

公告

最新自学视频 路飞Python免费小课

人生迷茫问题可以加Alex个人微信听鸡



Alex金角大王



扫一扫上面的二维码图案，加我微信

网管到CEO的10年逆袭之路

Alex给你清晰的事业规划和执行策略

Alex

老男孩Python教学总监 | 路飞学城CEO



面向对象编程

Alex | 前汽车之家架构师



面向对象开发原来如此简单

16人在学

进阶

12小时

Python常用模块

Alex | 前汽车之家架构师



Python开发中最常用的11个模块精讲

10人在学

进阶

6小时

Python开发21天

Alex | 前汽车之家架构师



跟随Alex金角大王3周上手Python开发

124人在学

入门

19小时

昵称： 金角大王

园龄： 5年6个月

粉丝： 10911

关注： 5

-取消关注

随笔 - 29 文章 - 64 评论 - 930

Python之路,Day9 - 异步IO\数据库\队列\缓存

本节内容

1. Gevent协程
2. Select\Poll\Epoll异步IO与事件驱动
3. Python连接Mysql数据库操作
4. RabbitMQ队列
5. Redis\Memcached缓存
6. Paramiko SSH
7. Twisted网络框架

引子

到目前为止，我们已经学了网络并发编程的2个套路， 多进程，多线程，这哥俩的优势和劣势都非常的明显，我们一起来回顾下

协程

协程，又称微线程，纤程。英文名Coroutine。一句话说明什么是线程：**协程是一种用户态的轻量级线程。**

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：

协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

协程的好处：

- 无需线程上下文切换的开销

• 无需原子操作锁定及同步的开销

◦ "原子操作(atomic operation)是不需要synchronized"，所谓原子操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切换到另一个线程）。原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序是不可以被打乱，或者切割掉只执行部分。视作整体是原子性的核心。

• 方便切换控制流，简化编程模型

• 高并发+高扩展性+低成本：一个CPU支持上万的协程都不是问题。所以很适合用于高并发处理。
- 缺点：
- 无法利用多核资源：协程的本质是个单线程,它不能同时将 单个CPU 的多个核用上,协程需要和进程配合才能运行在多CPU上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是cpu密集型应用。
- https://www.cnblogs.com/alex3714/articles/5248247.html

1/35

我的标签

职业发展(3)

创业(2)

随笔分类

□职业&生活随笔(22)

文章分类

Python全栈开发之路(12)

Python学习目录(4)

Python自动化开发之路(33)

爬虫(6)

最新评论

1. Re:编程要自学或报班这事你都想不明白, 那必然是你智商不够

牛逼克拉斯

--晋北高峰

2. Re:Django 14天从小白到进阶- Day3
搞定Views组件
不更新了吗

--30岁老古董

3. Re:编程要自学或报班这事你都想不明白, 那必然是你智商不够
大王性格直爽, 有目标, 肯付出行动! 偶像! 我要是早几年遇见大王就好了, 现在都已经三十了, 浪费了大好的青春!

--Xiyue666

4. Re:python 之路, Day11 - python
mysql and ORM
ALTER mytable ADD INDEX
[indexName] ON (username(length))
这句应改为: alter table mytable add
index index...

--原竹

5. Re:Python之路,Day3 - Python基础3
@我的恋人叫臭臭 淫角大王听说很厉害
吧? ...

--Xiyue666

阅读排行榜

1. python 之路, 200行Python代码写了个打飞机游戏! (52795)
2. Django + Uwsgi + Nginx 实现生产环境部署(31846)
3. Python Select 解析(27208)
4. 为什么很多IT公司不喜欢进过培训机构的人呢? (20593)
5. 编程要自学或报班这事你都想不明白, 那必然是你智商不够(16952)

推荐排行榜

1. 给一位做技术迷茫的同学回信(63)
2. 你做了哪些事, 导致老板下调了对你的评价? (51)

- 进行阻塞 (Blocking) 操作 (如IO时) 会阻塞掉整个程序

使用yield实现协程操作例子

```

1 import time
2 import queue
3 def consumer(name):
4     print("--->starting eating baozi...")
5     while True:
6         new_baozi = yield
7         print("[%s] is eating baozi %s" % (name,new_baozi))
8         #time.sleep(1)
9
10 def producer():
11
12     r = con.__next__()
13     r = con2.__next__()
14     n = 0
15     while n < 5:
16         n +=1
17         con.send(n)
18         con2.send(n)
19         print("\033[32;1m[producer]\033[0m is making baozi %s" %n )
20
21
22 if __name__ == '__main__':
23     con = consumer("c1")
24     con2 = consumer("c2")
25     p = producer()

```

看楼上的例子, 我问你这算不算做是协程呢? 你说, 我他妈哪知道呀, 你前面说了一堆废话, 但是并没告诉我协程的标准形态呀, 我脑洞一想, 觉得你说也对, 那好, 我们先给协程一个标准定义, 即符合什么条件就能称之为协程:

1. 必须在只有一个单线程里实现并发
2. 修改共享数据不需加锁
3. 用户程序里自己保存多个控制流的上下文栈
4. 一个协程遇到IO操作自动切换到其它协程

基于上面这4点定义, 我们刚才用yield实现的程并不能算是合格的线程, 因为它有一点功能没实现, 哪一点呢?

Greenlet

greenlet是一个用C实现的协程模块, 相比与python自带的yield, 它可以使你在任意函数之间随意切换, 而不需把这个函数先声明为generator

```

1 # -*- coding:utf-8 -*-
2
3
4 from greenlet import greenlet
5
6
7 def test1():
8     print(12)
9     gr2.switch()
10    print(34)
11    gr2.switch()
12
13
14 def test2():
15     print(56)
16     gr1.switch()
17     print(78)
18

```

3. 关于认识、格局、多维度发展的感触(46)
4. 为什么很多IT公司不喜欢进过培训机构的人呢? (37)
5. 编程要自学或报班这事你都想不明白,那必然是你智商不够(36)

```

19
20 gr1 = greenlet(test1)
21 gr2 = greenlet(test2)
22 gr1.switch()

```

感觉确实用着比generator还简单了呢,但好像还没有解决一个问题,就是遇到IO操作,自动切换,对不对?

Gevent

Gevent 是一个第三方库,可以轻松通过gevent实现并发同步或异步编程,在gevent中用到的主要模式是**Greenlet**,它是以C扩展模块形式接入Python的轻量级协程。

Greenlet全部运行在主程序操作系统进程的内部,但它们被协作式地调度。

```

1 import gevent
2
3 def func1():
4     print('\033[31;1m李闯在跟海涛搞...\033[0m')
5     gevent.sleep(2)
6     print('\033[31;1m李闯又回去跟继续跟海涛搞...\033[0m')
7
8 def func2():
9     print('\033[32;1m李闯切换到了跟海龙搞...\033[0m')
10    gevent.sleep(1)
11    print('\033[32;1m李闯搞完了海涛,回来继续跟海龙搞...\033[0m')
12
13
14 gevent.joinall([
15     gevent.spawn(func1),
16     gevent.spawn(func2),
17     #gevent.spawn(func3),
18 ])

```

输出:

```

李闯在跟海涛搞...
李闯切换到了跟海龙搞...
李闯搞完了海涛,回来继续跟海龙搞...
李闯又回去跟继续跟海涛搞...

```

同步与异步的性能区别

```

1 import gevent
2
3 def task(pid):
4     """
5     Some non-deterministic task
6     """
7     gevent.sleep(0.5)
8     print('Task %s done' % pid)
9
10 def synchronous():
11     for i in range(1,10):
12         task(i)
13

```

```

14 def asynchronous():
15     threads = [gevent.spawn(task, i) for i in range(10)]
16     gevent.joinall(threads)
17
18     print('Synchronous:')
19     synchronous()
20
21     print('Asynchronous:')
22     asynchronous()

```

上面程序的重要部分是将task函数封装到Greenlet内部线程的gevent.spawn。初始化的greenlet列表存放在数组threads中，此数组被传给gevent.joinall 函数，后者阻塞当前流程，并执行所有给定的greenlet。执行流程只会在 所有greenlet执行完后才会继续向下走。

遇到IO阻塞时会自动切换任务

```

1 from gevent import monkey; monkey.patch_all()
2 import gevent
3 from urllib.request import urlopen
4
5 def f(url):
6     print('GET: %s' % url)
7     resp = urlopen(url)
8     data = resp.read()
9     print('%d bytes received from %s.' % (len(data), url))
10
11 gevent.joinall([
12     gevent.spawn(f, 'https://www.python.org/'),
13     gevent.spawn(f, 'https://www.yahoo.com/'),
14     gevent.spawn(f, 'https://github.com/'),
15 ])

```

通过gevent实现单线程下的多socket并发

server side

```

1 import sys
2 import socket
3 import time
4 import gevent
5
6 from gevent import import socket, monkey
7 monkey.patch_all()
8
9
10 def server(port):
11     s = socket.socket()
12     s.bind(('0.0.0.0', port))
13     s.listen(500)
14     while True:
15         cli, addr = s.accept()
16         gevent.spawn(handle_request, cli)
17
18
19
20 def handle_request(conn):
21     try:
22         while True:
23             data = conn.recv(1024)
24             print("recv:", data)
25             conn.send(data)
26             if not data:
27                 conn.shutdown(socket.SHUT_WR)
28

