的单位，每个进程完成特定的任务。

进程的特性：独立性，动态性，并发行，结构性。

3) 进程：OS 对程序执行的一次抽象；

A. 进程的组成：

数据段，

代码段，

Shell Code:

```
char a[80];
```

进程控制块(PCB):其他的百度

1) 进程的标识符 pid, name;

2) 进程的调度信息（

进程的状态，

进程的优先级，linux -20~19，负数最高。window 相反

进程的关系）

3) 进程控制信息（资源清单，

信号量，

互斥量，

锁的信息，虚拟的地址，等）

4) 处理机的信息（通用寄存器，指令计数器等）；

进程是程序在计算机上的一次执行活动。当你运行一个程序，你就启动了一个进程。显然，程序是死的(静态的)，进程是活的(动态的)。进程可以分为系统进程和用户进程。凡是用于完成[操作系统的](#)各种功能的进程就是系统进程，它们就是处于运行状态下的[操作系统](#)本身；用户进程就是所有由你启动的进程。进程是[操作系统](#)进行资源分配的单位。

进程为应用程序的运行实例，是应用程序的一次动态执行。看似高深，我们可以简单地理解为：它是[操作系统](#)当前运行的[执行程序](#)。在系统当前运行的[执行程序](#)里包括：系统管理计算机个体和完成各种操作所必需的程序；用户开启、执行的额外程序，当然也包括用户不知道，而自动运行的非法程序（它们就有可能是病毒程序）。

实验：

B. 几类特殊的进程：

孤儿进程: 父进程创建子进程，自己（父进程）被杀或者自杀；

僵尸进程: 父进程创建子进程，子进程被杀或者自杀，还存在。

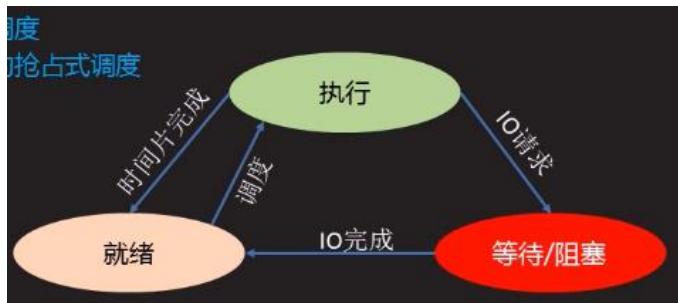
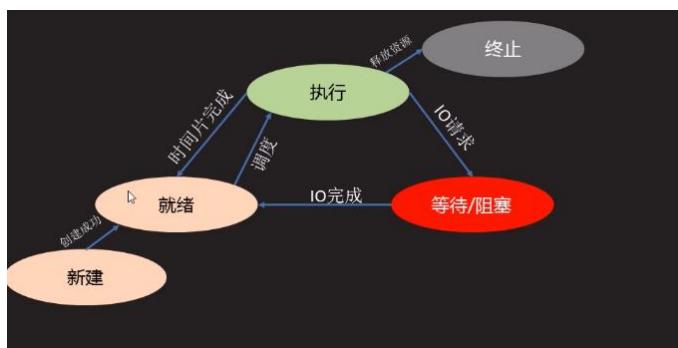
父进程没有上报，没有告诉系统回收资源；

这里与资源回收相关

4) 进程的状态：

三态: 就绪, 执行, 等待/阻塞；

五态: 新建, 就绪, 执行, 等待/阻塞, 终止；



程序与进程对应关系？

一对一， 一对多；

多对一， 多对多；

都有可能；

5) 解释几个名词：

多任务，并发

并发的关键是你有处理多个任务的能力，不一定要同时。

并行的关键是你有同时处理多个任务的能力。

【回顾】多任务

- 现代操作系统都是多任务系统
 - windows , linux , mac
 - 给我们的感觉是多个软件同时运行，完成不同的功能
- 单核计算机多任务实现
 - 多个任务分时占用CPU等资源
- 多核计算机多任务实现
 - 任务数小于等于CPU核数量
 - 当任务数大于CPU核数量

并发与并行

- 并发
 - 并发的实质是一个物理CPU(也可以多个物理CPU) 在若干道程序之间多路复用，并发性是对有限物理资源强制行使多用户共享以提高效率。
- 并行
 - 并行性指两个或两个以上事件或活动在同一时刻发生。在多道程序环境下，并行性使多个程序同一时刻可在不同CPU上同时执行。
 - 多个实体，多个事件，同时发生

异步与同步：

同步，是所有的操作都做完，才返回给用户结果。即写完数据库之后，在相应用户，用户体验不好。

异步，不用等所有操作等做完，就相应用户请求。即先相应用户请求，然后慢慢去写数据库，用户体验较好。

阻塞与非阻塞：一直等就是阻塞，干其他事就是非阻塞；

*** 当我们说阻塞和非阻塞的时候，是指一个进程的行为；

相对于我们的部门做好自己的事情，不需要外部干涉；

*** 当我们说异步和同步的时候，是指至少两个进程的行为；

这时需要在两个部门之间协调；同步的执行流程相对可控，

异步执行流程不可控，通常来说效率更高；

面试题：

1) 程序与进程的区别；

2) 并发，并行；阻塞与非阻塞；异步与同步的区别；京东通知你发货、自己去看发货没

有两种方式来实现并发性，

一种方式是让每个“任务”或“进程”在单独的内存空间中工作，每个都有自己的工作内存区域。
不过，虽然进程可在单独的内存空间中执行，但除非这些进程在单独的处理器上执行，否则，实际并

不是“同时”运行的。是由操作系统把处理器的时间片分配给一个进程，用完时间片后就需退出处理器等待另一个时间片的到来。

另一种方式是在程序中指定多个“执行线程”，让它们在相同的内存空间中工作。这称为“多线程处理”。线程比进程更有效，因为操作系统不必为每个线程创建单独的内存空间。

3) 死锁是怎么回事，怎么解决死锁？

Q: A 和 B 都需要 r1,r2 两个资源，A 占 r1，B 占 r2，都不放手，都没法用。**死锁**是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

解决死锁的方案

1.避免互斥 避免资源被一个进程使用；缺点 和资源本身的性质有关系；---增加资源或者排队

2.资源获取的原子性：保证所有的资源都拿到，否则都不要；---

缺点 拿资源的效率不高

举个例子：

A 想要从自己的帐户中转 1000 块钱到 B 的帐户里。那个从 A 开始转帐，到转帐结束的这一个过程，称之为一个事务。在这个事务里，要做如下操作：

1. 从 A 的帐户中减去 1000 块钱。如果 A 的帐户原来有 3000 块钱，现在就变成 2000 块钱了。

2. 在 B 的帐户 里加 1000 块钱。如果 B 的帐户如果原来有 2000 块钱，现在则变成 3000 块钱了。

如果在 A 的帐户已经减去了 1000 块钱的时候，忽然发生了意外，比如停电什么的，导致转帐事务意外终止了，而此时 B 的帐户里还没有增加 1000 块钱。那么，我们称这个操作失败了，要进行回滚。回滚就是回到事务开始之前的状态，也就是回到 A 的帐户还没减 1000 块的状态，B 的帐户的原来的状态。此时 A 的帐户仍然有 3000 块，B 的帐户仍然有 2000 块。

我们把这种要么一起成功（A 帐户成功减少 1000，同时 B 帐户成功增加 1000），要么一起失败（A 帐户回到原来状态，B 帐户也回到原来状态）的操作叫原子性操作。

如果把一个事务可看作是一个程序，它要么完整的被执行，要么完全不执行。这种特性就叫原子性

3.让资源有序获取：缺点 拿资源的效率不高

银行家算法；有 25，a 需求 50，b 需求 15，c 需求 10.会先借给 b 和 c。

上面 3 个在设计上尝试避免死锁发生；

4.超时：请求与保持条件 timeout，调度算法去处理，该放手就放手；

缺点 确定谁放手；

第 4 个是发生时怎么补救；

Python:

C⁺Python,Jython,pypy;

if, while, =; 图灵完备;

-----编程语言的层次-----

低级语言 机器语言, 汇编语言; 优点: 快, 开发效率低;

缺点 不可移植;

高级语言 C, C++, 编译型语言; 优点 开发效率高,

缺点 慢相对低级语言;

Java, Python; 编译解释性语言; 编译过程可以优化;

脚本语言 JS, VB, 解释性语言;

shell powershell, .bat, BShell, 轻量级粘合语言;

multiprocessing

是内置模块, 所以不用安装。

主要类包括:

Process 类: 多进程创建和启动

start 就绪 terminate 停止 is_alive 还活着 exitcode 退出

```
In [9]: p.          或者返回值t或f    进程名字
p.authkey   p.ident   p.name
p.daemon    p.is_alive p.pid   进程号 p.start  开始
p.exitcode  p.join    p.run   运行   p.terminate 停止
```

Pool 类: 进程池

Queue 类, pipe 类和 Event 类: 进程通信

Lock 类: 锁, 进程同步

Value 类和 Array 类: 共享内容

创建 Process 对象



- Process 构造函数

```
multiprocessing.Process(group=None, target=None, name=None,
args=(), kwargs={}, *, daemon=None)
```

- 创建Process对象必须使用 **关键字** 参数调用其构造函数

- group 始终为 None, 不需要传递, 兼容性
- **target** 指明进程运行时要执行的**函数**, 默认为None不调用任何函数
- **name** 是进程名称, 可选
- **args** 是目标调用的参数**元组**
- **kwargs** 是目标调用的关键字参数的**字典**
- **daemon**: True, False, 缺省None, 表示从父进程继承

```
from multiprocessing import Process

def t1():
    print("child process")

if __name__ == '__main__':
    p = Process(target=t1)
    p.start()
    print("main process")
```

```
运行函数t1

from multiprocessing import Process
import time

def t1():
    print("child process")

if __name__ == '__main__':
    p = Process(target=t1)
    p.start()      延迟一秒

    time.sleep(1)

    print("main process")
```

```
1 def t1(interval):
2     while True:
3         time.sleep(interval)
4         print(time.ctime())
5
6 if __name__ == '__main__':
7     p = Process(target=t1, args=(1,))  元组传参数,
8     p.start()
9
10    time.sleep(1)
11
12    print("main process")
```

