**001、列举10个常见的Linux命令？**

cd pwd touch ls mkdir rm help sudo ssh date clear vim ps cat more scp cp find mv grep echo

**002、Linux系统简介？**

/bin /home /dev /proc /mnt /usr /etc /lib

/bin:存储系统所使用命令的可执行文件

/home：普通用户的家目录 (cd home)；

/dev:外部设备（存的是设备的接口 通过接口能访问设备）；

/proc：虚拟目录 以进程为单位存储内存的映射--------一些统计信息；

/mnt：临时挂载点（将接口挂载到临时挂载点进行操作）；

/usr：第三方软件的一些文档；

/etc：系统配置目录；

/lib：库文件（静态库 共享库（动态库））

冯\*诺依曼 五大部件

（计算器 控制器）cpu (存储器)内存 主存------缓存数据（为了使cpu效率提高）（输入设备 输出设备）I/O 硬盘 显示器 键盘

数据从硬盘读到内存 再读到cpu去执行

Linux上一切皆文件：不以文件扩展名区分文件类型（将文件分为五大类）

          普通文件：-（.c     .h     .cpp   .java       .class       .txt     .pdf都属于普通文件）

        目录文件：d   文件夹

        链接文件:l(L)

        管道文件:p （进程间通讯的专用文件）

        设备文件 ：字符设备(c)           块设备(b)           套接字(s)

ls -l

文件提供给不同用户不同的权限（属主（创建者） 属组（组用户的名称） 其他用户

三种用户 ：三组权限

          u             g         o

r： 读权限

w：写权限

x：执行权限 普通文件（前提是他是可执行文件）

是否可进入   目录文件         针对文件不同，x的含义不同

文件操作命令：

文件创建：普通文件 （touch） 目录文件----文件夹（mkdir）

文件删除：普通文件（rm） 目录文件(rmdir-----删除空目录 rm -r 删除非空目录)

文件拷贝：普通文件（cp原文件路径+文件名 拷贝的目的地） 目录文件(cp -r 原文件路径+文件名 拷贝目的地的路径)把目录里面的文件都拷贝过去

文件剪切/重命名：普通文件（mv） 目录文件（mv）

边移动边重命名：如下图 stu3目录下本来就有一个project1所以剪切过来要重命名。这里的project1和project2是相同的文件

mv /home/stu1/project1 (这里中间要用空格隔开)/home/stu1/project2 单纯的重命名

修改属性：（文件类型一旦创建就不能修改，权限可以修改，链接数不能改，是随着操作变化的；属主可以改；属组可以修改；文件大小不能改，是你往里面加了东西才修改）

修改属主：只能root用户修改，chown newuser（改成谁） filename（所改的文件）

修改属组：只能root用户修改，chgrp newgroup（新的组，原来在那个组不用关心） filename

修改权限：chomd

1、用字符方式修改 chmod a（all所有的）/u/g/o +/-/=(赋值) r/w/x/rw/rx/wx/rwx

2、用数字方式修改：

每个n代表一个用户的权限，一共有3个n,每个n都是0-7的数字

chmod nnn filename

linux基本命令

cd 跟路径 ：切换目录

ls -l ：显示文件详细信息

ls -a ： 显示所有文件 （包含隐藏文件文件名以“.”开头的都是隐藏文件）

cd .. ：可以退回上级目录

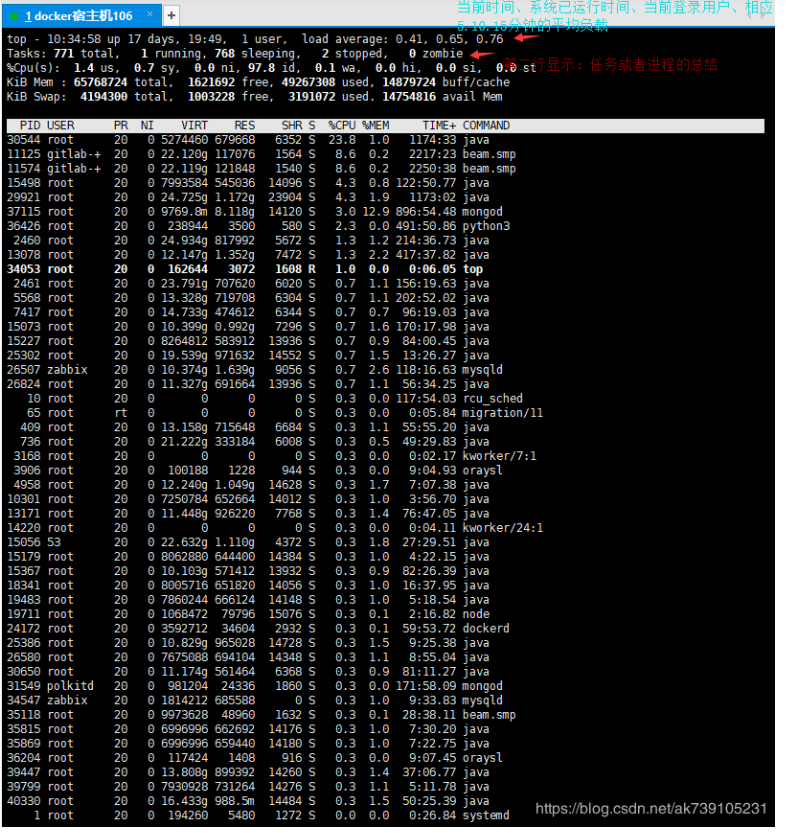
cd - ：两个目录之间来回切换

cd ~ ：直接返回家目录

pwd：查看当前目录的绝对路径/查看当前位置

**1.top**

**top是linux中自带的系统监控命令，实时监控系统各项指标**

****

第一行各字段含义：

这些字段显示：

当前时间

系统已运行的时间

当前登录用户的数量

相应最近5、10和15分钟内的平均负载。

Ps：1核cpu饱满负载为1，1以下均正常不会出现拥堵情况

# 总核数 = 物理CPU个数 X 每颗物理CPU的核数

# 总逻辑CPU数 = 物理CPU个数 X 每颗物理CPU的核数 X 超线程数

# 查看物理CPU个数cat /proc/cpuinfo| grep "physical id"| sort| uniq| wc -l

# 查看每个物理CPU中core的个数(即核数)cat /proc/cpuinfo| grep "cpu cores"| uniq

# 查看逻辑CPU的个数cat /proc/cpuinfo| grep "processor"| wc -l

1.物理cpu数：主板上实际插入的cpu数量，可以数不重复的 physical id 有几个（physical id）

2.cpu核数：单块CPU上面能处理数据的芯片组的数量，如双核、四核等 （cpu cores）

3.逻辑cpu数：一般情况下，逻辑cpu=物理CPU个数×每颗核数，如果不相等的话，则表示服务器的CPU支持超线程技术（HT：简单来说，它可使处理器中的1 颗内核如2 颗内核那样在操作系统中发挥作用。这样一来，操作系统可使用的执行资源扩大了一倍，大幅提高了系统的整体性能，此时逻辑cpu=物理CPU个数×每颗核数x2）

第二行字段含义：

第二行显示的是任务或者进程的总结。进程可以处于不同的状态。这里显示了全部进程的数量。除此之外，还有正在运行、睡眠、停止、僵尸进程的数量（僵尸是一种进程的状态）。

第三行字段含义：

第三行主要显示cpu信息：

us, user： 运行(未调整优先级的) 用户进程的CPU

sy，system: 运行内核进程的CPU

ni，niced：运行已调整优先级的用户进程的CPU

Id，空闲cpu

wa，IO wait: 用于等待IO完成的CPU

hi：处理硬件中断的CPU

si: 处理软件中断的CPU

st：这个虚拟机被hypervisor偷去的CPU（译注：如果当前处于一个hypervisor下的vm，实际上hypervisor也是要消耗一部分CPU处理时间的）。

第四行和第五行字段含义：

第四行是物理内存使用：

物理内存显示如下:全部可用内存、已使用内存、空闲内存、缓冲内存。

第五行是虚拟内存使用(交换空间)：

相似地：交换部分显示的是：全部、已使用、空闲和缓冲交换空间。

第四行中使用中的内存总量（used）指的是现在系统内核控制的内存数，空闲内存总量（free）是内核还未纳入其管控范围的数量。纳入内核管理的内存不见得都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到free中去，因此在linux上free内存会越来越少，但不用为此担心。

如果出于习惯去计算可用内存数，这里有个近似的计算公式：第四行的free + 第四行的buffers + 第五行的cached，按这个公式此台服务器的可用内存：

对于内存监控，在top里我们要时刻监控第五行swap交换分区的used，如果这个数值在不断的变化，说明内核在不断进行内存和swap的数据交换，这是真正的内存不够用了。

第7行各字段含义：

PID：进程ID，进程的唯一标识符

USER：进程所有者的实际用户名。

PR：进程的调度优先级。这个字段的一些值是'rt'。这意味这这些进程运行在实时态。

NI：进程的nice值（优先级）。越小的值意味着越高的优先级。负值表示高优先级，正值表示低优先级

VIRT：进程使用的虚拟内存。进程使用的虚拟内存总量，单位kb。VIRT=SWAP+RES

RES：驻留内存大小。驻留内存是任务使用的非交换物理内存大小。进程使用的、未被换出的物理内存大小，单位kb。RES=CODE+DATA

SHR：SHR是进程使用的共享内存。共享内存大小，单位kb

S：这个是进程的状态。它有以下不同的值:

D - 不可中断的睡眠态。

R – 运行态

S – 睡眠态

T – 被跟踪或已停止

Z – 僵尸态

%CPU：自从上一次更新时到现在任务所使用的CPU百分比。

%MEM：进程使用的可用物理内存百分比。

TIME+：任务启动后到现在所使用的全部CPU时间，精确到百分之一秒。

COMMAND：运行进程所使用的命令。进程名称（命令名/命令行）

Top其他快捷命令：

1.按1显示所有逻辑cpu的情况

2.按e显示各进程占用资源单位单位

3.按E显示整体占用资源单位

4.进程排序默认按%cpu进程，shift+”<”或”>”改变排序字段

5.d 指定每两次屏幕信息刷新之间的时间间隔。当然用户可以使用s交互命令来改变之。

         6.p 通过指定监控进程ID来仅仅监控某个进程的状态。

         7.i 使top不显示任何闲置或者僵死进程。

8.c 显示整个命令行而不只是显示命令名

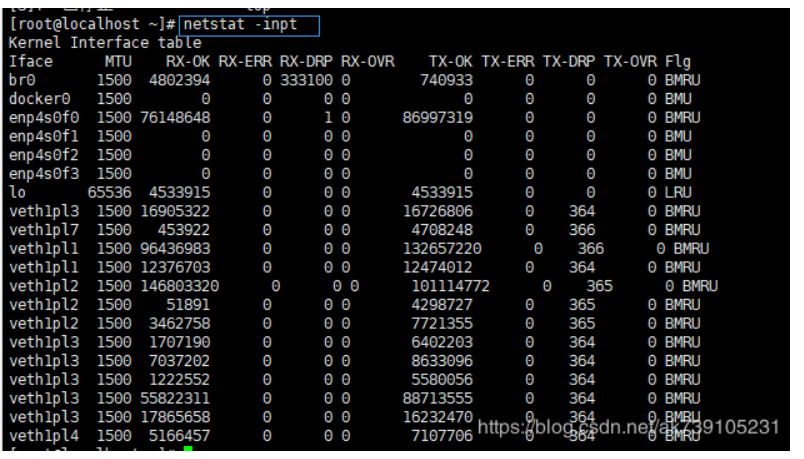
**2.查看进程的几种方式**

**1.ps -ef  查看当前运行的所有进程**

**2.ps -aux  查看正在内存中的进程**

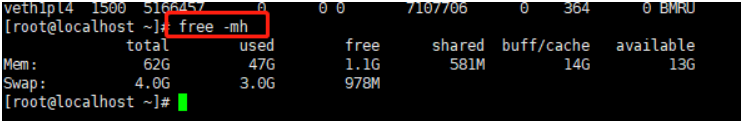
**3.查看系统端口**

**netstat  -lnpt**

****

**4.查看内存**

**free -mh**

****

**5.重定向**

**1.追加重定向 >>**

**2.覆盖重定向  >**

**6.查看磁盘空间**

**df -Th  统计磁盘空间**

**du -sh 统计当前目录大小**

**du -sh \*  列出当前目录下所有文件和文件夹大小**

**7.kill**

**kill是linux结束进程的命令，发送指定的信号到相应进程。**

**Kill使用：kill [option] [pid]**

**kill  [pid]          进程在退出之前可以清理并释放资源**

**Kill -9 [pid]  强制杀死该进程，可能会出现数据丢失情况**

**003、linux怎么查看端口？**

答：支支吾吾

**004、linux怎么查看内存占用情况？**

答：支支吾吾

**005、ping这个命令的底层协议是什么？**

答：tcp。

使用的是[ICMP](http://www.so.com/s?q=ICMP&ie=utf-8&src=internal_wenda_recommend_text)协议，是“Internet Control Message Protocol”（Internet控制消息协议）的缩写，是[TCP/IP协议族](http://www.so.com/s?q=TCP%2FIP%E5%8D%8F%E8%AE%AE%E6%97%8F&ie=utf-8&src=internal_wenda_recommend_text)的一个子协议，用于在IP主机、[路由器](http://www.so.com/s?q=%E8%B7%AF%E7%94%B1%E5%99%A8&ie=utf-8&src=internal_wenda_recommend_text)之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

它是用来检查网络是否通畅或者网络连接速度的命令。它所利用的原理是这样的：利用网络上机器IP地址的唯一性，给目标IP地址发送一个数据包，再要求对方返回一个同样大小的数据包来确定两台网络机器是否连接相通，时延是多少。

**006、tcp建立连接？**

答：三次握手四次挥手

**007、tcp的回执是什么包？**

答：不记得了。

传输控制协议（Transmission Control Protocol, TCP）

TCP协议主为了在主机间实现高可靠性的包交换传输协议。本文将描述协议标准和实现的一些方法。因为计算机网络在现代社会中已经是不可缺少的了，TCP协议主要在网络不可靠的时候完成通信，对军方可能特别有用，但是对于政府和商用部门也适用。

  TCP是面向连接的端到端的可靠协议。它支持多种网络应用程序。TCP对下层服务没有多少要求，它假定下层只能提供不可靠的数据报服务，它可以在多种硬件构成的网络上运行。下面的图是TCP在层次式结构中的位置，它的下层是IP协议，TCP可以根据IP协议提供的服务传送大小不定的数据，IP协议负责对数据进行分段，重组，在多种网络中传送。

TCP的上面就是应用程序，下面是IP协议，上层接口包括一系列类似于操作系统中断的调用。对于上层应用程序来说，TCP应该能够异步传送数据。下层接口我们假定为IP协议接口。为了在并不可靠的网络上实现面向连接的可靠的传送数据，TCP必须解决可靠性，流量控制的问题，必须能够为上层应用程序提供多个接口，同时为多个应用程序提供数据，同时TCP必须解决连接问题，这样TCP才能称得上是面向连接的，最后，TCP也必须能够解决通信安全性的问题。

网络环境包括由网关（或其它设备）连接的网络，网络可以是局域网也可以是一些城域网或广域网，但无论它们是什么，它们必须是基于包交换的。主机上不同的协议有不同的端口号，一对进程通过这个端口号进行通信。这个通信不包括计算机内的I/O操作，只包括在网络上进行的操作。

  网络上的计算机被看作包传送的源和目的结点。特别应该注意的是：计算机中的不同进程可能同时进行通信，这时它们会用端口号进行区别，不会把发向A进程的数据由B进程接收的。

进程为了传送数据会调用TCP，将数据和相应的参数传送给TCP，于是TCP会将数据传送到目的TCP那里，当然这是通过将TCP包打包在IP包内在网络上传送达到的。

  接收方TCP在接收到数据后会通信上层应用程序，TCP会保证接收数据顺序的正确性。虽然下层协议可能不会保证顺序是正确的。这里需要说明的是网关在接收到这个包后，会将包解开，看看是不是已经到目的地了，如果没有到，应该走什么路由达到目的地，在决定后，网关会根据下一个网络内的协议情况再次将TCP包打包传送，如果需要，还要把这个包再次分成几段再传送。

  这个落地检查的过程是一个耗时的过程。从上面，我们可以看出TCP传送的基本过程，当然具体过程可能要复杂得多。

在实现TCP的主机上，TCP可以被看成是一个模块，和文件系统区别不大，TCP也可以调用一些操作系统的功能，TCP不直接和网络打交道，控制网络的任务由专门的设备驱动模块完成。

  TCP只是调用IP接口，IP向TCP提供所有TCP需要的服务。通过下图我们可以更清楚地看到TCP协议的结构。

上面已经说过了，TCP连接是可靠的，而且保证了传送数据包的顺序，保证顺序是用一个序号来保证的。响应包内也包括一个序列号，表示接收方准备好这个序号的包。

  在TCP传送一个数据包时，它同时把这个数据包放入重发队列中，同时启动记数器，如果收到了关于这个包的确认信息，将此包从队列中删除，如果计时超时则需要重新发送此包。请注意，从TCP返回的确认信息并不保证最终接收者接收到数据，这个责任由接收方负责。

每个用于传送TCP的通道都有一个端口标记，因为这个标记是由每个TCP终端确定的，因此TCP可能不唯一，为了保证这个数值的唯一，要使用网络地址和端口号的组合达到唯一标识的目的，我们称这个为了套接字（Socket），一个连接由连接两端的套接字标识，本地的套接字可能和不同的外部套接字通信，这种通信是全双工的。

通过向本地端口发送OPEN命令及外部套接字参数建立连接，TCP返回一个标记这个连接的名称，以后如果用户需要使用这个名称标记这个连接。为了保存这个连接的信息，我们假设有一个称为传输控制块（Transmission Control Block，TCB）的东西来保存。

  OPEN命令还指定这个连接的建立是主动请求还是被动等待请求。下面我们要涉及具体的功能了，TCP段以internet数据报的形式传送。IP包头传送不同的信息域，包括源地址和目的地址。TCP头跟在internet包头后面，提供了一些专用于TCP协议的信息。

  下图是TCP包头格式图：

源端口：16位；

目的端口：16位

序列码：32位，当SYN出现，序列码实际上是初始序列码（ISN），而第一个数据字节是ISN+1；

确认码：32位，如果设置了ACK控制位，这个值表示一个准备接收的包的序列码；

数据偏移量：4位，指示何处数据开始；

保留：6位，这些位必须是0；

控制位：6位；

窗口：16位；

校验位：16位；

优先指针：16位，指向后面是优先数据的字节；

选项：长度不定；但长度必须以字节记；选项的具体内容我们结合具体命令来看；

填充：不定长，填充的内容必须为0，它是为了保证包头的结合和数据的开始处偏移量能够被32整除；

我们前面已经说过有一个TCB的东西了，TCB里有存储了包括发送方，接收方的套接字，用户的发送和接收的缓冲区指针等变量。

  除了这些还有一些变量和发送接收序列号有关：

发送序列变量

SND。UNA - 发送未确认

SND。NXT - 发送下一个

SND。WND - 发送窗口

SND。UP - 发送优先指针

SND。WL1 - 用于最后窗口更新的段序列号

SND。

  WL2 - 用于最后窗口更新的段确认号

ISS - 初始发送序列号

接收序列号

RCV。NXT - 接收下一个

RCV。WND - 接收下一个

RCV。UP - 接收优先指针

IRS - 初始接收序列号

下图会帮助您了解发送序列变量间的关系：

当前段变量

SEG。

  SEQ - 段序列号

SEG。ACK - 段确认标记

SEG。LEN - 段长

SEG。WND - 段窗口

SEG。UP - 段紧急指针

SEG。PRC - 段优先级

连接进程是通过一系列状态表示的，这些状态有：LISTEN，SYN-SENT，SYN-RECEIVED，ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT和 CLOSED。

  CLOSED表示没有连接，各个状态的意义如下：

LISTEN - 侦听来自远方TCP端口的连接请求；

SYN-SENT - 在发送连接请求后等待匹配的连接请求；

SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；

ESTABLISHED - 代表一个打开的连接，数据可以传送给用户；

FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；

FIN-WAIT-2 - 从远程TCP等待连接中断请求；

CLOSE-WAIT - 等待从本地用户发来的连接中断请求；

CLOSING - 等待远程TCP对连接中断的确认；

LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；

TIME-WAIT - 等待足够的时间以确保远程TCP接收到连接中断请求的确认；

CLOSED - 没有任何连接状态；

TCP连接过程是状态的转换，促使发生状态转换的是用户调用：OPEN，SEND，RECEIVE，CLOSE，ABORT和STATUS；传送过来的数据段，特别那些包括以下标记的数据段SYN，ACK，RST和FIN；还有超时，上面所说的都会时TCP状态发生变化。

下面的图表示了TCP状态的转换，但这图中没有包括错误的情况和错误处理，不要把这幅图看成是总说明了。

3。3。 序列号

请注意，我们在TCP连接中发送的字节都有一个序列号。因为编了号，所以可以确认它们的收到。对序列号的确认是累积性的，也就是说，如果用户收到对X的确认信息，这表示在X以前的数据（不包括X）都收到了。

  在每个段中字节是这样安排的：第一个字节在包头后面，按这个顺序排列。我们需要认记实际的序列空间是有限的，虽然很大，但是还是有限的，它的范围是0到2的32次方减1。我想熟悉编程的一定知道为什么要在计算两个段是不是相继的时候要使用2的32次方为模了。

  TCP必须进行的序列号比较操作种类包括以下几种：

(a) 决定一些发送了的但未确认的序列号；

(b) 决定所有的序列号都已经收到了；

(c) 决定下一个段中应该包括的序列号。

对于发送的数据TCP要接收确认，处理确认时必须进行下面的比较操作：

SND。

  UNA = 最老的确认了的序列号；

SND。NXT = 下一个要发送的序列号；

SEG。ACK = 接收TCP的确认，接收TCP期待的下一个序列号；

SEG。SEQ = 一个数据段的第一个序列号；

SEG。LEN = 数据段中包括的字节数；

SEG。

  SEQ+SEG。LEN-1 = 数据段的最后一个序列号。

请注意下面的关系：

SND。UNA 0

RCV。NXT =0

0

不可接受

>0

>0

RCV。NXT = B SYN 本方序列号是X

2) A B ACK 确认对方序列号

上面的第2步和第3步可以合并，这时可以成为3阶段，所以我们可以称它为三消息握手。

  这个过程是必须的，因为序列号不和全局时钟关联，TCP也可以有不同的机制选择ISN。接收到第一个SYN的接收方不可能知道这个数据段是不是被延时，除非它记住了在连接上使用的最近的序列号（这通常是不可能的），因此它必须要求发送者确认。

为了保证TCP获得的确认是刚才发送的段产生的，而不是仍然在网络中的老数据段产生的，因此TCP必须在MSL时间之内保持沉默。

  在本文中，我们假设MSL=2小时，这是出于工程的需要，如果用户觉得可以，他可以改变MSL。请注意如果TCP重新初始化，而内存中的序列号正在使用，不需要等待，但必须确认使用的序列号比当前使用的要大。

如果一台主机在未保留任何序列号的情况下失败，那么它应该在MSL时间之内不发出任何数据段。

  下面将会这一情况进行说明。TCP的实现可以不遵守这个规定，但是这会造成老数据被当成新数据接收，而新数据被当成老数据拒绝的情况。

每当数据段形成并进入输出队列，TCP会为它指定序列空间中的一个值。TCP中多复本检测和序列算法都依赖于这个地址空间，在对方发送或接收之前不会超过2的32次方个包存在于输出队列中。

  所有多余的数据段都会被删除。如果没有这个规定，会出现多个数据段被指定同一个序列号的情况，会造成混乱。数据段中序列号的多少和数据段中的字节数一样多。

在通常情况下，TCP保留下一个要发送的序列号和还未确认的最老的序列号，不要在没有确认的时候就再次使用，这样会有些风险，也正是因为这样的目的，所以序列空间很大。

  对于2M的网络，要4。5小时来耗尽序列空间，因为一个数据段可能的最大生存时间也不过十几分之一秒，这就留下了足够的空间；而在100M的网络上需要5。4分钟，虽然少了点，但也可以了。

如果在实现TCP时没有为保存序列号留下空间，那清除多余的包可能就不能实现了，因此推荐这种类型的TCP实现最好在失败后等待MSL时间，这样保证多余的包被删除。

  这种情况有时候也可能会出现在保留序列号的TCP实现中。如果TCP在选择一个另一个TCP连接正在使用的序列号时，这台主机突然失败了，这就产生了问题。这个问题的实质在于主机不知道它失败了多久，也不知道多余的复本是不是还在网络中。

处理这种问题的方法是等待MSL时间，如果不这样就要冒着对方错误接收数据的危险，要等待的时间也就称为“沉默时间”。

  实现者可以让用户选择是不是等待，但是无论用户如何也不见得非要等待MSL时间。

3。4。 建立一个连接

建立连接应用的是三消息握手。如果双方同时都发送SYN也没有关系，双方会发现这个SYN中没有确认，于是就知道了这种情况，通常来说，应该发送一个"reset"段来解决这种情况。

  三消息握手减少了连接失败的可能性。下面就是一个例子，在尖括号是的就是数据段中的内容和标记。其它的就不多说了。

在第2行，TCP A发送SYN初始化序列号，表示它要使用序列号100；第3行中，TCP B给出确认，并且期待着A的带有序列号101的数据段；第4行，TCP A给出确认，而在第5行，它也给出确认，并发送了一些数据，注意第4行的序列号与第5号的一样，因为ACK信息不占用序列号空间内的序列号。

  同时产生请求的情况如下图所示，只复杂一点。

使用三消息握手的主要原因是为了防止使用过期的数据段。为了这个目的，必须引入新的控制消息，RESET。如果接收TCP处理非同步状态，在接收到RESET后返回到LISTEN状态。如果TCP处理下面几种状态ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT时，放弃连接并通过用户。

  我们下面就详细说明后一种情况。

通过上面的例子，我们可以看出TCP连接是如何从过期数据段的干扰下恢复的。请注意第4行和第5行中的RST（RESET信号）。

半开连接和其它非正常状态

如果一方在未通过另一方的情况下关闭连接，或双方虽然失败而不同步的情况我们称为半开连接状态。

  在一方试图发送数据时连接会自动RESET。然而这种情况毕竟属于不正常情况。应该做出相应的处理。如果A处的连接已经关闭，B处并不知道。当B希望发送数据到A时，就会收到RESET信号，表示这个TCP连接有误，要中止当前连接。