```

```

29     except Exception as ex:
30         print(ex)
31     finally:
32         conn.close()
33 if __name__ == '__main__':
34     server(8001)

```

client side

```

1  import socket
2
3  HOST = 'localhost'    # The remote host
4  PORT = 8001           # The same port as used by the server
5  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6  s.connect((HOST, PORT))
7  while True:
8      msg = bytes(input(">>:"), encoding="utf8")
9      s.sendall(msg)
10     data = s.recv(1024)
11     #print(data)
12
13     print('Received', repr(data))
14 s.close()

```



```

import socket
import threading

def sock_conn():

    client = socket.socket()

    client.connect(("localhost", 8001))
    count = 0
    while True:
        #msg = input(">>:").strip()
        #if len(msg) == 0:continue
        client.send( ("hello %s" %count).encode("utf-8"))

        data = client.recv(1024)

        print("[%s]recv from server:" % threading.get_id(),data.decode())
        count +=1
    client.close()

for i in range(100):
    t = threading.Thread(target=sock_conn)
    t.start()

```

论事件驱动与异步IO

通常，我们写服务器处理模型的程序时，有以下几种模型：

- (1) 每收到一个请求，创建一个新的进程，来处理该请求；
- (2) 每收到一个请求，创建一个新的线程，来处理该请求；
- (3) 每收到一个请求，放入一个事件列表，让主进程通过非阻塞I/O方式来处理请求

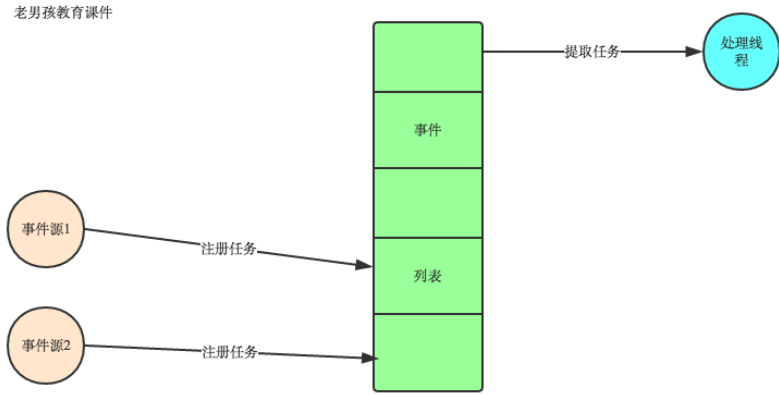
上面的几种方式，各有千秋，
第（1）中方法，由于创建新的进程的开销比较大，所以，会导致服务器性能比较差,但实现比较简单。
第（2）种方式，由于要涉及到线程的同步，有可能会面临死锁等问题。
第（3）种方式，在写应用程序代码时，逻辑比前面两种都复杂。
综合考虑各方面因素，一般普遍认为第（3）种方式是大多数网络服务器采用的方式

看图说话讲事件驱动模型

在UI编程中，常常要对鼠标点击进行相应，首先如何获得鼠标点击呢？
方式一：创建一个线程，该线程一直循环检测是否有鼠标点击，那么这种方式有以下几个缺点：
1. CPU资源浪费，可能鼠标点击的频率非常小，但是扫描线程还是会一直循环检测，这会造成很多的CPU资源浪费；如果扫描鼠标点击的接口是阻塞的呢？
2. 如果是堵塞的，又会出现下面这样的问题，如果我们不但要扫描鼠标点击，还要扫描键盘是否按下，由于扫描鼠标时被堵塞了，那么可能永远不会去扫描键盘；
3. 如果一个循环需要扫描的设备非常多，这又会引来响应时间的问题；
所以，该方式是非常不好的。

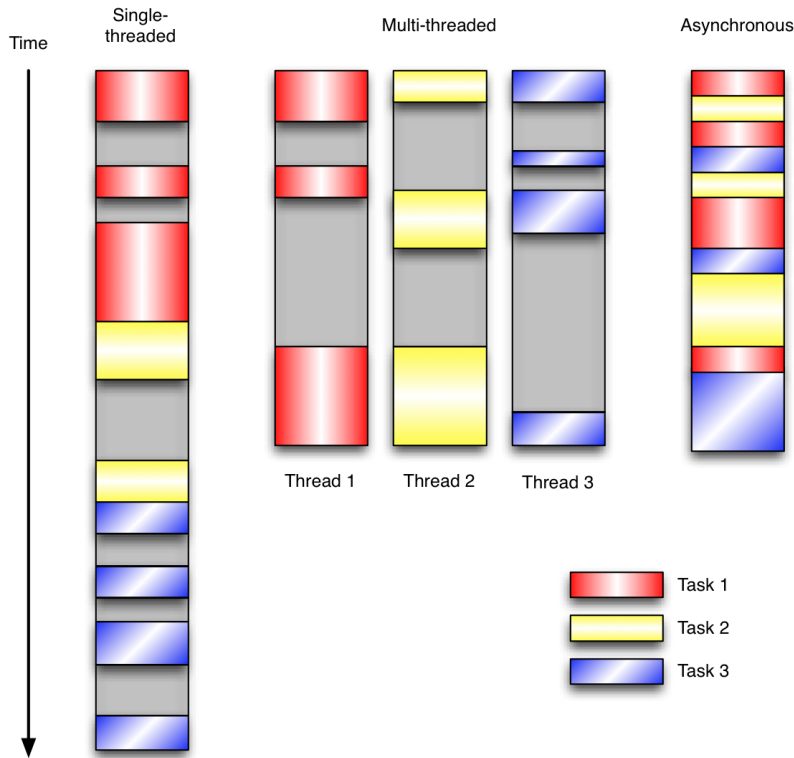
方式二：就是事件驱动模型

目前大部分的UI编程都是事件驱动模型，如很多UI平台都会提供onClick()事件，这个事件就代表鼠标按下事件。事件驱动模型大体思路如下：
1. 有一个事件（消息）队列；
2. 鼠标按下时，往这个队列中增加一个点击事件（消息）；
3. 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如onClick()、onKeyDown()等；
4. 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；



事件驱动编程是一种编程范式，这里程序的执行流由外部事件来决定。它的特点是包含一个事件循环，当外部事件发生时使用回调机制来触发相应的处理。另外两种常见的编程范式是（单线程）同步以及多线程编程。

让我们用例子来比较和对比一下单线程、多线程以及事件驱动编程模型。下图展示了随着时间的推移，这三种模式下程序所做的工作。这个程序有3个任务需要完成，每个任务都在等待I/O操作时阻塞自身。阻塞在I/O操作上所花费的时间已经用灰色框标示出来了。



在单线程同步模型中，任务按照顺序执行。如果某个任务因为I/O而阻塞，其他所有的任务都必须等待，直到它完成之后它们才能依次执行。这种明确的执行顺序和串行化处理的行为是很容易推断得出的。如果任务之间并没有互相依赖的关系，但仍然需要互相等待的话这就使得程序不必要的降低了运行速度。

在多线程版本中，这3个任务分别在独立的线程中执行。这些线程由操作系统来管理，在多处理器系统上可以并行处理，或者在单处理器系统上交错执行。这使得当某个线程阻塞在某个资源的同时其他线程得以继续执行。与完成类似功能的同步程序相比，这种方式更有效率，但程序员必须写代码来保护共享资源，防止其被多个线程同时访问。多线程程序更加难以推断，因为这类程序不得不通过线程同步机制如锁、可重入函数、线程局部存储或者其他机制来处理线程安全问题，如果实现不当就会导致出现微妙且令人痛不欲生的bug。

在事件驱动版本的程序中，3个任务交错执行，但仍然在一个单独的线程控制中。当处理I/O或者其他昂贵的操作时，注册一个回调到事件循环中，然后当I/O操作完成时继续执行。回调描述了该如何处理某个事件。事件循环轮询所有的事件，当事件到来时将它们分配给等待处理事件的回调函数。这种方式让程序尽可能的得以执行而不需要用到额外的线程。事件驱动型程序比多线程程序更容易推断出行为，因为程序员不需要关心线程安全问题。

当我们面对如下的环境时，事件驱动模型通常是一个好的选择：

- 1. 程序中有许多任务，而且...
- 2. 任务之间高度独立（因此它们不需要互相通信，或者等待彼此）而且...
- 3. 在等待事件到来时，某些任务会阻塞。

当应用程序需要在任务间共享可变的数据时，这也是一个不错的选择，因为这里不需要采用同步处理。

网络应用程序通常都有上述这些特点，这使得它们能够很好的契合事件驱动编程模型。

此处要提出一个问题，就是，上面的事件驱动模型中，只要一遇到IO就注册一个事件，然后主程序就可以继续干其它的事情了，只到io处理完毕后，继续恢复之前中断的任务，这本质上是怎么实现的呢？哈哈，下面我们就来一起揭开这神秘的面纱。。。。