将会一直打印 def 方法中的但前时间。

```

main process
child Sat Nov 4 15:11:52 2017
child Sat Nov 4 15:11:53 2017
main Sat Nov 4 15:11:54 2017
child Sat Nov 4 15:11:54 2017
child Sat Nov 4 15:11:55 2017
main Sat Nov 4 15:11:56 2017
child Sat Nov 4 15:11:56 2017
child Sat Nov 4 15:11:57 2017
main Sat Nov 4 15:11:58 2017
child Sat Nov 4 15:11:58 2017
child Sat Nov 4 15:11:59 2017
main Sat Nov 4 15:12:00 2017
child Sat Nov 4 15:12:00 2017
child Sat Nov 4 15:12:01 2017
main Sat Nov 4 15:12:02 2017
child Sat Nov 4 15:12:02 2017
child Sat Nov 4 15:12:03 2017
main Sat Nov 4 15:12:04 2017
child Sat Nov 4 15:12:04 2017

from multiprocessing import Process
import time

def t1(interval):
    while True:
        time.sleep(interval)
        print("child "+time.ctime())

if __name__ == '__main__':
    p = Process(target=t1, args=(1,))
    p.start()

    time.sleep(1)

    print("main process")
    while True:
        time.sleep(2)
        print("main "+time.ctime())

```

The diagram illustrates the execution flow of the code. The left terminal window shows a child process running from November 4, 2017, to November 5, 2017. The right terminal window shows a main process running from February 9, 2018, to February 10, 2018. Arrows point from the code blocks to the corresponding log entries in each terminal.

fork 创建

父进程所创建的进程叫子进程。每个进程都有一个不重复的进程 ID 号。或称 pid，它对进程进行标识。子进程与父进程完全相同，子进程从父进程继承了多个值的拷贝，如全局变量和环境变量。两个进程的唯一区别是 fork 的返回值。子进程接收返回值 0，而父进程接收子进程的 pid 作为返回值。os.fork 函数创建进程的过程是这样的。程序每次执行时，操作系统都会创建一个新进程来运行程序指令。进程还可调用 os.fork，要求操作系统新建一个进程。父进程是调用 os.fork 函数的进程。一个现有进程可以调用 fork 函数创建一个新进程。由 fork 创建的新进程被称为子进程（child process）。fork 函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回 0 值而父进程中返回子进程 ID。对于程序，只要判断 fork 的返回值，就知道自己是处于父进程中还是子进程中。

Local—>Enclosing--->Global--->Built-in

没有往大了找，但是不能在原来 scope 改变外边 scope 的值。

```
from multiprocessing import Process
import time

def t1(interval):
    while True:
        time.sleep(interval)
        print(time.ctime())
        print("child process")

if __name__ == "__main__":
    p = Process(target = t1, args=(3,))
    p.start()
    time.sleep(1)
    print("main process")

print("who are you")
print("!!!!!!")
```

```
who are you  
1111111  
main process  
who are you  
1111111  
Fri Feb 9 19:10:48 2018  
child process  
Fri Feb 9 19:10:51 2018  
child process
```

老师周一看到帮解答下，为什么会执行两次 `print ("who are you"); print ("111111")` .而不是一次。这两个语句不是主进程中的吗？

在 Windows 上，子进程会自动 import 启动它的这个文件，而在 import 的时候是会执行这些语句的

import 时如果你的代码没有写在 if __name__ == "__main": 里面的话会被自动执行

```
# 全局变量在多个进程中不共享
g_num = 100
def getTime(interval):
    global g_num
    while True:
        g_num += 100
        time.sleep(interval)
        print("in child num is %d"%g_num)

if __name__ == '__main__':
    p = Process(target=getTime, args=(1,))
    p.start()

    while True:
        g_num += 1
        print("Current num is %d"%g_num)
        time.sleep(1)
```

```
D:\Program Files\Python36\python.exe" "E:/redst/I
Current num is 101
Current num is 102
in child num is 200
Current num is 103
in child num is 300
同时
Current num is 104
in child num is 400
Current num is 105
in child num is 500
Current num is 106
in child num is 600
```

```
# 全局变量在多个进程中不共享
g_num = 100
def getTime(interval):
    global g_num
    while True:
        g_num += 100
        time.sleep(interval)
        print("in child num is %d"%g_num)

if __name__ == '__main__':
    p = Process(target=getTime, args=(2,))
    p.start()

    while True:
        g_num += 1
        print("Current num is %d"%g_num)
        time.sleep(1)
```

```
"D:\Program Files\Python36\python.exe" "E:/redhat/python/test.py"
Current num is 101
Current num is 102
Current num is 103
in child num is 200
Current num is 104
Current num is 105
in child num is 300
Current num is 106
```

```
g_num = 100
def getTime(interval):
    global g_num
    while True:
        g_num += 100
        time.sleep(interval)
        print("in child num is %d"%g_num)

if __name__ == '__main__':
    p = Process(target=getTime, args=(2,))
    p.start()

    while True:
        g_num += 1
        print("Current num is %d"%g_num)
        time.sleep(1)
```

```
Current num is 101
'Current num is 102
in child num is 200
Current num is 103
Current num is 104
in child num is 300
Current num is 105
Current num is 106
in child num is 400
Current num is 107
Current num is 108
in child num is 500
Current num is 109
Current num is 110
```

```
g_num = 100
def getTime(interval):
    global g_num
    while True:
        g_num += 100
        time.sleep(interval)
        print("in child num is %d"%g_num)

if __name__ == '__main__':
    p = Process(target=getTime, args=(3,))
    p.start()

    while True:
        g_num += 1
        print("Current num is %d"%g_num)
        time.sleep(1)
```

```
"D:\Program Files\Python36\python.exe" "E:/redhat/python/test.py"
Current num is 101
Current num is 102
Current num is 103
Current num is 104
in child num is 200
Current num is 105
Current num is 106
Current num is 107
in child num is 300
Current num is 108
```

```

import os

pid = os.fork()
if pid < 0:
    print("fork error")
elif pid == 0:
    print("child id %d, parent id %d"
          %(os.getpid(),           创建所在进程的id
              os.getppid()))
else:
    print("parent id %d, child id %d"
          %(os.getpid(), pid))

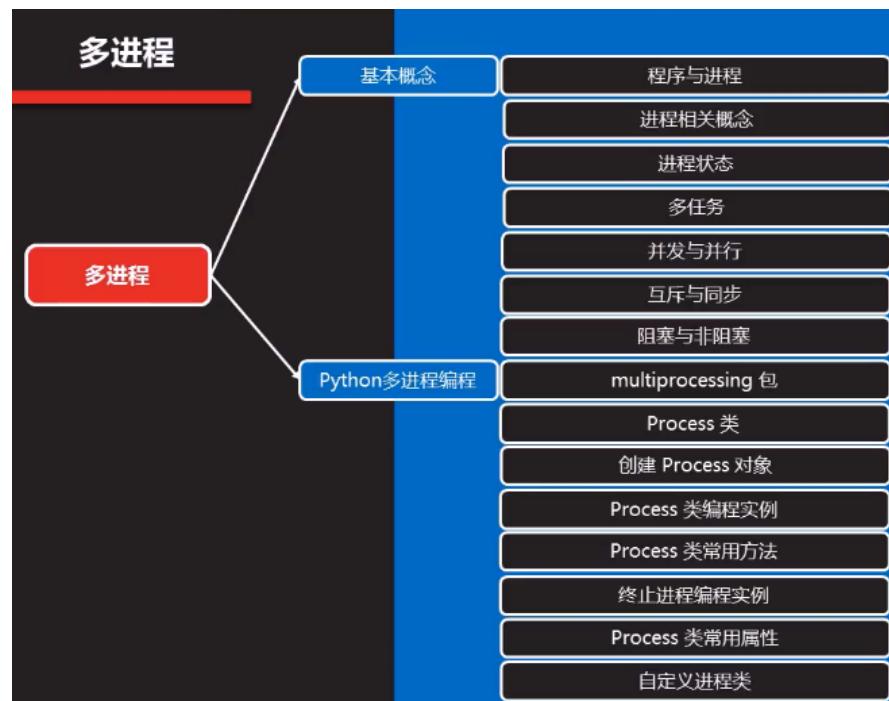
```

os.fork子进程创建返回0，若为父创建返回为子的pid

调用/调用fork，返回子pid

僵尸进程：百度 os 里面有

孤儿进程：解决问题方法，没人关了，系统分配一个进程管理



```

from multiprocessing import Process
import time

def t1(interval):
    for i in range(interval):
        time.sleep(1)
        print("child "+time.ctime())

if __name__ == '__main__':
    # 注意父进程会等子进程
    p = Process(target=t1, args=(5,))
    p.start()

    time.sleep(2)

    print("main process")
    #p.terminate() # Don't do it!!!
    #while True:
    #    time.sleep(2)
    #    print("main "+time.ctime())

```

```

import os
import time

pid = os.fork() # 注意父进程不会等子进程

if pid < 0:
    print("fork error")
elif pid == 0:
    time.sleep(1)
    print("child id %d, parent id %d"
          %(os.getpid(),
            os.getppid()))
else:
    print("parent id %d, child id %d"
          %(os.getpid(), pid))

```

6) 编程实践:

threading

A. Process 类的使用: myTimer.py

正常是主进程输入 print 完事，等上子进程循环 5 次，然后结束。

```

from multiprocessing import Process
import time

def t1(interval):
    for i in range(interval):
        time.sleep(1)
        print("child "+time.ctime())
        # 主进程结束，子进程马上结束。只能在这个位置

if __name__ == '__main__':
    # 注意父进程会等子进程
    p = Process(target=t1, args=(5,)) ↑
    p.daemon = True ↑
    p.start() ↑
    # 子进程开始
    #time.sleep(2)
    print("main") ↑
    print("main process") ↑