假设A和B两个进程相互通信的时候A的TCP发生了失败，A依靠操作系统支持TCP的存在，通常这种情况下会有恢复机制起作用，当TCP重新恢复的时候，A可能希望从恢复点开始工作。

  这样A可能会试图OPEN连接，然后在这个它认为还是打开的连接上传送数据，这时A会从本地（也就是A的）TCP上获得错误消息“未打开连接”。A的TCP将发送包括SYN的数据段。下面的例子将显示这一过程：

上面这个例子中，A方收到的信息并没有确认任何东西，这时候A发现出了问题，于是发送了RST控制信息。

  另一种情况是发生在A失败，而B方仍然试图发送数据时，下面的例子可以表示这种情况，请注意第2行中A对B发送来的信息不知所云。

在下面的例子中，A方和B方进行的被动连接，它们都在等待SYN信息。过期的包传送到B方使B回应了，而收到回应的A却发现不对头，传送RST控制信息，B方返回被动LISTEN状态。

现实中的情况太多了，我们列举一些产生RST控制信息的规则如下：通常情况下，RST在收到的信息不是期待的信息时产生。如果在不能确定时不要轻易发送RST控制信息。下面有三类情况：

如果连接已经不存在，而发送来的消息又不是RST，那么要返回RST。

  如果想拒绝对不存在的连接进行SYN，可以使用这种办法。如果到达的信息有一个ACK域，返回的RST信息可以从ACK域中取得序列号，如果没有这个域，就把RST的序列号设置为0，ACK域被设备为序列号和到达段长度之和。连接仍然处于CLOSE状态。

如果连接处于非同步状态（LISTEN，SYN-SENT，SYN-RECEIVED），而且收到的确认是对未发出包的确认或是接收到数据段的安全级别与不能连接要求的相一一致时，就发送RST。

  如果SYN未被确认时，而且收到的数据段的优先级比要求的优先级要高，那么要么提高本地优先级（得事先征得用户和系统的许可）要么发送RST；如果接收数据段的优先级比要求的优先级低，就算是匹配了，当然如果对方发现优先级不对提高了优先级，在下一个包中提高了优先级，这就不算是匹配了。

  如果连接已经进入SYN，那么接收到数据段的优先级必须和本地优先级一样，否则发送RST。如果到达的信息有一个ACK域，返回的RST信息可以从ACK域中取得序列号，如果没有这个域，就把RST的序列号设置为0，ACK域被设备为序列号和到达段长度之和。

  连接仍然处于与原来相同的状态。

如果连接处于同步状态（ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT），任何超出接收窗口的序列号的数据段都产生如下结果：发出一个空确认数据段，此段中包括当前发送序列号，另外还包括一个确认指出希望接收的下一个数据段的序列号，连接仍然保存在原来的状态。

  如果因为安全级，优先级之类的问题，那就发送RST信号然后进入CLOSED状态

**008、之前有干过测试或者运维吗？**

答：测试就是那样，运维的话开发过一个cmdb项目，还问了我一个测试的工具，我不知道，回答了个不了解。

**009、多进程多线程区别？**

答：进程是资源单位，线程是最小调度单位，然后扯到了gil锁，扯到了垃圾回收机制

**010、数据库怎么提高查询效率？**

答：

建立索引。主要通过三种途径提高数据库查询效率：1.SQL的性能优化，2.数据库的性能优化，3.查询优化

光说SQL语句,就要注意:

1、不要写效率低的语句，特别是有些复杂的，如果效率低就坚决不用，考虑用其他方法代替。

2、注意配合索引

3、存储过程和触发器的效率是由存储过程和触发器内的语句的效率决定的，特别是触发器，如果效率不好会影响整个服务器的性能。

**011、建立索引有什么缺点？**

答：这个不了解：

MySQL建立索引的优点和缺点

**建立索引的优缺点:**

为什么要创建索引呢？

        这是因为，创建索引可以大大提高系统的性能。

        第一、通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

        第二、可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。

        第三、可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

        第四、在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

        第五、通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

        也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？这种想法固然有其合理性，然而也有其片面性。虽然，索引有许多优点， 但是，为表中的每一个列都增加索引，是非常不明智的。

       这是因为，增加索引也有许多不利的一个方面:

        第一、创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

        第二、索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大。

        第三、当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

什么样的字段适合创建索引:

 索引是建立在数据库表中的某些列的上面。因此，在创建索引的时候，应该仔细考虑在哪些列上可以创建索引，在哪些列上不能创建索引。

       一般来说，应该在这些列上创建索引，例如：

       第一、在经常需要搜索的列上，可以加快搜索的速度；

       第二、在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

       第三、在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；

       第四、在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

       第五、在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

       第六、在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

       建立索引，一般按照select的where条件来建立，比如： select的条件是where f1 and f2，那么如果我们在字段f1或字段f2上简历索引是没有用的，只有在字段f1和f2上同时建立索引才有用等。

**什么样的字段不适合创建索引:**

同样，对于有些列不应该创建索引。一般来说，不应该创建索引的的这些列具有下列特点：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，

并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

       第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，

在查询的结果中，结果集的数据行占了表中数据行的很大比 例，即需要在表中搜索的数据行的比例很大。

增加索引，并不能明显加快检索速度。

       第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

        第四，当修改性能远远大于检索性能时，不应该创建索 引。这是因为，修改性能和检索性能是互相矛盾的。

当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。

因此，当修改性能远远大于检索性能时，不应该创建索引。

创建索引的方法::

1、创建索引，例如 create index <索引的名字> on table\_name (列的列表);

      2、修改表，例如 alter table table\_name add index[索引的名字] (列的列表);

      3、创建表的时候指定索引，例如create table table\_name ( [...], INDEX [索引的名字] (列的列表) );

查看表中索引的方法:

show index from table\_name; 查看索引

索引的类型及创建例子::

1.PRIMARY KEY （主键索引）

mysql> alter table table\_name add primary key ( `column` )

2.UNIQUE 或 UNIQUE KEY (唯一索引)

mysql> alter table table\_name add unique (`column`)

     3.FULLTEXT (全文索引)

     mysql> alter table table\_name add fulltext (`column` )

     4.INDEX (普通索引)

     mysql> alter table table\_name add index index\_name ( `column` )

5.多列索引 (聚簇索引)

      mysql> alter table `table\_name` add index index\_name ( `column1`, `column2`, `column3` )

**012、常用的web后台开发框架？**

答：django。

013、**url是怎么找到views的？**

答：讲了一下生命周期，然后讲了as\_view 和dispatch。

初始Django url+views

url是啥？

　　全球统一资源定位符

视图(views)是什么?

　　视图中存放处理逻辑函数等

url的基本格式？

　　url=协议+域名(ip)+端口+路径+参数

url在django中如何起作用？

* 一个请求首先来到项目目录下的urls.py，查找路由规则，按顺序运行每一个url模式，在第一个匹配的模式停止，一旦匹配，django导入并调用给定的视图，如果中间出错或没有匹配到，返回404
* urlpatterns变量是一个(django.urls.path,diango.url.re\_path)列表
* 实际使用过程中为了方便管理项目，一个请求首先来到项目目录下的urls.py，查找路由规则，按顺序运行每一个url模式，按规则匹配后会指向相应的应用urls，再去调用指定应用的视图（views）
* django搜索url搜索的是什么，只搜索路径部分，url的参数和域名不会去搜索

## 规则

* urls中的path，re\_path
* -path(route,view,kWARGS=NONE,name=none)
* -正则-re\_path(route,view,kWARGS=NONE,name=none)
* route:字符串的url规则
* view:是个视图
* kwargs:额外参数，传递给view,必须是一个字典
* name:url的命名 通过reverse实现不管route(字符串的url规则 )怎么变都指向同一个视图
* 
* **django搜索url搜索的是什么，只搜索路径部分，url的参数和域名不会去搜索，那么参数如何传递？**
* **-在url中捕获参数**
* 在url规则中使用‘<变量名>’可以捕获url中的值，传递给视图，注意捕获的值是字符串
* **django搜索url时想要用一条url来代替同种类型的url eg: 127.0.0.1/teacher/52 127.0.0.1/teacher/53...**
* **-路径转换器**
* 常用的路径转换器：url
* -str匹配除了‘/’路径分隔符之外的所有字符串
* -int 匹配整数
* -slug 匹配任意的acii字符 加上连字符和下划线
* -uuid 格式化的ID,区分资源取名用
* -pass匹配任意的非空字符
* python中正则表达式的分组命名 (?<name>pattern)
* 分组name可以不写，但一般写好，因为多了少了都会报错,且不写的时候是按顺序匹配的，容易赋值错误
* -包含其它URLconfs  include
* include截断当前传递剩余到其它地方，参数也会被传递到下一层的每一个views
* -传递额外参数注意
* path,re\_path方法中，传递一个kwargs的字典参数
* 当kwargs的key与url捕获中一致的时候一致的时候，以kwargs为准 ，当然一般不会把url捕获和额外参数写成一致
* 注意：根目录下传递参数会给下一级别每一个都加上参数
* **-url，命名**
* url命名可以修改访问路径的时候依然访问到同一个地方 reverse 反向解析
* 多个应用怕重名需要定义一个app名称 在urlconf下 app\_name = ''
* -页面重定向redirect
* 页面重定向，跳转页面， 登陆之后，某个操作之后 302 redirect
* 301 redirect: 301 代表永久性转移(Permanently Moved)
* 302 redirect: 302 代表暂时性转移(Temporarily Moved )
* ~~虚拟机和物理机外还有一层网络隔离保证不影响，需要通过端口映射来访问，映射到同一个网络层访问~~

## 实际例子：

* 项目目录文件夹下
* setting中
* 
* urls中指向应用urls
* 

**一、URL路由映射**

　　路由映射模块，主要完成url与views视图函数的映射。当一个url请求到来时，会按照这个模块中的url地址从上到下进行匹配，如果匹配成功，将执行映射试图中的函数；反之将返回404错误。

**1.基本介绍**

urlpatterns = [] #里面存放一条条的路由映射关系。

url(正则表达式, 映射函数，参数[可选]，别名[可选])

* 正则匹配中，如果带了括号，那么该括号中的内容会当作参数传递到对应的视图函数中去。
* 别名，在文件路径发生变化时用处非常大。

**2.配置路由：**　**常规正则匹配，路径命名，别名。**

### 3.路由应用分发

## 二、View视图

**视图函数，围绕着两个对象进行：HttpResponse和HttpRequest**

**1.HttpRequest**

* 属性

request.path # 获取访问文件路径

request.method属性 #获取请求中使用的HTTP方式（POST/GET）

request.GET # 获取HTTP GET方式请求传参（字典类型）

request.POST # 包含所有HTTP POST参数的类字典对象

request.COOKIES #包含所有cookies的标准Python字典对象；keys和values都是字符串。

request.FILES #包含所有上传文件的类字典对象

request.user #　是一个django.contrib.auth.models.User对象，代表当前登陆的用户

request.session　＃　唯一可读写的属性，代表当前会话的字典对象

**2.HttpResponse**

　　　对于HttpRequest请求对象来说，是由django自动创建的，但是，HttpResponse响应对象就必须我们自己创建。每个view请求处理方法必须返回一个HttpResponse响应对象。HttpResponse类在django.http.HttpResponse。

* **在HttpResponse对象上扩展的常用方法**

1. render(请求对象，'html文件和路径')方法，将指定页面渲染后返回给浏览器
2. render\_to\_response('html文件和路径')方法，将指定页面渲染后返回给浏览器
3. redirect('跳转路径和名称')方法，页面跳转
4. 注意：render与redirect两点区别：
5. 我们来模拟一个登陆网页，当我登陆成功后，跳转到另一个页面，分别用render与redirect来试试。
6. 总结两者区别：
7. 第一，render返回一个登陆成功后的页面，刷新该页面将回复到跳转前页面。而redirect则不会
8. 第二，render返回一个登陆成功页面，不会经过url路由分发系统，也就是说，不会执行跳转后url的试图函数。这样，返回的页面渲染不成功；而redirect是跳转到指定页面，当登陆成功后，会在url路由系统进行匹配，如果有存在的映射函数，就会执行对应的映射函数。

列表和元组有什么区别， 答：列表是可变类型，元组是不可变类型，但元组其实是可以改的，只不过默认的不去改

list和tuple都是一个可以放置任意数据类型的有序集合，都是既可以存放数字、字符串、对象等。

list和tuple的一些区别

* 列表是动态的，长度大小不固定，可以随意的增加、删除、修改元素
* 元组是静态的，长度在初始化的时候就已经确定不能更改，更无法增加、删除、修改元素

tuple 用于存储异构(heterogeneous)数据，当做没有字段名的记录来用；而列表一般用于存储同构数据(homogenous)，同构数据就是具有相同意义的数据。

列表和元组是Python中最常用的两种数据结构，字典是第三种。 相同点：

* 都是序列
* 都可以存储任何数据类型
* 可以通过索引访问

django是怎么实现异步任务的， 答：用celery+redis提交异步任务，然后他又问了一次是怎么实现的，我说因为他单独开了一个socket。

[Django设置异步任务](https://www.cnblogs.com/wuyan717/p/9317041.html)

1、安装Django-celery 包：pip install django-celery==3.2.2

2、开启redis服务

　　需要使用redis做broker，所以在使用异步和定时任务时需要开启redis服务器

3、配置信息

用redis+celery的时候，redis有什么异常吗。答：没遇到什么异常。

celery和redis常出现的错误是版本问题

使用 celery -A celery\_tasks.tasks worker -l info 后 处理的时候 出现以下错误

return iter(x.items())

AttributeError: 'str' object has no attribute 'items'

这是版本的问题！

安装pip install redis==2.10.6

django 1.8只能用2.10 不能用最新的版本，而此时用的是django1.8的

[浅谈工作中celery与Redis遇到的一些问题](https://www.cnblogs.com/Rglin/p/10786317.html)

celery的内存泄漏？

　　总结：   celery执行完任务不释放内存与原worker一直没有被销毁有关，因此CELERYD\_MAX\_TASKS\_PER\_CHILD可以适当配置小点，而任务并发数与CELERYD\_CONCURRENCY配置项有关，

　　每增加一个worker必然增加内存消耗，同时也影响到一个worker何时被销毁，因为celery是均匀调度任务至每个worker，因此也不宜配置过大，适当配置。

　　　　 CELERYD\_MAX\_TASKS\_PER\_CHILD

 　　　　 CELERYD\_CONCURRENCY = 20  # 并发worker数

  　　　　CELERYD\_FORCE\_EXECV = True    # 非常重要,有些情况下可以防止死锁

　　　　  CELERYD\_MAX\_TASKS\_PER\_CHILD = 100    # 每个worker最多执行万100个任务就会被销毁，可防止内存泄露

　　　　  CELERYD\_TASK\_TIME\_LIMIT = 60    # 单个任务的运行时间不超过此值，否则会被SIGKILL 信号杀死

  　　任务发出后，经过一段时间还未收到acknowledge , 就将任务重新交给其他worker执行

　　  　　CELERY\_DISABLE\_RATE\_LIMITS = True

 在Django中使用celery内存泄漏问题？

　　  在django下使用celery作为异步任务系统，十分方便。

  　　同时celery也提供定时任务机制，celery beat。使用celery beat 可以为我们提供 cron，schedule 形式的定时任务。

  　　在django下使用celery beat的过程中，发现了 celery beat进程 占用内存非常大，而且一直不释放。

  　　怀疑其有内存占用不释放的可能。

  　　因为之前使用django的时候，就知道在django中开启DEBUG模式，会为每次的SQL查询 缓存结果。   celery beat 作为 定时任务的timer和heartbeat程序，是长期运行的，而我使用了MYSQL作为存储定时任务的backend。

　　  因为每次heartbeat和timer产生的sql查询在开启了DEBUG模式下的django环境中，都会缓存查询结果集。因此 celery beat占用的 内存会一直不释放。在我的线上环境中 达到10G内存占用！

     　 解决： 关掉django的DEBUG模式，在setting中，设置DEBUG=False 即可。   关闭DEBUG模式后的celery beat程序 的内存占用大概 一直维持在150M左右。

数据库连接是单利吗？ 有必要实现多例吗？

 　   单例数据库连接在连接池中只有一个实例，对系统资源的开销比较少，甚至可以长时间保持连接不回收，以节省创建连接和回收连接的时间。

　　 但这样的连接在多用户并发时不能提供足够的效率，形象的来讲就是大家要排队。  初级程序员的作法是每个用户需求过来都打开一次连接，用完回收掉。

　 　进阶的做法是建立一个连接池，连接池里面给定一些已打开的连接，用程序控制这些连接的分配与调度

　     数据库链接用单例模式的原因：

        单例只保留一个对象，可以减少系统资源开销。

 　　提高创建速度，每次都获取已经存在的对象因此提高创建速度全局共享对象。

 　　单例在系统中只存在一个对象实例，因此任何地方使用此对象都是一个对象避免多实例创建使用时产生的逻辑错误。

　　 例模式是一种常用的软件设计模式，它的核心结构只包含一个被称为单例的特殊类。它的目的是保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

　　 单例模式有3种实现方式：懒汉式、饿汉式和双重锁的形式。

　　单例模式优点 只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等  因为类控制了实例化过程，所以类可以灵活更改实例化过程

　　单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。  单例的缺点  就是不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，

　　单例就会引起数据的错误，不能保存彼此的状态。 用单例模式，就是在适用其优点的状态下使用。

　　　　比如：   要进一个房间（数据库），就为这个房间开了一扇门（数据库类），一般情况下是一个人开一扇门，   不管你进出（数据库操作）这个房间多少次，门就这一扇（单例），

　　　　当然一个人也可以开很多扇门（非单例），   但你知道一个房间能开的门的数量是有限的，因此你不使用单例的话，一是性能慢一些，二是走别人的门，让别人无门可进。。

数据类型的堆栈存储？

　　　堆栈是一个后进先出的数据结构，其工作方式就像一堆汽车排队进去一个死胡同里面，最先进去的一定是最后出来。

　　　队列是一种先进先出的数据类型，它的跟踪原理类似于在超市收银处排队，队列里的的第一个人首先接受服务，新的元素通过入队的方式添加到队列的末尾，而出队就是将队列的头元素删除。

　　　栈：是一种容器，可存入数据元素、访问元素、删除元素

　　　特点：只能从顶部插入（入栈）数据和删除（出栈）数据

　　　原理：LIFO(Last In First Out)后进先出  栈可以使用顺序表实现也可使用链表实现 使用python列表实现代码：  class Stack(object):   """   栈   使用python列表实现   """

  　　　　def \_\_init\_\_(self):    self.items = list()

 　　　　 def is\_empty(self):    """判空"""    return self.items == []

  　　　　def size(self):    """获取栈元素个数"""    return len(self.items)

  　　　　def push(self, item):    """入栈"""    self.items.append(item)

  　　　　def pop(self):    """出栈"""    self.items.pop()

 　　 　　def peek(self):    """获取栈顶元素"""    if self.is\_empty():     raise IndexError("stack is empty")    return self.items[-1]

flask的jwt?

　　一篇文章需求分析：(Flask + flask-jwt 实现基于Json Web Token的用户认证授权)

 　　jwt是flask的一个第三方库：flask-jwt-------->可以实现基于Json Web Token的用户认证授权

　　 使用 JWT 让你的 RESTful API 更安全  什么是 JWT ？

　　　　JWT 及时 JSON Web Token，它是基于 RFC 7519 所定义的一种在各个系统中传递紧凑和自包含的 JSON 数据形式。

　　　　紧凑（Compact） ：由于传送的数据小，JWT 可以通过GET、POST 和 放在 HTTP 的 header 中，同时也是因为小也能传送的更快。

　　　　自包含（self-contained） : Payload 中能够包含用户的信息，避免数据库的查询。  JSON Web Token 由三部分组成使用 .

　　　　分割开：   Header   Payload   Signature     一个 JWT 形式上类似于下面的样子：   xxxxx.yyyy.zzzz

　　　　Header 一般由两个部分组成：   alg   typ     alg 是是所使用的 hash 算法例如 HMAC SHA256 或 RSA，typ 是 Token 的类型自然就是 JWT。   {     "alg": "HS256",     "typ": "JWT"   }

　　JSON Web Token 的工作流程：

 　　　　在用户使用证书或者账号密码登入的时候一个 JSON Web Token 将会返回，同时可以把这个 JWT 存储在local storage、或者 cookie 中，用来替代传统的在服务器端创建一个 session 返回一个 cookie。

 　　　　当用户想要使用受保护的路由时候，应该要在请求得时候带上 JWT ，一般的是在 header 的 Authorization 使用 Bearer 的形式，一个包含的 JWT 的请求头的 Authorization 如下：

 　　　　　   Authorization: Bearer <token>

　　　　　　 这是一中无状态的认证机制，用户的状态从来不会存在服务端，在访问受保护的路由时候回校验 HTTP header 中 Authorization 的 JWT，

　　　　　　同时 JWT 是会带上一些必要的信息，不需要多次的查询数据库。

　　　　　　这种无状态的操作可以充分的使用数据的 APIs，甚至是在下游服务上使用，这些 APIs 和哪服务器没有关系，

　　　　　　因此，由于没有 cookie 的存在，所以在不存在跨域（CORS, Cross-Origin Resource Sharing）的问题。

gil锁的局限性和打破方式？

　　　　局限性:

　　　　　　在Cpython解释器中，同一个进程下开启的多线程，同一时刻只能有一个线程执行，无法利用多核优势

　　　　　　GIL存在原因   CPython在执行多线程的时候并不是线程安全的，所以为了程序的稳定性，加一把全局解释锁，能够确保任何时候都只有一个Python线程执行。

　　　　　　GIL的弊端

  　　　　　  　　GIL对计算密集型的程序会产生影响。因为计算密集型的程序，需要占用系统资源。GIL的存在，相当于始终在进行单线程运算，这样自然就慢了。

　　　　　　　　IO密集型影响不大的原因在于，IO，input/output，这两个词就表明程序的瓶颈在于输入所耗费的时间，线程大部分时间在等待，所以它们是多个一起等（多线程）还是单个等（单线程）无所谓的。

　　　　　　　　 这就好比，你在公交站等公交时，你们排队等公交（单线程）还是沿着马路一字排开等（多线程）是无所谓的。公交车（即input，即输入的资源）没来，哪种方式都是瞎折腾。

　　　　解决方案(打破方式)

　　　　　　multiprocessing

　　　　　　multiprocessing是一个多进程模块，开多个进程，每个进程都带一个GIL，就相当于多线程来用了。

　　　　　　multiprocessing的弊端   多线程与多进程一个不同点在于：

  　　　　　　　  多线程是共享内存的，即这些线程共用一个内存地址。好处在于便于线程间数据通信和数据同步。

　　　　　　　　多进程，各个进程地址之间是独立的内存地址。这样不存内存地址之间通信就麻烦了。

　　　　　　　　综上所述，如果是IO密集型且对数据通信有需求，使用python 的threading模块也是可以的。

　　　　解决方法：   1. 使用多进程执行，此将要面临解决共享数据的问题，多用queue或pipe解决；   2. 使用Python多线程load C的module执行。

 　　　　　　　　 from ctypes import \*   form threading import Thread

  　　　　　　　　#加载动态库   lib = cdll.LoadLibrary("./libdeadloop.so")

 　　　　　　　　 #创建一个子线程，让其执行c语言编写的函数，此函数是一个死循环   t = Thread(target=lib.DeadLoop)   t.start()

  　　　　　　　　while True:    pass

 网友博客理解：

　　IL是限制同一个进程中只有一个线程进入Python解释器。。。。。

　　而线程锁是由于在线程进行数据操作时保证数据操作的安全性(同一个进程中线程之间可以共用信息，如果同时对数据进行操作，则会出现公共数据错误)

　　其实线程锁完全可以替代GIL，但是Python的后续功能模块都是加在GIL基础上的，所以无法更改或去掉GIL,这就是Python语言最大的bug…只能用多进程或协程改善，或者直接用其他语言写这部分

　　追问

　　GIL本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。

　　保护不同的数据的安全，就应该加不同的锁。

　　每执行一个python程序，就是开启一个进程，在一个python的进程内，不仅有其主线程或者由该主线程开启的其他线程，还有解释器开启的垃圾回收等解释器级别的线程，   所有的线程都运行在这一个进程内，

　　　　所以：

　　　　1、所有数据都是共享的，这其中，代码作为一种数据也是被所有线程共享的（test.py的所有代码以及Cpython解释器的所有代码）

　　　　2、所有线程的任务，都需要将任务的代码当做参数传给解释器的代码去执行，即所有的线程要想运行自己的任务，首先需要解决的是能够访问到解释器的代码。

　　在python的原始解释器CPython中存在着GIL（Global Interpreter Lock，全局解释器锁），因此在解释执行python代码时，会产生互斥锁来限制线程对共享资源的访问，

　　直到解释器遇到I/O操作或者操作次数达到一定数目时才会释放GIL。   所以，虽然CPython的线程库直接封装了系统的原生线程，但CPython整体作为一个进程，同一时间只会有一个获得GIL的线程在跑，

　　其他线程则处于等待状态。   这就造成了即使在多核CPU中，多线程也只是做着分时切换而已。   不过muiltprocessing的出现，已经可以让多进程的python代码编写简化到了类似多线程的程度了

我对 GIL的理解：

　　解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁。

　　GIL锁开始运作主线程做操作主线程完成操作GIL锁释放资源   所以多线程共同操作共享资源的时候，有一个线程竞得了资源，它就被GIL锁保护起来，其他线程只能是在那里等着，

　　但是这个时候，线程的休眠唤醒，全部会消耗CPU资源，所以嘞，就会慢。   Python语言和GIL解释器锁没有关系，它是在实现Python解析器(CPython)时所引入的一个概念，

　　同样一段代码可以通过CPython，PyPy，Psyco等不同的Python执行环境来执行，然而因为CPython是大部分环境下默认的Python执行环境。所以在很多人的概念里CPython就是Python，

　　也就想当然的把GIL归结为Python语言的缺陷，所有GIL并不是python的特性，仅仅是因为历史原因在Cpython解释器中难以移除。

  　  GIL保证同一时刻只有一个线程执行代码，每个线程在执行过程中都要先获取GIL

  　 线程释放GIL锁的情况：

　  　　在IO操作等可能会引起阻塞的system call之前,可以暂时释放GIL,但在执行完毕后,必须重新获取GIL Python 3.x使用计时器（执行时间达到阈值后，当前线程释放GIL）或Python 2.x，tickets计数达到100