Select\Poll\Epoll异步IO

<http://www.cnblogs.com/alex3714/p/4372426.html>

番外篇 <http://www.cnblogs.com/alex3714/articles/5876749.html>

select 多并发socket 例子

```
#!/usr/bin/env python
#_*_coding:utf-8_*_
__author__ = 'Alex Li'

import select
import socket
import sys
import queue

server = socket.socket()
server.setblocking(0)

server_addr = ('localhost', 10000)

print('starting up on %s port %s' % server_addr)
server.bind(server_addr)

server.listen(5)

inputs = [server, ] #自己也要监测呀,因为server本身也是个fd
outputs = []

message_queues = {}

while True:
    print("waiting for next event...")

    readable, writeable, exceptional = select.select(inputs, outputs, inputs)

    for s in readable: #每个s就是一个socket

        if s is server: #别忘记,上面我们server自己也当做一个fd放在了inputs列表里,传
            #就是有活动了, 什么情况下它才有活动? 当然 是有新连接进来的时候 呀
            #新连接进来了,接受这个连接
            conn, client_addr = s.accept()
            print("new connection from", client_addr)
            conn.setblocking(0)
            inputs.append(conn) #为了不阻塞整个程序,我们不会立刻在这里开始接收客户端
            #就会被交给select去监听,如果这个连接的客户端发来了数据,那这个连接的fd在se
            #readable 列表里,然后你就可以loop readable列表,取出这个连接,开始接收数据

            message_queues[conn] = queue.Queue() #接收到客户端的数据后,不立刻返回

        else: #s不是server的话,那就只能是一个 与客户端建立的连接的fd了
            #客户端的数据过来了,在这接收
            data = s.recv(1024)
            if data:
                print("收到来自[%s]的数据:" % s.getpeername()[0], data)
                message_queues[s].put(data) #收到的数据先放到queue里,一会返回给客
                if s not in outputs:
                    outputs.append(s) #为了不影响处理与其它客户端的连接,这里不立刻

            else: #如果收不到data代表什么呢? 代表客户端断开了呀
```



```

        print("客户端断开了",s)

        if s in outputs:
            outputs.remove(s) #清理已断开的连接

        inputs.remove(s) #清理已断开的连接

        del message_queues[s] ##清理已断开的连接

for s in writeable:
    try :
        next_msg = message_queues[s].get_nowait()

    except queue.Empty:
        print("client [%s]" %s.getpeername()[0], "queue is empty..")
        outputs.remove(s)

    else:
        print("sending msg to [%s]"%s.getpeername()[0], next_msg)
        s.send(next_msg.upper())

for s in exeptional:
    print("handling exception for ",s.getpeername())
    inputs.remove(s)
    if s in outputs:
        outputs.remove(s)
    s.close()

    del message_queues[s]

```



```

# -*- coding:utf-8 -*-
__author__ = 'Alex Li'

import socket
import sys

messages = [ b'This is the message. ',
             b'It will be sent ',
             b'in parts.',
             ]
server_address = ('localhost', 10000)

# Create a TCP/IP socket
socks = [ socket.socket(socket.AF_INET, socket.SOCK_STREAM),
          socket.socket(socket.AF_INET, socket.SOCK_STREAM),
          ]

# Connect the socket to the port where the server is listening
print('connecting to %s port %s' % server_address)
for s in socks:
    s.connect(server_address)

for message in messages:

    # Send messages on both sockets
    for s in socks:
        print('%s: sending "%s"' % (s.getsockname(), message) )
        s.send(message)

```

```
# Read responses on both sockets
for s in socks:
    data = s.recv(1024)
    print( '%s: received "%s"' % (s.getsockname(), data) )
    if not data:
        print(sys.stderr, 'closing socket', s.getsockname() )
```



selectors模块

This module allows high-level and efficient I/O multiplexing, built upon the `select` module primitives. Users are encouraged to use this module instead, unless they want precise control over the OS-level primitives used.

```
1 import selectors
2 import socket
3
4 sel = selectors.DefaultSelector()
5
6 def accept(sock, mask):
7     conn, addr = sock.accept() # Should be ready
8     print('accepted', conn, 'from', addr)
9     conn.setblocking(False)
10    sel.register(conn, selectors.EVENT_READ, read)
11
12 def read(conn, mask):
13     data = conn.recv(1000) # Should be ready
14     if data:
15         print('echoing', repr(data), 'to', conn)
16         conn.send(data) # Hope it won't block
17     else:
18         print('closing', conn)
19         sel.unregister(conn)
20         conn.close()
21
22 sock = socket.socket()
23 sock.bind(('localhost', 10000))
24 sock.listen(100)
25 sock.setblocking(False)
26 sel.register(sock, selectors.EVENT_READ, accept)
27
28 while True:
29     events = sel.select()
30     for key, mask in events:
31         callback = key.data
32         callback(key.fileobj, mask)
```

数据库操作与Paramiko模块

<http://www.cnblogs.com/wupeiqi/articles/5095821.html>

RabbitMQ队列

安装 <http://www.rabbitmq.com/install-standalone-mac.html>

安装python rabbitMQ module

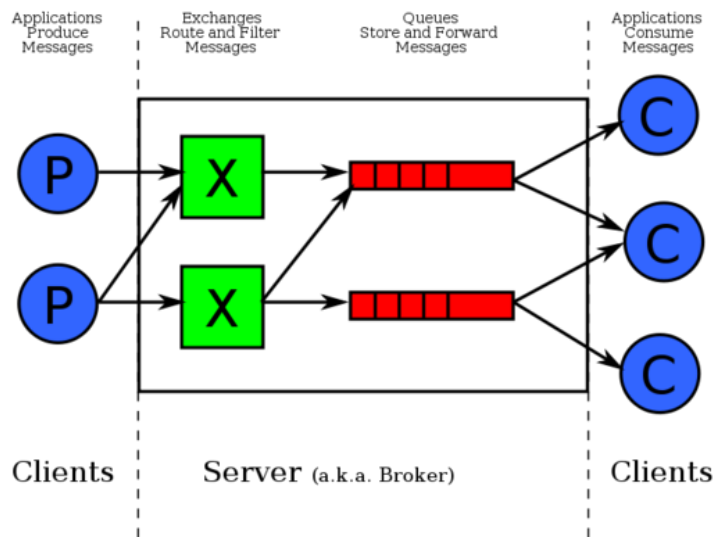
```
1 pip install pika
```

```

2 | or
3 | easy_install pika
4 | or
5 | 源码
6 |
7 | https://pypi.python.org/pypi/pika

```

实现最简单的队列通信



sunjun041640.blog.163.com

send端

```

1 | #!/usr/bin/env python
2 | import pika
3 |
4 | connection = pika.BlockingConnection(pika.ConnectionParameters(
5 |     'localhost'))
6 | channel = connection.channel()
7 |
8 | #声明queue
9 | channel.queue_declare(queue='hello')
10 |
11 | #n RabbitMQ a message can never be sent directly to the queue, it always ne
12 | channel.basic_publish(exchange='',
13 |                       routing_key='hello',
14 |                       body='Hello World!')
15 | print(" [x] Sent 'Hello World!'")
16 | connection.close()

```

receive端

```

1 | #_coding:utf-8_*_
2 | __author__ = 'Alex Li'
3 | import pika
4 |
5 | connection = pika.BlockingConnection(pika.ConnectionParameters(
6 |     'localhost'))
7 | channel = connection.channel()
8 |
9 |
10 | #You may ask why we declare the queue again - we have already declared it i
11 | # We could avoid that if we were sure that the queue already exists. For ex
12 | #was run before. But we're not yet sure which program to run first. In such
13 | # practice to repeat declaring the queue in both programs.
14 | channel.queue_declare(queue='hello')

```

```

15
16 def callback(ch, method, properties, body):
17     print(" [x] Received %r" % body)
18
19     channel.basic_consume(callback,
20                             queue='hello',
21                             no_ack=True)
22
23     print(' [*] Waiting for messages. To exit press CTRL+C')
24     channel.start_consuming()

```

远程连接rabbitmq server的话，需要配置权限 噢

首先在rabbitmq server上创建一个用户

```
1 | sudo rabbitmqctl add_user alex alex3714
```

同时还要配置权限，允许从外面访问

```
1 | sudo rabbitmqctl set_permissions -p / alex ".*" ".*" ".*"
```

set_permissions [-p vhost] {user} {conf} {write} {read}

vhost

The name of the virtual host to which to grant the user access, defaulting to /.

user

The name of the user to grant access to the specified virtual host.

conf

A regular expression matching resource names for which the user is granted configure permissions.

write

A regular expression matching resource names for which the user is granted write permissions.

read

A regular expression matching resource names for which the user is granted read permissions.