```

主进程没有睡两秒，那么直接执行print。然后程序结束
有睡两秒，那么看看子进程，他睡一秒，那么可以执行，
要是多于两秒，拿酒完了，不执行

```

1 from multiprocessing import Process
2 import time
3
4 def t1(interval):
5     for i in range(interval):
6         time.sleep(1)
7         print("child "+time.ctime())
8
9 if __name__ == '__main__':
10    # 注意父进程会等子进程
11    p = Process(target=t1, args=(5,))
12    # 守卫进程
13    p.start()
14

```

```

[1] tarena@edu:~/aid1709$ python3 mt1.py
[1] Traceback (most recent call last):
[1]   File "mt1.py", line 14, in <module>
[1]     p.daemon = True
[1]   File "/usr/lib/python3.5/multiprocessing/process.py", line 158,
[1]     in daemon
[1]     assert self._popen is None, 'process has already started'
[1] AssertionError: process has already started
[1] child Sat Nov  4 17:07:54 2017
[1] child Sat Nov  4 17:07:55 2017
[1] child Sat Nov  4 17:07:56 2017

```

写在下面会报错。

```
#!/usr/bin/python
from multiprocessing import Process
import time
def CallTime():
    while True:
        print("Now this is "+time.ctime())
        time.sleep(27000)
if __name__=="__main__":
    print("you are tired, you need some time to relax")
    p = Process(target=CallTime)
    p.daemon = False
    p.start()
    p.join()
```

这句话的含义

在大多数编排得好一点的脚本或者程序里面都有这段 if __name__ == 'main':，虽然一直知道他的作用，但是一直比较模糊，收集资料详细理解之后与打架分享。

1、这段代码的功能

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。

举个例子，下面在 test.py 中写入如下代码：

```
print "I'm the first."
if __name__=="__main__":
    print "I'm the second."
并直接执行 test.py，结果如下图，可以成功 print 两行字符串。即，if  
__name__=="__main__": 语句之前和之后的代码都被执行。
```

```
C:\Users\Evan\Desktop>python test.py
I'm the first.
I'm the second.

C:\Users\Evan\Desktop>
```

然后在同一文件夹新建名称为 import_test.py 的脚本，只输入如代码：

```
import test
```

执行 import_test.py 脚本，输出结果如下：

```
C:\Users\Evan\Desktop>python import_test.py
I'm the first.
```

只输出了第一行字符串。即，if __name__=="__main__": 之前的语句被执行，之后的没有被执行。

2、运行的原理

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，

```
test.py - C:\Users\Evan\Desktop\test.py (2.7.11)
File Edit Format Run Options Window Help
print "I'm the first."
print __name__
if __name__ == "__main__":
    print "I'm the second."



C:\选择命令提示符

C:\Users\Evan\Desktop>python test.py
I'm the first.
main
I'm the second.
```

可以看出，此时变量 `__name__` 的值为 “`__main__`”；

再执行 import_test.py，模块内容和执行结果如下：

```
import_test.py - C:\Users\Evan\Desktop\import_test.py (2.7.11)
File Edit Format Run Options Window Help
import test


C:\选择命令提示符

C:\Users\Evan\Desktop>python import_test.py
I'm the first.
test
```

此时，test.py 中的 `__name__` 变量值为 `test`，不满足 `__name__ == "__main__"` 的条件，因此，无法执行其后的代码。

Python daemon 进程：

1. 主进程已结束，子进程就跟着结束；

2. 可以随时被杀死;
3. 写在 start 之前, 系统默认是 False;
4. 后台进程;
5. 与 linux 机制进程不同, 一般不把写入东西之类的进程设置为 daemon 进程。

`p.join()`会让父进程等待子进程, 是让 daemon 失效; 与 daemon 同时存在, 那么 daemon 就无效了。他是阻塞方法。

```
from multiprocessing import Process
import time

def t1(interval):
    for i in range(interval):
        time.sleep(1)
        print("child "+time.ctime())

if __name__ == '__main__':
    # 注意父进程会等子进程
    p = Process(target=t1, args=(5,))
    p.daemon = True # 守卫进程
    p.start()
    p.join()

# time.sleep(2)

print("main process")
# p.terminate() # Don't do it!!!
# while True:
#     time.sleep(2)
#     print("main "+ time.ctime())
```

优先队列 priority Queue;

知识

队列: FIFO (First In First Out) 先进先出;

堆栈, 栈 stack: LIFO (Last In First Out) 后进先出; 堆 heap

```
In [3]: q.
q.all_tasks_done      q.maxsize          q.qsize
q.empty               q.mutex             q.queue
q.full                q.not_empty        q.task_done
q.get                 q.not_full         q.unfinished_tasks
q.get_nowait          q.put
q.join
```

Repr 和 str 区别

`__repr__` 和 `__str__` 这两个方法都是用于显示的, `__str__` 是面向用户的, 而 `__repr__` 面向程序员。

打印操作会首先尝试 `__str__` 和 `str` 内置函数(`print` 运行的内部等价形式), 它通常应该返回一个友好的显示。

`__repr__` 用于所有其他的环境中: 用于交互模式下提示回应以及 `repr` 函数, 如果没有使用 `__str__`, 会使用 `print` 和 `str`。它通常应该返回一个编码字符串, 可以用来重新创建对象, 或者给开发者详细的显示。

当我们想所有环境下都统一显示的话，可以重构`__repr__`方法；当我们想在不同环境下支持不同的显示，例如终端用户显示使用`__str__`，而程序员在开发期间则使用底层的`__repr__`来显示，实际上`__str__`只是覆盖了`__repr__`以得到更友好的用户显示。

```
class Test(object):
    def __init__(self, value='hello, world!'):
        self.data = value

>>> t = Test()
>>> t
<__main__.Test at 0x7fa91c307190>
>>> print t
<__main__.Test object at 0x7fa91c307190>

# 看到了么？上面打印类对象并不是很友好，显示的是对象的内存地址
# 下面我们重构下该类的__repr__以及__str__，看看它们俩有啥区别
```

```
# 重构__repr__
class TestRepr(Test):
    def __repr__(self):
        return 'TestRepr(%s)' % self.data

>>> tr = TestRepr()
>>> tr
TestRepr(hello, world!)
>>> print tr
TestRepr(hello, world!)

# 重构__str__
class TestStr(Test):
    def __str__(self):
        return '[Value: %s]' % self.data

>>> ts = TestStr()
>>> ts
<__main__.TestStr at 0x7fa91c314e50>
>>> print ts
[Value: hello, world!]
```

队列的总结

`queue.PriorityQueue` 优先队列；

`multiprocessing.Queue` 进程间通信时进程间传递的数据；

`multiprocessing.Manager.Queue` 进程池的进程间传递数据来用；

队列的总结

进程池中的进程和我们自己创建的进程：

进程池中不需要关心我们任务分配给进程的策略；

而自己创建的进程则需要关心细节；

对于进程池而言，`python` 提供了一些配套措施，比如对于队列的管理；

`multiprocessing.Manager.Queue`

B. 进程调度：

时间片：就绪等待执行等待的东西；

优先级：

7) 进程池初步：

Pool.apply

实践：

8) 简单说明下进程间通信：

IPC, RPC;

******进程使用的总结******

- 1.在构建较大应用程序之前，仔细阅读在线文档，官方文档中涉及了一些可能出现的更奇怪的问题；
- 2.确保进程之间传递的所有数据都能序列化；
- 3.避免使用共享数据，尽可能使用消息传递和队列；在使用消息传递时，不必过于担心同步，锁定和其他问题。当进程的数量增长时，往往还能提供更好的扩展；
- 4.在必须运行在单独进程中的函数内部，不要使用全局变量而应当显示的传递参数；
- 5.注意关闭进程的方式。使用一个良好的终止模式，不要使用 terminate();
可能导致僵尸进程；
- 6.multiprocessing.Manger 管理器和代理的使用与分布式计算中的多个概念密切相关（如分布式对象）。参考与分布式计算的相关书籍会有所帮助；
- 7.os 尽管此模块可以在 Windows 上工作，但还是要读官方的文档看一些微妙的细节。fork();
- 8.最重要一点：**尽量让事情变得简单**；

作业：

1) 我们的优先级 1-20；我们要做一个模拟 OS 的调度程序：

某一时刻，四个进程：

Watch TV; random 优先级 import random

Listen to music; random 优先级

Print Doc; 4+q.qsize()*0.5

Write Doc; 4+q.qsize()*0.5

优先队列 PriorityQueue

***** from queue import PriorityQueue *****

4+q.qsize()*0.5 == radom.randint(1,20)+q.qsize()*0.5

***** Found a bug****:

Print Doc 和 Write Doc 出现乱序；

input: 四个进程的信息;

output: 打印出四个进程谁先输出谁后输出;

2) 多进程定期器: 每一个小时, 你的定时器提示你:

不要 Coding, 休息一下眼睛吧(

提示的同时显示当前进程的 pid,name)

os 模块设置优先级;

3) 大作业: 通过一个进程池来实现

文件夹下大量文件的拷贝; 至少 1000 个文件;

有方法证明你拷贝的文件是 Ok 的; (三次课之后)

****hash 哈希:

```
import hashlib

digest = hashlib.sha512() #! 创建 sha512 算法

f = open("test.txt")

while True:

    chunk = f.read(BUFSIZE)

    if not chunk:

        break

    digest.update(chunk.encode('utf8')) #!

f.close()

print(digest.hexdigest()) #!
```

override 可以翻译为覆盖, 从字面就可以知道, 它是覆盖了一个方法并且对其重写, 以求达到不同的作用。对我们来说最熟悉的覆盖就是对接口方法的实现, 在接口中一般只是对方法进行了声明, 而我们在实现时, 就需要实现接口声明的所有方法。除了这个典型的用法以外, 我们在继承中也可能会在子类覆盖父类中的方法。在覆盖要注意以下的几点:

- 1、覆盖的方法的标志必须要和被覆盖的方法的标志完全匹配, 才能达到覆盖的效果;
- 2、覆盖的方法的返回值必须和被覆盖的方法的返回一致;
- 3、覆盖的方法所抛出的异常必须和被覆盖方法所抛出的异常一致, 或者是其子类;
- 4、被覆盖的方法不能为 **private**, 否则在其子类中只是新定义了一个方法, 并没有对其进行覆盖。