  　　　Python使用多进程是可以利用多核的CPU资源的。   多线程爬取比单线程性能有提升，因为遇到IO阻塞会自动释放GIL锁

　　　  GIL只对计算密集型的程序有作用，对IO密集型的程序并没有影响，因为遇到IO阻塞会自动释放GIL锁   当需要执行计算密集型的程序时，

　　　  可以选择：1.换解释器，2.扩展C语言，3.换多进程等方案

　　　 GIL（Global Interpreter Lock）：全局解释器锁，python解释器在执行python字节码的时候会锁住解释器，导致其它的线程不能使用解释器，从而多线程情况下CPU上不去。

　　　 lupa是一个python调用lua的第三方库（https://pypi.python.org/pypi/lupa），lua\_code是一段纯CPU计算的lua代码片段，之后开启了3个线程，可以发现CPU利用率达到了300%  之前在写一些python程序的时候，

　　　 如果是cpu密集的常常会使用多进程的方式，但是这样会有一些缺点：

　　　　　　1. 进程间共享数据特别麻烦，虽然multiprocessing库提供了很多进程间共享数据的方法，但是这些方法最后自己会成为瓶颈

 　　　　　   2. 编程复杂度比较高

  　　　　　  3. 主进程和子进程必然需要通信，进程间数据隔离，所以数据需要内存拷贝，成本高  相应的python代码：

 　　　　　　　　 #python lupa load

  　　　　　　　　import lupa   lua = lupa.LuaRuntime()

 　　　　　　　　 LIBS = [    "./scripts/foo.lua",   ]

　　　　　　　　  llibs = {}

 　　　　　　　　def get\_file\_name(filename):

　　　　　　　　　　import os

　　　　　　　　　　(\_, tmp) = os.path.split(filename)

　　　　　　　　　　(f\_name, ext) = os.path.splittext(tmp)

　　　　　　　　　　return f\_name

　　　　　　　　 def load\_libs():

　　　　　　　　　　global LIBS, llibs    for lib\_p in LIBS:     f = open(lib\_p, 'r')

　　　　　　　　　　code\_str = f.readlines()     filename = get\_file\_name(lib\_p)

　　　　　　　　　　llibs[filename] = lua.execute('\n'.join(code\_str))

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

　　　　　　　　　　 load\_libs()

　　　　　　　　　　print llibs['foo'].sayhi()

　　　　　　　　　　print llibs['foo'].callback(100, 200, 300, 400)

　　　　　　　　　　--foo.lualibfoo = {}

  　　　　　　　　　  function libfoo.sayhi()

　　　　　　　　　　return "hi from lupa"   end  function libfoo.callback(a, b, c, d)    return a \* b + c - d   endreturn libfoo

　　这样做有几点好处：

 　　　　1. python写框架，lua写回调，每次调用一遍load\_libs就相当于一次热更新

 　　　　2. lua代码本身特别简单，可以交给策划配置，与热更新结合效果更好

 　　　　3. python多线程结合lua使用可以突破python GIL的限制，后面补充一个demo

　　　　　　　　线程安全：   线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。   不会出现数据不一致或者数据污染。

　　　　　　　　线程不安全：就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

celery在某一时刻突然执行2回? 为什么？ 否 怎么解决？

　　　　Celery是一个用Python开发的异步的分布式任务调度模块

　　网友遇到的工作问题以及解决方案：

　　　　使用 Celery Once 来防止 Celery 重复执行同一个任务   在使用 Celery 的时候发现有的时候 Celery 会将同一个任务执行两遍，我遇到的情况是相同的任务在不同的 worker 中被分别执行，并且时间只相差几毫秒。

　　　　这问题我一直以为是自己哪里处理的逻辑有问题，后来发现其他人 也有类似的问题，然后基本上出问题的都是使用 Redis 作为 Broker 的，而我这边一方面不想将 Redis 替换掉，就只能在 task 执行的时候加分布式锁了。

  　　　 不过在 Celery 的 issue 中搜索了一下，有人使用 Redis 实现了分布式锁，然后也有人使用了 Celery Once。 大致看了一下 Celery Once ，发现非常符合现在的情况，就用了下。

　　　　Celery Once 也是利用 Redis 加锁来实现, Celery Once 在 Task 类基础上实现了 QueueOnce 类，该类提供了任务去重的功能，

　　　　所以在使用时，我们自己实现的方法需要将 QueueOnce 设置为 base   @task(base=QueueOnce, once={'graceful': True})

　　　　后面的 once 参数表示，在遇到重复方法时的处理方式，默认 graceful 为 False，那样 Celery 会抛出 AlreadyQueued 异常，手动设置为 True，则静默处理。

  　　　  另外如果要手动设置任务的 key，可以指定 keys 参数

    　　　@celery.task(base=QueueOnce, once={'keys': ['a']})   def slow\_add(a, b):    sleep(30)    return a + b

celery 任务突然不执行是为什么？

　　问题：

　　　　别人推送很多消息给我，用 tornado 接收然后传到 celery 里面处理   celery 进程刚启动还是没问题，运行一天半天 突然里面的任务都不处理了   重新启动下 就能把之前接收到的推送 一个个继续处理。。。

　　　　看不出是什么问题。。   打算修改下配置的处理任务的超时时间，看看能不能解决这个问题。

　　目前解决方案：

　　　　在服务器加一个定时刷新，开启时间长，处理大量消息，出现任务超时，端口占用，任务没有放掉，导致后期任务无法执行，需要服务器定时任务重启

　　问题：

　　　　最近在写一个分布式微博爬虫，主要就是使用celery做的分布式任务调度。celery确实比较好用，但是也遇到一些问题，我遇到的问题主要集中在定时任务和任务路由这两个部分。

　　　　本文不会讲解celery的基本使用，如果需要看celery入门教程的话，请点击这里跳转。   celery worker -A app\_name -l info必须推荐在项目的根目录运行而且,这里的app\_name必须是项目中的Celery实例的完整引用路径\*。

　　　　如果不在项目根目录运行，那么相关的调用也得切换到app同级目录下，这一点可以通过命令行进行佐证

　　　　celery的定时任务会有一定时间的延迟。比如，我规定模拟登陆新浪微博任务每隔10个小时执行一次，那么定时任务第一次执行就会在开启定时任务之后的10个小时后才会执行。

　　　　而我抓取微博需要马上执行，需要带上cookie，所以不能等那1个小时。这个没有一个比较好的解决方法，可以使用celery的crontab()来代替schdule做定时，它会在启动的时候就执行。

　　解决方案

　　　　我采用的方法是第一次手动执行该任务，然后再通过schedule执行。      celery的定时任务可能会让任务重复。定时器一定只能在一个节点启动，否则会造成任务重复。

　　　　另外，如果当前worker节点都停止了，而beat在之后才停止，那么下一次启动worker的时候，它还会执行上一次未完成的任务，可能会有重复。

　　　　由于抓取用户和抓取用户关注、粉丝的任务耗时和工作量不同，所以需要使用任务路由，将任务按比重合理分配到各个分布式节点上，这就需要使用到celery提供的task queue。

　　　　如果单独使用task queue还好，但是和定时任务一起使用，就可能出现问题。我遇到的问题就是定时任务压根就不执行！开始我的配置大概就是这样

　　　　　　app.conf.update(

　　　　　　　　CELERY\_TIMEZONE='Asia/Shanghai',

　　　　　　　　CELERY\_ENABLE\_UTC=True,

　　　　　　　　CELERY\_ACCEPT\_CONTENT=['json'],

　　　　　　　　CELERY\_TASK\_SERIALIZER='json',

　　　　　　　　CELERY\_RESULT\_SERIALIZER='json',

　　　　　　　　CELERYBEAT\_SCHEDULE={

　　　　　　　　　　 　　'user\_task': {       'task': 'tasks.user.excute\_user\_task',       'schedule': timedelta(minutes=3),      },

　　　　　　　　　　　　 'login\_task': {       'task': 'tasks.login.excute\_login\_task',       'schedule': timedelta(hours=10),      },

　　　　　　　　　　　　 },

　　　　　　　　CELERY\_QUEUES=(

　　　　　　　　　　　　 Queue('login\_queue', exchange=Exchange('login', type='direct'), routing\_key='for\_login'),

　　　　　　　　　　　　 Queue('user\_crawler', exchange=Exchange('user\_info', type='direct'), routing\_key='for\_user\_info'),

　　　　　　　　　　　　 Queue('fans\_followers', exchange=Exchange('fans\_followers', type='direct'), routing\_key='for\_fans\_followers')     )    )

　　　　结果过了一天发现定时任务并没有执行，后来把task加上了一个option字段，指定了任务队列，就可以了，

　　　　比如     'user\_task': {      'task': 'tasks.user.excute\_user\_task',      'schedule': timedelta(minutes=3),      'options': {'queue': 'fans\_followers', 'routing\_key': 'for\_fans\_follwers'}     },

 　　　  部分分布式节点一直出现Received task，但是却不执行其中的任务的情况。这种情况下重启worker节点一般就可以恢复。

　　　　但是最好查查原因。通过查看flower的失败任务信息，才发现是插入数据的时候有的异常未被处理。这一点严格说来并不是celery的bug，不过也很令人费解。

　　　　所以推荐在使用celery的时候配合使用flower做监控。

Redis是否保证事务的一致性？

　　　　网友遇到问题：

　　　　　　”redis设计之初是简单高效，所以说在事务操作时命令是不会出错的，出错的可能性就是程序的问题“   那这样的意思就是把锅抛给程序咯？

　　　　　　如果程序能保证百分之百不出错那么关系型数据库还要啥事务呢？

　　　　　　redis事务报错时仍会执行所有命令，这样怎么保证一致性呢？

　　　　　　或者说白了redis根本就不支持事务只是冠以事务的名号而已。以上纯属个人见解   又专业人士可以解释大家讨论。

　　　　网友理解：

　　　　　　1、Redis事务主要用于不间断执行多条命令，即是存在引发错误的命令。Redis先执行命令，命令执行成功后才会记录日志，所以出现错误时无法回滚。

　　　　　　　　支持完整的acid会让Redis变得复杂也可能导致性能较低。 此外，使用lua脚本也可以保证Redis不间断执行多条命令。

　　　　　　2、找到网上这个解释比较到位：      单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。

　　　　　　　　事务可以理解为一个打包的批量执行脚本，但批量指令并非原子化的操作，中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。

 redis事务

  　　　　1、基本概念

  　　　　　　1）什么是redis的事务？

 　　　　　　　　 简单理解，可以认为redis事务是一些列redis命令的集合，并且有如下两个特点：

 　　　　　　　　 a）事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

 　　　　　　　　 b）事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

 　　　　　　 2）事务的性质ACID

  　　　　　　　　一般来说，事务有四个性质称为ACID，分别是原子性，一致性，隔离性和持久性。

 　　　　　　　　 a）原子性atomicity：redis事务保证事务中的命令要么全部执行要不全部不执行。有些文章认为redis事务对于执行错误不回滚违背了原子性，是偏颇的。

 　　　　　　　　 b）一致性consistency：redis事务可以保证命令失败的情况下得以回滚，数据能恢复到没有执行之前的样子，是保证一致性的，除非redis进程意外终结。

  　　　　　　　　c）隔离性Isolation：redis事务是严格遵守隔离性的，原因是redis是单进程单线程模式，可以保证命令执行过程中不会被其他客户端命令打断。

 　　　　　　　　 d）持久性Durability：redis事务是不保证持久性的，这是因为redis持久化策略中不管是RDB还是AOF都是异步执行的，不保证持久性是出于对性能的考虑。

　　　　　　  3）redis事务的错误

  　　　　　　　　使用事务时可能会遇上以下两种错误：

  　　　　　　　　　　a）入队错误：事务在执行 EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），

　　　　　　　　　　　　　　或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）。

  　　　　　　　　　　b）执行错误：命令可能在 EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。

 　　　　　　　　　　 注：第三种错误，redis进程终结，本文并没有讨论这种错误。

 　　　　2、redis事务的用法

  　　　　　　　redis事务是通过MULTI，EXEC，DISCARD和WATCH四个原语实现的。

  　　　　　　　MULTI命令用于开启一个事务，它总是返回OK。

  　　　　　　　MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

  　　　　　　　另一方面，通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务。

Redis的持久化方式？

　　　　两种持久化方式：

　　　　　　RDB和AOF  深入了解RDB和AOF的作用原理，剩下的就是根据实际情况来制定合适的策略了，再复杂一点，也就是定制一个高可用的，数据安全的策略了。

　　　　　　在RDB方式下，你有两种选择，一种是手动执行持久化数据命令来让redis进行一次数据快照，另一种则是根据你所配置的配置文件 的 策略，达到策略的某些条件时来自动持久化数据。

　　　　　　而手动执行持久化命令，你依然有两种选择，那就是save命令和bgsave命令。   save操作在Redis主线程中工作，因此会阻塞其他请求操作，应该避免使用。

　　　　　　bgSave则是调用Fork,产生子进程，父进程继续处理请求。子进程将数据写入临时文件，并在写完后，替换原有的.rdb文件。

　　　　　　Fork发生时，父子进程内存共享，所以为了不影响子进程做数据快照，在这期间修改的数据，将会被复制一份，而不进共享内存。

　　　　　　所以说，RDB所持久化的数据，是Fork发生时的数据。在这样的条件下进行持久化数据，如果因为某些情况宕机，则会丢失一段时间的数据。

　　　　　　如果你的实际情况对数据丢失没那么敏感，丢失的也可以从传统数据库中获取或者说丢失部分也无所谓，那么你可以选择RDB持久化方式。

　　　　　　再谈一下配置文件的策略，实际上它和bgsave命令持久化原理是相同的。

　　　　　　AOF持久化方式：

　　　　　　配置文件中的appendonly修改为yes。开启AOF持久化后，你所执行的每一条指令，都会被记录到appendonly.aof文件中。

　　　　　　但事实上，并不会立即将命令写入到硬盘文件中，而是写入到硬盘缓存，在接下来的策略中，配置多久来从硬盘缓存写入到硬盘文件。

　　　　　　所以在一定程度一定条件下，还是会有数据丢失，不过你可以大大减少数据损失。  这里是配置AOF持久化的策略。

　　　　　　redis默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

　　　　　　而no则是不主动进行同步操作，是默认30s一次。当然always一定是效率最低的，个人认为everysec就够用了，数据安全性能又高。

　　　Redis也允许我们同时使用两种方式，再重启redis后会从aof中恢复数据，因为aof比rdb数据损失小嘛。

　　　　　　深入理解Redis的两种持久化方式:  RDB每次进行快照方式会重新记录整个数据集的所有信息。RDB在恢复数据时更快，可以最大化redis性能，子进程对父进程无任何性能影响。

　　　　　　AOF有序的记录了redis的命令操作。意外情况下数据丢失甚少。他不断地对aof文件添加操作日志记录，你可能会说，这样的文件得多么庞大呀。

　　　　　　是的，的确会变得庞大，但redis会有优化的策略，比如你对一个key1键的操作，set key1 001 ,  set key1 002, set key1 003。那优化的结果就是将前两条去掉咯，

　　　　　　那具体优化的配置在配置文件中对应的是  https://images2015.cnblogs.com/blog/686162/201608/686162-20160809211516715-145676984.png

　　　　　　前者是指超过上一次aof重写aof文件大小的百分之多少，会再次优化，如果没有重写过，则以启动时为主。

　　　　　　后者是限制了允许重写的最小aof文件大小。bgrewriteaof命令是手动重写命令，会fork子进程，在临时文件中重建数据库状态，对原aof无任何影响，

　　　　　　当重建旧的状态后，也会把fork发生后的一段时间内的数据一并追加到临时文件，最后替换原有aof文件，新的命令继续向新的aof文件中追加。

　　Redis数据库简介：

　　　　　　Redis是一种高级key-value数据库。它跟memcached类似，不过数据可以持久化，而且支持的数据类型很丰富。

　　　　　　有字符串，链表，集 合和有序集合。支持在服务器端计算集合的并，交和补集(difference)等，还支持多种排序功能。

　　　　　　所以Redis也可以被看成是一个数据结构服务 器。   Redis的所有数据都是保存在内存中，然后不定期的通过异步方式保存到磁盘上(这称为“半持久化模式”)；

　　　　　　也可以把每一次数据变化都写入到一个append only file(aof)里面(这称为“全持久化模式”)。

　　Redis数据库简介：

　　　　　　默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

　　　　　　默认使用everysec，就是说每秒持久化一次，而always则是每次操作都会立即写入aof文件中。

　　　　　　由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，

　　　　　　当redis重启后，可以从磁盘中恢复数据。redis提供两种方式进行持久化，一种是RDB持久化（原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化），

　　　　　　另外一种是AOF（append only file）持久化（原理是将Reids的操作日志以追加的方式写入文件）。

　　　　　　那么这两种持久化方式有什么区别呢，改如何选择呢？网上看了大多数都是介绍这两种方式怎么配置，怎么使用，就是没有介绍二者的区别，在什么应用场景下使用。

二者的区别   RDB存在哪些优势呢？

 　　　　　　1). 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这对于文件备份而言是非常完美的。

　　　　　　　　比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。

　　　　　　　　通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常容易的进行恢复。

 　　　　　　2). 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

 　　　　　　3). 性能最大化。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

 　　　　　　4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

 　　　　RDB又存在哪些劣势呢？

 　　　　　　1). 如果你想保证数据的高可用性，即最大限度的避免数据丢失，那么RDB将不是一个很好的选择。因为系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

 　　　　　　2). 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

 　　　　AOF的优势有哪些呢？

 　　　　　　1). 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。

　　　　　　　　事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。

　　　　　　　　而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。

 　　　　　　2). 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。

　　　　　　　　然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。

 　　　　　　3). 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。

　　　　　　　　因此在进行rewrite切换时可以更好的保证数据安全性。

　　　　　　 4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

 　　　　AOF的劣势有哪些呢？

 　　　　　　1). 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

 　　　　　　2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

 　　　　　　二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。

　　　　 　　 rdb这个就更有些 eventually consistent的意思了。

常用配置

 　　　　RDB持久化配置

 　　　　Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

 　　　　　　save 900 1              #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

 　　　　　　save 300 10            #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

 　　　　　　save 60 10000        #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

 　　　　AOF持久化配置

 　　　　在Redis的配置文件中存在三种同步方式，它们分别是：

 　　　　　　appendfsync always     #每次有数据修改发生时都会写入AOF文件。

 　　　　　　appendfsync everysec  #每秒钟同步一次，该策略为AOF的缺省策略。

 　　　　　　appendfsync no          #从不同步。高效但是数据不会被持久化。

同理celery也是只能用3.1.25 不能用4版本以上的，django1.11才可以用4以上的

数据库事务的了解 答：事务特性，举例银行

深入理解数据库事务

#数据库事务（简称‘事务’）

数据库事务是指作为单个逻辑工作单元执行的**一系列数据库操作**

Transaction作为关系型数据库的核心组成，在数据安全方面有着非常重要的作用，本文会一步步解析事务的核心特性，以获得对事务更深的理解。

什么是事务

数据库几乎是所有系统的核心模块，它将数据有条理地保存在储存介质(磁盘）中，

并在逻辑上，将数据以结构化的形态呈现给用户。支持数据的增、删、改、查，并在过程中保障数据的正确且可靠。

要做到这点并非易事，常见的例子就是银行转账，A账户给B账户转账一个亿(T1)，买一块地盖房子。在这种交易的过程中，有几个问题值得思考：

* 如何同时保证上述交易中，A账户总金额减少一个亿，B账户总金额增加一个亿？ A
* A账户如果同时在和C账户交易(T2)，如何让这两笔交易互不影响？ I
* 如果交易完成时数据库突然崩溃，如何保证交易数据成功保存在数据库中？ D
* 如何在支持大量交易的同时，保证数据的合法性(没有钱凭空产生或消失) ？ C

要保证交易正常可靠地进行，数据库就得解决上面的四个问题，这也就是事务诞生的背景，它能解决上面的四个问题，对应地，它拥有四大特性：

* 原子性（Atomicity）: 事务要么全部完成，要么全部取消。 如果事务崩溃，状态回到事务之前（事务回滚）。
* 隔离性（Isolation）: 如果2个事务 T1 和 T2 同时运行，事务 T1 和 T2 最终的结果是相同的，不管 T1和T2谁先结束。
* 持久性（Durability）: 一旦事务提交，不管发生什么（崩溃或者出错），数据要保存在数据库中。
* 一致性（Consistency）: 只有合法的数据（依照关系约束和函数约束）才能写入数据库。

ACID

接下来详细地了解这四大特性：

* 原子性，确保不管交易过程中发生了什么意外状况（服务器崩溃、网络中断等），不能出现A账户少了一个亿，但B账户没到帐，或者A账户没变，但B账户却凭空收到一个亿（数据不一致）。A和B账户的金额变动要么同时成功，要么同时失败(保持原状)。
* 隔离性，如果A在转账1亿给B（T1），同时C又在转账3亿给A（T2），不管T1和T2谁先执行完毕，最终结果必须是A账户增加2亿，而不是3亿，B增加1亿，C减少3亿。
* 持久性，确保如果 T1 刚刚提交，数据库就发生崩溃，T1执行的结果依然会保持在数据库中。
* 一致性，确保钱不会在系统内凭空产生或消失， 依赖原子性和隔离性。

可以看出，原子性、隔离性、一致性的根本问题，是不同的事务同时对同一份数据(A账户)进行写操作(修改、删除、新增)，如果事务中都只是读数据的话，那么它们可以随意地同时进行，反正读到的数据都是一样的。

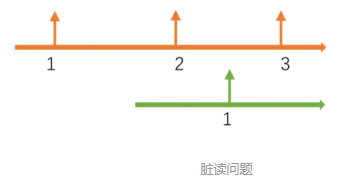
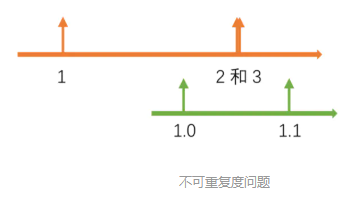
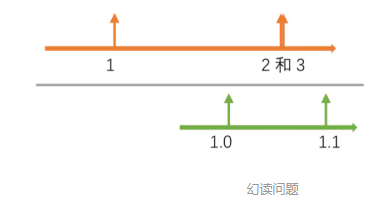
如果，几个互不知晓的事务在同时修改同一份数据，那么很容易出现后完成的事务覆盖了前面的事务的结果，导致不一致。 事务在最终提交之前都有可能会回滚，撤销所有修改：

1. 如果T1事务修改了A账户的数据，
2. 这时T2事务读到了更新后的A账户数据，并进行下一步操作，
3. 但此时T1事务却回滚了，撤销了对A账户的修改，
4. 那么T2读取到的A账户数据就是非法的，这会导致数据不一致。

这些问题都是事务需要避免的。

所以，在事务中更新某条数据获得的互斥锁，只有在事务提交或失败之后才会释放，在此之前，其他事务是只能读，不能写这条数据的。

这就是隔离性的关键，针对隔离性的强度，有以下四的级别([引用自这篇文章](https://link.zhihu.com/?target=http%3A//blog.jobbole.com/100349/))：

* 读取未提交（Read uncommitted）：最低级别的隔离，是读取已提交+新的隔离突破。如果事务A读取了数据D，然后数据D被事务B修改（但并未提交，事务B仍在运行中），事务A再次读取数据D时，数据修改是可见的。如果事务B回滚，那么事务A第二次读取的数据D是无意义的，因为那是事务B所做的从未发生的修改（已经回滚了嘛）。这叫脏读（dirty read）。Read uncommit：（读未提交）有脏读问题
* 读取已提交（Read committed，Oracle、PostgreSQL、SQL Server默认模式）：可重复读+新的隔离突破。如果事务A读取了数据D，然后数据D被事务B修改（或删除）并提交，事务A再次读取数据D时数据的变化（或删除）是可见的。这叫不可重复读（non-repeatable read）。Read commit：（读已提交）解决脏读问题，有不可重复读问题
* 可重复读（Repeatable read，MySQL默认模式）：如果一个事务成功执行并且添加了新数据(事务提交)，这些数据对其他正在执行的事务是可见的。但是如果事务成功修改了一条数据，修改结果对正在运行的事务不可见。所以，事务之间只是在新数据方面突破了隔离，对已存在的数据仍旧隔离。Repeatable read：（可重复读）解决不可重复读问题，有幻读问题
* 串行化(Serializable，SQLite默认模式）：最高级别的隔离。两个同时发生的事务100%隔离，每个事务有自己的"世界", 串行执行。串行化）解决幻读问题，所有问题都解决
* 由于事务可以分为三个阶段：读取、修改和提交，那么下面根据事务的三阶段，通过进行分析对应问题（脏读等问题）和采用对应的策略（行级锁等策略）来学习。如下图所示，其中我们后面就将读取阶段简称为1，修改阶段简称为2，提交节点简称为3。横轴表示时间线
* 
* 事务的三个阶段
* 下面我们用两个事务的时间线来讲解没有事务的一些隔离级别的时候存在的一些问题。
* 事务A：橘色
* 事务B：绿色
* 脏读
* 问题：
* 如果事务B的读取阶段位于事务A的修改和提交阶段（其实就是读取到了其他事务修改但是没有提交的数据），紧接着这个时候事务A发生回滚，那么事务B读取到的数据就是脏数据——脏读。
* 
* 脏读问题
* **解决：**
* 在事务修改和提交（也就是2和3阶段）之间增加写锁（就是排它锁，在读取期间不允许其他事务读取事务），这样就解决了脏读问题——其实就是Read commit（提交读）。
* **不可重复度**
* **问题：**
* 脏读问题解决了，但是在脏读问题之上又存在新的问题，比如在事务B第一次读取和第二次读取期间，事务A发生了修改和提交（这里我们站在解决脏读的问题上来看，事务A的2和3加了排它锁，我这里将两者合在一起），导致事务B第一次和第二次读取到的数据不一样（数据本身的值发生了变化）——不可重复读。
* 
* 不可重复度问题
* 解决：
* 1.通过添加行级共享锁
* 通过在事务B的读取到提交（就是阶段1~3，上图我们就只显示了读取阶段）期间加上读锁，在读锁这个共享锁加锁期间，事务A是不能添加排它锁的，也就是不能进行修改，进而每次事务B读取到的都是最新提交的数据。但是这种仍然存在幻读（下面介绍）。
* 2.通过用mvcc策略
* mvcc 可参考相关文档
* 幻读
* 问题：
* 不可重复读问题解决了，但是这里还有幻读问题，就是事务第一次读取和第二次读取，发现读取到的数据量不一样——幻读（这个跟不可重复读相似，但是这个强调的是数量的变化）。
* 下图我用灰色线表示两个事务处理的是不同的数据行（因为可重复读的隔离级别要求了一个事务读取到结束都加上了行级共享锁，其他事务是无法在中间修改）。
* 
* 幻读问题
* 当其中事务A在某个时间修改了其他行的数据，而此修改之前事务B读取了一次，但是这个数据修改后却满足了事务B的查询要求，就会引起事务B在第一次和第二次查询期间数据量不一样，就产生了幻读的问题。
* 解决：
* 在一个事务的执行中，添加表级锁，读添加表级共享锁，写添加表级排它锁，这样事务之间的执行就会完全无交叉。但是这种表级锁对应引起的性能会比较低，因为所有的事务就是串行化的执行（这也许就是叫串行化的原因）。下面我们会讲解下，一个新的解决方法：mvcc。就是在解决幻读的情况下，能让隔离级别保持在不可重复度的阶段。