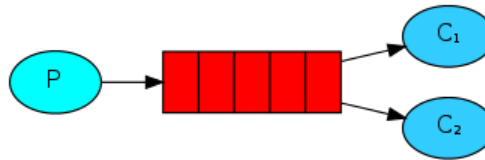
客户端连接的时候需要配置认证参数

```

1 | credentials = pika.PlainCredentials('alex', 'alex3714')
2
3
4 | connection = pika.BlockingConnection(pika.ConnectionParameters(
5 |     '10.211.55.5', 5672, '/', credentials))
6 | channel = connection.channel()

```

Work Queues



在这种模式下, RabbitMQ会默认把p发的消息依次分发给各个消费者(c),跟负载均衡差不多

消息提供者代码

```

1  import pika
2  import time
3  connection = pika.BlockingConnection(pika.ConnectionParameters(
4      'localhost'))
5  channel = connection.channel()
6
7  # 声明queue
8  channel.queue_declare(queue='task_queue')
9
10 # n RabbitMQ a message can never be sent directly to the queue, it always n
11 import sys
12
13 message = ' '.join(sys.argv[1:]) or "Hello World! %s" % time.time()
14 channel.basic_publish(exchange='',
15                       routing_key='task_queue',
16                       body=message,
17                       properties=pika.BasicProperties(
18                           delivery_mode=2, # make message persistent
19                       )
20                       )
21 print(" [x] Sent %r" % message)
22 connection.close()
  
```

消费者代码

```

1  #_*_coding:utf-8_*_
2
3  import pika, time
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6      'localhost'))
7  channel = connection.channel()
8
9
10 def callback(ch, method, properties, body):
11     print(" [x] Received %r" % body)
12     time.sleep(20)
13     print(" [x] Done")
14     print("method.delivery_tag",method.delivery_tag)
15     ch.basic_ack(delivery_tag=method.delivery_tag)
16
17
18 channel.basic_consume(callback,
19                       queue='task_queue',
20                       no_ack=True
21                       )
22
23 print(' [*] Waiting for messages. To exit press CTRL+C')
24 channel.start_consuming()
  
```

此时，先启动消息生产者，然后再分别启动3个消费者，通过生产者多发送几条消息，你会发现，这几条消息会被依次分配到各个消费者身上

Doing a task can take a few seconds. You may wonder what happens if one of the consumers starts a long task and dies with it only partly done. With our current code once RabbitMQ delivers message to the customer it immediately removes it from memory. In this case, if you kill a worker we will lose the message it was just processing. We'll also lose all the messages that were dispatched to this particular worker but were not yet handled.

But we don't want to lose any tasks. If a worker dies, we'd like the task to be delivered to another worker.

In order to make sure a message is never lost, RabbitMQ supports message *acknowledgments*. An *ack*(nowledgement) is sent back from the consumer to tell RabbitMQ that a particular message had been received, processed and that RabbitMQ is free to delete it.

If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an *ack*, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

There aren't any message timeouts; RabbitMQ will redeliver the message when the consumer dies. It's fine even if processing a message takes a very, very long time.

Message acknowledgments are turned on by default. In previous examples we explicitly turned them off via the `no_ack=True` flag. It's time to remove this flag and send a proper acknowledgment from the worker, once we're done with a task.

```
1 def callback(ch, method, properties, body):
2     print " [x] Received %r" % (body,)
3     time.sleep( body.count('.') )
4     print " [x] Done"
5     ch.basic_ack(delivery_tag = method.delivery_tag)
6
7 channel.basic_consume(callback,
8                        queue='hello')
```

Using this code we can be sure that even if you kill a worker using CTRL+C while it was processing a message, nothing will be lost. Soon after the worker dies all unacknowledged messages will be redelivered

消息持久化

We have learned how to make sure that even if the consumer dies, the task isn't lost (by default, if wanna disable use `no_ack=True`). But our tasks will still be lost if RabbitMQ server stops.

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

First, we need to make sure that RabbitMQ will never lose our queue. In order to do so, we need to declare it as *durable*:

```
1 channel.queue_declare(queue='hello', durable=True)
```

Although this command is correct by itself, it won't work in our setup. That's because we've already defined a queue called hello which is not durable. RabbitMQ doesn't allow you to redefine an existing queue with different parameters and will return an error to any program that tries to do that. But there is a quick workaround - let's declare a queue with different name, for example task_queue:

```
1 | channel.queue_declare(queue='task_queue', durable=True)
```

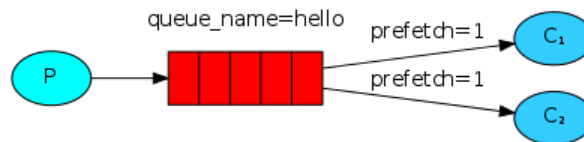
This queue_declare change needs to be applied to both the producer and consumer code.

At that point we're sure that the task_queue queue won't be lost even if RabbitMQ restarts. Now we need to mark our messages as persistent - by supplying a delivery_mode property with a value 2.

```
1 | channel.basic_publish(exchange='',
2 |                       routing_key="task_queue",
3 |                       body=message,
4 |                       properties=pika.BasicProperties(
5 |                           delivery_mode = 2, # make message persistent
6 |                       ))
```

消息公平分发

如果Rabbit只管按顺序把消息发到各个消费者身上，不考虑消费者负载的话，很可能出现，一个机器配置不高的消费者那里堆积了很多消息处理不完，同时配置高的消费者却一直很轻松。为解决此问题，可以在各个消费者端，配置prefetch=1，意思就是告诉RabbitMQ在我这个消费者当前消息还没处理完的时候就不要再给我发新消息了。



```
1 | channel.basic_qos(prefetch_count=1)
```

带消息持久化+公平分发的完整代码

生产者端

```
1 | #!/usr/bin/env python
2 | import pika
3 | import sys
4 |
5 | connection = pika.BlockingConnection(pika.ConnectionParameters(
6 |     host='localhost'))
7 | channel = connection.channel()
8 |
9 | channel.queue_declare(queue='task_queue', durable=True)
10 |
11 | message = ' '.join(sys.argv[1:]) or "Hello World!"
12 | channel.basic_publish(exchange='',
13 |                      routing_key='task_queue',
14 |                      body=message,
15 |                      properties=pika.BasicProperties(
16 |                          delivery_mode = 2, # make message persistent
17 |                      ))
18 | print(" [x] Sent %r" % message)
```

```
19 | connection.close()
```

消费者端

```
1 | #!/usr/bin/env python
2 | import pika
3 | import time
4 |
5 | connection = pika.BlockingConnection(pika.ConnectionParameters(
6 |     host='localhost'))
7 | channel = connection.channel()
8 |
9 | channel.queue_declare(queue='task_queue', durable=True)
10 | print(' [*] Waiting for messages. To exit press CTRL+C')
11 |
12 | def callback(ch, method, properties, body):
13 |     print(" [x] Received %r" % body)
14 |     time.sleep(body.count(b'.'))
15 |     print(" [x] Done")
16 |     ch.basic_ack(delivery_tag = method.delivery_tag)
17 |
18 | channel.basic_qos(prefetch_count=1)
19 | channel.basic_consume(callback,
20 |                       queue='task_queue')
21 |
22 | channel.start_consuming()
```

Publish\Subscribe(消息发布\订阅)

之前的例子都基本都是1对1的消息发送和接收，即消息只能发送到指定的queue里，但有些时候你想让你的消息被所有的Queue收到，类似广播的效果，这时候就要用到exchange了，

An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the *exchange type*.

Exchange在定义的时候是有类型的，以决定到底是哪些Queue符合条件，可以接收消息

fanout: 所有bind到此exchange的queue都可以接收消息

direct: 通过routingKey和exchange决定的那个唯一的queue可以接收消息

topic: 所有符合routingKey(此时可以是一个表达式)的routingKey所bind的queue可以接收消息

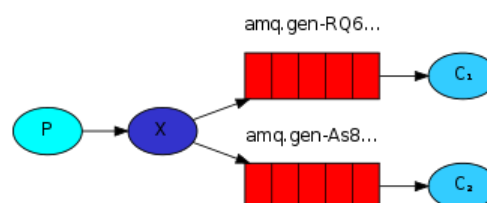
表达式符号说明: #代表一个或多个字符, *代表任何字符

例: #.a会匹配a.a, aa.a, aaa.a等

*.a会匹配a.a, b.a, c.a等

注: 使用RoutingKey为#, Exchange Type为topic的时候相当于使用fanout

headers: 通过headers 来决定把消息发给哪些queue



消息publisher


```

1 import pika
2 import sys
3
4 connection = pika.BlockingConnection(pika.ConnectionParameters(
5     host='localhost'))
6 channel = connection.channel()
7
8 channel.exchange_declare(exchange='logs',
9     type='fanout')
10
11 message = ' '.join(sys.argv[1:]) or "info: Hello World!"
12 channel.basic_publish(exchange='logs',
13     routing_key='',
14     body=message)
15 print(" [x] Sent %r" % message)
16 connection.close()