`overload` 对我们来说可能比较熟悉，可以翻译为重载，它是指我们可以定义一些名称相同的方法，通过定义不同的输入参数来区分这些方法，然后再调用时，VM 就会根据不同的参数样式，来选择合适的方法执行。在使用重载要注意以下的几点：

- 1、在使用重载时只能通过不同的参数样式。例如，不同的参数类型，不同的参数个数，不同的参数顺序（当然，同一方法内的几个参数类型必须不一样，例如可以是 `fun(int, float)`，但是不能为 `fun(int, int)`）；
 - 2、不能通过[访问权限](#)、[返回类型](#)、抛出的异常进行重载；
 - 3、方法的异常类型和数目不会对重载造成影响；
 - 4、对于继承来说，如果某一方在父类中是[访问权限](#)是 `priavte`，那么就不能在子类对其进行重载，如果定义的话，也只是定义了一个新方法，而不会达到重载的效果。
- 1、方法的覆盖是子类和父类之间的关系，是垂直关系；方法的重载是同一个类中方法之间的关系，是水平关系
- 2、覆盖只能由一个方法，或只能由一对方法产生关系；方法的重载是多个方法之间的关系。
 - 3、覆盖要求参数列表相同；重载要求参数列表不同。
 - 4、覆盖关系中，调用那个方法体，是根据对象的类型（对象对应存储空间类型）来决定；重载关系，是根据调用时的实参表与形参表来选择方法体的。

day02

1) 进程与线程

进程：CPU 分配资源的单位；

线程：CPU 执行流程的单位；是轻量级的进程；

3) `threading` 模块的使用：

```
Thread(group=None, target=None, name=None, args=(), kwargs={}):
```

创建一个新的实例。Group 的值是 `None`，为以后的扩展而保留。

2) Python 中线程的问题

GIL(Global Interperter Lock) 全局解释锁；

由于历史原因，CPython(C)天生有这样一个缺陷，

Python 的多线程并不是真正的多线程；

例子：切换虚拟机的核数看代码实验；

在真正的并行中它不是真正的并行；

我们在用 Python(CPython)编程时，什么时候用多线程，什么时候用多进程：

***** 1) 在 CPU 计算密集情况下，使用多进程；

能用进程池就用进程池，通过实践检验到底需要几个进程效果最佳；

对于 Python 的每一个进程，Python 会单独起一个解释器；

多进程可以真正实现并行；

***** 2) 在 I/O 密集的情况下，使用多线程；

突破 GIL，可以尝试自己来用 C 语言来突破；

4) 一些简单的 threading 的 Demo 演示

1. 用 thread 来实现定时器；

2. 用面向对象的方式实现定时器；

3. thread 使用 Queue 的 Demo

5) Python 多线程，多进程的简单总结：

1. Python 多线程适合于 I/O 密集型的场景；

2. Python 多进程适合于 CPU 计算密集型的差干净；

6). 关于进程与线程的一点内容补充

1.关于 OS 模块：OS 模块可以获取信息很多相关的信息，但是和 OS 相关性比较；

7) 关于线程进程的可能遇到的面试题：

1.进程有哪三态，哪五态；相互之间怎么切换？

2.什么是死锁，例举出一个死锁的例子,怎么避免死锁？

3.进程和线程有什么区别？

4.多线程同步与互斥有几种实现方法？都是什么？

5.Python 的多进程，多线程分别适合于什么场景下使用？爬虫适合使用多进程还是多线程？

大作业：通过一个进程池来实现

文件夹下大量文件的拷贝；至少 1000 个文件；

有方法证明你拷贝的文件是 Ok 的；

****hash 哈希：

```
import hashlib
```

```
digest = hashlib.sha512() #! 创建 sha512 算法
```

```
f = open("test.txt")

while True:
    chunk = f.read(BUFSIZE)
    if not chunk:
        break
    digest.update(chunk.encode('utf8')) #!
f.close()

print(digest.hexdigest()) #!
```

```
生产者消费者模型;

From multiprocessing import Queue

From multiprocessing import Process

#消费者逻辑

Def consumer(input_q):

    While not input_q.empty():
        Print(input_q.get())

    #生产者逻辑

Def produce(output_p):

    For I in range(seqeunce):
        Output_p.put(i) #放入东西

    If __name__=="__main__":
        Q=Quene()
        Con_p=Process(target=consumer,args=(q,))
        Sequence =[1,2,3,4,5]
        Producer(sequence,q)
```

```
# 消费者逻辑
def consumer(input_q):
    time.sleep(2)
    while not input_q.empty():
        print(input_q.get())

# 生产者逻辑
def producer(sequeunce, output_p):
    for i in sequeunce:
        output_p.put(i)

if name == ' main ':
    q = Queue()
    con_p = Process(target=consumer, args=(q,))
    con_p.start()

sequence = [1,2,3,4,5]
producer(sequence, q)

from multiprocessing import Queue
from multiprocessing import Process
import time

# 消费者逻辑
def consumer(input_q):
    while True:
        # if not input_q.empty():
        #     print(input_q.get())
        # else:
        #     break
        item = input_q.get()
        # 用None作为标志，来判断生产者是否已经结束了生产
        if item == None:
            break
        print(item)

# 生产者逻辑
def producer(sequeunce, output_p):
    for i in sequeunce:
        output_p.put(i)

if name == ' main ':
    q = Queue()
    con_p = Process(target=consumer, args=(q,))
    con_p.start()
```

生产者消费者模型：

```

from multiprocessing import Queue
from multiprocessing import Process
import time

# 消费者逻辑
def consumer(input_q):
    while True:
        # if not input_q.empty():
        #     print(input_q.get())
        # else:
        #     break 遍历put传入的值
        item = input_q.get()
        # 用None作为标志, 来判断生产者是否已经结束了生产
        if item == None:
            input_q.put(6) 在末尾加上一个元素6
            空则跳出 break
            print(item)

# 生产者逻辑
def producer(sequence, output_p):
    for i in sequence:
        output_p.put(i) 加到了q对象队列中了。

```

```

if __name__ == '__main__':
    q = Queue()
    con_p = Process(target=consumer, args=(q,))
    # 例如多进程的一部分
    con_p.start() 主进程生成子进程

    sequence = [1, 2, 3, 4, 5]
    producer(sequence, q)
    # 在这里加入生产结束标志
    q.put(None)

    con_p.join() 等待子进程结束, 我感觉父进程已经运行
    print(q.get()) 而没有输出

```

银行存取款模型：

队列 queue FIFO	管道 pipe	文件 file	互斥锁 mutex
信号 Semaphore	时间 event	共享内存 share memory	

1) 进程间通信的常用方式:

进程间传递数据: A, B, C

A*队列: FIFO(first in, first out)

这里好好看看 queue, 老师没好好讲

**from multiprocessing import Queue

multiprocessing.Queue

multiprocessing.Queue 与 queue.PriorityQueue

multiprocessing.Queue 与一般数据结构中的队列的差别:

前者要符合多进程的特性: 序列化和反序列化等;

对数据做了一些通信上的加工, 保证多进程下安全性;

所以我们在使用多进程的数据通信, 不需要考虑多进程可能带来的安全隐患, 直接使用

multiprocessing.Queue 在进程间传递数据是 Python 比较提倡的一种方式;

1. from Queue import Queue

这个是普通的队列模式, 类似于普通列表, 先进先出模式, get 方法会阻塞请求, 直到有数据 get 出来为止

```
2.from multiprocessing.Queue import Queue
```

这个是多进程并发的 Queue 队列，用于解决多进程间的通信问题。普通 Queue 实现不了。

例如来跑多进程对一批 IP 列表进行运算，运算后的结果都存到 Queue 队列里面，这个就必须使用 multiprocessing 提供的 Queue 来实现。

模块 Queue

队列Queue (续2)



- 一些常用方法
 - q.get([block [, timeout]])
如果q为空，此方法将阻塞，直到队列中有项可用为止；
 - q.get_nowait() 同q.get(False) 将引发Queue.Empty异常；
 - join()阻塞调用线程，直到队列中的所有任务被处理掉。只要有数据被加入队列，未完成的任务数就会增加。当消费者线程调用task_done()
(意味着有消费者取得任务并完成任务)，未完成的任务数就会减少。
当未完成的任务数降到0，join()解除阻塞；
 - q.close() 关闭队列，防止队列中加入更多的数据。调用此方法时，后台线程将继续写入那些已入队列但尚未写入的数据，但将在方法完成时马上关闭。如果q被垃圾回收，将自动调用此方法。关闭队列不会再在队列消费者中生成任何类型的数据结束信号或异常；
 - q.empty() 判空，不一定可靠；
 - q.qsize() 不一定可靠。在某些系统中可能会引起NotFoundError
异常；

q.get([block[,timeout]])。如果 q 为空，此方法将阻塞，直到队列中有项可用为止

q.get_nowait() 同 q.get(False)

q.join()阻塞调用程序，直到队列中的所有任务都处理掉。只要有数据被加入队列，未完成的任务数就会增加。当消费者线程调用 task_done()，未完成的任务数就会减少，当未完成的任务数量降到 0，join()结束阻塞。

q.close()关闭队列，防止队列中加入更多的数据，调用此方法时，后台线程将继续写入那些已在队列但尚未写入的数据，但将在方法完成时马上关闭。如果 q 被垃圾回收，将自动调用此方法，关闭队列不会再在队列消费者中生成任何类型的数据结束信号或者异常。

q.put()放入队列

q.get()获取队列

q.empty()判断是否为空，不可靠

q.qsize()队列长度，不可靠，在某些系统中会引用 NotImplementedError

B.文件:

open, read, write, seek, tell, close;

mode:模式

r;读取

w;写入

a;最佳

b;二进制

posix 早期时各种硬件设备都是私有的接口;

文件化;