如何保证持久性

隔离性的问题解决了，但是如果在事务提交后，事务的数据还没有真正落到磁盘上，此时数据库奔溃了，事务对应的数据会不会丢？

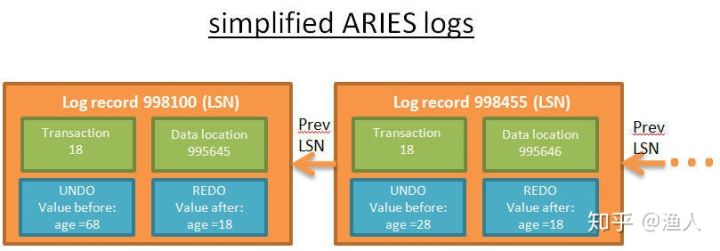
事务会保证数据不会丢，当数据库因不可抗拒的原因奔溃后重启，它会保证：

* 成功提交的事务，数据会保存到磁盘
* 未提交的事务，相应的数据会回滚

事务日志

数据库通过事务日志来达到这个目标。 事务的每一个操作（增/删/改）产生一条日志，内容组成大概如下：

* LSN：一个按时间顺序分配的唯一日志序列号，靠后的操作的LSN比靠前的大。
* TransID：产生操作的事务ID。
* PageID：被修改的数据在磁盘上的位置，数据以页为单位存储。
* PrevLSN：同一个事务产生的上一条日志记录的指针。
* UNDO：取消本次操作的方法，按照此方法回滚。
* REDO：重复本次操作的方法，如有必要，重复此方法保证操作成功。

来自jobbole.com

磁盘上每个页（保存数据的，不是保存日志的）都记录着最后一个修改该数据操作的LSN。数据库会通过解析事务日志，将修改真正落到磁盘上(写盘)，随后清理事务日志(正常情况下)。

这也是数据库在保证数据安全和性能这两个点之前的折中办法：

* 如果每次更新都写盘，由于数据是随机的，会造成大量的随机IO，性能会非常差
* 如果每次更新不马上写盘，那一旦数据库崩溃，数据就会丢失

折中的办法就是：

* 将数据的变更以事务日志的方式，按照时间先后追加到日志缓冲区，由特定算法写入事务日志，这是顺序IO，性能较好
* 通过数据管理器解析事务日志，由特定的算法择机进行写盘

数据库恢复

当数据库从崩溃中恢复时，会有以下几个步骤：

* 解析存在的事务日志，分析哪些事务需要回滚，哪些需要写盘(还没来得及写盘，数据库就崩溃了)。
* Redo，进行写盘。检测对应数据所在数据页的LSN，如果数据页的LSN>=事务操作的LSN，说明已经写过盘，不然进行写盘操作。
* Undo, 按照LSN倒序进行回滚

经过这几个阶段，在数据库恢复后，可以达到奔溃前的状态，也保证了数据的一致性。

#传播行为

1、PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

2、PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

3、PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

4、PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

5、PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

6、PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

7、PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

介绍一下自己，在之前公司有什么收获，学到什么 答：扯了一波犊子

挑一个你的项目来讲一下 答：看着简历上写的直接分析一波，尽可能的多讲了很多

你要怎么来开多线程 答：直接用Threading模块开启多线程 或者用线程池的方式。

线程有 **就绪、阻塞、运行** 三种基本状态。

1. 就绪状态是指线程具备运行的所有条件，逻辑上可以运行，在等待处理机；

2. 运行状态是指线程占有处理机正在运行；

3. 阻塞状态是指线程在等待一个事件（如某个信号量），逻辑上不可执行。

GIL 本质就是一把互斥锁，既然是互斥锁，所有互斥锁的本质都一样，都是将并发运行变成串行，以此来控制同一时间内共享数据只能被一个任务所修改，进而保证数据安全。

Python 提供多线程编程的模块有以下几个：

- \_thread

- threading

- Queue

- multiprocessing

般来说，使用线程有两种模式，一种是创建线程要执行的函数，把这个函数传递进Thread对象里，让它来执行；另一种是直接从**threading.Thread**继承，创建一个新的class，把线程执行的代码放到这个新的 class里。

讲一下python的Gil 答：讲了多线程不能利用多核，多进程可以，gil是用来解决垃圾回收机制问题的。

在进行GIL讲解之前，我们可以先回顾一下并行和并发的区别：

并行：多个CPU同时执行多个任务，就好像有两个程序，这两个程序是真的在两个不同的CPU内同时被执行。

并发：CPU交替处理多个任务，还是有两个程序，但是只有一个CPU，会交替处理这两个程序，而不是同时执行，只不过因为CPU执行的速度过快，而会使得人们感到是在“同时”执行，执行的先后取决于各个程序对于时间片资源的争夺。

并行和并发同属于多任务，目的是要提高CPU的使用效率。这里需要注意的是，一个CPU永远不可能实现并行，即一个CPU不能同时运行多个程序，但是可以在随机分配的时间片内交替执行（并发），就好像一个人不能同时看两本书，但是却能够先看第一本书半分钟，再看第二本书半分钟，这样来回切换。

多线程并不会充分调用两个CPU，而是会像在一个CPU上充分运转，而多进程则是会完全调用两个CPU，同时执行；

Guido van Rossum（吉多·范罗苏姆）创建python时就只考虑到单核cpu，解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁， 于是有了GIL这把超级大锁。因为cpython解析只允许拥有GIL全局解析器锁才能运行程序，这样就保证了保证同一个时刻只允许一个线程可以使用cpu。由于大量的程序开发者接收了这套机制,现在代码量越来越多,已经不容易通过c代码去解决这个问题。

什么是GIL？

即全局解释器所（global interpreter lock），每个线程在执行时候都需要先获取GIL，保证同一时刻只有一个线程可以执行代码，即同一时刻只有一个线程使用CPU，也就是说多线程并不是真正意义上的同时执行。

那么，我们改如何解决GIL锁的问题呢？

1.更换cpython为jpython(不建议)

2.使用多进程完成多线程的任务

3.在使用多线程可以使用c语言去实现

以下是几个面试会遇到的问题，希望对大家有所帮助：

问题1: 什么时候会释放Gil锁,

答 :

1 遇到像 i/o操作这种 会有时间空闲情况 造成cpu闲置的情况会释放Gil

2 会有一个专门ticks进行计数 一旦ticks数值达到100 这个时候释放Gil锁 线程之间开始竞争Gil锁(说明:

    ticks这个数值可以进行设置来延长或者缩减获得Gil锁的线程使用cpu的时间)

问题2: 互斥锁和Gil锁的关系

Gil锁  : 保证同一时刻只有一个线程能使用到cpu

互斥锁 : 多线程时,保证修改共享数据时有序的修改,不会产生数据修改混乱

首先假设只有一个进程,这个进程中有两个线程 Thread1,Thread2, 要修改共享的数据date, 并且有互斥锁

执行以下步骤

(1)多线程运行，假设Thread1获得GIL可以使用cpu，这时Thread1获得 互斥锁lock,Thread1可以改date数据(但并没有开始修改数据)

(2)Thread1线程在修改date数据前发生了 i/o操作 或者 ticks计数满100 (注意就是没有运行到修改data数据),这个时候 Thread1 让出了Gil,Gil锁可以被竞争

(3) Thread1 和 Thread2 开始竞争 Gil (注意:如果Thread1是因为 i/o 阻塞 让出的Gil Thread2必定拿到Gil,如果Thread1是因为ticks计数满100让出Gil 这个时候 Thread1 和 Thread2 公平竞争)

(4)假设 Thread2正好获得了GIL, 运行代码去修改共享数据date,由于Thread1有互斥锁lock，所以Thread2无法更改共享数据date,这时Thread2让出Gil锁 , GIL锁再次发生竞争

(5)假设Thread1又抢到GIL，由于其有互斥锁Lock所以其可以继续修改共享数据data,当Thread1修改完数据释放互斥锁lock,Thread2在获得GIL与lock后才可对data进行修改

以上描述了 互斥锁和Gil锁的 一个关系

IO密集型的python比计算密集型的程序更能利用多线程环境带来的便利。

1.2 GIL对线程执行的影响

多线程环境中，python虚拟机按照以下方式执行：

1. 设置GIL
2. 切换到一个线程去执行
3. 运行代码，这里有两种机制：
   1. 指定数量的字节码指令（100个）
   2. 固定时间15ms线程主动让出控制
4. 把线程设置为睡眠状态
5. 解锁GIL
6. 再次重复以上步骤

## GIL介绍

1. python全局解释器锁(global interpreter lock, GIL)限制了任何时候只能有一个thread处于运行状态，这对于cpu密集型和多线程程序并不友好，会带来性能瓶颈。

### GIL解决的问题

1. python用引用计数来管理内存对象。当对象的引用计数变量为0的时候，对象占用的内存方可释放。引用计数变量是一个竞态条件，多个线程同时访问的时候需要进行互斥。如果不互斥，可能导致内存泄漏。
2. 这个问题可以通过所有对象加锁来解决，但是这会导致死锁，性能等其他更复杂的问题。
3. GIL是给解释器自身加锁，任何python代码的执行都需要先获得解释器锁。这就解决了死锁问题(只有一个锁)，并且不会带来额外的性能问题。但是却导致cpu型的任务，任意时刻只能同时运行一个thread。
4. GIL并不是这个问题的唯一解决方案。线程安全的内存管理除了引用计数，也可以通过垃圾回收机制解决。但是这样会移除GIL带来的优势，单线程程序和IO型多线程程序的性能损失。

### 为什么选择GIL

1. python的设计就是简单易用，快速开发，让更多的开发者参与其中。
2. 很多extensions需要GIL的线程安全内存管理。一些不是线程安全的C libraries可以很容易的集成到python中。并且GIL的实现很简单。对于单线程的程序也有性能的提升, GIL也是促使python如此流行的一个因素。

### 对于多线程程序的影响

1. 在cpu型多线程程序中，GIL阻止了线程的并行执行。对于IO型的程序，GIL并没有太大的影响，因为当等待IO操作的时候，会进行线程切换，锁是在线程之间共享的。

### GIL为什么没有被移除

1. 移除GIL存在遗留的兼容性问题，还有很多C extensions依赖于GIL的方案。新的方案替代GIL，也会损失单线程程序和IO型多线程程序的性能，没人会希望新的版本反而导致已有程序的性能下降。

### 怎么解决GIL带来的影响

1. 利用多进程模块multiprocessing。多进程会带来显著的性能提升，但是不是成倍的，因为进程比线程更重，有其他开销。GIL存在于CPython，如果条件允许，也可以尝试用其他语言实现的python版本，譬如java实现的版本Jython。

## 深入理解GIL

### pyhton线程

1. python线程是真正的系统线程，posix threads(pthreads), windows threads。
2. 完全由os管理。线程在运行时候持有GIL，在等待IO操作的时候会释放GIL。
3. python并没有自己的线程调度机制，所有的线程调度依赖于OS。这里会有另一个问题，就是signal的处理，signal只能在main thread中被处理，而python解释器无法控制线程调度，所以只能期望更快的切换线程，让主线程得以运行。

### cpu型任务

1. 对于cpu型的任务，解释器会定时的执行check动作，进行线程的切换。这里的定时单位是tick，tick是python解释器的一个指令运行时间。python指令可以通过dis模块查看。

对于广大写Python的人来说，**GIL**(**Global Interpreter Lock**, 全局解释器锁)肯定不陌生，但未必清楚GIL的历史和全貌是怎样的，今天我们就来梳理一下GIL。

**1. 什么是GIL**

GIL的全称是 Global Interpreter Lock，全局解释器锁。之所以叫这个名字，是因为**Python的执行依赖于解释器**。Python最初的设计理念在于，**为了解决多线程之间数据完整性和状态同步的问题，设计为在任意时刻只有一个线程在解释器中运行。**而当执行多线程程序时，由GIL来控制同一时刻只有一个线程能够运行。即Python中的多线程是表面多线程，也可以理解为fake多线程，不是真正的多线程。

可能有的同学会问，**同一时刻只有一个线程能够运行，那么是怎么执行多线程程序的呢？其实原理很简单：解释器的分时复用**。即多个线程的代码，轮流被解释器执行，只不过切换的很频繁很快，给人一种多线程“同时”在执行的错觉。聊的学术化一点，其实就是“**并发**”。

再拓展一点“并发”和“并行”的概念：

普通解释：

并发：交替做不同事情的能力

并行：同时做不同事情的能力

专业术语：

**并发：不同的代码块交替执行**

**并行：不同的代码块同时执行**

那么问题来了，Python为什么要如此设计呢？即**为什么要保证同一时刻只有一个线程在解释器中运行呢**？

答案是为了**Python解释器中原子操作的线程安全**。

**2. 什么是线程安全，什么又是原子操作？**

**2.1 线程安全**

我们首先要搞清楚什么是进程，什么是线程。**进程是系统资源分配的最小单位，线程是程序执行的最小单位**。

举一个例子，如果我们把跑程序比作吃饭，那么进程就是摆满了饭菜的桌子，线程就是吃饭的那个人。

在多线程环境中，当各线程不共享数据的时候，那么一定是线程安全的。问题是这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

线程安全一般都涉及到synchronized，就是**多线程环境中，共享数据同一时间只能有一个线程来操作 不然中间过程可能会产生不可预制的结果**。

接着刚才的**例子**，桌子上有三碗米饭，一个人正在吃，吃了两碗米饭，但是还没有吃完，因此桌子上米饭的数量还没有更新；此时第二个人也想吃米饭，如果没有线程安全方面的考虑，第二个人要是想直接拿三碗米饭吃，就会出错。

**2.2 原子操作**

**2.2.1 什么是原子操作**

**原子操作就是不会因为进程并发或者线程并发而导致被中断的操作**。**原子操作**的特点就是**要么一次全部执行，要么全不执行**。不存在执行了一半而被中断的情况。

**当对全局资源存在写操作时，如果不能保证写入过程的原子性，会出现脏读脏写的情况。**

3. GIL的优点与缺点

GIL的优点是显而易见的，GIL可以保证我们在多线程编程时，无需考虑多线程之间数据完整性和状态同步的问题。

GIL缺点是：我们的多线程程序执行起来是“并发”，而不是“并行”。因此执行效率会很低，会不如单线程的执行效率。

网上很多人都提到过这样的疑问：”为什么我多线程Python程序运行得比其只有一个线程的时候还要慢?“显然，大家觉得一个具有两个线程的程序要比其只有一个线程时要快。事实上,这个问题是确实存在的，原因在于GIL的存在使得Python多线程程序的执行效率甚至比不上单线程的执行效率。很简单，由于GIL使得同一时刻只有一个线程在运行程序，再加上切换线程和竞争GIL带来的开销，显然Python多线程的执行效率就比不上单线程的执行效率了。

4. 为什么会有GIL，GIL的历史

大家显然会继续思考，为什么GIL需要保证只有一个线程在某一时刻处于运行中?难道不可以添加细粒度的锁来阻止多个独立对象的同时访问?并且为什么之前没有人去尝试过类似的事情?

这些实用的问题有着十分有趣的回答。首先要明确一点, Python解释器的实现是有多个版本的：CPython, Jpython等。CPython就是用C语言实现Python解释器，JPython是用Java实现Python解释器。那么 GIL的问题实际上是存在于CPython中的。GIL的问题得不到解决，一方面是因为CPython中一开始就使用GIL的设计理念，并且很多Package依赖于CPython甚至依赖于GIL。因此造成尾大不掉，实际上是个历史问题。

为了利用多核，Python开始支持多线程。而解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁。 于是有了GIL这把超级大锁，而当越来越多的代码库开发者接受了这种设定后，他们开始大量依赖这种特性（即默认python内部对象是thread-safe的，无需在实现时考虑额外的内存锁和同步操作）。

慢慢的这种实现方式被发现是蛋疼且低效的。但当大家试图去拆分和去除GIL的时候，发现大量库代码开发者已经重度依赖GIL而非常难以去除了。有多难？做个类比，像MySQL这样的“小项目”为了把Buffer Pool Mutex这把大锁拆分成各个小锁也花了从5.5到5.6再到5.7多个大版为期近5年的时间，本且仍在继续。MySQL这个背后有公司支持且有固定开发团队的产品走的如此艰难，那又更何况Python这样核心开发和代码贡献者高度社区化的团队呢？

GIL对诸如当前线程状态和为垃圾回收而用的堆分配对象这样的东西的访问提供着保护。这是该实现的一种典型产物。现在也有其它的Python解释器(和编译器)并不使用GIL。虽然,对于CPython来说,自其出现以来已经有很多不使用GIL的解释器。

那么为什么不抛弃GIL呢?许多人也许不知道,在1999年,针对Python 1.5,一个经常被提到但却不怎么理解的“free threading”补丁已经尝试实现了这个想法,该补丁来自Greg Stein。在这个补丁中,GIL被完全的移除,且用细粒度的锁来代替。然而,GIL的移除给单线程程序的执行速度带来了一定的代价。当用单线程执行时,速度大约降低了40%。使用两个线程展示出了在速度上的提高,但除了这个提高,这个收益并没有随着核数的增加而线性增长。由于执行速度的降低,这一补丁被拒绝了,并且几乎被人遗忘。

不过,“free threading”这个补丁是有启发性意义的,其证明了一个关于Python解释器的基本要点:移除GIL是非常困难的。由于该补丁发布时所处的年代,解释器变得依赖更多的全局状态,这使得想要移除当今的GIL变得更加困难。值得一提的是,也正是因为这个原因,许多人对于尝试移除GIL变得更加有兴趣。困难的问题往往很有趣。

但是这可能有点被误导了。让我们考虑一下:如果我们有了一个神奇的补丁,其移除了GIL,并且没有对单线程的Python代码产生性能上的下降,那么我们将会获得我们一直想要的:一个线程API可能会同时利用所有的处理器。但这确实是一个好事吗?

基于线程的编程毫无疑问是困难的。在编码过程中，总是会悄无声息的出现一些新的问题。因此有一些非常知名的语言设计者和研究者已经总结得出了一些线程模型。就像某个写过多线程应用的人可以告诉你的一样,不管是多线程应用的开发还是调试都会比单线程的应用难上数倍。程序员通常所具有的顺序执行的思维模恰恰就是与并行执行模式不相匹配。GIL的出现无意中帮助了开发者免于陷入困境。在使用多线程时仍然需要同步的情况下,GIL事实上帮助我们保持不同线程之间的数据一致性问题。

所以简单的说GIL的存在更多的是历史原因。如果推到重来，多线程的问题依然还是要面对，但是至少会比目前GIL这种方式会更优雅。

5. 如何规避GIL带来的影响

用multiprocess（多进程）替代Thread（推荐）

multiprocess库的出现很大程度上是为了弥补thread库因为GIL而低效的缺陷。它完整的复制了一套thread所提供的接口方便迁移。唯一的不同就是它使用了多进程而不是多线程。每个进程有自己的独立的GIL，因此也不会出现进程之间的GIL争抢。

当然multiprocess也不是万能良药。它的引入会增加程序实现时线程间数据通讯和同步的困难。就拿计数器来举例子，如果我们要多个线程累加同一个变量，对于thread来说，申明一个global变量，用thread.Lock的context包裹住三行就搞定了。而multiprocess由于进程之间无法看到对方的数据，只能通过在主线程申明一个Queue，put再get或者用share memory的方法。这个额外的实现成本使得本来就非常痛苦的多线程程序编码，变得更加痛苦了。

用其他解析器（不推荐）

之前也提到了既然GIL只是CPython的产物，那么其他解析器是不是更好呢？没错，像JPython和IronPython这样的解析器由于实现语言的特性，他们不需要GIL的帮助。然而由于用了Java/C#用于解析器实现，他们也失去了利用社区众多C语言模块有用特性的机会。所以这些解析器也因此一直都比较小众。毕竟功能和性能大家在初期都会选择前者，Done is better than perfect。

GIL与互斥锁

值得注意的是GIL 并不会保护开发者自己编写的代码。这是因为同一时刻固然只能有一个 Python 线程得到执行，但是，当这个线程正在操作某个数据结构的时候，其他线程可能会打断它，一旦发生这种现象，就会破坏程序的状态，从而使相关的数据结构无法保持其一致性。为了保证所有线程能够得到公平地执行，Python 解释器会给每个线程分配大致相等的处理器时间。为了达到这样的分配策略，Python 系统可能当某个线程正在执行的时候将其暂停，然后使另一个线程继续往下执行。由于我们无法提前获知 Python 系统会在何时暂停这些线程，所以我们无法控制程序中某些操作是原子操作。

为了防止线程中出现数据竞争的行为，使开发者可以保护自己的数据结构不受破坏，Python 在 threading 模块中提供了最简单、最有用的工具：Lock 类，该类相当于互斥锁。

在开发中我们可以使用互斥锁来保护某个对象，使得在多线程同时访问某个对象的时候，不会将该对象破坏。因为同一时刻，只有一个线程能够获得这把锁。也就是说对将要访问的对象进行隔离，那么使用线程隔离的意义在于：是当前线程能够正确的引用到它自己创造的对象，而不是引用到其它线程锁创建的对象。

总结

Python GIL其实是功能和性能之间权衡后的产物，它尤其存在的合理性，也有较难改变的客观因素。我们可以做以下一些简单的总结：

* 因为GIL的存在，只有IO Bound场景下得多线程会得到较好的性能
* 如果对并行计算性能较高的程序可以考虑把核心部分也成C模块，或者索性用其他语言实现
* 在Python编程中，如果想利用计算机的多核提高程序执行效率，用多进程代替多线程
* 即使有GIL存在，由于GIL只保护Python解释器的状态，所以对于非原子操作，在Python进行多线程编程时也需要使用互斥锁（如thread中的lock）保证线程安全。
* GIL在较长一段时间内将会继续存在，但是会不断对其进行改进

讲一下数据库的连表有哪些 答：左连 右脸 内联 自关联

1.左右连接：以哪个表为主，结果集为“主表”的全部记录百+“副表”与“主表”相匹配的记录，如果“副表”中没有和“主表”相匹配的记录，则相对应的记录显示为NULL

2.左连接：左边表全部行+右边表相匹配的行，如果左边表中度的某一行，在右边表中没有匹配的行，则显示NULL（left join 或者left outer join）

3.右连接：和左连接相反。（right join 或者right outer join）

4.内连接：它返回字段ID（连接条件）同时存在于两个表中的记录，也就是说，仅当至少有一个同属于两表的行符合联接条件时，内联接才返回行，内联接消除与另一个表中的任何行不匹配的行。（inner　join或者join）

5.全连接：不管匹配不匹配，全部都显示出来回。（full join或者full outer join）

6.交叉连接：没有WHERE 子句的交叉联接将产生答联接所涉及的表的笛卡尔积。第一个表的行数乘以第二个表的行数等于笛卡尔积结果集的大小。（cross join不带where）

7.自连接：给自己取个别名，一个表当两个表来使用。

内存里的东西是退出python之后全都消失吗 答：不一定，有些相互引用了的东西不会被回收

答案是否定的。那些具有对象循环引用或者全局命名空间引用的变量，在 Python 退出是往往不会被释放，另外不会释放 C 库保留的部分内容。例如：Python退出，理论上就会释放资源占有的内存，但是Python中也会出现僵尸进程，这样就会出现有些资源无法释放，那样有些内存就无法释放。

除了Python，对于任何的程序，在退出时操作系统都会负责回收相关的资源。也就是说，即使解释器没有回收内存，操作系统也会回收。

当然例外也是有的，比如你在程序里启动了一个新进程。在Python退出时这个新进程不一定退出，操作系统也就不会回收这个新进程相关的资源。这对于任何语言同样适用的，不过我想这应该也不在讨论范围以内。

答案是：否定的。

那些具有对象循环引用或者全局命名空间引用的变量，在 Python 退出是往往不会被释放

另外不会释放 C 库保留的部分内容。

先给大家普及一波关于python关于内存分配释放的知识：

关于Python中的内存释放问题

首先就不得不提到Python解释器在何种情况下会释放变量的内存。Python引用了内存计数这一简单的计数来控制。

下面是引用计数的知识：

增加引用计数 当对象被创建并（将其引用）赋值给变量时，该对象的引用计数被设置为1。

对象的引用计数增加的情况：

对象被创建：x = 3.14

另外的别名被创建：y = x

被作为参数传递给函数（新的本地引用）：foobar(x)

成为容器对象的一个元素：myList = [123, x, ‘xyz’]

减少引用计数 对象的引用计数减少的情况：

一个本地引用离开了其作用范围。如foobar()函数结束时

对象的别名被显式销毁：del y

对象的一个别名被赋值给其他对象：x = 123

对象被从一个窗口对象中移除：myList.remove(x)

窗口对象本身被销毁：del myList

del语句 Del语句会删除对象的一个引用

python的程序有两种退出方式：os.\_exit()， sys.exit()。本文介绍这两种方式的区别和选择。

os.\_exit()会直接将python程序终止，之后的所有代码都不会继续执行。

sys.exit()会引发一个异常：SystemExit，如果这个异常没有被捕获，那么python解释器将会退出。如果有捕获此异常的代码，那么这些代码还是会执行。

### 区别

综上，sys.exit()的退出比较优雅，调用后会引发SystemExit异常，可以捕获此异常做清理工作。os.\_exit()直接将python解释器退出，余下的语句不会执行。

一般情况下使用sys.exit()即可，一般在fork出来的子进程中使用os.\_exit()