```

消息subscriber

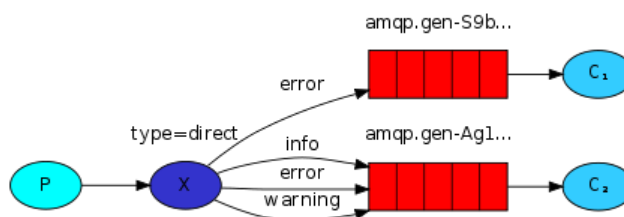
```

1 # -*- coding:utf-8 -*-
2 __author__ = 'Alex Li'
3 import pika
4
5 connection = pika.BlockingConnection(pika.ConnectionParameters(
6     host='localhost'))
7 channel = connection.channel()
8
9 channel.exchange_declare(exchange='logs',
10     type='fanout')
11
12 result = channel.queue_declare(exclusive=True) #不指定queue名字,rabbit会随机:
13 queue_name = result.method.queue
14
15 channel.queue_bind(exchange='logs',
16     queue=queue_name)
17
18 print(' [*] Waiting for logs. To exit press CTRL+C')
19
20 def callback(ch, method, properties, body):
21     print(" [x] %r" % body)
22
23 channel.basic_consume(callback,
24     queue=queue_name,
25     no_ack=True)
26
27 channel.start_consuming()

```

有选择的接收消息(exchange type=direct)

RabbitMQ还支持根据关键字发送，即：队列绑定关键字，发送者将数据根据关键字发送到消息exchange，exchange根据 关键字 判定应该将数据发送至指定队列。



publisher

```

1 import pika

```

```

2 import sys
3
4 connection = pika.BlockingConnection(pika.ConnectionParameters(
5     host='localhost'))
6 channel = connection.channel()
7
8 channel.exchange_declare(exchange='direct_logs',
9     type='direct')
10
11 severity = sys.argv[1] if len(sys.argv) > 1 else 'info'
12 message = ' '.join(sys.argv[2:]) or 'Hello World!'
13 channel.basic_publish(exchange='direct_logs',
14     routing_key=severity,
15     body=message)
16 print(" [x] Sent %r:%r" % (severity, message))
17 connection.close()

```

subscriber

```

1 import pika
2 import sys
3
4 connection = pika.BlockingConnection(pika.ConnectionParameters(
5     host='localhost'))
6 channel = connection.channel()
7
8 channel.exchange_declare(exchange='direct_logs',
9     type='direct')
10
11 result = channel.queue_declare(exclusive=True)
12 queue_name = result.method.queue
13
14 severities = sys.argv[1:]
15 if not severities:
16     sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
17     sys.exit(1)
18
19 for severity in severities:
20     channel.queue_bind(exchange='direct_logs',
21         queue=queue_name,
22         routing_key=severity)
23
24 print(' [*] Waiting for logs. To exit press CTRL+C')
25
26 def callback(ch, method, properties, body):
27     print(" [x] %r:%r" % (method.routing_key, body))
28
29 channel.basic_consume(callback,
30     queue=queue_name,
31     no_ack=True)
32
33 channel.start_consuming()

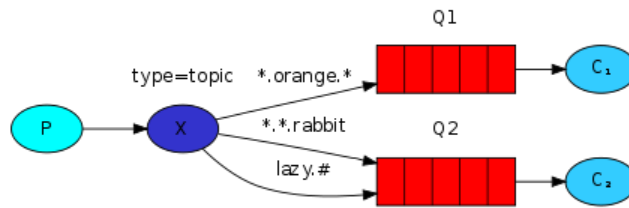
```

更细致的消息过滤

Although using the direct exchange improved our system, it still has limitations - it can't do routing based on multiple criteria.

In our logging system we might want to subscribe to not only logs based on severity, but also based on the source which emitted the log. You might know this concept from the syslog unix tool, which routes logs based on both severity (info/warn/crit...) and facility (auth/cron/kern...).

That would give us a lot of flexibility - we may want to listen to just critical errors coming from 'cron' but also all logs from 'kern'.



publisher

```

1 import pika
2 import sys
3
4 connection = pika.BlockingConnection(pika.ConnectionParameters(
5     host='localhost'))
6 channel = connection.channel()
7
8 channel.exchange_declare(exchange='topic_logs',
9     type='topic')
10
11 routing_key = sys.argv[1] if len(sys.argv) > 1 else 'anonymous.info'
12 message = ' '.join(sys.argv[2:]) or 'Hello World!'
13 channel.basic_publish(exchange='topic_logs',
14     routing_key=routing_key,
15     body=message)
16 print(" [x] Sent %r:%r" % (routing_key, message))
17 connection.close()
  
```

subscriber

```

1 import pika
2 import sys
3
4 connection = pika.BlockingConnection(pika.ConnectionParameters(
5     host='localhost'))
6 channel = connection.channel()
7
8 channel.exchange_declare(exchange='topic_logs',
9     type='topic')
10
11 result = channel.queue_declare(exclusive=True)
12 queue_name = result.method.queue
13
14 binding_keys = sys.argv[1:]
15 if not binding_keys:
16     sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
17     sys.exit(1)
18
19 for binding_key in binding_keys:
20     channel.queue_bind(exchange='topic_logs',
21         queue=queue_name,
22         routing_key=binding_key)
23
24 print(' [*] Waiting for logs. To exit press CTRL+C')
25
26 def callback(ch, method, properties, body):
27     print(" [x] %r:%r" % (method.routing_key, body))
28
29 channel.basic_consume(callback,
30     queue=queue_name,
31     no_ack=True)
32
33 channel.start_consuming()
  
```

To receive all the logs run:

```
python receive_logs_topic.py "#"
```

To receive all logs from the facility "kern":

```
python receive_logs_topic.py "kern.*"
```

Or if you want to hear only about "critical" logs:

```
python receive_logs_topic.py "/*.critical"
```

You can create multiple bindings:

```
python receive_logs_topic.py "kern.*" "/*.critical"
```

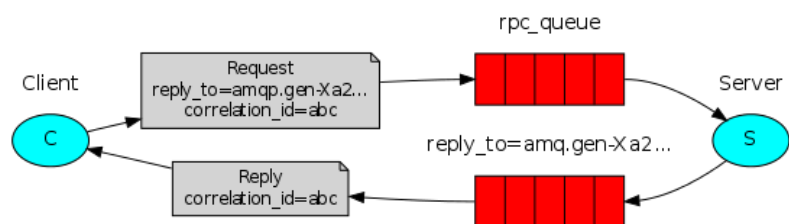
And to emit a log with a routing key "kern.critical" type:

```
python emit_log_topic.py "kern.critical" "A critical kernel error"
```

Remote procedure call (RPC)

To illustrate how an RPC service could be used we're going to create a simple client class. It's going to expose a method named call which sends an RPC request and blocks until the answer is received:

```
1 fibonacci_rpc = FibonacciRpcClient()
2 result = fibonacci_rpc.call(4)
3 print("fib(4) is %r" % result)
```



RPC server

```
1 #_*_coding:utf-8*_
2 __author__ = 'Alex Li'
3 import pika
4 import time
5 connection = pika.BlockingConnection(pika.ConnectionParameters(
6     host='localhost'))
7
8 channel = connection.channel()
9
10 channel.queue_declare(queue='rpc_queue')
11
12 def fib(n):
13     if n == 0:
14         return 0
15     elif n == 1:
16         return 1
17     else:
18         return fib(n-1) + fib(n-2)
19
20 def on_request(ch, method, props, body):
21     n = int(body)
22
23     print(" [.] fib(%s)" % n)
24     response = fib(n)
25
26     ch.basic_publish(exchange='',
```

```

27         routing_key=props.reply_to,
28         properties=pika.BasicProperties(correlation_id = \
29                                         props.correlation_
30                                         body=str(response))
31     ch.basic_ack(delivery_tag = method.delivery_tag)
32
33     channel.basic_qos(prefetch_count=1)
34     channel.basic_consume(on_request, queue='rpc_queue')
35
36     print(" [x] Awaiting RPC requests")
37     channel.start_consuming()

```

RPC client

```

1  import pika
2  import uuid
3
4  class FibonacciRpcClient(object):
5      def __init__(self):
6          self.connection = pika.BlockingConnection(pika.ConnectionParameters
7              host='localhost'))
8
9          self.channel = self.connection.channel()
10
11         result = self.channel.queue_declare(exclusive=True)
12         self.callback_queue = result.method.queue
13
14         self.channel.basic_consume(self.on_response, no_ack=True,
15                                     queue=self.callback_queue)
16
17     def on_response(self, ch, method, props, body):
18         if self.corr_id == props.correlation_id:
19             self.response = body
20
21     def call(self, n):
22         self.response = None
23         self.corr_id = str(uuid.uuid4())
24         self.channel.basic_publish(exchange='',
25                                   routing_key='rpc_queue',
26                                   properties=pika.BasicProperties(
27                                       reply_to = self.callback_queue,
28                                       correlation_id = self.corr_id,
29                                       ),
29                                   body=str(n))
30
31         while self.response is None:
32             self.connection.process_data_events()
33         return int(self.response)
34
35     fibonacci_rpc = FibonacciRpcClient()
36
37     print(" [x] Requesting fib(30)")
38     response = fibonacci_rpc.call(30)
39     print(" [.] Got %r" % response)