文件操作方式:

```
posix
早期时各种硬件设备都是私有的接口;
    文件化;
f = open("test.txt")
f.read()
f.write()
f.close() #!!! 别忘了close
#with open("test.txt") as f:
```

with open("test.txt") as f: #提倡使用文件这种方式最最笨, 很慢;

但最普世, 最通用;

C*.共享内存:

优点: 快速在进程间传递数据, 比队列;

缺点: 有安全风险, 需要考虑加锁的问题;

B. 管道: 一端只能发送, 一端只能接收, 不用的一端 close;, 也就是说, 接收端必须把输出功能关了

进程间通讯 IPC(Inter-Process Communication)两个进程或者线程间传递数据或者信号的一些技术或者方法。

进程间通讯 RPC(Remote Procedure Protocol)远程过程调用协议, 通过一种网络从远程计算机程序请求服务, 而不是需要了解底层网络技术的协议。RPC 协议假定某些传输协议存在, 例如 TCP 或者 UDP, 为了通讯程序间携带信息数据。在 OSI 网络通讯模型中, RPC 跨越了传输层和应用层。RPC 使的开发包括网络分布式多程序在内的应用程序更加容易。

匿名管道: 父子进程,

命名管道: 没有亲缘关系的进程之间也可以通信;

管道Pipe

- 管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。调用pipe系统函数即可创建一个管道。有如下特质：
 1. 其本质是一个伪文件(实为内核缓冲区)
 2. 由两个文件描述符引用，一个表示读端，一个表示写端
 3. 规定数据从管道的写端流入管道，从读端流出。
 - 管道的原理：管道实为内核使用环形队列机制，借助内核缓冲区(4k)实现。
 - 管道的局限性：
 - A 数据自己读不能自己写；
 - B 数据一旦被读走，便不在管道中存在，不可反复读取；
 - C 由于管道采用半双工通信方式。因此，数据只能在一个方向上流动。
 - D 只能在有公共祖先的进程间使用管道。
- 常见的通信方式有，单工通信、半双工通信、全双工通信。

进程间传递信号

D*.互斥量：

```
from multiprocessing import Lock

lock = Lock()

# 注意锁的粒度，尽快最小化，代价最小

lock.acquire()# 上锁

money.value += 1

lock.release()#解锁

# 递归锁
```

RLock:

```
def f():

    lock.acquire() # 上锁

    f1()

    money.value += 1

    lock.release()#解锁

def f1():

    f0()

    # 递归锁
```

E.信号量：可以同时提供给 N 个消费者服务，

互斥量其实是一种特殊的信号量，特殊在一次只有一个消费者；

E.事件：Event

一个进程通过 Event 通知另一个进程做某件事件的时机到了，我们就可以做这件事了。

例子：一个写的程序在多个读的进程中，可以通过这种方式来通知这些进程们，

可以来读数据了。e.wait([timeout]), e.set();

网络编程

H.Socket:

面试题：

实现一个函数 f

输入：一个 list:

输出：一个 list，list 中每一个元素除以下标为 0 那个元素

(1) 除以 0 怎么办

(2) 测试用例

一个函数

1) 前置条件 : assert

2) 做事

3) 后置检查

****进程间通信****:

在多个进程中传递数据一般而言用队列就可以了（multiprocessing.Queue）；

如果要快速高效的话可以考虑共享内存；

互斥量(multiprocessing.Lock)主要用于在多个进程间加锁；

在多个进程中传递信号可以用信号量（multiprocessing.Semaphore），事件 Event；

互斥量可以看作一种特殊的信号量；互斥量可以看作是一个消费者只有一个的信号量；

不同的进程通信方式有不同的优点和缺点。因此，对于不同的应用问题，要根据问题本身的情况来选择进程间的通信方式。

为了高效，可以考虑：共享内存+信号量；

****进程池****:

fork: 父进程不会等子进程，父子进程都要做事；

Process: 父进程会等子进程，父子进程都要做事；

Pool: 父进程等子进程，父进程不做事；

```
p = Pool()
```

```
p.apply_async() p.map()
```

```
p.close()
```

```
p.join()
```

apply 不要用，不然可能比单进程还慢；

面试题：

实现一个函数 f;

输入：一个整形数 list;

输出：一个整形数 list,

list 中的每一个元素除以下标为 0 的那个元素；

1) 判断 list 是否为 0, 除以 0 怎么办；

2) 测试用例； for 怎么写的；

一个函数的实现规范：

1) 前置条件； assert;

2) 做事；

3) 后置检查； 返回结果；

面试题：

有 1000 个一模一样的瓶子，其中 999 瓶是普通的水，有一瓶毒药，任何动物喝下毒药后一周的时间会死亡。现在你有 10 只小老鼠，给你一周的时间，你怎么找出那瓶毒药。

1) 对瓶子编号： 0 -- 999

0000000000 1111100111

1 -- 1000:

0000000001 m1

0000000010 m2

0000000011 m1,m2

0000000100 m3
0000000101 m1, m3
...
1111101000

0001111111 m1,m2,m3,m4,...m7

dict = {...0001111111:127}

dict[0001111111] = 127;

2) 对老鼠编号:

m1 - m10

作业: 1) 使用队列实现生产者消费者模型;

生产者生产[0,1,2,3,4,5,6,7,8,9,10],

消费者能够在处理完了 list 序列后, 退出;

2) 使用互斥锁去完成银行存取款操作,

存款 10000, A 取 10 次, 每次取 100;

B 存 5 次, 每次存 200;

3) 用面向对象的方式实现第一个次作业的第二题 myTimer;

-----201711月7日

1)把共享内存 Demo.py 中的数组 Array 中的元素值改为 0, 1, 2, ..., 9;

2)想办法显示一下进程池 apply.py 中进程池的调度细节;

3)运行代码:

进程池 mapDemo.py

进程池 apply.py

进程池 apply_async.py

体会各个 API 使用过程;

day03

回顾

队列Queue (续)

- 一些常用方法
 - task_done()意味着之前入队的一个任务已经完成。由队列的消费者线程调用。每一个get()调用得到一个任务，接下来的task_done()调用告诉队列该任务已经处理完毕。如果当前一个join()正在阻塞，它将在队列中的所有任务都处理完时恢复执行（即每一个由put()调用入队的任务都有一个对应的task_done()调用）。
 - join()阻塞调用线程，直到队列中的所有任务被处理掉。只要有数据被加入队列，未完成的任务数就会增加。当消费者线程调用task_done()（意味着有消费者取得任务并完成任务），未完成的任务数就会减少。当未完成的任务数降到0，join()解除阻塞。

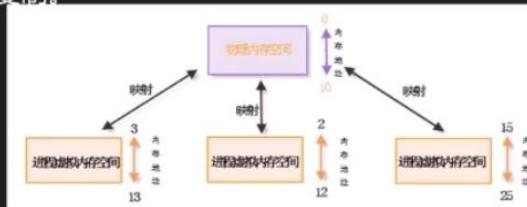
队列Queue (续2)

- 一些常用方法
 - q.get([block [, timeout]])
如果q为空，此方法将阻塞，直到队列中有项可用为止；
 - q.get_nowait() 同q.get(False) 将引发Queue.Empty异常；
 - join()阻塞调用线程，直到队列中的所有任务被处理掉。只要有数据被加入队列，未完成的任务数就会增加。当消费者线程调用task_done()（意味着有消费者取得任务并完成任务），未完成的任务数就会减少。当未完成的任务数降到0，join()解除阻塞；
 - q.close() 关闭队列，防止列队中加入更多的数据。调用此方法时，后台线程将继续写入那些已入队列但尚未写入的数据，但将在方法完成时马上关闭。如果q被垃圾回收，将自动调用此方法。关闭队列不会在队列消费者中生成任何类型的数据结束信号或异常；
 - q.empty() 判空，不一定可靠；
 - q.qsize() 不一定可靠。在某些系统中可能会引发NotImplementedError 异常；

信号量

共享内存Share Memory

- 最简单的进程间通信方式，它允许多个进程访问相同的内存，一个进程改变其中的数据后，其他的进程都可以看到数据的变化。
- 共享内存是进程间最快速的通信方式：
 - 进程共享同一块内存空间。
 - 访问共享内存和访问私有内存一样快。
 - 不需要系统调用和内核入口。
 - 不造成不必要的内存复制。



信号量Semaphore

- 信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。

信号量Semaphore (续)

- 特点
 - 1.信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。
 - 2.信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。
 - 3.每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

互斥量Mutex

- 互斥量是一个可以处于两态之一的变量：解锁和加锁。
- 如果不需要信号量的计数能力，有时可以使用信号量的一个简化版本，称为互斥量（mutex）。互斥量仅仅适用于管理共享资源或一小段代码。由于互斥量在实现时既容易又有效，这使得互斥量在实现用户空间线程包时非常有用。



信号量 multiprocessing.semaphore(value=4)

以一个停车场的运作为例。简单起见，假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆直接进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复。

在这个停车场系统中，车位是公共资源，每辆车好比一个[线程](#)，看门人起的就是信号量的作用。

semaphore acquire 加锁 get_value release 解锁

multiprocessing.current_process().name 获取当前进程的名字。

multiprocessing.os.get_pid() 获取当前进程的 pid getppid

```

import multiprocessing
import time

def consumer(s):
    # 信号量的使用与互斥锁类似
    s.acquire()      进程名字
    print(multiprocessing.current_process().name
          +"get")
    time.sleep(1)
    s.release()
    print(multiprocessing.current_
          +"release")

if name == ' main ':
    # 把信号值置为2, 一次可以给两个消费者提供服务
    s = multiprocessing.Semaphore(2)

    # 起5进程, 也就是说有5个消费者
    for i in range(5):
        p = multiprocessing.Process(target=consumer,
                                    args=(s,))
        p.start()

    print("Main End")

```

Process-1get
 Process-2get
 Main End
 Process-1release
 Process-3get
 Process-2release
 Process-4get
 Process-3release
 Process-5get
 Process-4release
 Process-5release

互斥量 multiprocessing.mutex

一次只能有一次进程执行，完了才能下一个。

事件 event

```
e=multiprocessing.event()
```

```
e.clear()
```

```
e.is_set()
```

```
e.set()
```

```
e.wait()
```