装饰器是怎么样的 答：两大原则，装饰器作用都讲了。**一、闭包**

闭包是语法闭包的简称，是引用自由变量的函数。这个被引用的自由变量将与这个函数一同存在，即使已经离开创造它的环境也不例外。所以，闭包是由函数和与其相关的引用环境组合而成的实体。闭包是函数式编程的重要语法结构。Python也支持这一特性。

闭包的总结：

* 闭包函数必须有嵌套函数；
* 嵌套函数需要引用上一级函数的变量；
* 闭包函数必须返回嵌套函数。

**二、装饰器**

装饰器本质上就是函数，功能是为其他函数添加附加功能。

通俗的来讲，装饰器就是等于高阶函数+函数嵌套+闭包的综合实现。

装饰器有两大原则：第一，不修改被修饰函数的源代码；第二，不修改被修饰函数的调用方法。

3.装饰器调用顺序

使用多个装饰器修饰同个函数时，装饰器的调用顺序与语法糖@的声明顺序相反。

了解虚拟化技术吗？容器这些 答：docker比较擅长的。

虚拟化技术与容器技术

作为一种部署应用程序的全新方式，容器技术在短时间内获得了广泛关注，但是这种技术同样存在某些限制，并且和虚拟机存在本质上的不同。

       虚拟化技术已经改变了现代计算方式，它能够提升系统资源使用效率、消除应用程序和底层硬件之间的依赖关系，同时加强负载的可以移植性和安全性，但是hypervisor和虚拟机只是部署虚拟负载的方式之一。作为一种能够替代传统虚拟化技术的解决方案，容器虚拟化技术凭借其高效性和可靠性得到了快速发展，它能够提供新的特性，并且帮助数据中心专家解决新的顾虑。

　　容器和虚拟机之间的主要区别在于虚拟化层的位置和操作系统资源的使用方式。

　　虚拟机依赖于hypervisor，其通常被安装在“裸金属”系统硬件之上，这导致hypervisor在某些方面被认为是一种操作系统。一旦 hypervisor安装完成， 就可以从系统可用计算资源当中分配虚拟机实例了，每台虚拟机都能够获得唯一的操作系统和负载(应用程序)。

　　通常，第一台虚拟机都将会被用来运行系统管理程序，比如Microsoft System Center。而之后的虚拟机可能包含其他企业负载，比如数据库、ERP、CRM、邮件服务器、媒体服务器、web服务器或者其他业务应用。虚拟机之间是完全隔离的——没有任何一台虚拟机知道(或者依赖)相同系统当中的另外一台虚拟机的存在——恶意软件、应用程序崩溃和其他问题只能影响一台虚拟机。虚拟机从一个虚拟系统被移动到另外一个当中，而不用考虑系统硬件和操作系统等因素。

　　而容器环境的工作方式则有所不同。对于容器环境来说，需要首先安装主机操作系统，之后将容器层(比如LXC或libcontainer)安装在主机操作系统(通常是Linux变种)之上。在安装完容器层之后，就可以从系统可用计算资源当中分配容器实例了，并且企业应用可以被部署在容器当中。但是，每个容器化应用都会共享相同的操作系统(单个主机操作系统)。

　　相比于虚拟机，容器拥有更高的资源使用效率，因为它并不需要为每个应用分配单独的操作系统——实例规模更小、创建和迁移速度也更快。这意味相比于虚拟机，单个操作系统能够承载更多的容器。云提供商十分热衷于容器技术，因为在相同的硬件设备当中，可以部署数量更多的容器实例。然而，单个操作系统有可能引起影响所有相关实例的单点事故。比如，恶意软件或者主机操作系统崩溃可能禁用或者影响所有容器。此外，容器易于迁移，但是只能被迁移到具有兼容操作系统内核的其他服务器当中，这样会无形中减少迁移选择。

　　从实际应用的角度来说，容器和虚拟机可以在同一个数据中心当中共存，因此这两种技术被认为是互补的——为现代应用程序架构师和数据中心管理员添加了可用工具集，通过不同的方式为应用负载提供支持。

python数据类型说一下 答：巴拉巴拉。

ython基本数据类型一般分为：数字、字符串、列表、元组、字典、集合这六种基本数据类型。

数字

|  |  |
| --- | --- |
| 1  2  3  4  5 | 数字类型是不可更改的对象。对变量改变数字值就是生成/创建新的对象。Python支持多种数字类型：  整型(标准整型和长整型(Python2.7及之前的有这种类型))、布尔型、双精度浮点型、十进制浮点型、复数。  标准整型  int，标准整型，在大多数32位机器上标准整型取值范围是-2^31到2^31-1，也就是-2147483648~2147483647，如果在64位机器使用64位  编译器，那么这个系统的标准整型将是64位。 |

布尔型

bool，从Python2.3开始Python中添加了布尔类型。布尔类型有两种True和False。对于没有\_\_nozero\_\_方法的对象默认是True。

对于值为0的数字、空集（空列表、空元组、空字典等）在Python中的布尔类型中都是False。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> bool(1)  True  >>> bool('a')  True  >>> bool(0)  False  >>> bool('')  False |

浮点型

|  |  |
| --- | --- |
| 1  2  3  4  5 | float，每个浮点型占8个字节（64位），完全遵守IEEE754号规范（52M/11E/1S）,其中52个位用于表示底，11个位用于表示指数（可表示  的范围大约是±10\*\*308.25），剩下的一个位表示符号。这看上去相当完美，然而，实际精度依赖于机器架构和创建Python解释器的编译  器。  浮点型值通常都有一个小数点和一个可选的后缀e（大写或小写，表示科学计数法）。在e和指数之间可以用正（+）或负（-）表示指数  的正负（正数的话可以省略符号）。 |

以上是Python核心编程的对浮点型(双精度浮点型)的说明。经过Python实测浮点型默认长度是24字节如果超出这个范围会自动

复数类型

complex，在复数中虚数不能单独存在，它们总是和一个值为0.0的实数部分一起来构成一个复数。

|  |  |
| --- | --- |
| 1  2  3  4 | 复数由实数部分和虚数部分构成。  表示虚数的语法：real+imagj。  实数部分和虚数部分都是浮点型。  虚数部分必须有后缀j或J。 |

字符串

|  |  |
| --- | --- |
| 1  2  3  4 | 字符串(string)是不可变类型，就是说改变一个字符串的元素需要新建一个新的字符串。字符串是由独立的字符组成的，并且这些字符  可以通过切片操作顺序地访问。Python里面通过在引号间包含字符的方式创建字符串，单引号和双引号的作用是相同的。  Python用“原始字符串”操作符来创建直接量字符串，所以再做区分就没什么意义了。其他的语言，比如C语言里面用单引号来标示  字符，双引号标示字符串，而在Python里面没有字符这个类型。这可能是双引号和单引号在Python里面被视作一样的另一个原因。 |

Python实际上有三类字符串。通常意义的字符串（str） 和Unicode字符串（unicode） 实际上都是抽象类basestring的子类。这个basestring是不能实例化的。

列表

|  |  |
| --- | --- |
| 1  2 | 列表(list)像字符串类型一样，列表类型也是序列式的数据类型。字符串只能由字符组成，而且是不可变的（不能单独改变它的某  个值），而列表则是能保留任意数目的Python对象的灵活的容器。 |

列表不仅可以包含Python的标准类型，而且可以用用户定义的对象作为自己的元素。列表可以包含不同类型的对象，而且要比C或者Python自己的数组类型（包含在array扩展包中）都要灵活，因为数组类型所有的元素只能是一种类型。列表可以执行pop,sort、reverse等操作。列表也可以添加或者减少元素，还可以跟其他的列表结合或者把一个列表分成几个。可以对单独一个元素或者多个元素执行insert、update或remove操作。

元组

|  |  |
| --- | --- |
| 1  2  3 | 元组类型在很多操作上都跟列表一样，许多用在列表上的例子在元组上照样能跑，我们有一节内容专门讲解元组类型。它们的主要不同  在于元组是不可变的，或者说是只读的，所以那些用于更新列表的操作，比如用切片操作来更新一部分元素的操作，就不能用于元  组类型。 |

字典

字典是Python语言中唯一的映射类型。映射类型对象里哈希值（键，key） 和指向的对象（值。value） 是一对多的关系。一个字典对象是可变的，它是一个容器类型，能存储任意个数的Python对象，其中也包括其他容器类型。字典类型和序列类型容器类（列表、元组） 的区别是存储和访问数据的方式不同。序列类型只用数字类型的键（从序列的开始起按数值顺序索引） 。映射类型可以用其他对象类型做键，一般最常见的是用字符串做键。和序列类型的键不同，映像类型的键直接或间接地和存储的数据值相关联。但因为在映射类型中，我们不再用“序列化排序”的键，所以映像类型中的数据是无序排列的映射类型不要求用数字值做索引以从一个容器中获取对应的数据项。你可以用键直接“映射”到值，这就是为什么叫映射类型（“mappingtype”） 的原因。映射类型通常被称做哈希表，是因为字典对象就是哈希类型的。字典是Python中最强大的数据类型之一。

字典常见的操作有：字典创建、字典赋值、字典中值的访问、字典更新、字典元素删除等操作。

集合

集合最早出现在Python2.3版本中，通过集合模块来创建，并通过ImmutableSet类和Set类进行访问。

集合有两种不同的类型，可变集合（set）和不可变集合（frozenset）。可变集合，你可以添加和删除元素，对不可变集合则不允许这样做。请注意，可变集合不是可哈希的，因此既不能用做字典的键也不能做其他集合中的元素。不可变集合则正好相反，即，他们有哈希值，能被用做字典的键或是作为集合中的一个成员。集合对象是一组无序排列的可哈希的值。

集合支持用in和not in操作符检查成员，由len()内建函数得到集合的基数（大小） ，用for循环迭代集合的成员。但是因为集合本身是无序的，你不可以为集合创建索引或执行切片（slice） 操作，也没有键可用来获取集合中元素的值。更多学习内容，请点击[Python学习网。](https://www.py.cn/)

见一下列表元祖和字典 答：讲一下可变不可变，有序无序

定义方法的不同：

列表    定义方法：list = ['abc',  123 ]

元组    定义方法：tup = ('abc',  123)

字典    定义方法：dict = {'abc': 123, 'efg': 456}

集合    定义方法：parame = {‘abc’, 123 }   或者    set(value)

列表——list：类型相同的元素，可以改变元素的值，即可进行增删改查操作。用中括号表示[num1,num2,num3]

元组——tuple：和列表功能相似，但不能改变其元素的值，即不可以进行增删改的操作，只能执行查询操作。用小括号表示(num1,num2,num3)

字典——infor：类型不同的元素，由键值对组成，与列表一样可以进行增删改查。用大括号表示{key1:value1,key2:value2,key3：value3}

列表中的元素可以是元组和字典。

另外，在python中，大部分参数是不可以被修改的，如字符串，数字，元组，而列表和字典中的内容是可以被修改的，所以在字典中，列表和字典是不可以被当作key值的。

列表和元组是Python中最常用的两种数据结构，字典是第三种。 相同点：

都是序列

都可以存储任何数据类型

可以通过索引访问

语法差异

使用方括号[]创建列表，而使用括号()创建元组

是否可变

列表是可变的，而元组是不可变的，这标志着两者之间的关键差异。

我们可以修改列表的值，但是不修改元组的值。

由于列表是可变的，我们不能将列表用作字典中的key。 但可以使用元组作为字典key。

重用与拷贝

元组无法复制。 原因是元组是不可变的。 如果运行tuple(tuple\_name)将返回自己。

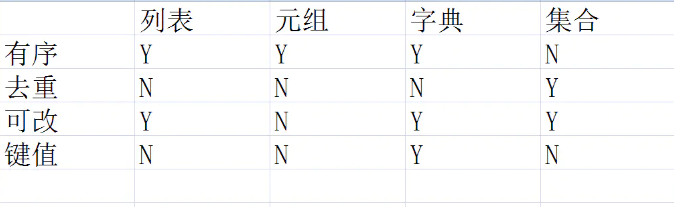
大小差异

Python将低开销的较大的块分配给元组，因为它们是不可变的。 对于列表则分配小内存块。 与列表相比，元组的内存更小。 当你拥有大量元素时，元组比列表快。列表的长度是可变的。

同构与异构

习惯上元组多用于用于存储异构元素，异构元素即不同数据类型的元素，比如(ip,port)。 另一方面，列表用于存储异构元素，这些元素属于相同类型的元素，比如[int1,in2,in3]。

各自特性的异同：



字典内部是怎么存储数据的 答：开辟一块内存空间，申请一个名称空间，指向内存空间。

前几天出去面试的时候，有被问到字典类型的数据在内存中是如何存储的，说实话从事开发工作3年来还真没有研究过这个问题，于是我也是看了几篇文章才学到的，那么我就来讲解一下这个问题

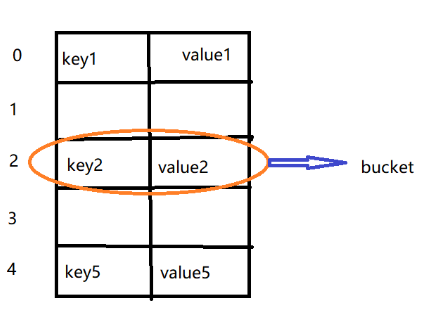
在这之前呢，我们说说什么是hash（摘要百度百科）：

Hash，一般翻译做散列、杂凑，或音译为哈希，是把任意长度的输入（又叫做预映射pre-image）通过散列算法变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

字典的底层原理

字典对象的核心其实是个散列表，而散列表是一个稀疏数组(不是每个位置都有值),每个单元叫做bucket,每个bucket有两部分:一个是键对象的引用,一个是值对象的引用，由于,所有bucket结构和大小一致,我们可以通过偏移量来指定bucket的位置

散列表如图：



将一对键值放入字典的过程：

先定义一个字典，再写入值

dict = {}

dict["age"] = "ming"

在执行第二行时，键name = 值ming放入字典中时，第一步就是计算name的散列值，python中可以用hash函数得到hash值，再用再将得到的值放入bin函数，返回int类型的二进制

print(bin(hash("age")))

# 结果：0b100010001110001000100010110000101101011110100010000101011001111

假设数组长度为10，我们取出计算出的散列值,最右边3位数作为偏移量,即111,十进制是数字7,我们查看偏移量为7对应的bucket的位置是否为空,如果为空,则将键值放进去,如果不为空,依次取右边3位作为偏移量001,十进制是数字1,再查看偏移量1的bucket是否为空,直到单元为空的bucket将键值放进去

以上就是字典的存储原理

字典的查询原理

dict['age'], dict.get('age')

散列值：0b100010001110001000100010110000101101011110100010000101011001111

查询就很简单了，第一步与存储一样，先计算键的散列值，取出后三位111，十进制为7的偏移量，找到对应的bucket的位置，查看是否为空，如果为空就返回None，不为空就获取键并计算键的散列值，计算后将刚计算的散列值与要查询的键的散列值比较，相同就返回对应bucket位置的value，不同就往前再取三位重新计算偏移量，依次取完后还是没有结果就返回None

可以做键的对象条件

键必须可散列

       数字,字符串,元组,都是可散列的

自定义对象做键的条件

       对象内部，有\_\_hash\_\_()和\_\_eq\_\_()方法。

Python 的字典有好多个名称（"映射"、"哈希"、"散列"或者"关系数组"），那你知道为什么字典会被称为 Hash（翻译为"哈希"或"散列"）吗？

你知道为什么字典对于键（Key）的存储数据要求比较严格，但对于对应的值（Value）的存储却要求很宽松吗？

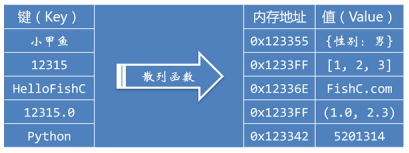
读完这篇文章，你将深刻理解这些问题背后的真相！

首先我们来解释一下什么是 Hash，来自维基百科：

散列函数（或散列算法，又称哈希函数，英语：Hash Function）是一种从任何一种数据中创建小的数字“指纹”的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。该函数将数据打乱混合，重新创建一个叫做散列值的指纹。散列值通常用来代表一个短的随机字母和数字组成的字符串。好的散列函数在输入域中很少出现散列冲突。在散列表和数据处理中，不抑制冲突来区别数据，会使得数据库记录更难找到。

世界上没有两片完全相同的树叶，也没有两个相同的指纹，散列函数是用于从数据中创建小的数字指纹的方法。

我们看下图示：



如图，Python 调用内部的散列函数，将键（Key）作为参数进行转换，得到一个唯一的地址（这也就解释了为什么给相同的键赋值会直接覆盖的原因，因为相同的键转换后的地址是一样滴），然后将值（Value）存放到该地址中。

对于 Python 来说，键（Key）必须是可哈希的，换句话说就是要可以通过散列函数计算出唯一地址的。那如果拿一个变量当键（Key）可以吗？肯定不行。因为变量随时都可能改变，不符合可哈希原则！

同样的，列表、字典、集合这些都是可变的，所以都不能做为键（Key）来使用。

那有朋友可能会问，那元组呢？元组总该是不变的吧？

其实不然，因为元组里边可以存放列表这类可变因素，所以如果实在想拿元组当字典的键（Key），那必须对元祖做限制：元组中只包括像数字和字符串这样的不可变元素时，才可以作为字典中有效的键（Key）。

另外还需要注意的一点是，Python 的哈希算法对相同的值计算得到的结果是一样的，也就是说 12315 和 12315.0 的值相同，他们被认为是相同的键（Key）。

多进程和多线程怎么开 答：直接开或者池，扯到gil锁。

多进程

关键字

开启进程的俩种方式

进程之间的内存空间是隔离的

进程中的join方法使用

进程对象的其它属性和方法

守护进程的使用

进程的互斥锁

进程实现队列

生产者和消费者模型

多线程

使用多线程实现tcp并发

开启线程的俩种方式

进程和线程的俩种区别

区别一：启动数据快

区别二：线程间资源共享，进程间资源独立

守护线程的使用

线程的互斥锁

线程的GIL锁（解释器锁）

paramiko模块实现ssh登录

死锁和递归锁RLock

信号量Semaphore

Event事件

定时器

线程的queue

进程池ProcessPoolExecutor

线程池ThreadPoolExecutor

协程

greenley模块

gevent模块

协程实现单线程实现并发实例

Python的标准库提供了两个模块：thread和threading，thread是低级模块，threading是高级模块，对thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。

启动一个线程就是把一个函数传入并创建Thread实例，然后调用start()开始执行：

使用Thread类，可以有多种方法创建线程：

* 创建Thread类的实例，传递一个函数
* 创建Thread类的实例，传递一个可调用的类实例
* 派生Thread类的子类，并创建子类的实例

一般的，我们会采用第一种或者第三种方法。

**前面也说了python多线程的弊端和GIL的内容，适合IO密集型，而如果解决计算密集型时候的多线程呢？那就是多进程。**

#### multiprocessing模块

python中多线程无法利用多核优势，如果想要充分地使用多核cpu的资源（os.cpu\_count()），在python中大部分情况需要使用多进程，python提供了multiprocessing。

multiprocessing并非是python的一个模块，而是python中多进程管理的一个包

multiprocessing模块用来开启子进程，并在子进程中执行我们定制的任务（比如函数），该模块与多线程模块threading的编程接口类似。

multiprocessing模块的功能众多：支持子进程，通信和共享数据，执行不同形式的同步，提供了process、Queue、Lock等组件。

需要再次强调的一点是：与线程不同，进程没有任何共享状态，进程修改的数据，改动仅限与该进程内。

数据分析有做过吗 答：用三个模块，清洗了数据，给前端做了个图表返回去

数据库有做过优化吗 答：业务量大的时候，建立主从，实现读写分离。

优化手段:

a)  创建索引:创建合适的索引，我们就可以现在索引中查询，查询到以后直接找对应的记录。

b)  分表   ：当一张表的数据比较多或者一张表的某些字段的值比较多并且很少使用时，采用水平分表和垂直分表来优化

c)  读写分离：当一台服务器不能满足需求时，采用读写分离的方式进行集群。

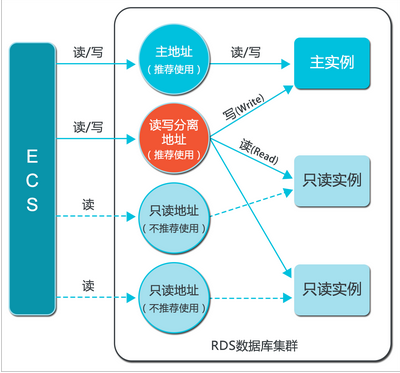
d)  缓存:使用  redis来进行缓存等等。

在互联网项目中，当业务规模越来越大，数据也越来越多，随之而来的就是数据库压力会越来越大。

我们可能会采取各种方式去优化，比如之前文章提到的缓存方案，SQL优化等等，除了这些方式以外，这里再分享几个针对数据库优化的常规手段：「数据读写分离」与「数据库Sharding」。这两点基本上是大中型互联网项目中应用的非常普遍的方案了。

下面我们来详细看一看，

**一、从读写分离到CQRS**



（图片来源阿里云）

由于互联网业务场景，大多数是读多写少，因此进行数据库的读写分离是一件非常简单且有效率的方案。

读写分离简单点来说就是把对数据的读操作和写操作进行分开来，让这两种操作去访问不同的数据库，这样的话，就可以减轻数据库的压力了。

例如上图中，数据库会有一个「主实例」，这个主要用来提供写操作的（偶尔也会承担一点读操作），除了「主实例」以外，还会有多个「从实例」（在图中显示的是 只读实例），「从实例」的功能只是用来承担读操作的。

那上面就出现了多个数据库了，在多个数据库之间的数据是怎么保证一致性的呢？

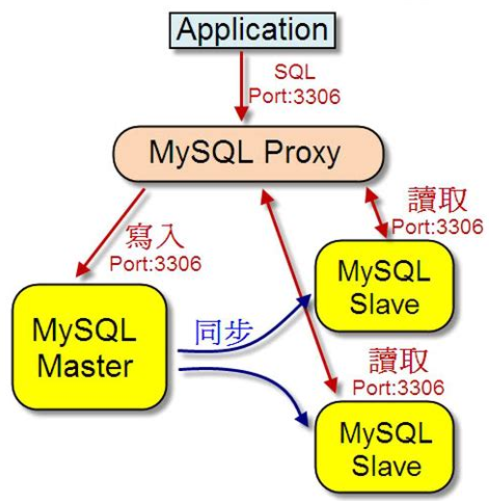
其实，我们常用的数据库就自带这类同步功能，比如 Mysql，它自己有一个master-slave功能，可以实现主库与从库数据的自动同步，是基于二进制日志复制来实现的。在主库进行的写操作，会形成二进制日志，然后Mysql会把这个日志异步的同步到从库上，从库再自动执行一遍这个二进制日志，那么数据就跟主库一致了。

除了Mysql以外，像Oracle等商业数据库都有类似的功能，甚至是网络上还有很多开源的第三方数据同步工具，也有很多成熟好用的。

好了，「主实例」与「从实例」之间的数据同步问题解决了，那现在还有一个问题就是，项目中是怎样让 写请求 去访问「主实例」，让 读请求 去访问「从实例」的，这个路由规则是怎么实现的呢？

常规的有2种方式：

1. **使用编码方式**
2. 这个方式主要是靠开发同学在编码的时候，根据读写不同的操作需求，去调用不同的数据源。例如在数据操作层（DAO层）将读数据与写数据分开为两个方法（函数），然后为这两个方法分别指定不同的数据库即可。
3. 但是这种方式有点硬编码的味道了，而且对开发同学而言还得额外关注这个事情，多了一个编码成本且容易不小心忽略掉。
4. **使用中间件**
5. 这种方式就是在后端数据库的前面，前置一个 数据库代理服务，如下图的：MySQL-Proxy 是Mysql提供的一个中间件，用于实现读写分离请求，但这个组件实际用的人不多，我们可以选择其它的一些开源的组件替代，例如：MyCat、ProxySQL 等等，但大致的原理比较类似，通过这个图很容易理解这个模式。

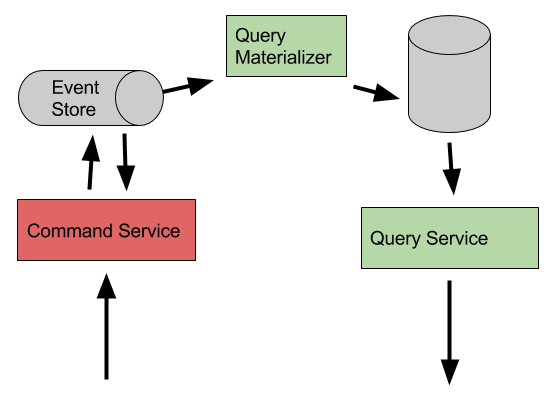


好了，基础的读写分离就讲完了，但感觉这个方式虽然实用是实用，就是不怎么有逼格。

OK，想要有逼格是吧，满足你，那我们就来聊一聊另一个有逼格的读写分离概念: 「 CQRS 」

**CQRS**：Command Query Responsibility Segregation

**命令（增删改）和查询的责任分离**



我们还是先看图，通过上图可以简单的理解一下CQRS

CQRS 重点强调的就是 Query（读） 和 Command（写）的分离，在业务上将职责分离清晰，Command 主要做业务逻辑的执行，Query来负责数据查询和展示。同时 这两种操作是基于不同的数据源，甚至是一个是数据库，另外一个是NoSQL都可以，Query去查询的数据源可以直接按照领域模型进行存储，而并不是按照数据模型去存储，这样查询出来就立即可以展示，而不用转换，且查询效率高。

其实CQRS是由鼎鼎大名的 Martin Fowler 提出，搞计算机的应该都认识。想要更深入的去学习CQRS，可以翻看Martin Fowler公开的资料。

**二、Sharding（分库分表）**

上面讲完了数据库的读写分离，现在我们来聊一下数据库的Sharding。

随着数据库里的数据越来越大，单表查询的性能已经不能满足业务要求了，这个时候就需要进行分表处理了，将大表拆分为若干个小表，不同的分表中数据也不一样，这样可以分散查询压力，提高处理效率。

然而，当表越来越多，所有的数据都在一个数据库上时，网络IO以及文件IO也都会集中在一个数据库上，可能会超过单台服务器的容量， CPU、内存、文件IO、网络IO 都会成为系统的瓶颈，QPS/TPS也会超过单数据库实例的处理极限。那么这个时候就需要对数据库进行分片处理。