```

Memcached & Redis使用

memcached

<http://www.cnblogs.com/wupeiqi/articles/5132791.html>

redis 使用

<http://www.cnblogs.com/alex3714/articles/6217453.html>

Twisted异步网络框架

Twisted是一个事件驱动的网络框架，其中包含了诸多功能，例如：网络协议、线程、数据库管理、网络操作、电子邮件等。

Package	application	Configuration objects for Twisted Applications.
Package	conch	Twisted Conch: The Twisted Shell. Terminal emulation, SSHv2 and telnet.
Module	copyright	Copyright information for Twisted.
Package	cred	Twisted Cred: Support for verifying credentials, and providing services to user based on those credentials.
Package	enterprise	Twisted Enterprise: Database support for Twisted services.
Package	internet	Twisted Internet: Asynchronous I/O and Events.
Package	logger	Twisted Logger: Classes and functions to do granular logging.
Package	mail	Twisted Mail: Servers and clients for POP3, ESMTP, and IMAP.
Package	manhole	Twisted Manhole: interactive interpreter and direct manipulation support for Twisted.
Package	names	Twisted Names: DNS server and client implementations.
Package	news	Twisted News: A NNTP-based news service.
Package	pair	Twisted Pair: The framework of your ethernet.
Package	persisted	Twisted Persisted: Utilities for managing persistence.
Module	plugin	Plugin system for Twisted.
Package	plugins	Plugins for services implemented in Twisted.
Package	positioning	Twisted Positioning: Framework for applications that make use of positioning.
Package	protocols	Twisted Protocols: A collection of internet protocol implementations.
Package	python	Twisted Python: Utilities and Enhancements for Python.
Package	runner	Twisted Runner: Run and monitor processes.
Package	scripts	Subpackage containing the modules that implement the command line tools.
Package	spread	Twisted Spread: Spreadable (Distributed) Computing.
Package	tap	Twisted TAP: Twisted Application Persistence builders for other Twisted servers.
Package	test	Twisted's unit tests.
Package	trial	Twisted Trial: Asynchronous unit testing framework.
Package	web	Twisted Web: HTTP clients and servers, plus tools for implementing them.
Package	words	Twisted Words: Client and server implementations for IRC, XMPP, and other chat services.

事件驱动

简而言之，事件驱动分为二个部分：第一，注册事件；第二，触发事件。

自定义事件驱动框架，命名为：“弑君者”：

```
1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3
4  # event_drive.py
5
6  event_list = []
7
8
9  def run():
10     for event in event_list:
11         obj = event()
12         obj.execute()
13
14
15  class BaseHandler(object):
16     """
17     用户必须继承该类，从而规范所有类的方法（类似于接口的功能）
18     """
19     def execute(self):
20         raise Exception('you must overwrite execute')
21
22  最牛逼的事件驱动框架
```

程序员使用“弑君者框架”：

```
1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3
4  from source import event_drive
5
6
7  class MyHandler(event_drive.BaseHandler):
8
```

```

9         def execute(self):
10             print 'event-drive execute MyHandler'
11
12
13     event_drive.event_list.append(MyHandler)
14     event_drive.run()

```

Protocols

Protocols描述了如何以异步的方式处理网络中的事件。HTTP、DNS以及IMAP是应用层协议中的例子。Protocols实现了IProtocol接口，它包含如下的方法：

makeConnection	在transport对象和服务端之间建立一条连接
connectionMade	连接建立起来后调用
dataReceived	接收数据时调用
connectionLost	关闭连接时调用

Transports

Transports代表网络中两个通信结点之间的连接。Transports负责描述连接的细节，比如连接是面向流式的还是面向数据报的，流控以及可靠性。TCP、UDP和Unix套接字可作为transports的例子。它们被设计为“满足最小功能单元，同时具有最大程度的可复用性”，而且从协议实现中分离出来，这让许多协议可以采用相同类型的传输。Transports实现了ITransports接口，它包含如下的方法：

write	以非阻塞的方式按顺序依次将数据写到物理连接上
writeSequence	将一个字符串列表写到物理连接上
loseConnection	将所有挂起的数据写入，然后关闭连接
getPeer	取得连接中对端的地址信息
getHost	取得连接中本端的地址信息

将transports从协议中分离出来也使得对这两个层次的测试变得更加简单。可以通过简单地写入一个字符串来模拟传输，用这种方式来检查。

EchoServer

```

1  from twisted.internet import protocol
2  from twisted.internet import reactor
3
4  class Echo(protocol.Protocol):
5      def dataReceived(self, data):
6          self.transport.write(data)
7
8  def main():
9      factory = protocol.ServerFactory()
10     factory.protocol = Echo
11
12     reactor.listenTCP(1234, factory)
13     reactor.run()
14
15 if __name__ == '__main__':
16     main()

```

EchoClient

```

1  from twisted.internet import reactor, protocol
2
3
4  # a client protocol
5
6  class EchoClient(protocol.Protocol):
7      """Once connected, send a message, then print the result."""

```

```

8
9     def connectionMade(self):
10         self.transport.write("hello alex!")
11
12     def dataReceived(self, data):
13         "As soon as any data is received, write it back."
14         print "Server said:", data
15         self.transport.loseConnection()
16
17     def connectionLost(self, reason):
18         print "connection lost"
19
20 class EchoFactory(protocol.ClientFactory):
21     protocol = EchoClient
22
23     def clientConnectionFailed(self, connector, reason):
24         print "Connection failed - goodbye!"
25         reactor.stop()
26
27     def clientConnectionLost(self, connector, reason):
28         print "Connection lost - goodbye!"
29         reactor.stop()
30
31
32 # this connects the protocol to a server running on port 8000
33 def main():
34     f = EchoFactory()
35     reactor.connectTCP("localhost", 1234, f)
36     reactor.run()
37
38 # this only runs if the module was *not* imported
39 if __name__ == '__main__':
40     main()

```

运行服务器端脚本将启动一个TCP服务器，监听端口1234上的连接。服务器采用的是Echo协议，数据经TCP transport对象写出。运行客户端脚本将对服务器发起一个TCP连接，回显服务器端的回应然后终止连接并停止reactor事件循环。这里的Factory用来对连接的双方生成protocol对象实例。两端的通信是异步的，connectTCP负责注册回调函数到reactor事件循环中，当socket上有数据可读时通知回调处理。

一个传送文件的例子

server side

```

1  # -*- coding:utf-8 -*-
2  # This is the Twisted Fast Poetry Server, version 1.0
3
4  import optparse, os
5
6  from twisted.internet.protocol import ServerFactory, Protocol
7
8
9  def parse_args():
10     usage = """usage: %prog [options] poetry-file
11
12     This is the Fast Poetry Server, Twisted edition.
13     Run it like this:
14
15     python fastpoetry.py <path-to-poetry-file>
16
17     If you are in the base directory of the twisted-intro package,
18     you could run it like this:
19
20     python twisted-server-1/fastpoetry.py poetry/ecstasy.txt
21
22     to serve up John Donne's Ecstasy, which I know you want to do.

```



```

23  """
24
25  parser = optparse.OptionParser(usage)
26
27  help = "The port to listen on. Default to a random available port."
28  parser.add_option('--port', type='int', help=help)
29
30  help = "The interface to listen on. Default is localhost."
31  parser.add_option('--iface', help=help, default='localhost')
32
33  options, args = parser.parse_args()
34  print("--arg:", options, args)
35
36  if len(args) != 1:
37      parser.error('Provide exactly one poetry file.')
38
39  poetry_file = args[0]
40
41  if not os.path.exists(args[0]):
42      parser.error('No such file: %s' % poetry_file)
43
44  return options, poetry_file
45
46
47  class PoetryProtocol(Protocol):
48
49      def connectionMade(self):
50          self.transport.write(self.factory.poem)
51          self.transportloseConnection()
52
53
54  class PoetryFactory(ServerFactory):
55
56      protocol = PoetryProtocol
57
58      def __init__(self, poem):
59          self.poem = poem
60
61
62  def main():
63      options, poetry_file = parse_args()
64
65      poem = open(poetry_file).read()
66
67      factory = PoetryFactory(poem)
68
69      from twisted.internet import reactor
70
71      port = reactor.listenTCP(options.port or 9000, factory,
72                              interface=options.iface)
73
74      print 'Serving %s on %s.' % (poetry_file, port.getHost())
75
76      reactor.run()
77
78
79  if __name__ == '__main__':
80      main()

```

client side

```

1  # This is the Twisted Get Poetry Now! client, version 3.0.
2
3  # NOTE: This should not be used as the basis for production code.
4
5  import optparse