Multiprocessing.Process(name="block",target="",args=())

Name 给进程命名。

懒惰与贪婪

```
In [9]: print(sys.path)  当前路径
[ '/usr/bin', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-linux-gnu', '/usr/lib/python3.5/lib-dynload', '/home/tarena/.local/lib/python3.5/site-packages', '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages', '/usr/lib/python3/dist-packages/IPython/extensions', '/home/tarena/.ipython' ]
```

共享内存

```
import multiprocessing
from datetime import datetime, timedelta

def trans(a, size):
    # 打印出具体花了多久访问这些数组元素
    t = datetime.now()
    for i in range(size):
        print(a[i])
    print('消耗了 %s' % (datetime.now() - t))  # 运行时差

if __name__ == '__main__':
    print("test share memory:")
    num = 10  # 数组的类型是整形的，大小是100
    a = multiprocessing.Array('i', num)  # 类型 长度
    p = multiprocessing.Process(target=trans,
                                args=(a, num))
    p.start()
```

进程池

用好了比单进程快。

同步？异步？映射？

进程池简介

- 什么是进程池
 - 预先创建一组子进程，当有新任务来时，主进程通过某种方法分配给该组进程中的一个进程完成此任务。这种成组管理子进程的模式与实现称为进程池
- 为什么需要进程池
 - 进程创建、销毁需要CPU的时间开销
 - 预先创建，以**空间换时间**，提供性能
 - 通过合理分配任务，可以提高性能

Python 进程池的 map 机制

Pool 类

- Python 进程池实现，Pool 类
 - 管理一个包含工作进程的进程池
 - 支持同步、异步、映射方式添加任务到进程池
 - 方便对多个函数或分布式数据并行执行和运算
- 导入方法参考
 - `from multiprocessing import Pool`
 - `import multiprocessing`
`multiprocessing.Pool`
- 注意：
 - Pool类必须有 `_main_` 模块导入
 - 进程池对象的方法只能由创建进程池的进程调用

Pool 类 (续1)

- Pool 类主要方法
 - 构造函数，创建进程池

```
class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]]])
```

 - `processes`：进程池中进程数量，缺省则和CPU数量相同
 - `initializer`：进程初始化函数，如果`initializer`不是`None`，则每个工作进程在启动时将调用`initializer(*initargs)`。
 - `maxtasksperchild`：3.2版本以后支持，工作进程在退出并由新工作进程替换之前可以完成的任务数，以使未使用的资源可以释放。
默认 `maxtasksperchild` 是`None`，表示工作进程的生存时间将与进程池相同。
 - `context`：指定用于启动工作进程的上下文，通常缺省，3.4 版本以后

Pool 类 (续2)

达内

- Pool 类主要方法

- apply(func[, args[, kwds]])
向线程池添加任务，阻塞方式
func 任务执行的函数，args参数元组，和kwds关键字参数
- apply_async(func[, args[, kwds[, callback[, error_callback]]]])
apply()方法的一个变体，它返回一个结果对象，非阻塞 异步
callback，如果有任务结束将调用此函数，该函数一个参数
- map(func, iterable[, chunkszie]) 生成迭代对象类似
- close()
关闭进程池，阻止任何更多的任务提交到进程池。一旦所有任务完成，工作进程将退出。
- terminate()
立即停止工作进程，而不需要完成未完成的工作。当进程池对象被垃圾收集，terminate()将被立即调用。
- join()
等待进程池所有进程结束，在使用join()之前，必须调用close()或terminate()。

进程池编程实例

- 创建进程池完成：apply 添加任务

```
# -*- coding: utf-8 -*-
from multiprocessing import Pool
import time

def func(str):
    print("Pool:", str)
    time.sleep(3)
    print("Pool:", str, "end")

if __name__ == "__main__":
    p = Pool(processes = 3)
    time1 = time.time()
    for i in range(4):
        msg = "apply %d" % (i)
        p.apply(func, (msg,))
    print("for end")
    p.close()
    p.join()
    print("Main end")
    print(time.time() - time1)
```

进程池编程实例 (续1)

- 创建进程池完成：apply_async 添加任务

```
# -*- coding: utf-8 -*-
from multiprocessing import Pool
import time

def func(str):
    print("Pool:", str)
    time.sleep(3)
    print("Pool:", str, "end")

if __name__ == "__main__":
    p = Pool(processes = 3)
    time1 = time.time();
    for i in range(4):
        msg = "apply_async %d" % (i)
        p.apply_async(func, (msg,))
    print("for end")
    pool.close()
    pool.join()
    print("Main end")
    print(time.time() - time1)
```

阻塞非阻塞，同步异步没关系

```
from multiprocessing import Pool
import time

def Cal(n):
    # 前n个数的平方和
    sum = 0
    for i in range(n):
        sum += i*i
    return sum

if __name__ == '__main__':
    calCount = 20000 #计算到calCount-1的平方和
    checkNum = 1001 #验证下标为checkNum的list元素下标

    # 用进程池来算平方和，看看我们的计算时间
    t1 = time.time()#运行开始时间
    p = Pool()
    result = p.map(Cal, range(calCount)) # 计算到9999
    p.close() # close 必须在join之前，不在接受
    p.join() # join阻塞等待
    print(result[checkNum])
    print("进程池花费了 %f"%(time.time()-t1))

    # 用单个进程做平方和，看看我们的计算时间
    t2 = time.time()
    result2 = []
    for i in range(calCount):
        result2.append(Cal(i))
    print(result2[checkNum])
    print("单个进程花费了 %f"%(time.time()-t2))#时差
```

day04

管理器 manger.queue

队列的总结**

queue.PriorityQueue 优先队列：优先级别，模仿系统调度

multiprocessing.Queue 进程间通信时进程间传递的数据进程间通讯

multiprocessing.Manager.Queue 进程池的进程间传递数据来用：

```

if name == "main":
    pool = Pool(processes=3)
    time1 = time.time()
    for i in range(4):
        msg = "apply async %d" % (i)
        pool.apply_async(func, (msg, ), callback=cbFunc)

    #print ("for end")                                cfun和func异步

#pool.close()# close 必须在join之前
#pool.join()# join阻塞等待
print ("Main end")
print(time.time() - time1)

```

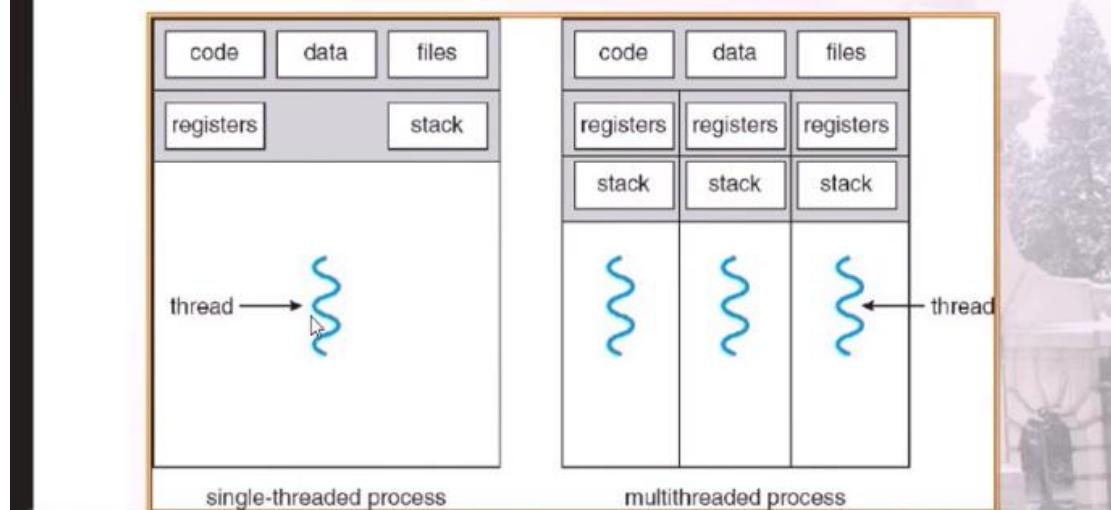
线程

什么是线程

达内教育

- 线程是CPU执行的基础单元，是轻量级的进程

- 线程是进程的执行单元，一个进程中可以有多个线程
- 进程所具有的动态含义，是通过线程来体现的。



threading模块的使用

- Thread(group=None, target=None, name=None, args=(), kwargs={}):
- 创建一个新的实例。Group的值是None，为以后的扩展而保留。target是一个可调用对象，线程启动时，run()方法将调用此对象，默认值是None；name是线程名称；默认将创建一个“Thread-N”格式的唯一名称。args是传递给target函数的参数元组。kwargs是传递给target的关键字参数的字典；
- Thread实例t支持以下方法和属性：
 - 1.start()
 - 2.run()
 - 3.join([timeout]) 等待直到线程终止或出现超时为止。timeout是一个浮点数，用于指定以秒为单位的超时时间。线程不能连接自身，而且在线程启动之前就连接它将出现错误；
 - a. is_alive
 - b. name
 - c. ident 整数线程标识符；
 - d. daemon 线程的布尔型后台标志；

```
from threading import Thread
import time

def getTime(interval):
    while True:
        time.sleep(interval)
        print("child thread %s"%time.ctime())

if __name__ == '__main__':
    # 创建线程计时器
    t = Thread(target=getTime, args=(1,))
    t.start()
    t
```

```
from threading import Thread
import time

g_num = 100
def f1():
    global g_num
    for i in range(5):
        g_num += 1
    print("in f1 g num is %d"%g_num)

def f2():
    time.sleep(1)
    global g_num
    print("in f1 g num is %d"%g_num)

if name == ' main ':
    # 创建两个线程
    t1 = Thread(target=f1)
    t1.start()
    t2 = Thread(target=f2)
    t2.start()
```

```
from threading import Thread
import time

def f1():
    g_num = 100
    g_num += 1
    print("in f1 g num is %d"%g_num)

def f2():
    g_num = 100
    time.sleep(1)
    print("in f2 g num is %d"%g_num)

if name == ' main ':
    # 创建两个线程
    t1 = Thread(target=f1)
    t1.start()

    t2 = Thread(target=f2)
    t2.start()
```