因为分表和分库的思路类似，因此下面统一来聊技术方案。

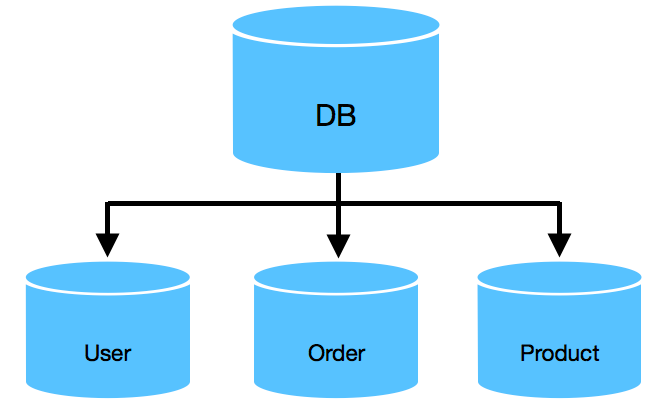
其实分库分表只是我们通俗的便于理解的说话，正确的描述应该是：数据分片

数据的分片主要有2种模式：

* **垂直拆分**
* **水平拆分**

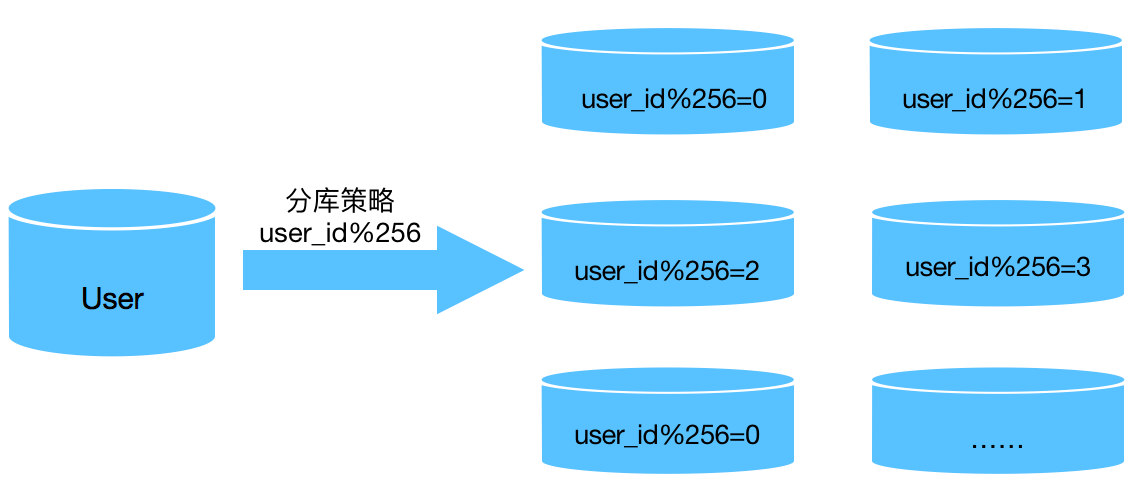
两种拆分应用的场景是不同的：

1. **垂直拆分**，是指按照业务模块进行拆分。简单来讲，就是把业务紧密的模块的字段/表放在一起，放在同一个数据库或者服务器上。将不同业务的字段/表进行独立，拆到不同的数据库或者服务器上。比如一个游戏系统中，可以将玩家基本信息与道具公会等信息进行拆分。
2. 如图示例：



（图片来源网络）

1. **水平拆分**，是指纯粹的按照某种数据规则/格式进行拆分。例如 按照数据唯一ID的哈希散列拆分、按照数据的日期拆分、按照某种范围拆分等等。水平拆分需要注意的是，随着数据动态的变化，分片数量可能需要随之动态调整，另外就是水平分片是没有考虑业务特征的，因此在进行业务汇总查询或者分片中事物处理的时候就比较麻烦一些。
2. 如图示例：



另外，在实际应用中，两种拆分模式一般会结合在一起使用，效果更佳。

以上就是数据库性能优化之「数据读写分离」与「数据库Sharding」方法，欢迎大家一起交流。

分类: [SQL-数据库](https://www.cnblogs.com/lonelyxmas/category/403355.html)

用了哪些数据库 答：mysql redis

讲一下left join 和inner join的区别 答：一个左连是左边所有数据都展示出来，内联两个表都展示。

1 WHERE子句中使用的连接语句，在数据库语言中，被称为隐性连接。INNER JOIN……ON子句产生的连接称为显性连接。（其他JOIN参数也是显性连接）WHERE和INNER JOIN产生的连接关系，没有本质区别，结果也一样。但是！隐性连接随着数据库语言的规范和发展，已经逐渐被淘汰，比较新的数据库语言基本上已经抛弃了隐性连接，全部采用显性连接了。

a inner join：理解为“有效连接”，两张表中都有的数据才会显示left join：理解为“有左显示”，比如on a.field=b.field，则显示a表中存在的全部数据及a\\b中都有的数据，A中有、B没有的数据以null显示

b right join：理解为“有右显示”，比如on a.field=b.field，则显示B表中存在的全部数据及a\\b中都有的数据，B中有、A没有的数据以null显示

c full join：理解为“全连接”，两张表中所有数据都显示，实际就是inner (left-inner) (right-inner)

3 join可以分主次表 外联接有三种类型：完全外联，左联，右联．

完全外联包含两张表的所有记录．

左联是以左边的表为主，右边的为辅，右联则相反

4一般要使得数据库查询语句性能好点遵循一下原则:

在做表与表的连接查询时，大表在前，小表在后

不使用表别名，通过字段e799bee5baa6e59b9ee7ad9431333363393063前缀区分不同表中的字段

查询条件中的限制条件要写在表连接条件前

尽量使用索引的字段做为查询条件

知道union 和union all 的区别吗 答：之前用过，忘记了。

union和union all的区别是,union会自动压缩多个结果集合中的重复结果，而union all则将所有的结果全部显示出来，不管是不是重复。

Union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；

UNION在进行表链接后会筛选掉重复的记录，所以在表链接后会对所产生的结果集进行排序运算，删除重复的记录再返回结果。

实际大部分应用中是不会产生重复的记录，最常见的是过程表与历史表UNION

Union All：对两个结果集进行并集操作，包括重复行，不进行排序；

如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据了。

Union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；

Union All：对两个结果集进行并集操作，包括重复行，不进行排序；

进程之间怎么通信 答：通过队列或者管道，生产者消费者模型。

网络和进程间通信

本章介绍的模块提供了网络和进程间通信的机制。

某些模块仅适用于同一台机器上的两个进程，例如 [signal](https://docs.python.org/zh-cn/3/library/signal.html#module-signal) 和 [mmap](https://docs.python.org/zh-cn/3/library/mmap.html#module-mmap) 。 其他模块支持两个或多个进程可用于跨机器通信的网络协议。

本章中描述的模块列表是：

* [asyncio --- 异步 I/O](https://docs.python.org/zh-cn/3/library/asyncio.html)
* [socket --- 底层网络接口](https://docs.python.org/zh-cn/3/library/socket.html)
* [ssl --- 套接字对象的TLS/SSL封装](https://docs.python.org/zh-cn/3/library/ssl.html)
* [select --- 等待 I/O 完成](https://docs.python.org/zh-cn/3/library/select.html)
* [selectors --- 高级 I/O 复用库](https://docs.python.org/zh-cn/3/library/selectors.html)
* [asyncore --- 异步socket处理器](https://docs.python.org/zh-cn/3/library/asyncore.html)
* [asynchat --- 异步 socket 指令/响应 处理器](https://docs.python.org/zh-cn/3/library/asynchat.html)
* [signal --- 设置异步事件处理程序](https://docs.python.org/zh-cn/3/library/signal.html)
* [mmap --- 内存映射文件支持](https://docs.python.org/zh-cn/3/library/mmap.html)

python提供了多种进程通信的方式，包括信号，管道，消息队列，信号量，共享内存，socket等

主要Queue和Pipe这两种方式，Queue用于多个进程间实现通信，Pipe是两个进程的通信。

1.管道：分为匿名管道和命名管道

匿名管道：在内核中申请一块固定大小的缓冲区，程序拥有写入和读取的权利，一般使用fock函数实现父子进程的通信

命名函数：在内存中申请一块固定大小的缓冲区，程序拥有写入和读取的权利，没有血缘关系的进程也可以进程间通信

特点：面向字节流；生命周期随内核；自带同步互斥机制；半双工，单向通信，两个管道实现双向通信

2.消息队列：在内核中创建一个队列，队列中每个元素是一个数据报，不同的进程可以通过句柄去访问这个队列。消息队列提供了一个从一个进程向另外一个进程发送一块数据的方法。每个数据块都被认为是有一个类型，接收者进程接收的数据块可以有不同的类型。消息队列也有管道一样的不足，就是每个消息的最大长度是有上限的，每个消息队列的总的字节数是有上限的，系统上消息队列的总数也有一个上限

特点：消息队列可以被认为是一个全局的一个链表，链表节点中存放着数据报的类型和内容，有消息队列的标识符进行标记；消息队列允许一个或多个进程写入或读取消息；消息队列的生命周期随内核；消息队列可实现双向通信

3.信号量：在内核中创建一个信号量集合（本质上是数组），数组的元素（信号量）都是1，使用P操作进行-1，使用V操作+1

P(sv)：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该程序的执行

V(sv)：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而挂起，就给它加1

PV操作用于同一个进程，实现互斥；PV操作用于不同进程，实现同步

功能：对临界资源进行保护

4.共享内存：将同一块物理内存一块映射到不同的进程的虚拟地址空间中，实现不同进程间对同一资源的共享。共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式

特点：不同从用户态到内核态的频繁切换和拷贝数据，直接从内存中读取就可以；共享内存是临界资源，所以需要操作时必须要保证原子性。使用信号量或者互斥锁都可以；生命周期随内核

### 1.1 Queue有两个方法：

* Put方法：以插入数据到队列中，他还有两个可选参数：blocked和timeout。详情自行百度
* Get方法：从队列读取并且删除一个元素。同样，他还有两个可选参数：blocked和timeout。详情自行百度

### 1.2 Pipe通信机制，

### \* Pipe常用于两个进程，两个进程分别位于管道的两端

### \* Pipe方法返回（conn1,conn2）代表一个管道的两个端，Pipe方法有duplex参数，默认为True，即全双工模式，若为FALSE，conn1只负责接收信息，conn2负责发送，

### \* send和recv方法分别为发送和接收信息。

二面：

讲一下python的线程 答：讲了多线程 多进程 gil

jwt是干嘛的，怎么实现的 答：用户认证，加密，三个部分加密（最重要），校验。

mysql的乐观锁悲观锁 答：逻辑实现的乐观锁，物理实现的悲观锁

讲一下具体应用场景（乐观锁） 答：讲了一个订单

讲一下事务 答：银行转账

讲一下事务的隔离级别 答：不了解

讲一下脏读幻读，不可重复读 答：我回答错了

讲一下协程是怎么实现的 答：用第三方包gevent，然后他说是yield，我解释了一下。

讲一下垃圾回收机制 答：引用计数，相互引用，分代回收

你多线程是怎么实现的，三方包吗？ 答：对，直接开启多线程或者用线程池

redis的数据类型 答：数字，字符串，字典，集合，有序集合

redis主从讲一下 答：一主多从 一主一从

主从的缺点是什么 答：主库挂了就完蛋了

redis你都用来做了些什么 答：做消息队列，做缓存

redis数据怎么持久化存储 答：忘记了

redis的哨兵讲一下 答：哨兵用来实现高可用巴拉巴拉

python现在是什么版本了 答：3.8

3.7和3.8有什么区别 答：不知道

装饰器是什么 答：两大原则

深浅拷贝 答：一层两层的说

一个函数超过五秒就退出 答：不记得了

linux怎么查看端口 答：支支吾吾

linux怎么查看内存占用情况 答：支支吾吾

ping这个命令的底层协议是什么 答：tcp

tcp建立连接 答：三次握手四次挥手

tcp的回执是什么包 答：不记得了

django请求生命周期 答：讲了一下生命周期

wsgiref和uWSGI的区别 答：并发量不同

redis五大数据类型 答：巴拉巴拉

redis如果存多了要怎么办 答：持久化存储

celery讲一下 答：异步框架

多进程和多线程的区别 答：巴拉巴拉

进程之间怎么通信 答：队列或管道

索引有哪些 答：只回答了主键索引，唯一索引

索引是怎么加快查询速度的 答：支支吾吾，它是一种新的数据结构

索引底层是什么数据结构 答：不知道

你在orm里写原生sql，怎么判断他走了索引或者是全局查了一次 答：不知道

你们之前的项目上线了是怎么看他死了没 答：不清楚，这是运维做的。

讲一个项目 答：cmdb

怎么起一个容器 答：docker start 容器

怎么新建一个容器 答：。。。。

全局异常捕获 答：。。。。。

装饰器 答：。。。。

http状态码 答：2 3 4 5

秒杀用什么锁 答：悲观锁，其实应该用乐观锁

讲项目 答：cmdb

写题目给你写

装饰器怎么写

with管理上下文的源码怎么写，类似的就行

让你开发微信二维码电脑端登录，你要怎么实现

property是什么

\_\_setattr\_\_

一个字符串，让你倒序输出，例子 my name is >>> is name my

判断括号是否合法，类似于leetcode上的，[](){}，每个括号都要成对

get-post定义的规范

接口评审

接口管理规范

架构管理规范

最有成就感的项目，学到经验，让自己比较踏实，让自己得到启发的项目

接口文档怎么维护，如何维护三端（小程序，安卓，网页等）连接，以一套接口为原则，接口冲突怎么解决

mysql存储过程

小程序的类型，什么类型的项目

mysql的主从复制，主从服务

项目发布的过程（上传-发布），上线，发布，测试，谁发布测试

pandas库有哪些函数，介绍pandas函数的作用

rpa技术

介绍项目过程中，测试完之后是由运维发布，测试让运维发版

有什么能力竞争这个岗位

如何学习一门新技术，学习方式,举一个类子

接触自动化测试

逻辑能力和算法能力

如果一个评估项目，需要一个月的时间，用什么心态和方法来圆满完成。

讲一讲项目组之间如果出现了进度紧张，应该怎么处理

做web项目不懂的，如何让一个不认识的同事来帮你，如何请教他

如何和前端有冲突，比如一个方法的争议，如何处理

自己的缺点和优点

简单一句话介绍自己的工作上和生活。讲明自己是一个什么样的人

讲一下自己的俩年到五年的计划

应该做哪些努力来提升自己

二面：

讲一下python的线程 答：讲了多线程 多进程 gil。

jwt是干嘛的，怎么实现的 答：用户认证，加密，三个部分加密（最重要），校验。

一、优势简介

JSON Web Tokens简称jwt，是rest接口的一种安全策略。本身有很多的优势：

解决跨域问题：这种基于Token的访问策略可以克服cookies的跨域问题。

服务端无状态可以横向扩展，Token可完成认证，无需存储Session。

系统解耦，Token携带所有的用户信息，无需绑定一个特定的认证方案，只需要知道加密的方法和密钥就可以进行加密解密，有利于解耦。

防止跨站点脚本攻击，没有cookie技术，无需考虑跨站请求的安全问题。

二、原理简介

JSON Web Tokens的格式组成，jwt是一段被base64编码过的字符序列，用点号分隔，一共由三部分组成，头部header，消息体playload和签名sign。

要如何用php实现JWT认证，那我们首先就来认识一下什么是JWT。

**什么是JWT**JWT(json web token)是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准。

JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源。比如用在用户登录上。

JWT定义了一种用于简洁，自包含的用于通信双方之间以 JSON 对象的形式安全传递信息的方法。JWT 可以使用 HMAC 算法或者是 RSA 的公钥密钥对进行签名。

**JWT有两个特点：**

自包含(Self-contained)：负载中包含了所有用户所需要的信息，避免了多次查询数据库

简洁(Compact)：可以通过URL, POST 参数或者在 HTTP header 发送，因为数据量小，传输速度快

**JWT组成**

JWT由header，payload，signature三个部分，下面我们用官网的实例先来讲解一个这三个部分的用法。

header部分：

jwt的头部承载两部分信息：

声明类型，这里是jwt

声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：

{

 "alg": "HS256",

 "typ": "JWT"

}

对应base64UrlEncode编码为：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

说明：该字段为json格式。alg字段指定了生成signature的算法，默认值为HS256，typ默认值为JWT

**payload部分：**

载荷就是存放有效信息的地方。

标准中注册的声明 (建议但不强制使用) ：

iss: jwt签发者

sub: jwt所面向的用户

aud: 接收jwt的一方

exp: jwt的过期时间，这个过期时间必须要大于签发时间

nbf: 定义在什么时间之前，该jwt都是不可用的

iat: jwt的签发时间

jti: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

{

 "sub": "1234567890",

 "name": "John Doe",

 "iat": 1516239022

}

对应base64UrlEncode编码为：

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ

说明：该字段为json格式，表明用户身份的数据，可以自己自定义字段，很灵活。sub 面向的用户，name 姓名 ,iat 签发时间。例如可自定义示例如下：

{

  "iss": "admin",     //该JWT的签发者

  "iat": 1535967430,    //签发时间

  "exp": 1535974630,    //过期时间

  "nbf": 1535967430,     //该时间之前不接收处理该Token

  "sub": "www.admin.com",  //面向的用户

  "jti": "9f10e796726e332cec401c569969e13e"  //该Token唯一标识

}

signature部分：

jwt的第三部分是一个签证信息，这个签证信息由三部分组成：

* header (base64后的)
* payload (base64后的)
* secret

HMACSHA256(

 base64UrlEncode(header) + "." +

 base64UrlEncode(payload),

 123456

)

对应的签名为：

keH6T3x1z7mmhKL1T3r9sQdAxxdzB6siemGMr\_6ZOwU

最终得到的JWT的json为(header.payload.signature)：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.keH6T3x1z7mmhKL1T3r9sQdAxxdzB6siemGMr\_6ZOwU

说明：对header和payload进行base64UrlEncode编码后进行拼接。通过key（这里是123456）进行HS256算法签名。

**JWT的使用流程**

* 初次登录：用户初次登录，输入用户名密码
* 密码验证：服务器从数据库取出用户名和密码进行验证
* 生成JWT：服务器端验证通过，根据从数据库返回的信息，以及预设规则，生成JWT
* 返还JWT：服务器的HTTP RESPONSE中将JWT返还
* 带JWT的请求：以后客户端发起请求，HTTP REQUEST
* HEADER中的Authorizatio字段都要有值，为JWT
* 服务器验证JWT

## 起源

说起JWT，我们应该来谈一谈基于token的认证和传统的session认证的区别。

传统的session认证

我们知道，http协议本身是一种无状态的协议，而这就意味着如果用户向我们的应用提供了用户名和密码来进行用户认证，那么下一次请求时，用户还要再一次进行用户认证才行，因为根据http协议，我们并不能知道是哪个用户发出的请求，所以为了让我们的应用能识别是哪个用户发出的请求，我们只能在服务器存储一份用户登录的信息，这份登录信息会在响应时传递给浏览器，告诉其保存为cookie,以便下次请求时发送给我们的应用，这样我们的应用就能识别请求来自哪个用户了,这就是传统的基于session认证。

但是这种基于session的认证使应用本身很难得到扩展，随着不同客户端用户的增加，独立的服务器已无法承载更多的用户，而这时候基于session认证应用的问题就会暴露出来.

基于session认证所显露的问题

Session: 每个用户经过我们的应用认证之后，我们的应用都要在服务端做一次记录，以方便用户下次请求的鉴别，通常而言session都是保存在内存中，而随着认证用户的增多，服务端的开销会明显增大。

扩展性: 用户认证之后，服务端做认证记录，如果认证的记录被保存在内存中的话，这意味着用户下次请求还必须要请求在这台服务器上,这样才能拿到授权的资源，这样在分布式的应用上，相应的限制了负载均衡器的能力。这也意味着限制了应用的扩展能力。

CSRF: 因为是基于cookie来进行用户识别的, cookie如果被截获，用户就会很容易受到跨站请求伪造的攻击。

基于token的鉴权机制

基于token的鉴权机制类似于http协议也是无状态的，它不需要在服务端去保留用户的认证信息或者会话信息。这就意味着基于token认证机制的应用不需要去考虑用户在哪一台服务器登录了，这就为应用的扩展提供了便利。

流程上是这样的：

* 用户使用用户名密码来请求服务器
* 服务器进行验证用户的信息
* 服务器通过验证发送给用户一个token
* 客户端存储token，并在每次请求时附送上这个token值
* 服务端验证token值，并返回数据

这个token必须要在每次请求时传递给服务端，它应该保存在请求头里， 另外，服务端要支持CORS(跨来源资源共享)策略，一般我们在服务端这么做就可以了Access-Control-Allow-Origin: \*。

那么我们现在回到JWT的主题上。

JWT长什么样？

JWT是由三段信息构成的，将这三段信息文本用.链接一起就构成了Jwt字符串。就像这样:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

JWT的构成

第一部分我们称它为头部（header),第二部分我们称其为载荷（payload, 类似于飞机上承载的物品)，第三部分是签证（signature).

header

jwt的头部承载两部分信息：

* 声明类型，这里是jwt
* 声明加密的算法 通常直接使用 HMAC SHA256

完整的头部就像下面这样的JSON：

{

'typ': 'JWT',

'alg': 'HS256'

}

然后将头部进行base64加密（该加密是可以对称解密的),构成了第一部分.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

playload

载荷就是存放有效信息的地方。这个名字像是特指飞机上承载的货品，这些有效信息包含三个部分

* 标准中注册的声明
* 公共的声明
* 私有的声明

标准中注册的声明 (建议但不强制使用) ：

* iss: jwt签发者
* sub: jwt所面向的用户
* aud: 接收jwt的一方
* exp: jwt的过期时间，这个过期时间必须要大于签发时间
* nbf: 定义在什么时间之前，该jwt都是不可用的.
* iat: jwt的签发时间
* jti: jwt的唯一身份标识，主要用来作为一次性token,从而回避重放攻击。

公共的声明 ：

公共的声明可以添加任何的信息，一般添加用户的相关信息或其他业务需要的必要信息.但不建议添加敏感信息，因为该部分在客户端可解密.

私有的声明 ：

私有声明是提供者和消费者所共同定义的声明，一般不建议存放敏感信息，因为base64是对称解密的，意味着该部分信息可以归类为明文信息。

定义一个payload:

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

然后将其进行base64加密，得到Jwt的第二部分。

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9

signature

jwt的第三部分是一个签证信息，这个签证信息由三部分组成：

* header (base64后的)
* payload (base64后的)
* secret

这个部分需要base64加密后的header和base64加密后的payload使用.连接组成的字符串，然后通过header中声明的加密方式进行加盐secret组合加密，然后就构成了jwt的第三部分。

// javascript

var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(payload);

var signature = HMACSHA256(encodedString, 'secret'); // TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

将这三部分用.连接成一个完整的字符串,构成了最终的jwt:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

注意：secret是保存在服务器端的，jwt的签发生成也是在服务器端的，secret就是用来进行jwt的签发和jwt的验证，所以，它就是你服务端的私钥，在任何场景都不应该流露出去。一旦客户端得知这个secret, 那就意味着客户端是可以自我签发jwt了。

如何应用

一般是在请求头里加入Authorization，并加上Bearer标注：

fetch('api/user/1', {

headers: {

'Authorization': 'Bearer ' + token

}

})

服务端会验证token，如果验证通过就会返回相应的资源。整个流程就是这样的:

jwt-diagram

总结

优点

* 因为json的通用性，所以JWT是可以进行跨语言支持的，像JAVA,JavaScript,NodeJS,PHP等很多语言都可以使用。
* 因为有了payload部分，所以JWT可以在自身存储一些其他业务逻辑所必要的非敏感信息。
* 便于传输，jwt的构成非常简单，字节占用很小，所以它是非常便于传输的。
* 它不需要在服务端保存会话信息, 所以它易于应用的扩展

安全相关

* 不应该在jwt的payload部分存放敏感信息，因为该部分是客户端可解密的部分。
* 保护好secret私钥，该私钥非常重要。
* 如果可以，请使用https协议

mysql的乐观锁悲观锁 答：逻辑实现的乐观锁，物理实现的悲观锁。

悲观锁

悲观锁（Pessimistic Lock），顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。

Java synchronized 就属于悲观锁的一种实现，每次线程要修改数据时都先获得锁，保证同一时刻只有一个线程能操作数据，其他线程则会被block。

乐观锁

乐观锁（Optimistic Lock），顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。乐观锁适用于读多写少的应用场景，这样可以提高吞吐量。

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。

乐观锁一般来说有以下2种方式：

1. 使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。
2. 使用时间戳（timestamp）。乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

Java JUC中的atomic包就是乐观锁的一种实现，AtomicInteger 通过CAS（Compare And Set）操作实现线程安全的自增。

MySQL隐式和显示锁定

MySQL InnoDB采用的是两阶段锁定协议（two-phase locking protocol）。在事务执行过程中，随时都可以执行锁定，锁只有在执行 COMMIT或者ROLLBACK的时候才会释放，并且所有的锁是在同一时刻被释放。前面描述的锁定都是隐式锁定，InnoDB会根据事务隔离级别在需要的时候自动加锁。

另外，InnoDB也支持通过特定的语句进行显示锁定，这些语句不属于SQL规范：

* SELECT ... LOCK IN SHARE MODE
* SELECT ... FOR UPDATE

## 乐观锁与悲观锁的区别

* 乐观锁的思路一般是表中增加版本字段，更新时where语句中增加版本的判断，算是一种CAS（Compare And Swep）操作，商品库存场景中number起到了版本控制（相当于version）的作用（ AND number=#{number}）。
* 悲观锁之所以是悲观，在于他认为本次操作会发生并发冲突，所以一开始就对商品加上锁（SELECT ... FOR UPDATE），然后就可以安心的做判断和更新，因为这时候不会有别人更新这条商品库存。

## 小结

* 这里我们通过 MySQL 乐观锁与悲观锁 解决并发更新库存的问题，当然还有其它解决方案，例如使用 **分布式锁**。目前常见分布式锁实现有两种：基于Redis和基于Zookeeper，基于这两种 业界也有开源的解决方案，例如 [Redisson Distributed locks](https://link.jianshu.com?t=https:/github.com/redisson/redisson/wiki/8.-distributed-locks-and-synchronizers) 、 [Apache Curator Shared Lock](https://link.jianshu.com?t=http:/curator.apache.org/curator-recipes/shared-lock.html) ，这里就不细说，网上Google 一下就有很多资料。