```

```
6
7 from twisted.internet.protocol import Protocol, ClientFactory
8
9
10 def parse_args():
11     usage = """usage: %prog [options] [hostname]:port ...
12
13 This is the Get Poetry Now! client, Twisted version 3.0
14 Run it like this:
15
16 python get-poetry-1.py port1 port2 port3 ...
17 """
18
19     parser = optparse.OptionParser(usage)
20
21     _, addresses = parser.parse_args()
22
23     if not addresses:
24         print parser.format_help()
25         parser.exit()
26
27     def parse_address(addr):
28         if ':' not in addr:
29             host = '127.0.0.1'
30             port = addr
31         else:
32             host, port = addr.split(':', 1)
33
34         if not port.isdigit():
35             parser.error('Ports must be integers.')
36
37         return host, int(port)
38
39     return map(parse_address, addresses)
40
41
42 class PoetryProtocol(Protocol):
43
44     poem = ''
45
46     def dataReceived(self, data):
47         self.poem += data
48
49     def connectionLost(self, reason):
50         self.poemReceived(self.poem)
51
52     def poemReceived(self, poem):
53         self.factory.poem_finished(poem)
54
55
56 class PoetryClientFactory(ClientFactory):
57
58     protocol = PoetryProtocol
59
60     def __init__(self, callback):
61         self.callback = callback
62
63     def poem_finished(self, poem):
64         self.callback(poem)
65
66
67 def get_poetry(host, port, callback):
68     """
69     Download a poem from the given host and port and invoke
70
71     callback(poem)
```

```
72
73     when the poem is complete.
74     """
75     from twisted.internet import reactor
76     factory = PoetryClientFactory(callback)
77     reactor.connectTCP(host, port, factory)
78
79
80 def poetry_main():
81     addresses = parse_args()
82
83     from twisted.internet import reactor
84
85     poems = []
86
87     def got_poem(poem):
88         poems.append(poem)
89         if len(poems) == len(addresses):
90             reactor.stop()
91
92     for address in addresses:
93         host, port = address
94         get_poetry(host, port, got_poem)
95
96     reactor.run()
97
98     for poem in poems:
99         print poem
100
101
102 if __name__ == '__main__':
103     poetry_main()
```

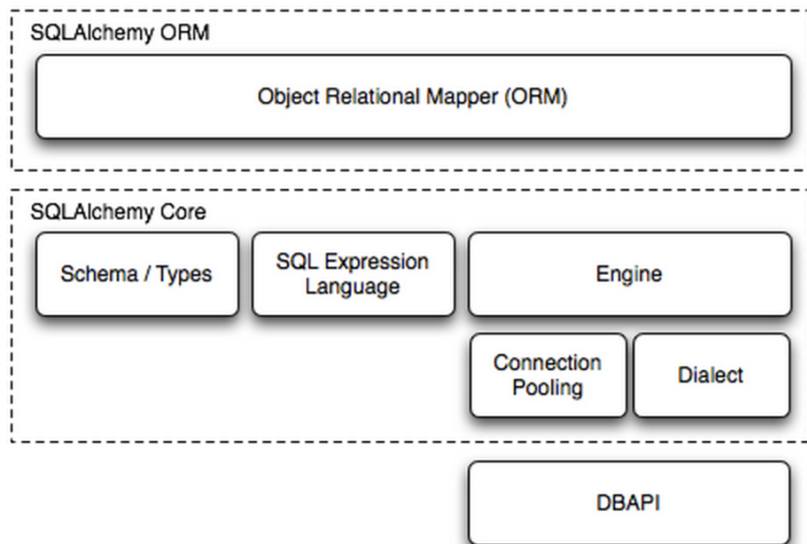
Twisted深入

<http://krondo.com/an-introduction-to-asynchronous-programming-and-twisted/>

<http://blog.csdn.net/hanhuili/article/details/9389433>

SQLAlchemy ORM

SQLAlchemy是Python编程语言下的一款ORM框架，该框架建立在数据库API之上，使用关系对象映射进行数据库操作，简言之便是：将对象转换成SQL，然后使用数据API执行SQL并获取执行结果



Dialect用于和数据API进行交流，根据配置文件的不同调用不同的数据库API，从而实现
对数据库的操作，如：

```

1 MySQL-Python
2     mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
3
4 pymysql
5     mysql+pymysql://<username>:<password>@<host>/<dbname>[?<options>]
6
7 MySQL-Connector
8     mysql+mysqlconnector://<user>:<password>@<host>[:<port>]/<dbname>
9
10 cx_Oracle
11     oracle+cx_oracle://user:pass@host:port/dbname[?key=value&key=value...]
12
13 更多详见: http://docs.sqlalchemy.org/en/latest/dialects/index.html
  
```

步骤一：

使用 Engine/ConnectionPooling/Dialect 进行数据库操作，Engine使用
ConnectionPooling连接数据库，然后再通过Dialect执行SQL语句。

```

1 #!/usr/bin/env python
2 # -*- coding:utf-8 -*-
3
4 from sqlalchemy import create_engine
5
6
7 engine = create_engine("mysql+mysqldb://root:123@127.0.0.1:3306/s11", max_o
8
9 engine.execute(
10     "INSERT INTO ts_test (a, b) VALUES ('2', 'v1')"
11 )
12
13 engine.execute(
14     "INSERT INTO ts_test (a, b) VALUES (%s, %s)",
15     ((555, "v1"), (666, "v1")),
16 )
17 engine.execute(
18     "INSERT INTO ts_test (a, b) VALUES (%(id)s, %(name)s)",
19     id=999, name="v1"
20 )
21
22 result = engine.execute('select * from ts_test')
23 result.fetchall()
  
```

步骤二:

使用 Schema Type/SQL Expression

Language/Engine/ConnectionPooling/Dialect 进行数据库操作。Engine使用 Schema Type创建一个特定的结构对象，之后通过SQL Expression Language将该对象转换成SQL语句，然后通过 ConnectionPooling 连接数据库，再然后通过 Dialect 执行SQL，并获取结果。

```

1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3
4  from sqlalchemy import create_engine, Table, Column, Integer, String, MetaData
5
6  metadata = MetaData()
7
8  user = Table('user', metadata,
9      Column('id', Integer, primary_key=True),
10     Column('name', String(20)),
11 )
12
13 color = Table('color', metadata,
14     Column('id', Integer, primary_key=True),
15     Column('name', String(20)),
16 )
17 engine = create_engine("mysql+mysqldb://root@localhost:3306/test", max_over
18
19 metadata.create_all(engine)

```

增删改查

```

1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3
4  from sqlalchemy import create_engine, Table, Column, Integer, String, MetaData
5
6  metadata = MetaData()
7
8  user = Table('user', metadata,
9      Column('id', Integer, primary_key=True),
10     Column('name', String(20)),
11 )
12
13 color = Table('color', metadata,
14     Column('id', Integer, primary_key=True),
15     Column('name', String(20)),
16 )
17 engine = create_engine("mysql+mysqldb://root:123@127.0.0.1:3306/s11", max_o
18
19 conn = engine.connect()
20
21 # 创建SQL语句, INSERT INTO "user" (id, name) VALUES (:id, :name)
22 conn.execute(user.insert(), {'id': 7, 'name': 'seven'})
23 conn.close()
24
25 # sql = user.insert().values(id=123, name='wu')
26 # conn.execute(sql)
27 # conn.close()
28
29 # sql = user.delete().where(user.c.id > 1)
30
31 # sql = user.update().values(fullname=user.c.name)
32 # sql = user.update().where(user.c.name == 'jack').values(name='ed')
33

```

```

34 # sql = select([user, ])
35 # sql = select([user.c.id, ])
36 # sql = select([user.c.name, color.c.name]).where(user.c.id==color.c.id)
37 # sql = select([user.c.name]).order_by(user.c.name)
38 # sql = select([user]).group_by(user.c.name)
39
40 # result = conn.execute(sql)
41 # print result.fetchall()
42 # conn.close()

```

一个简单的完整例子

```

1 from sqlalchemy import create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3 from sqlalchemy import Column, Integer, String
4 from sqlalchemy.orm import sessionmaker
5
6 Base = declarative_base() #生成一个SqlORM 基类
7
8
9 engine = create_engine("mysql+mysqldb://root@localhost:3306/test",echo=False)
10
11
12 class Host(Base):
13     __tablename__ = 'hosts'
14     id = Column(Integer,primary_key=True,autoincrement=True)
15     hostname = Column(String(64),unique=True,nullable=False)
16     ip_addr = Column(String(128),unique=True,nullable=False)
17     port = Column(Integer,default=22)
18
19 Base.metadata.create_all(engine) #创建所有表结构
20
21 if __name__ == '__main__':
22     SessionCls = sessionmaker(bind=engine) #创建与数据库的会话session class
23     session = SessionCls()
24     #h1 = Host(hostname='localhost',ip_addr='127.0.0.1')
25     #h2 = Host(hostname='ubuntu',ip_addr='192.168.2.243',port=20000)
26     #h3 = Host(hostname='ubuntu2',ip_addr='192.168.2.244',port=20000)
27     #session.add(h3)
28     #session.add_all([h1,h2])
29     #h2.hostname = 'ubuntu_test' #只要没提交,此时修改也没问题
30     #session.rollback()
31     #session.commit() #提交
32     res = session.query(Host).filter(Host.hostname.in_(['ubuntu2','localhos
33     print(res)

```

更多内容详见:

<http://www.jianshu.com/p/e6bba189fcbd>

http://docs.sqlalchemy.org/en/latest/core/expression_api.html

注: SQLAlchemy无法修改表结构, 如果需要可以使用SQLAlchemy开发者开源的另外一个软件Alembic来完成。

步骤三:

使用 ORM/Schema Type/SQL Expression

Language/Engine/ConnectionPooling/Dialect 所有组件对数据进行操作。根据类创建对象, 对象转换成SQL, 执行SQL。

```

1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3
4  from sqlalchemy.ext.declarative import declarative_base
5  from sqlalchemy import Column, Integer, String
6  from sqlalchemy.orm import sessionmaker
7  from sqlalchemy import create_engine
8
9  engine = create_engine("mysql+mysqldb://root:123@127.0.0.1:3306/s11", max_o
10
11  Base = declarative_base()
12
13
14  class User(Base):
15      __tablename__ = 'users'
16      id = Column(Integer, primary_key=True)
17      name = Column(String(50))
18
19  # 寻找Base的所有子类, 按照子类的结构在数据库中生成对应的数据表信息
20  # Base.metadata.create_all(engine)
21
22  Session = sessionmaker(bind=engine)
23  session = Session()
24
25
26  # ##### 增 #####
27  # u = User(id=2, name='sb')
28  # session.add(u)
29  # session.add_all([
30  #     User(id=3, name='sb'),
31  #     User(id=4, name='sb')
32  # ])
33  # session.commit()
34
35  # ##### 删除 #####
36  # session.query(User).filter(User.id > 2).delete()
37  # session.commit()
38
39  # ##### 修改 #####
40  # session.query(User).filter(User.id > 2).update({'cluster_id' : 0})
41  # session.commit()
42  # ##### 查 #####
43  # ret = session.query(User).filter_by(name='sb').first()
44
45  # ret = session.query(User).filter_by(name='sb').all()
46  # print ret
47
48  # ret = session.query(User).filter(User.name.in_(['sb','bb'])).all()
49  # print ret
50
51  # ret = session.query(User.name.label('name_label')).all()
52  # print ret,type(ret)
53
54  # ret = session.query(User).order_by(User.id).all()
55  # print ret
56
57  # ret = session.query(User).order_by(User.id)[1:3]
58  # print ret
59  # session.commit()

```

外键关联

A one to many relationship places a foreign key on the child table referencing the parent.`relationship()` is then specified on the parent, as referencing a collection of items represented by the child

```
from sqlalchemy import Table, Column, Integer, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
```

```
Base = declarative_base()
```

```
1 <br>class Parent(Base):
2     __tablename__ = 'parent'
3     id = Column(Integer, primary_key=True)
4     children = relationship("Child")
5
6 class Child(Base):
7     __tablename__ = 'child'
8     id = Column(Integer, primary_key=True)
9     parent_id = Column(Integer, ForeignKey('parent.id'))
```

To establish a bidirectional relationship in one-to-many, where the "reverse" side is a many to one, specify an additional `relationship()` and connect the two using the `relationship.back_populates` parameter:

```
1 class Parent(Base):
2     __tablename__ = 'parent'
3     id = Column(Integer, primary_key=True)
4     children = relationship("Child", back_populates="parent")
5
6 class Child(Base):
7     __tablename__ = 'child'
8     id = Column(Integer, primary_key=True)
9     parent_id = Column(Integer, ForeignKey('parent.id'))
10    parent = relationship("Parent", back_populates="children")
```

Child will get a `parent` attribute with many-to-one semantics.

Alternatively, the `backref` option may be used on a single `relationship()` instead of using `back_populates`:

```
1 class Parent(Base):
2     __tablename__ = 'parent'
3     id = Column(Integer, primary_key=True)
4     children = relationship("Child", backref="parent")
```

附，原生sql join查询

几个Join的区别 <http://stackoverflow.com/questions/38549/difference-between-inner-and-outer-joins>

- **INNER JOIN:** Returns all rows when there is at least one match in BOTH tables
- **LEFT JOIN:** Return all rows from the left table, and the matched rows from the right table
- **RIGHT JOIN:** Return all rows from the right table, and the matched rows from the left table

```
1 select host.id,hostname,ip_addr,port,host_group.name from host right join h
```

in SQLAlchemy

```
1 session.query(Host).join(Host.host_groups).filter(HostGroup.name=='t1').gro
```

group by 查询


```
1 | select name,count(host.id) as NumberOfHosts from host right join host_group
```

in SQLAlchemy

```
1 | from sqlalchemy import func
2 | session.query(HostGroup, func.count(HostGroup.name)).group_by(HostGroup.name)
3 |
4 | #another example
5 | session.query(func.count(User.name), User.name).group_by(User.name).all()
6 | FROM users GROUP BY users.name
```

[更多ORM内容猛点这里](#)

本节作业一

题目:IO多路复用版FTP

需求:

1. 实现文件上传及下载功能
2. 支持多连接并发传文件
3. ☐使用select or selectors

本节作业二

题目: rpc命令端

需求:

1. 可以异步的执行多个命令
2. 对多台机器

```
>>:run "df -h" --hosts 192.168.3.55 10.4.3.4
```

```
task id: 45334
```

```
>>: check_task 45334
```

```
>>:
```

分类: [Python自动化开发之路](#)

好文要顶

已关注

收藏该文

金角大王

关注 - 5

粉丝 - 10911

我在关注他 取消关注

2

0

posted @ 2016-03-06 19:14 金角大王 阅读(46666) 评论(12) 编辑 收藏

评论列表

- #1楼 2016-07-10 12:11 大阿拉伯人

okokok

回复 引用

支持(0) 反对(0)
- #2楼 2016-07-10 12:12 大阿拉伯人

大王，悟空来了

回复 引用

支持(0) 反对(0)
- #3楼 2016-09-19 09:27 Dus

吃瓜群众路过。

回复 引用

支持(0) 反对(0)
- #4楼 2016-09-24 15:54 屌丝逆袭记

666666666666

回复 引用

支持(0) 反对(0)
- #5楼 2017-03-19 12:07 兴趣使然的coder

alex用腚眼都能想明白的问题 你们学了这么久还不懂！？

回复 引用

支持(0) 反对(0)
- #6楼 2017-03-20 20:42 小祎

@ 兴趣使然的coder
你的意思是说他的腚眼比我脑袋都大？？？

回复 引用

支持(0) 反对(0)
- #7楼 2017-09-15 12:09 Alano的自嘲

想问一下 为什么我执行Rabbitmq 会报错

```
Traceback (most recent call last):
  File "E:\Python\anaconda\pika_send.py", line 6, in <module>
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
  File "C:\Users\dell\AppData\Local\Programs\Python\Python36\lib\site-packages\pika\adapters\blocking_connection.py", line 374, in __init__
    self._process = self._start_connection_getting()
  File "C:\Users\dell\AppData\Local\Programs\Python\Python36\lib\site-packages\pika\adapters\blocking_connection.py", line 414, in _process_is_for_run
    self._open_error_result.is_ready()
  File "C:\Users\dell\AppData\Local\Programs\Python\Python36\lib\site-packages\pika\adapters\blocking_connection.py", line 468, in _flush_output
    raise exceptions.ConnectionClosed(maybe_exception)
pika.exceptions.ConnectionClosed: Connection to 127.0.0.1:5672 failed: timeout

Process finished with exit code 1
```

支持(2) 反对(0)
- #8楼 2017-12-14 14:47 一只火眼金睛的男猴

Mark一下，晚上下班搞一搞。

回复 引用

支持(0) 反对(0)
- #9楼 2018-01-12 17:15 橙子味的萝卜

TypeError: exchange_declare() got an unexpected keyword argument 'type'

这里面的type参数必须指定为exchange_type才行，改完之后运行正常，有大神说明下吗

支持(0) 反对(0)
- #10楼 2018-04-22 11:09 沐阳静枫

大王，为什么会出现这种错误，网上搜了没有解决。

```
raise exceptions.ConnectionClosed(maybe_exception)
pika.exceptions.ConnectionClosed: Connection to 127.0.0.1:5672 failed: timeout
```

支持(0) 反对(0)
- #11楼 2018-05-22 13:13 余波可

回复 引用

帖子有个地方错了，有选择的接收消息(exchange type=direct) 这部分，exchange_type 错写成 type了

支持(0) 反对(1)

#12楼 2020-01-14 16:15 yongqi-911 [回复](#) [引用](#)

应该把: type='fanout' 改成 change_type='fanout'
channel.exchange_declare(exchange='logs_fanout', exchange_type='fanout')





版本不一样，就会报错了

支持(1) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

发表评论

编辑 预览

B    

支持 Markdown

[提交评论](#) [退出](#) [订阅评论](#)

[Ctrl+Enter快捷键提交]

- 【推荐】超50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】2019热门技术盛会400则演讲资料全收录

相关博文:

- 异步IO\数据库\队列\缓存
 - 异步io\数据库\队列\缓存
 - 异步IO\数据库\队列\缓存
 - python-----异步IO\数据库\队列\缓存
 - Python之路,Day10 - 异步IO\数据库\队列\缓存
- » 更多推荐...

最新 IT 新闻:

- 搜狗2020年Q2财报: 营收18亿 输入法日活4.84亿
 - 搜狐Q2总营收4.21亿美元 张朝阳: 搜狗业务符合预期
 - 俞渝夫妇被儿子告上法庭! 这家人又上演什么大戏?
 - 倪光南: 没有网络安全就没有新基建的安全 自主可控是新基建前提
 - LG U+将商业化5G“自动驾驶机器人”
- » 更多新闻...