线程共享变量 和这个无关。

附录

multiprocessing 模块

Process 类：进程

p=Process(target=目标函数,args=(参数,))

p.start()方法，进程就绪

p.is_alive()是否还存在

p.join()等待子进程结束，在执行主进程。当有 join 时候，daemon 无效。

p.terminate()进程停止

p.daemon=True/False 后台进程；随时被杀死；只要主进程结束，子进程结束。必须在 start 之前。

```
In [9]: p.          或者返回值或f    进程名字
      p.authkey   p.ident    p.name
      p.daemon    p.is_alive p.pid     p.sentinel
      p.exitcode  p.join?   加入?    p.start  开始
                           p.run    运行    p.terminate 停止
```

Pool 类：进程池

Queue 类：队列：进程通信

队列Queue (续)

- 一些常用方法
 - `task_done()`意味着之前入队的一个任务已经完成。由队列的消费者线程调用。每一个`get()`调用得到一个任务，接下来的`task_done()`调用告诉队列该任务已经处理完毕。如果当前一个`join()`正在阻塞，它将在队列中的所有任务都处理完时恢复执行（即每一个由`put()`调用入队的任务都有一个对应的`task_done()`调用）。
 - `join()`阻塞调用线程，直到队列中的所有任务被处理掉。只要有数据被加入队列，未完成的任务数就会增加。当消费者线程调用`task_done()`（意味着有消费者取得任务并完成任务），未完成的任务数就会减少。当未完成的任务数降到0，`join()`解除阻塞。

队列Queue (续2)

- 一些常用方法
 - `q.get([block [, timeout]])`
如果q为空，此方法将阻塞，直到队列中有项可用为止；
 - `q.get_nowait()` 同`q.get(False)` 将引发`Queue.Empty`异常；
 - `join()`阻塞调用线程，直到队列中的所有任务被处理掉。只要有数据被加入队列，未完成的任务数就会增加。当消费者线程调用`task_done()`（意味着有消费者取得任务并完成任务），未完成的任务数就会减少。当未完成的任务数降到0，`join()`解除阻塞；
 - `q.close()` 关闭队列，防止队列中加入更多的数据。调用此方法时，后台线程将继续写入那些已入队列但尚未写入的数据，但将在方法完成时马上关闭。如果q被垃圾回收，将自动调用此方法。关闭队列不会在队列消费者中生成任何类型的数据结束信号或异常；
 - `q.empty()` 判空，不一定可靠；
 - `q.qsize()` 不一定可靠。在某些系统中可能会引发`NotImplementedError` 异常；

队列Queue (续3)

- `JoinableQueue([maxsize])` 创建可连接的共享进程队列
类似一个`Queue`对象，但队列允项的消费者通知生产项已被成功处理。

通知进程使用共享的信号和条件变量来实现的。

`JoinableQueue`还有下面方法：

`q.task_done()` 消费者使用此方法发送信号，表示`q.get()`返回项已经被处理；如果调用此方法的次数大于从队列中删除的项数量，将会引发`ValueError`异常；

`q.join()`生产者使用此方法进行阻塞，直到队列中所有的项均被处理。阻塞将持续到队列中的每个项目均调用`q.task_done()`方法为止。

当一个队列为空的时候如果再用`get`取则会堵塞，所以取队列的时候一般是用到

`get_nowait()`方法，这种方法在向一个空队列取值的时候会抛一个 `Empty` 异常

所以更常用的方法是先判断一个队列是否为空，如果不为空则取值

队列中常用的方法

`Queue.qsize()` 返回队列的大小

`Queue.empty()` 如果队列为空，返回 `True`, 反之 `False`

`Queue.full()` 如果队列满了，返回 `True`, 反之 `False`

`Queue.get([block[, timeout]])` 获取队列，`timeout` 等待时间

`Queue.get_nowait()` 相当 `Queue.get(False)`

非阻塞 `Queue.put(item)` 写入队列，`timeout` 等待时间

`Queue.put_nowait(item)` 相当 `Queue.put(item, False)`

pipe 类：管道：进程通信

管道Pipe

ieau
达

- 管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。调用`pipe`系统函数即可创建一个管道。有如下特质：
 1. 其本质是一个伪文件(实为内核缓冲区)
 2. 由两个文件描述符引用，一个表示读端，一个表示写端
 3. 规定数据从管道的写端流入管道，从读端流出。
- 管道的原理: 管道实为内核使用环形队列机制，借助内核缓冲区(4k)实现。
- 管道的局限性：
 - A 数据自己读不能自己写；
 - B 数据一旦被读走，便不在管道中存在，不可反复读取；
 - C 由于管道采用半双工通信方式。因此，数据只能在一个方向上流动。
 - D 只能在有公共祖先的进程间使用管道。

常见的通信方式有，单工通信、半双工通信、全双工通信。

匿名管道

- 只能用于具有血缘关系的进程间通信，因此在这里我们可以调用fork函数，创建一个子进程，然后让父子进程通过管道进行通信；
- 同样可以在multiprocessing的Process中使用匿名管道来做父子进程之间的通信；

命名管道

- 首先，这个管道只能是具有血缘关系的进程之间通信；第二，它只能实现一个进程写另一个进程读，而如果需要两者同时进行时，就得重新打开一个管道。
- 为了使任意两个进程之间能够通信，就提出了命名管道（named pipe 或 FIFO）。
 - 1、与管道的区别：提供了一个路径名与之关联，以FIFO文件的形式存储于文件系统中，能够实现任何两个进程之间通信。而匿名管道对于文件系统是不可见的，它仅限于在父子进程之间的通信。
 - 2、FIFO是一个设备文件，在文件系统中以文件名的形式存在，因此即使进程与创建FIFO的进程不存在血缘关系也依然可以通信，前提是可以通过该路径。
 - 3、FIFO(first input first output)总是遵循先进先出的原则，即第一个进来的数据会第一个被读走。

multiprocessing.Pipe ([双工])

返回一对的代表的端部的对象管。 (conn1, conn2) Connection

如果双工是True（默认），那么管道是双向的。如果是双工，False那么管道是单向的：conn1只能用于接收消息，conn2只能用于发送消息。

class ([maxsize]) multiprocessing.Queue

返回使用管道和一些锁/信号量实现的进程共享队列。当一个进程首先把一个项目放到队列中时，一个进给线程被启动，将一个缓冲区中的对象传送到管道中。

标准库模块的通常queue.Empty和queue.Full例外都会queue引发超时。

Queue实现queue.Queue除了task_done()和之外的所有方法join()。

qsize ()

返回队列的近似大小。由于多线程/多处理语义，这个数字是不可靠的。

请注意，NotImplementedError在Mac OS X等Unix平台上可能会出现这种sem_getvalue()情况。

conn1.recv()Return an object sent from the other end of the connection using send().

Blocks until there is something to receive. Raises `EOFError` if there is nothing left to receive and the other end was closed.

Event 类: 事件; 进程通信

Lock 类: 锁, 进程同步

信号量Semaphore (续)

- 特点

- 1.信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。
- 2.信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。
- 3.每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

Value 类: 共享内容

Array 类: 共享内容

semaphore

acquire 加锁 get_value release 解锁

os

o=os

fork()

get_pid

get_ppid

time 模块

time 类

t=time()

t.ctime()当前时间, 精确到年-秒

t.sleep(s)睡眠 s 秒

queue 模块 help(queue)

priorityqueue 类

```
empty()  
full()  
get(self,block=true,timeout=None) 从队列中移除并返回一个项目  
get_nowait()  
join()  
put("进程", 优先级 int) 存入数列值  
put_nowait()  
qsize()
```

Queue 和 pipe 区别。

pipe 操作中，拿一个取一个，每一段只能执行一个职责，因为只有一个缓存

queue，没有限制随时可以取，多进程时，考虑封装序列化等问题。

multiprocessing 和 threading 模块很多相似。process 换成 thread 可以直接用

`multiprocessing` 是一个使用类似于 `threading` 模块的 API 来支持产卵过程的软件包。该软件包提供本地和远程并发，通过使用子进程而不是线程有效地侧移 全局解释器锁。由于这个原因，该模块允许程序员充分利用给定机器上的多个处理器。它可以在 Unix 和 Windows 上运行。`multiprocessing`

该模块还引入了模块中没有模拟量的 API。一个很好的例子就是 提供了一个方便的手段来并行执行跨多个输入值的函数，在输入数据之间分配输入数据（数据并行性）。以下示例演示了在模块中定义这些函数的常见做法，以便子进程可以成功导入该模块。这个数据并行性的基本例子，`multiprocessing` `threading` `Pool`

```
class ( group = None , target = None , name = None , args = ( ) , kwargs = {} ,  
* , daemon = None ) multiprocessing.Process
```

17.2.1.1. 本 Process 类

在，进程是通过创建一个 对象，然后调用它的方法产生的。遵循API的。一个多进程程序的简单例子是 `multiprocessing` `Process` `start()` `Process` `threading`. `Thread`

有方法的等价
只是为了兼容

在版本3.3中进行了更改：添加了守护进程参数。

run()

表示进程活动的方法。

您可以在子类中重写此方法。标准run()方法调用传递给对象构造函数的可调用对象作为目标参数，如果有的话，分别从args和kwargs参数中获取顺序和关键字参数。

start()

开始进程的活动。

每个进程对象最多只能调用一次。它安排对象的run()方法在一个单独的进程中被调用。 ↴

is_alive()

返回过程是否存在。

粗略地说，从start()方法返回的那一刻起，一个进程对象是活着的，直到子进程终止。

daemon

进程的守护进程标志，一个布尔值。这必须在start()调用之前设置。

初始值是从创建过程继承的。

当进程退出时，它将尝试终止其所有守护进程的子进程。

请注意，守护进程不允许创建子进程。否则，如果父进程退出时被终止，则守护进程将使其子进程成为孤儿。此外，这些不是 Unix守护进程或服务，它们是正常的进程，如果非守护进程已经退出，它将被终止（而不是加入）。