悲观锁：正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）的修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。

如何选择乐观锁还是悲观锁:

1.响应速度：如果需要非常高的响应速度，建议采用乐观锁方案，成功就执行，不成功就失败，不需要等待其他并发去释放锁

2.冲突频率：如果冲突频率非常高，建议采用悲观锁，保证成功率，如果冲突频率大，乐观锁会需要多次重试才能成功，代价比较大

3.重试代价：如果重试代价大，建议采用悲观锁

## 乐观锁

总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

## 悲观锁

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

注：要使用数据库的悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。

**悲观锁分为两种：共享锁和排它锁**

**共享锁是其它事务可以读但是不能写**

**排他锁是只有自己得事务有权限对此数据进行读写**

讲一下具体应用场景（乐观锁） 答：讲了一个订单。

俗话说金三银四，不管是为了升职加薪还是打算换个环境，现在正是广大程序员们的跳槽季，IT行业正暗流涌动腥风血雨，互联网寒冬似乎并没有过去，很多企业依然在裁员，还有部分企业光明正大的实行着"996"，这一行可能正面临着最大的竞争压力和生存压力。我们无法改变环境，也无法左右市场，但是我们能改变和左右的是自己的知识深度和广度，只有武装到牙齿，不断的充实自己，才能在面试这场江湖纷争里游刃有余。

你是不是经常被面试官突如其来的问题问的语无伦次或者毫无底气？进而在一整场面试里都失去了信心。最近有个群员就遇到这个问你，面试官让他自我介绍以后，就突然问他mysql的悲观锁和乐观锁区别，而这正好是他的盲区，第一个问题就被堵住了，这场面试的结果也可想而知。下面就从面试角度来浅谈一下，该怎样回答这个问题。

一、概念上区别

乐观锁（\*\* Optimistic Locking\*\*）：顾名思义，对加锁持有一种乐观的态度，即先进行业务操作，不到最后一步不进行加锁，"乐观"的认为加锁一定会成功的，在最后一步更新数据的时候再进行加锁。

悲观锁\*\*（Pessimistic Lock）\*\*：正如其名字一样，悲观锁对数据加锁持有一种悲观的态度。因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

二、实现方式：

乐观锁：

* version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

sql实现代码：

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

* CAS（定义见后）操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。（这种方式作者也是最近刚知道，惭愧惭愧）

悲观锁：是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了（原理：共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程），如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

三、使用场景

* 乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。
* 悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

四、特点

* 乐观锁：乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。
* 悲观锁：悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。通常所说的“一锁二查三更新”即指的是使用悲观锁。

五、sql实现代码和案例

因为本文主要是讲解的是面试的时候怎样回答，所以就不将具体sql代码和案例贴出来了，感兴趣的同学可以自行去看一下，如果以后有时间，我也会单独写一篇详细的实现案例出来。

总结：以上，基本回答完前四点后，这道题已经能在面试官心里达到八九十分了。但是我们的人生不仅仅是为了面试，希望同学们一定要弄懂其中的原理和区别，悲观锁和乐观锁都有自己的优缺点，**简而言之记得一句话：读取频繁使用乐观锁，写入频繁使用悲观锁。乐观锁不能解决脏读的问题。**

如果觉得本文有用，请推荐给更多有需要的人，谢谢！如果发现问题，欢迎留言，请随时批评改正，谢谢！

现在我有一个购买商品的需求，我们知道当我们购买商品时，后台会进行减库存和增加购买记录的操作。我们分别在无锁和乐观锁和悲观锁进行相应的代码演示来说明问题。

ySql的乐观锁：

    我们在使用乐观锁时会假设在极大多数情况下不会形成冲突，只有在数据提交的时候，才会对数据是否产生冲突进行检验。如果数据产生冲突了，则返回错误信息，进行相应的处理。

    实现：MySql最经常使用的乐观锁时进行版本控制，也就是在数据库表中增加一列，记为version，当我们将数据读出时，将版本号一并读出，当数据进行更新时，会对这个版本号进行加1，当我们提交数据时，会判断数据库表中当前的version列值和当时读出的version是否相同，若相同说明没有进行更新的操作，不然，则取消这次的操作。

三、悲观锁

    MySql的悲观锁就是打开事务，当启动事务时，如果事务中的sql语句涉及到索引并用索引进行了条件判断，那么会使用行级锁锁定所要修改的行，否则使用表锁锁住整张表。

我们可以在不同的场合使用不同的处理方法，乐观锁并发高并且性能也很好，而悲观锁虽然并发不是很高，但是它不允许脏读，所以各有各的优点。

讲一下事务 答：银行转账。

讲一下事务的隔离级别 答：不了解

讲一下脏读幻读，不可重复读 答：我回答错了

讲一下协程是怎么实现的 答：用第三方包gevent，然后他说是yield，我解释了一下。

**协程**

协程是一种用户态的轻量级线程，又称微线程。

协程拥有自己的寄存器上下文和栈，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。

优点：

1. 无需线程上下文切换的开销
2. 无需原子操作锁定及同步的开销
3. 方便切换控制流，简化编程模型
4. 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。

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。

缺点：

1. 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。
2. 进行阻塞（Blocking）操作（如IO时）会阻塞掉整个程序

### 协程实现方式

1. 1) yield
2. 2) Greenlet
3. 3) Gevent

### 协程的优缺点

1. 优点： 无需线程上下文切换的开销
2. 无需原子操作锁定及同步的开销
3. 缺点： 无法利用多核资源
4. 进行阻塞操作

## 2.yield实现的协程

1. 传统的生产者－消费者模型是一个线程生成消息，一个线程取得消息，能过锁机制控制队列和等待，但一不小心就有可能死锁。
2. 如果改用协程，生产者生产消息后，直接通过yield跳转到消费者开始执行，待消费者执行完毕后，切换加生产者继续生产，效率较高。
3. 整个流程是由一个线程执行，producer和consumer协作完成任务，所以称为协程，而不是线程中的抢占式多任务。
4. 基于协程的定义，刚才使用yield实现的协程并不算合格的协程。

yield 是类似于return的关键字，不同的是，return关键字是直接结束了函数的执行。而yield 可以通过调用 send(), \_\_next\_\_() 让函数继续从yield关键字下一条语句继续执行。

send()函数的用法 是可以传参的，send(param) 。即传一个参数给yield生成器。

\_\_next\_\_() 函数和send()不同的是它不传参。

## 3.由greenlet模块实现的协程

greenlet机制的主要思想是：生成器函数或者协程函数中的yield语句挂起函数的执行，直到稍后使用next()或send()操作进行恢复主止。可以使用一个调度器循环在一组生成器函数之间协作多个任务。greenlet是python中实现协程的一个模块。

greenlet不是一种真正的并发机制，而是在同一线程内，在不同函数的执行代码块之间切换

当出现阻塞时，就显式切换到没有被阻塞的代码段执行，直到另一端代码再显示的切换到本段代码时，这段代码才会继续执行。

## 4.基于greenlet框架，gevent模块实现协程

python通过yield提供了对协程的基本支持，但是不完全。第三方的gevent模块提供了协程支持。

gevent是第三方库，通过greenlet实现协程。

当一个greenlet遇到IO操作时，比如访问网络，就自动切换到其他的greenlet，等到IO操作完成，再在适当的时候切换回来继续执行。由于IO操作非常耗时，经常使程序处于等待状态，有了gevent自动切换协程，就保证总有greenlet在运行，而不是等待IO。

gevent.spawn()方法spawn一些任务，然后通过gevent.joinall将任务加入协程执行队列中等待执行。

gevent的核心greenlet，并试用 epoll机制使线程实现自动切换，并保证始终有greenlet在运行，而不是等待io。

gevent.spawn()方法会创建并运行一个新的greenlet协程对象。

gevent.joinall()方法的参数是一个协程对象列表，等待所有的协程都执行完成后退出

## 5.协程的优点：

无需线程上下文切换造成的资源的浪费。

无需原子操作锁定及同步的开销。

方便切换控制流，简化编程模型。

高并发及高扩展性加低成本：一个CPU支持上万的协程都可以，于高并发处理。

## 6.协程的缺点：

无法利用多核资源，协程的本质是单个线程，不能同时使用多核CPU。

协程需要与进程配合才能运行在多CPU上。

程序一旦阻塞，会阻塞整个代码段。

讲一下垃圾回收机制 答：引用计数，相互引用，分代回收。简单来说python的内存管理机制有三种

1）引用计数

2）垃圾回收。①  引用计数，②  标记清除，③  分代回收。

3）内存池

Python 使用标记清除（mark-sweep）算法和分代收集（generational），来启用针对循环引用的自动垃圾回收。

先来看标记清除算法。我们先用图论来理解不可达的概念。对于一个有向图，如果从一个节点出发进行遍历，并标记其经过的所有节点；那么，在遍历结束后，所有没有被标记的节点，我们就称之为不可达节点。显而易见，这些节点的存在是没有任何意义的，自然的，我们就需要对它们进行垃圾回收。

当然，每次都遍历全图，对于 Python 而言是一种巨大的性能浪费。所以，在 Python 的垃圾回收实现中，标记清除算法使用双向链表维护了一个[数据结构](http://c.biancheng.net/data_structure/)，并且只考虑容器类的对象（只有容器类对象才有可能产生循环引用）。

而分代收集算法，则是将 Python 中的所有对象分为三代。刚刚创立的对象是第 0 代；经过一次垃圾回收后，依然存在的对象，便会依次从上一代挪到下一代。而每一代启动自动垃圾回收的阈值，则是可以单独指定的。当垃圾回收器中新增对象减去删除对象达到相应的阈值时，就会对这一代对象启动垃圾回收。

事实上，分代收集基于的思想是，新生的对象更有可能被垃圾回收，而存活更久的对象也有更高的概率继续存活。因此，通过这种做法，可以节约不少计算量，从而提高 Python 的性能。

内存池机制

Python提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统。

Python中所有小于256个字节的对象都使用pymalloc实现的分配器，而大的对象则使用系统的 malloc。另外Python对象，如整数，浮点数和List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数。

一 、什么是内存管理和垃圾回收

Python GC主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

现在的高级语言如java，c#等，都采用了垃圾收集机制，而不再是c，c++里用户自己管理维护内存的方式。自己管理内存极其自由，可以任意申请内存，但如同一把双刃剑，为大量内存泄露，悬空指针等bug埋下隐患。

对于一个字符串、列表、类甚至数值都是对象，且定位简单易用的语言，自然不会让用户去处理如何分配回收内存的问题。

python里也同java一样采用了垃圾收集机制，不过不一样的是:

最关键的一句话：

python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

## 四、垃圾回收与性能调优

1.手动垃圾回收

2.调高垃圾回收阈值

3.避免循环引用（手动解循环引用和使用弱引用）

你多线程是怎么实现的，三方包吗？ 答：对，直接开启多线程或者用线程池。

redis的数据类型 答：数字，字符串，字典，集合，有序集合。string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

redis主从讲一下 答：一主多从 一主一从。

1、为什么要搭建主从系统

这已经是业界很多应用的通用做法了，跟mysql一样搭建主从常见的就是可以实现读写分离，至于为什么要做读写分离从redis的本身来讲：redis的操作具有原子性，同时使用的是单线程(这样可以避免多线程之间切换消耗)操作,过多的写会阻塞线程影响redis的整体性能同时也会影响redis读性能。

此外当我们集群中或者单机的主节点宕机了或者无法跟集群的其它节点就行通讯的话，如果没有从节点则这部分缓存数据丢失，如果有从节点时我们可以用节点数据继续服务因为几乎所有的数据主节点都同步到从节点啦，可以正常使用。

2、主从特性

1、一个 master 可以拥有一个或者多个 slave。

2、slave 可以接收来自其他 slave 的连接。意思是不是就是说 slave 在接收其他的slave的连接之后成为 master ？等下我们来验证。

3、redis 复制在 master 这一端是非阻塞的，也就是说在和 slave 同步数据的时候，master 仍然可以执行客户端的操作命令而不受其影响。

4、redis 复制在 slave 这一端也是非阻塞的。在配置文件里面有 slave-serve-stale-data 这一项，如果它为 yes ，slave 在执行同步时，它可以使用老版本的数据来处理查询请求，如果是 no ，slave 将返回一个错误。在完成同步后，slave 需要删除老数据，加载新数据，在这个阶段，slave 会阻止连接进来。

5、使用复制可以避免 master 因为需要把全部的数据集写入磁盘而造成的开销，因此可以把 master 中 save 配置项全部注释掉，不让它进行保存，然后配置 slave ，让 slave 保存。虽然有这个特性，但是我们好像一般不这么做。

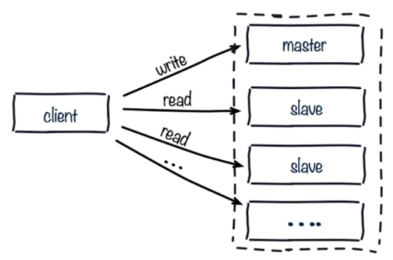
3、实现过程略。

主从实现原理剖析：

当你设置了主从关系后，slave 在第一次连接或者重新连接 master 时，slave 都会发送一条同步指令给 master ；master 接到指令后，开始启动后台保存进程保存数据，接着收集所有的数据修改指令。后台保存完了，master 就把这份数据发送给 slave，slave 先把数据保存到磁盘，然后把它加载到内存中，master 接着就把收集的数据修改指令一行一行的发给 slave，slave 接收到之后重新执行该指令，这样就实现了数据同步。slave 在与 master 失去联系后，自动的重新连接。如果 master 收到了多个 slave 的同步请求，它会执行单个后台保存来为所有的 slave 服务。

主从的缺点是什么 答：主库挂了就完蛋了。

**读写分离的问题**



1.数据复制的延迟

读写分离时，master会异步的将数据复制到slave，如果这是slave发生阻塞，则会延迟master数据的写命令，造成数据不一致的情况

解决方法：可以对slave的偏移量值进行监控，如果发现某台slave的偏移量有问题，则将数据读取操作切换到master，但本身这个监控开销比较高，所以关于这个问题，大部分的情况是可以直接使用而不去考虑的。

2.读到过期的数据

我们知道redis在删除过期key的时候，是有两种策略，第一种是懒惰型策略，即只有当redis操作这个key的时候，发现这个key过期，就会把这个key删除。第二种是定期采样一些key进行删除。

针对上面说的两种过期策略，会有个问题，即如果我们过期key的数量非常多，而采样速度根本比不上过期key的生成速度时会造成很多过期数据没有删除，但在redis里master和slave达成一种协议，slave是不能处理数据的（即不能删除数据）而我们的客户端没有及时读到到过期数据同步给master将key删除，就会导致slave读到过期的数据（这个问题已经在redis3.2版本中解决）

主从配置不一致

这个问题一般很少见，但如果有，就会发生很多诡异的问题

例如：

1. maxmemory配置不一致：这个会导致数据的丢失

原因：例如master配置4G，slave配置2G，这个时候主从复制可以成功，但，如果在进行某一次全量复制的时候，slave拿到master的RDB加载数据时发现自身的2G内存不够用，这时就会触发slave的maxmemory策略，将数据进行淘汰。更可怕的是，在高可用的集群环境下，如果我们将这台slave升级成master的时候，就会发现数据已经丢失了。

2. 数据结构优化参数不一致（例如hash-max-ziplist-entries）：这个就会导致内存不一致

原因：例如在master上对这个参数进行了优化，而在slave没有配置，就会造成主从节点内存不一致的诡异问题。

规避全量复制

首先，我们知道，redis复制有全量复制和部分复制两种（这个我前面博客有写到）而全量复制的开销是很大的。那么我们来看看，如何尽量去规避全量复制。

1.第一次全量复制

当我们某一台slave第一次去挂到master上时，是不可避免要进行一次全量复制的，那么，我们如何去想办法降低开销呢？

方案1：小主节点，例如我们把redis分成2G一个节点，这样一来，会加速RDB的生成和同步，同时还可以降低我们fork子进程的开销（master会fork一个子进程来生成同步需要的RDB文件，而fork是要拷贝内存快的，如果主节点内存太大，fork的开销就大）。

方案2：既然第一次不可以避免，那我们可以选在集群低峰的时间（凌晨）进行slave的挂载。

2.节点RunID不匹配

例如我们主节点重启（RunID发生变化），对于slave来说，它会保存之前master节点的RunID，如果它发现了此时master的RunID发生变化，那它会认为这是master过来的数据可能是不安全的，就会采取一次全量复制

解决办法：对于这类问题，我们只有是做一些故障转移的手段，例如master发生故障宕掉，我们选举一台slave晋升为master（哨兵或集群）

3.复制积压缓冲区不足

我在全量复制与部分复制那篇文章提到过，master生成RDB同步到slave，slave加载RDB这段时间里，master的所有写命令都会保存到一个复制缓冲队列里（如果主从直接网络抖动，进行部分复制也是走这个逻辑），待slave加载完RDB后，拿offset的值到这个队列里判断，如果在这个队列中，则把这个队列从offset到末尾全部同步过来，这个队列的默认值为1M。而如果发现offset不在这个队列，就会产生全量复制。

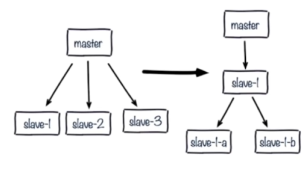
解决办法：增大复制缓冲区的配置 rel\_backlog\_size 默认1M，我们可以设置大一些，从而来加大我们offset的命中率。这个值，我们可以假设，一般我们网络故障时间一般是分钟级别，那我们可以根据我们当前的QPS来算一下每分钟可以写入多少字节，再乘以我们可能发生故障的分钟就可以得到我们这个理想的值。

规避复制风暴

什么是复制风暴？举例：我们master重启，其master下的所有slave检测到RunID发生变化，导致所有从节点向主节点做全量复制。尽管redis对这个问题做了优化，即只生成一份RDB文件，但需要多次传输，仍然开销很大。

1.单主节点复制风暴：主节点重启，多从节点全量复制

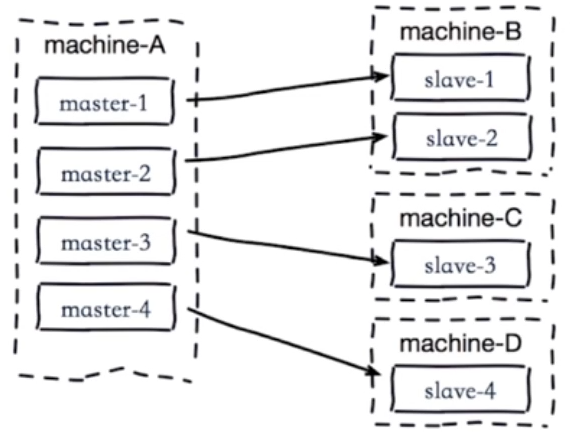
解决：更换复制拓扑如下图：



1.我们将原来master与slave中间加一个或多个slave，再在slave上加若干个slave，这样可以分担所有slave对master复制的压力。（这种架构还是有问题：读写分离的时候，slave1也发生了故障，怎么去处理？）

2.如果只是实现高可用，而不做读写分离，那当master宕机，直接晋升一台slave即可。

2.单机器复制风暴：机器宕机后的大量全量复制，如下图：



当machine-A这个机器宕机重启，会导致该机器所有master下的所有slave同时产生复制。（灾难）

解决：

1.主节点分散多机器（将master分散到不同机器上部署）

2.还有我们可以采用高可用手段（slave晋升master）就不会有类似问题了。

转载于:https://my.oschina.net/u/3371837/blog/1789452

redis你都用来做了些什么 答：做消息队列，做缓存。

一、redis是什么

　　Redis（全称：Remote Dictionary Server 远程字典服务）是一个开源的使用ANSI [C语言](https://baike.baidu.com/item/C%E8%AF%AD%E8%A8%80)编写、支持网络、可基于内存亦可持久化的日志型、Key-Value nosql [数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/103728)，并提供多种语言的API。

二、redis的应用场景

　　1、缓存

　　　　缓存现在几乎是所有中大型网站都在用的必杀技，合理的利用缓存不仅能够提升网站访问速度，还能大大降低数据库的压力。Redis提供了键过期功能，也提供了灵活的键淘汰策略，所以，现在Redis用在缓存的场合非常多。

　　2、排行榜

　　　　很多网站都有排行榜应用的，如京东的月度销量榜单、商品按时间的上新排行榜等。Redis提供的有序集合数据类构能实现各种复杂的排行榜应用。

　　3、计数器

　　　　什么是计数器，如电商网站商品的浏览量、视频网站视频的播放数等。为了保证数据实时效，每次浏览都得给+1，并发量高时如果每次都请求数据库操作无疑是种挑战和压力。

　　　　Redis提供的incr命令来实现计数器功能，内存操作，性能非常好，非常适用于这些计数场景。

　　4、分布式会话

　　　　集群模式下，在应用不多的情况下一般使用容器自带的session复制功能就能满足，当应用增多相对复杂的系统中，一般都会搭建以Redis等内存数据库为中心的session服务，session不再由容器管理，而是由session服务及内存数据库管理。

　　5、分布式锁

　　　　　　在很多互联网公司中都使用了分布式技术，分布式技术带来的技术挑战是对同一个资源的并发访问，如全局ID、减库存、秒杀等场景，并发量不大的场景可以使用数据库的悲观锁、乐观锁来实现，但在并发量高的场合中，利用数据库锁来控制资源的并发访问是不太理想的，大大影响了数据库的性能。可以利用Redis的setnx功能来编写分布式的锁，如果设置返回1说明获取锁成功，否则获取锁失败，实际应用中要考虑的细节要更多。

　　6、 社交网络

　　　　点赞、踩、关注/被关注、共同好友等是社交网站的基本功能，社交网站的访问量通常来说比较大，而且传统的关系数据库类型不适合存储这种类型的数据，Redis提供的哈希、集合等数据结构能很方便的的实现这些功能。

　　7、最新列表

　　　　Redis列表结构，LPUSH可以在列表头部插入一个内容ID作为关键字，LTRIM可用来限制列表的数量，这样列表永远为N个ID，无需查询最新的列表，直接根据ID去到对应的内容页即可。

　　8、消息系统(不推荐使用)

　　　　消息队列是大型网站必用中间件，如ActiveMQ、RabbitMQ、Kafka等流行的消息队列中间件，主要用于业务解耦、流量削峰及异步处理实时性低的业务。Redis提供了发布/订阅及阻塞队列功能，能实现一个简单的消息队列系统。

三、redis的优点和缺点

　　优点：

　　（1）速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O1)

　　（2）支持丰富数据类型，支持 string，list，set，Zset，hash 等

　　（3）支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

　　（4）丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除，支持数据的持久化，支持数据的备份，即master-slave模式的数据备份。

　　缺点：

　　（1）由于 Redis 是内存数据库，所以，单台机器，存储的数据量，跟机器本身的内存大小。虽然 Redis 本身有 Key 过期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。

　　（2）redis是单线程的，单台服务器无法充分利用多核服务器的CPU。

四、redis为什么快？

　　1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

　　 2、数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；

　　 3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

　　 4、使用多路I/O复用模型，非阻塞IO；

　　 5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求；

## 1. 获取最新的n个数据，2. 获取 TOP N 的数据，3. 设置精准的抢购时间，4. 轻松实现计数器

## ，5. 去除重复值，6. 利用set命令制作反垃圾系统，7. 利用发布订阅的中的SUBSCRIBE和PUBLISH命令构建实时消息推送系统，8. 构建队列系统，

redis可以用来做什么？以博客帖子为例大概说说，当然redis的其他功能还有很多，后续在做介绍。

Redis 是互联网技术领域使用最为广泛的存储中间件，它是「**Re**mote **Di**ctionary **S**ervice」的首字母缩写，也就是「远程字典服务」。

记录帖子的点赞数、评论数和点击数 (hash) --针对帖子的一对多记录， key为数名称，value为各种数的值。

记录用户的帖子 ID 列表 (排序)，便于快速显示用户的帖子列表 (zset) -- 记录id而不是文章是为了需要减少内存，同时zset 的score可以用来排序。

记录帖子的标题、摘要、作者和封面信息，用于列表页展示 (hash) -- 针对帖子的一对多记录，key为必要信息的名称，value为各种值。

记录帖子的点赞用户 ID 列表，评论 ID 列表，用于显示和去重计数 (zset) -- 单个字段记录，不重复。

缓存近期热帖内容 (帖子内容空间占用比较大)，减少数据库压力 (hash) -- 热帖数量不多，所以内容全部缓存，key为id，value为帖子内容。

记录帖子的相关文章 ID，根据内容推荐相关帖子 (list) -- 相关文章，有相关度的概念，所以需要有序，而list可以作为队列和栈来使用。

如果帖子 ID 是整数自增的，可以使用 Redis 来分配帖子 ID(计数器)。 收藏集和帖子之间的关系 (zset) -- 有序不重复。

记录热榜帖子 ID 列表，总热榜和分类热榜 (zset) -- 有序不重复。

缓存用户行为历史，进行恶意行为过滤 (zset,hash) -- 有序不重复，针对用户一对多行为记录。

redis数据怎么持久化存储 答：忘记了。

从严格意义上说，Redis服务提供四种持久化存储方案：RDB、AOF、虚拟内存（VM）和　DISKSTORE。虚拟内存（VM）方式，从Redis Version 2.4开始就被官方明确表示不再建议使用，Version 3.2版本中更找不到关于虚拟内存（VM）的任何配置范例，Redis的主要作者Salvatore Sanfilippo还专门写了一篇论文，来反思Redis对虚拟内存（VM）存储技术的支持问题。

　　至于DISKSTORE方式，是从Redis Version 2.8版本开始提出的一个存储设想，到目前为止Redis官方也没有在任何stable版本中明确建议使用这用方式。在Version 3.2版本中同样找不到对于这种存储方式的明确支持。从网络上能够收集到的各种资料来看，DISKSTORE方式和RDB方式还有着一些千丝万缕的联系，不过各位读者也知道，除了官方文档以外网络资料很多就是大抄。

> 为什么要做持久化存储?

持久化存储是将 Redis 存储在内存中的数据存储在硬盘中，实现数据的永久保存。我们都知道 Redis 是一个基于内存的 nosql 数据库，内存存储很容易造成数据的丢失，因为当服务器关机等一些异常情况都会导致存储在内存中的数据丢失。

> 持久化存储分类

在 Redis 中，持久化存储分为两种。一种是 aof 日志追加的方式，另外一种是 rdb 数据快照的方式。

> RDB持久化存储

什么是RDB持久化存储

RDB持久化存储即是将redis存在内存中的数据以快照的形式保存在本地磁盘中。

.RDB持久化存储分为自动备份和手动备份

1.手动备份通过 save 命令和 bgsave 命令。save是同步阻塞，而 bgsave 是非阻塞(阻塞实际发生在 fork 的子进程中)。因此，在我们实际过程中大多是使用bgsave命令实现备份.