除了 threading.Thread API之外，Process对象还支持以下属性和方法：

`join([timeout])`

如果可选参数`timeout`是`None`（缺省值），则该方法将阻塞，直到其`join()`方法被调用的进程终止。如果超时是一个正数，它最多会阻塞超时秒数。请注意，`None`如果方法终止或方法超时，则方法返回。检查流程`exitcode`，确定是否终止。

一个过程可以连接多次。

进程无法自行加入，因为这会导致死锁。尝试在启动之前加入进程是错误的。

`name`

该进程的名称。名称是一个字符串，仅用于识别目的。它没有语义。多个进程可以被赋予相同的名称。

初始名称由构造函数设置。如果没有明确的名字被提供给构造函数，就会构造一个名为“Process- $N_1 : N_2 : \dots : N_k$ ”的表单，其中每个 N_k 是其父代的第 N 个孩子。

`pid`

返回进程ID。在这个过程产生之前，这将是`None`。

`multiprocessing.Pipe([双工])`

返回一对的代表的配管的端部的对象。`(conn1, conn2)` `Connection`

如果双工是`True`（默认），那么管道是双向的。如果是`双工`，`False`那么管道是单向的：`conn1`只能用于接收消息，`conn2`只能用于发送消息。

`class([maxsize])` `multiprocessing.Queue`

`queue`

`class([maxsize])` `multiprocessing.Queue`

返回使用管道和一些锁/信号量实现的进程共享队列。当一个进程首先把一个项目放到队列中时，一个进给线程被启动，将一个缓冲区中的对象传送到管道中。

标准库模块的通常`queue.Empty`和`queue.Full`例外都会`queue`引发超时。

`Queue`实现`queue.Queue`除了`task_done()`和之外的所有方法`join()`。

`qsize()`

返回队列的近似大小。由于多线程/ 多处理语义，这个数字是不可靠的。

请注意，`NotImplementedError` 在 Mac OS X 等 Unix 平台上可能会出现这种 `sem_getvalue()` 情况。

`empty()`

`True` 如果队列为空 `False` 则返回，否则返回。由于多线程/ 多处理语义，这是不可靠的。

`full()`

`True` 如果队列已满 `False` 则返回，否则返回。由于多线程/ 多处理语义，这是不可靠的。

`put(obj [, block [, timeout]])`

将 `obj` 放入队列中。如果可选参数 `块` 是 `True`（缺省值）并且 `超时` 是 `None`（缺省值），则在需要时禁止，直到空闲插槽可用。如果 `超时` 是一个正数，则最多会阻塞 `超时秒数`，`queue.Full` 如果在此时间内没有空闲插槽，则会引发异常。否则（`块` 是 `False`），如果一个空闲插槽立即可用，则在队列中放置一个项目，否则引发 `queue.Full` 异常（在这种情况下 `超时` 被忽略）。

`put_nowait(obj)`

相当于。`put(obj, False)`

`get([block [, timeout]])`

从队列中移除并返回一个项目。如果可选 `ARGS` 块是 `True`（默认值），`超时` 是 `None`（默认），块如有必要，直到一个项目是可用的。如果 `超时` 是一个正数，则最多会阻塞 `超时秒数`，`queue.Empty` 如果在该时间内没有可用项目，则会引发异常。否则（`块` 是 `False`），返回一个项目，如果一个是立即可用的，否则引发 `queue.Empty` 异常（在这种情况下 `超时` 被忽略）。

`get_nowait()`

相当于 `get(False)`。

`multiprocessing.Queue` 有一些其他的方法没有找到 `queue.Queue`。大多数代码通常不需要这些方法：

`close()`

表明当前进程不会有更多的数据放在这个队列中。一旦将所有缓冲的数据刷新到管道，后台线程将退出。当队列被垃圾收集时，这被自动调用。

`join_thread()`

加入后台线程。这只能在`close()`被调用后才能使用。它阻塞，直到后台线程退出，确保缓冲区中的所有数据已经刷新到管道。

默认情况下，如果一个进程不是队列的创建者，那么在退出时它将尝试加入队列的后台线程。该过程可以调用`cancel_join_thread()`做出`join_thread()`什么也不做。

线程相关

`cancel_join_thread()`

Pr事件`join_thread()`阻止。特别是，这个pr事件是后台线程在进程退出时自动加入 - 参见`join_thread()`。

这个方法的名字可能更好`allow_exit_without_flush()`。这可能会导致入队数据丢失，而且几乎可以肯定不需要使用它。如果您需要当前进程立即退出而不等待将排入队列的数据刷新到底层管道，并且您不关心丢失的数据，那么只有在那里。

```
class ([maxsize]) multiprocessing.JoinableQueue  
JoinableQueue, 一个Queue子类, 是另外有一个队列task_done()和join()方法。
```

`task_done()`

表明以前排队的任务已经完成。由队列使用者使用。对于每个`get()`用于获取任务的对象，随后的调用都会`task_done()`告诉队列，任务的处理已完成。

如果a`join()`当前被阻塞，则在处理所有项目时（意味着`task_done()`已经接收到已经`put()`进入队列的每个项目的呼叫），则将恢复。

提出了一个`ValueError`好象叫更多的时间比中放入队列中的项目。

自己研究 <http://vdisk.weibo.com/s/BMJiCHvNNxwvG>

`multiprocessing.current_process()`

返回`Process`当前进程对应的对象。

一个类似的`threading.current_thread()`。

类 multiprocessing.Event

克隆的。threading.Event

类 multiprocessing.Lock

一个非递归锁对象：一个非常类似的threading.Lock。一旦一个进程或线程获得一个锁，随后的尝试从任何进程或线程获取它将被阻塞，直到它被释放；任何进程或线程可能会释放它。threading.Lock应用于线程的概念和行为在这里被复制，因为它适用于进程或线程，除非另有说明。multiprocessing.Lock

请注意，这Lock实际上是一个工厂函数，它返回一个使用默认上下文初始化的实例。multiprocessing.synchronize.Lock

Lock支持上下文管理器协议，因此可以用在with语句中。

acquire(block = True , timeout = None)

获取一个锁，阻塞或不阻塞。

在block参数设置为True（默认）的情况下，方法调用将被阻塞直到锁处于解锁状态，然后将其设置为锁定并返回True。请注意，这第一个参数的名称不同于threading.Lock.acquire()。

将块参数设置为False，方法调用不会阻止。如果锁目前处于锁定状态，则返回False；否则将锁设置为锁定状态并返回True。

当用正数，浮点值超时调用时，只要无法获取锁定，最多阻塞超时指定的秒数。具有负值的超时调用相当于超时零。超时值None（默认值）的调用将超时期限设置为无限。请注意，处理负值或超时None值与实施的行为不同。该超时参数有，如果没有实际意义块参数被设置为，并因此忽略。返回threading.Lock.acquire() False True如果锁已被获取或False超时时间已过。

release()

释放一个锁。这可以从任何进程或线程调用，不仅是最初获取锁的进程或线程。

threading.Lock.release() 除了在解锁的锁上调用a时，行为与其相同 ValueError。

类 multiprocessing.RLock



递归锁对象：一个紧密的类比threading.RLock。递归锁必须由获取它的进程或线程释放。一旦一个进程或线程获得递归锁定，相同的进程或线程可以再次获取它，而不会阻塞；该进程或线程必须每次释放一次它被获取。

请注意，这RLock实际上是一个工厂函数，它返回一个使用默认上下文初始化的实例。multiprocessing.synchronize.RLock

RLock支持上下文管理器协议，因此可以用在with语句中。

17.2.2.6。共享ctypes对象

可以使用可以由子进程继承的共享内存来创建共享对象。

```
multiprocessing.Value ( typecode_or_type , * args , lock = True )
```

返回ctypes从共享内存分配的对象。默认情况下，返回值实际上是对象的同步包装器。对象本身可以通过a的value属性来访问Value。

typecode_or_type决定了返回对象的类型：它是一个ctypes类型或array模块使用的一种字符类型。*参数传递给类型的构造函数。

如果锁是True（默认），则会创建一个新的递归锁对象来同步对该值的访问。如果锁是一个Lock或一个RLock对象，那么将用于同步访问值。如果是锁，False那么访问返回的对象将不会被锁自动保护，因此它不一定是“过程安全的”。

操作就像+=它涉及的读取和写入不是原子。因此，例如，如果你想自动增加一个共享的值，这是不够的

```
multiprocessing.Array ( typecode_or_type , size_or_initializer , * , lock = True )
```

返回从共享内存分配的ctypes数组。默认情况下，返回值实际上是数组的同步包装器。

typecode_or_type确定返回数组元素的类型：它是模块使用的类型或单字符类型array。如果size_or_initializer是一个整数，那么它决定了数组的长度，数组将被初始化为零。否则，size_or_initializer是一个用来初始化数组的长度，它的长度决定了数组的长度。

如果锁是True（默认），则创建一个新的锁对象来同步对该值的访问。如果锁是一个Lock或一个RLock对象，那么将用于同步访问值。如果是锁，False那么访问返回的对象将不会被锁自动保护，因此它不一定是“过程安全的”。

请注意，锁定是仅有关键字的参数。

17.2.2.7。经理

管理者提供了一种方法来创建可以在不同进程之间共享的数据，包括在不同机器上运行的进程之间通过网络进行共享。管理员对象控制管理共享对象的服务器进程。其他进程可以通过使用代理来访问共享对象。

`multiprocessing.Manager()`

返回`SyncManager`可用于在进程之间共享对象的已启动对象。返回的管理对象对应于一个衍生的子进程，并具有创建共享对象并返回相应代理的方法。

`Manager`进程在垃圾收集或其父进程退出时将立即关闭。管理员类在模块中定义：

`multiprocessing.managers`

`class ([address [, authkey]]) multiprocessing.managers.BaseManager`

创建一个`BaseManager`对象。

一旦创建，应该调用`start()`或`get_server().serve_forever()`确保管理器对象引用已启动的管理器进程。