RDB持久化的优势与劣势

优势:

1.文件实现的数据快照，全量备份，便于数据的传输.比如我们需要把A服务器上的备份文件传输到B服务器上面，直接将rdb文件拷贝即可.

2.文件采用压缩的二进制文件，当重启服务时加载数据文件，比aof方式更快.

劣势:

1.rbd采用加密的二进制格式存储文件，由于Redis各个版本之间的兼容性问题也导致rdb由版本兼容问题导致无法再其他的Redis版本中使用.

2.时效性差，容易造成数据的不完整性.因为rdb并不是实时备份，当某个时间段Redis服务出现异常，内存数据丢失，这段时间的数据是无法恢复的，因此易导致数据的丢失.

redis的持久化——RDB和AOF。redis有两种方式支持持久化，分别是RDB和AOF。

redis的哨兵讲一下 答：哨兵用来实现高可用巴拉巴拉。

sentinel是redis高可用的解决方案，sentinel系统（N个sentinel实例，N >= 1）可以监视一个或者多个redis master服务，以及这些master服务的所有从服务；当某个master服务下线时，自动将该master下的某个从服务升级为master服务替代已下线的master服务继续处理请求。

Redis的主从[架构](http://lib.csdn.net/base/16)，如果master发现故障了，还得手动将slave切换成master继续服务，手动的方式容易造成失误，导致数据丢失，那Redis有没有一种机制可以在master和slave进行监控，并在master发送故障的时候，能自动将slave切换成master呢？有的，那就是哨兵。

哨兵的作用：

1、监控redis进行状态，包括master和slave

2、当master down机，能自动将slave切换成master

为了达到redis的高可用，有两种部署方式：主从复制+哨兵机制；集群模式。哨兵机制是redis2.8开始支持。集群模式是redis3.0开始支持。

    主从复制的意义：

    主从复制可以把主节点的数据复制给从节点。从节点可以备份主节点的数据，起到主节点down调，顶上来接替主节点工作的作用。也可以起到分担主节点读压力的作用。

    没有哨兵机制的时候，主从复制结构部署存在的问题是什么？也可以说redis主节点发生故障如何解决？

    如果主节点down调，主从切换需要人工介入。

    主从切换步骤为：

    1、启用从节点为主节点。命令：slaveof no one

    2、旧主节点的其他从节点变成新主节点的从节点。命令：slaveof new master

    3、通知应用方redis主节点变成了新主节点。 修改客户端调用的地址并重启客户端。

    4、旧主节点变成新主节点的从节点。  命令：slaveof new master

    哨兵机制存在的意义：

    为了实现redis故障转移的自动化。自动发现，自动转移。不需要人工参与。

功能作用

1. 监控(monitoring)：Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
2. 提醒(Notifation)：当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
3. 自动故障转移(Automatic failover)：当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会将失效主服务器的其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。

## 之前有说到redis的主从分离架构以实现提高redis的高水平扩展能力，但是单单是这样的主从架构是存在着一些问题的：

### master（主）节点挂了会发生什么？

1. master挂了，那么master下的从节点同样的处于不可用状态了。即master那一片都挂了。因为slave（从）节点不能再接收到新的数据

### slave节点挂了会怎么样？

1. 如果是一个slave节点挂了，那么还有其他的slave节点对外提供服务，不至于所有的请求都发向后台的数据库中导致数据库压力突然增大。

### redis进程挂掉了会怎么样？

1. 这就比较厉害了，进程挂掉了redis就没用了，有的时候redis进程挂掉了，而你有恰好没有给redis设置数据的持久化，redis可能会丢失较多的数据，除此外这台机器的这个进程的redis肯定也不能对外提供服务了。带来的问题就和前面两个一样的了，如果是master的进程，同master节点挂了；如果是slave的进程，同slave节点挂了。

### 解决办法：

1. 哨兵，对zookeeper有了解的话会很容易理解这个机制，在这里简要的说明一下哨兵的工作。这里以zookeeper来做哨兵模式的讲解。首先zookeeper是一个分布式锁的服务，根据CPA理论，在分布式系统中，系统只能达到CAP理论中的两个要求，无法满足全部三个。
2. 这个不难理解。C：数据的一致性，P：数据的分区容错性，A:数据的可用性。zookeeper实现的主要是CAP中的一致性（C）和分区容错性（P）不过你就算姑且认为zookeeper主要实现的是一致性（C）也ok，zookeeper在A方面的确不算强调的地方。同时由于zookeeper的非高可用性，zookeeper被认为不适合作为服务发现的系统。当然并非说zookeeper不能用，只是他的可用性不算好而已。zookeeper实现其强一致

Redis Sentinel（哨兵集群）是一种高可用的redis部署方案。在集群中的redis-master服务挂掉时，无需人为干预，即可通过哨兵集群的自我调整，实现redis服务的持续可用。

哨兵集群的功能:

1. 监控：持续监测集群中的master和slave是否可用。
2. 通知：通知系统管理员，集群中的部分服务不可用。
3. 故障切换：若master挂掉，哨兵集群会启动故障切换，选出其中一个slave作为新的master。使用redis的应用程序也会被告知使用新的master。
4. 提供配置信息：哨兵集群为redis客户提供服务发现功能，客户端访问哨兵集群，会被告知当前的master地址。

哨兵集群的分布式特性：多个哨兵进程协同工作。

多个哨兵进程的好处：

1. 故障需要多个进程同时确认，降低了发现故障时出错的概率。
2. 部分哨兵进程挂掉后，集群同样可以维持工作，从而使系统更加健壮。

 工作原理：

首先来了解哨兵集群中两个重要的基本概念：

SDOWN(主观宕机)：哨兵实例的PING命令没有返回+PONG，-LOADING error，-MASTERDOWN error。

ODOWN(客观宕机)：足够数量（大于等于sentinel.conf文件中的配置）的哨兵实例判断出发生了SDOWN。

发生客观宕机后开始故障切换。

哨兵集群可以自动发现各个哨兵实例以及redis master的slave

python现在是什么版本了 答：3.8

3.7和3.8有什么区别 答：不知道

很多人在学习了基本的Python语言知识后，就转入应用阶段了，后期很少对语言本身的新变化、新内容进行跟踪学习和知识更新，甚至连已经发布了好几年的Python3.6的新特性都缺乏了解。

本文列举了Python3.6、3.7、3.8三个版本的新特性，学习它们有助于提高对Python的了解，跟上最新的潮流。

一、Python3.6新特性

1、新的格式化字符串方式

新的格式化字符串方式，即在普通字符串前添加 f 或 F 前缀，其效果类似于str.format()。比如

name = "red"

print(f"He said his name is {name}.")

# 'He said his name is red.'

相当于：

print("He said his name is {name}.".format(\*\*locals()))

此外，此特性还支持嵌套字段，比如：

import decimal

width = 10

precision = 4

value = decimal.Decimal("12.34567")

print(f"result: {value:{width}.{precision}}")

#'result: 12.35'

2、变量声明语法

可以像下面一样声明一个变量并指定类型：

from typing import List, Dict

primes: List[int] = []

captain: str # 此时没有初始值

class Starship:

stats: Dict[str, int] = {}

3、数字的下划线写法

允许在数字中使用下划线，以提高多位数字的可读性。

a = 1\_000\_000\_000\_000\_000 # 1000000000000000

b = 0x\_FF\_FF\_FF\_FF # 4294967295

除此之外，字符串格式化也支持\_选项，以打印出更易读的数字字符串：

'{:\_}'.format(1000000) # '1\_000\_000'

'{:\_x}'.format(0xFFFFFFFF) # 'ffff\_ffff'

4、异步生成器

在Python3.5中，引入了新的语法 async 和 await 来实现协同程序。但是有个限制，不能在同一个函数体内同时使用 yield 和 await。Python3.6中，这个限制被放开了，允许定义异步生成器：

async def ticker(delay, to):

"""Yield numbers from 0 to \*to\* every \*delay\* seconds."""

for i in range(to):

yield i

await asyncio.sleep(delay)

5、异步解析器

允许在列表list、集合set 和字典dict 解析器中使用 async 或 await 语法。

result = [i async for i in aiter() if i % 2]

result = [await fun() for fun in funcs if await condition()]

6、新增加模块

标准库（The Standard Library）中增加了一个新的模块：secrets。该模块用来生成一些安全性更高的随机数，用于管理passwords, account authentication, security tokens, 以及related secrets等数据。

7、其他新特性

* 新的 PYTHONMALLOC 环境变量允许开发者设置内存分配器，以及注册debug钩子等。
* asyncio模块更加稳定、高效，并且不再是临时模块，其中的API也都是稳定版的了。
* typing模块也有了一定改进，并且不再是临时模块。
* datetime.strftime 和 date.strftime 开始支持ISO 8601的时间标识符%G, %u, %V。
* hashlib 和 ssl 模块开始支持OpenSSL1.1.0。
* hashlib模块开始支持新的hash算法，比如BLAKE2, SHA-3 和 SHAKE。
* Windows上的 filesystem 和 console 默认编码改为UTF-8。
* json模块中的 json.load() 和 json.loads() 函数开始支持 binary 类型输入。

更多内容参考官方文档：[What's New In Python 3.6](https://docs.python.org/3.6/whatsnew/3.6.html)

二、Python3.7新特性

Python 3.7于2018年6月27日发布， 包含许多新特性和优化，增添了众多新的类，可用于数据处理、针对脚本编译和垃圾收集的优化以及更快的异步I/O，主要如下：

* 用类处理数据时减少样板代码的数据类。
* 一处可能无法向后兼容的变更涉及处理生成器中的异常。
* 面向解释器的“开发模式”。
* 具有纳秒分辨率的时间对象。
* 环境中默认使用UTF-8编码的UTF-8模式。
* 触发调试器的一个新的内置函数。

1、新增内置函数breakpoint()

使用该内置函数，相当于通过代码的方式设置了断点，会自动进入Pbd调试模式。

如果在环境变量中设置PYTHONBREAKPOINT=0会忽略此函数。并且，pdb 只是众多可用调试器之一，你可以通过设置新的 PYTHONBREAKPOINT 环境变量来配置想要使用的调试器。

下面有一个简单例子，用户需要输入一个数字，判断它是否和目标数字一样：

"""猜数字游戏"""

def guess(target):

user\_guess = input("请输入你猜的数 >>> ")

if user\_guess == target:

return "你猜对了!"

else:

return "猜错了"

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

a = 100

print(guess(a))

不幸的是，即使猜的数和目标数一样，打印的结果也是‘猜错了’，并且没有任何异常或错误信息。

为了弄清楚发生了什么，我们可以插入一个断点，来调试一下。以往一般通过print大法或者IDE的调试工具，但现在我们可以使用 breakpoint()。

"""猜数字游戏"""

def guess(target):

user\_guess = input("请输入你猜的数 >>> ")

breakpoint() //加入这一行

if user\_guess == target:

return "你猜对了!"

else:

return "猜错了"

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

a = 100

print(guess(a))

在 pdb 提示符下，我们可以调用 locals() 来查看当前的本地作用域的所有变量。（pdb 有大量的命令，你也可以在其中运行正常的Python 语句）

请输入你猜的数 >>> 100

> d:\work\for\_test\py3\_test\test.py(7)guess()

-> if user\_guess == target:

(Pdb) locals()

{'target': 100, 'user\_guess': '100'}

(Pdb) type(user\_guess)

<class 'str'>

搞明白了，target是一个整数，而user\_guess 是一个字符串，这里发生了类型对比错误。

2、类型和注解

从 Python 3.5 开始，类型注解就越来越受欢迎。对于那些不熟悉类型提示的人来说，这是一种完全可选的注释代码的方式，以指定变量的类型。

什么是注解？它们是关联元数据与变量的语法支持，可以是任意表达式，在运行时被 Python 计算但被忽略。注解可以是任何有效的 Python 表达式。

下面是个对比的例子：

# 不带类型注解

def foo(bar, baz):

# 带类型注解

def foo(bar: 'Describe the bar', baz: print('random')) -> 'return thingy':

上面的做法，其实是Python对自身弱类型语言的强化，希望获得一定的类型可靠和健壮度，向Java等语言靠拢。

在 Python 3.5 中，注解的语法获得标准化，此后，Python 社区广泛使用了注解类型提示。

但是，注解仅仅是一种开发工具，可以使用 PyCharm 等 IDE 或 Mypy 等第三方工具进行检查，并不是语法层面的限制。

我们前面的猜数程序如果添加类型注解，它应该是这样的：

"""猜数字游戏"""

def guess(target:str):

user\_guess:str = input("请输入你猜的数 >>> ")

breakpoint()

if user\_guess == target:

return "你猜对了!"

else:

return "猜错了"

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

a:int = 100

print(guess(a))

PyCharm会给我们灰色的规范错误提醒，但不会给红色的语法错误提示。

用注解作为类型提示时，有两个主要问题：启动性能和前向引用。

* 在定义时计算大量任意表达式相当影响启动性能，而且 typing 模块非常慢
* 你不能用尚未声明的类型来注解

typing 模块如此缓慢的部分原因是，最初的设计目标是在不修改核心 CPython 解释器的情况下实现 typing 模块。随着类型提示变得越来越流行，这一限制已经被移除，这意味着现在有了对 typing 的核心支持。

而对于向前引用，看下面的例子：

class User:

def \_\_init\_\_(self, name: str, prev\_user: User) -> None:

pass

错误在于 User类型还没有被声明，此时的 prev\_user 不能定义为 User 类型。

为了解决这个问题，Python3.7 将注解的评估进行了推迟。并且，这项改动向后不兼容，需要先导入annotations，只有到Python 4.0后才会成为默认行为。

from \_\_future\_\_ import annotations

class User:

def \_\_init\_\_(self, name: str, prev\_user: User) -> None:

pass

或者如下面的例子：

class C:

def validate\_b(self, obj: B) -> bool:

...

class B:

...

3、新增dataclasses模块

这个特性可能是 Python3.7以后比较常用的，它有什么作用呢？

假如我们需要编写一个下面的类：

from datetime import datetime

import dateutil

class Article(object):

def \_\_init\_\_(self, \_id, author\_id, title, text, tags=None,

created=datetime.now(), edited=datetime.now()):

self.\_id = \_id

self.author\_id = author\_id

self.title = title

self.text = text

self.tags = list() if tags is None else tags

self.created = created

self.edited = edited

if type(self.created) is str:

self.created = dateutil.parser.parse(self.created)

if type(self.edited) is str:

self.edited = dateutil.parser.parse(self.edited)

def \_\_eq\_\_(self, other):

if not isinstance(other, self.\_\_class\_\_):

return NotImplemented

return (self.\_id, self.author\_id) == (other.\_id, other.author\_id)

def \_\_lt\_\_(self, other):

if not isinstance(other, self.\_\_class\_\_):

return NotImplemented

return (self.\_id, self.author\_id) < (other.\_id, other.author\_id)

def \_\_repr\_\_(self):

return '{}(id={}, author\_id={}, title={})'.format(

self.\_\_class\_\_.\_\_name\_\_, self.\_id, self.author\_id, self.title)

大量的初始化属性要定义默认值，可能还需要重写一堆魔法方法，来实现类实例的打印、比较、排序和去重等功能。

如果使用dataclasses进行改造，可以写成这个样子：

from dataclasses import dataclass, field

from typing import List

from datetime import datetime

import dateutil

@dataclass(order=True) //注意这里

class Article(object):

\_id: int

author\_id: int

title: str = field(compare=False)

text: str = field(repr=False, compare=False)

tags: List[str] = field(default=list(), repr=False, compare=False)

created: datetime = field(default=datetime.now(), repr=False, compare=False)

edited: datetime = field(default=datetime.now(), repr=False, compare=False)

def \_\_post\_init\_\_(self):

if type(self.created) is str:

self.created = dateutil.parser.parse(self.created)

if type(self.edited) is str:

self.edited = dateutil.parser.parse(self.edited)

这使得类不仅容易设置，而且当我们创建一个实例并打印出来时，它还可以自动生成优美的字符串。在与其他类实例进行比较时，它也会有适当的行为。这是因为dataclasses除了帮我们自动生成 \_\_init\_\_ 方法外，还生成了一些其他特殊方法，如 repr、eq 和 hash 等。

Dataclasses 使用字段 field来完提供默认值，手动构造一个 field() 函数能够访问其他选项，从而更改默认值。例如，这里将 field 中的 default\_factory 设置为一个 lambda 函数，该函数提示用户输入其名称。

from dataclasses import dataclass, field

class User:

name: str = field(default\_factory=lambda: input("enter name"))

4、生成器异常处理

在Python 3.7中，生成器引发StopIteration异常后，StopIteration异常将被转换成RuntimeError异常，那样它不会悄悄一路影响应用程序的堆栈框架。这意味着如何处理生成器的行为方面不太敏锐的一些程序会在Python 3.7中抛出RuntimeError。在Python 3.6中，这种行为生成一个弃用警告；在Python 3.7中，它将生成一个完整的错误。

一个简易的方法是使用try/except代码段，在StopIteration传播到生成器的外面捕获它。更好的解决方案是重新考虑如何构建生成器――比如说，使用return语句来终止生成器，而不是手动引发StopIteration。

5、开发模式

Python解释器添加了一个新的命令行开关：-X，让开发人员可以为解释器设置许多低级选项。

这种运行时的检查机制通常对性能有重大影响，但在调试过程中对开发人员很有用。

-X 激活的选项包括：

* asyncio模块的调试模式。这为异步操作提供了更详细的日志记录和异常处理，而异常操作可能很难调试或推理。
* 面向内存分配器的调试钩子。这对于编写CPython扩展件的那些人很有用。它能够实现更明确的运行时检查，了解CPython如何在内部分配内存和释放内存。
* 启用faulthandler模块，那样发生崩溃后，traceback始终转储出去。

6、 高精度时间函数

Python 3.7中一类新的时间函数返回纳秒精度的时间值。尽管Python是一种解释型语言，但是Python的核心开发人员维克多•斯廷纳（Victor Stinner）主张报告纳秒精度的时间。最主要的原因是，在处理转换其他程序（比如数据库）记录的时间值时，可以避免丢失精度。

新的时间函数使用后缀\_ns。比如说，time.process\_time()的纳秒版本是time.process\_time\_ns()。请注意，并非所有的时间函数都有对应的纳秒版本。

7、其他新特性

* 字典现在保持插入顺序。这在 3.6 中是非正式的，但现在成为了官方语言规范。在大多数情况下，普通的 dict 能够替换 collections.OrderedDict。
* .pyc 文件具有确定性，支持可重复构建 —— 也就是说，总是为相同的输入文件生成相同的 byte-for-byte 输出。
* 新增contextvars模块，针对异步任务提供上下文变量。
* \_\_main\_\_中的代码会显示弃用警告（DeprecationWarning）。
* 新增UTF-8模式。在Linux/Unix系统，将忽略系统的locale，使用UTF-8作为默认编码。在非Linux/Unix系统，需要使用-X utf8选项启用UTF-8模式。
* 允许模块定义\_\_getattr\_\_、\_\_dir\_\_函数，为弃用警告、延迟import子模块等提供便利。
* 新的线程本地存储C语言API。
* 更新Unicode数据到11.0。

三、Python3.8新特性

Python3.8版本于2019年10月14日发布，以下是 Python 3.8 相比 3.7 的新增特性。

1、海象赋值表达式

新的语法 :=，将值赋给一个更大的表达式中的变量。它被亲切地称为 “海象运算符”(walrus operator)，因为它长得像海象的眼睛和象牙。

“海象运算符” 在某些时候可以让你的代码更整洁，比如：

在下面的示例中，赋值表达式可以避免调用 len () 两次:

if (n := len(a)) > 10:

print(f"List is too long ({n} elements, expected <= 10)")

类似的好处还可体现在正则表达式匹配中需要使用两次匹配对象的情况中，一次检测用于匹配是否发生，另一次用于提取子分组:

discount = 0.0

if (mo := re.search(r'(\d+)% discount', advertisement)):

discount = float(mo.group(1)) / 100.0

此运算符也可用于配合 while 循环计算一个值，来检测循环是否终止，而同一个值又在循环体中再次被使用的情况:

# Loop over fixed length blocks

while (block := f.read(256)) != '':

process(block)

或者出现于列表推导式中，在筛选条件中计算一个值，而同一个值又在表达式中需要被使用:

[clean\_name.title() for name in names

if (clean\_name := normalize('NFC', name)) in allowed\_names]

请尽量将海象运算符的使用限制在清晰的场合中，以降低复杂性并提升可读性。

2、仅限位置形参

新增一个函数形参语法 / 用来指明某些函数形参必须使用仅限位置而非关键字参数的形式。

这种标记语法与通过 help () 所显示的使用 Larry Hastings 的 Argument Clinic 工具标记的 C 函数相同。

在下面的例子中，形参 a 和 b 为仅限位置形参，c 或 d 可以是位置形参或关键字形参，而 e 或 f 要求为关键字形参:

def f(a, b, /, c, d, \*, e, f):

print(a, b, c, d, e, f)

以下是合法的调用:

f(10, 20, 30, d=40, e=50, f=60)

但是，以下均为不合法的调用:

f(10, b=20, c=30, d=40, e=50, f=60) # b 不可以是一个关键字参数

f(10, 20, 30, 40, 50, f=60) # e 必须是一个关键字参数

这种标记形式的一个用例是它允许纯 Python 函数完整模拟现有的用 C 代码编写的函数的行为。例如，内置的 pow () 函数不接受关键字参数:

def pow(x, y, z=None, /):

"Emulate the built in pow() function"

r = x \*\* y

return r if z is None else r%z

另一个用例是在不需要形参名称时排除关键字参数。例如，内置的 len () 函数的签名为 len (obj, /)。这可以排除如下这种笨拙的调用形式:

len(obj='hello') # The "obj" keyword argument impairs readability

另一个益处是将形参标记为仅限位置形参将允许在未来修改形参名而不会破坏客户的代码。例如，在 statistics 模块中，形参名 dist 在未来可能被修改。这使得以下函数描述成为可能:

def quantiles(dist, /, \*, n=4, method='exclusive')

...

由于在 / 左侧的形参不会被公开为可用关键字，其他形参名仍可在 \*\*kwargs 中使用:

>>> def f(a, b, /, \*\*kwargs):

... print(a, b, kwargs)

...

>>> f(10, 20, a=1, b=2, c=3) # a and b are used in two ways

10 20 {'a': 1, 'b': 2, 'c': 3}

这极大地简化了需要接受任意关键字参数的函数和方法的实现。例如，下面是 collections 模块中的代码摘录：

class Counter(dict):

def \_\_init\_\_(self, iterable=None, /, \*\*kwds):

# Note "iterable" is a possible keyword argument

3、f 字符串支持 =

增加 = 说明符用于 f-string。形式为 f'{expr=}' 的 f 字符串将扩展表示为表达式文本，加一个等于号，再加表达式的求值结果。例如：

>>> user = 'eric\_idle'

>>> member\_since = date(1975, 7, 31)

>>> f'{user=} {member\_since=}'

"user='eric\_idle' member\_since=datetime.date(1975, 7, 31)"

f 字符串格式说明符允许更细致地控制所要显示的表达式结果:

>>> delta = date.today() - member\_since

>>> f'{user=!s} {delta.days=:,d}'

'user=eric\_idle delta.days=16,075'

= 说明符将输出整个表达式，以便详细演示计算过程:

>>> print(f'{theta=} {cos(radians(theta))=:.3f}')

theta=30 cos(radians(theta))=0.866

4、 typing模块的改进

Python是动态类型语言，但可以通过typing模块添加类型提示，以便第三方工具验证Python代码。Python 3.8给typing添加了一些新元素，因此它能够支持更健壮的检查：

* final修饰器和Final类型标注表明，被修饰或被标注的对象在任何时候都不应该被重写、继承，也不能被重新赋值。
* Literal类型将表达式限定为特定的值或值的列表（不一定是同一个类型的值）。
* TypedDict可以用来创建字典，其特定键的值被限制在一个或多个类型上。注意这些限制仅用于编译时确定值的合法性，而不能在运行时进行限制。

5、多进程共享内存

multiprocessing模块新增SharedMemory类，可以在不同的Python进城之间创建共享的内存区域。

在旧版本的Python中，进程间共享数据只能通过写入文件、通过网络套接字发送，或采用Python的pickle模块进行序列化等方式。共享内存提供了进程间传递数据的更快的方式，从而使得Python的多处理器和多内核编程更有效率。

共享内存片段可以作为单纯的字节区域来分配，也可以作为不可修改的类似于列表的对象来分配，其中能保存数字类型、字符串、字节对象、None对象等一小部分Python对象。

6、 新版本的pickle协议

Python的pickle模块提供了一种序列化和反序列化Python数据结构或实例的方法，可以将字典原样保存下来供以后读取。不同版本的Python支持的pickle协议不同，而3.8版本的支持范围更广、更强大、更有效的序列化。

Python 3.8引入的第5版pickle协议可以用一种新方法pickle对象，它能支持Python的缓冲区协议，如bytes、memoryviews或Numpy array等。新的pickle避免了许多在pickle这些对象时的内存复制操作。

NumPy、Apache Arrow等外部库在各自的Python绑定中支持新的pickle协议。新的pickle也可以作为Python 3.6和3.7的插件使用，可以从PyPI上安装。

7、性能改进

* 许多内置方法和函数的速度都提高了20%~50%，因为之前许多函数都需要进行不必要的参数转换。
* 一个新的opcode缓存可以提高解释器中特定指令的速度。但是，目前实现了速度改进的只有LOAD\_GLOBAL opcode，其速度提高了40%。以后的版本中也会进行类似的优化。
* 文件复制操作如shutil.copyfile()和shutil.copytree()现在使用平台特定的调用和其他优化措施，来提高操作速度。
* 新创建的列表现在平均比以前小了12%，这要归功于列表构造函数如果能提前知道列表长度的情况下，可以进行优化。
* Python 3.8中向新型类（如class A(object)）的类变量中的写入操作变得更快。operator.itemgetter()和collections.namedtuple()也得到了速度优化。

装饰器是什么 答：两大原则。python装饰器简单来说可以在不改变某个函数内部实现和原来调用方式的前提下对该函数增加一些附件的功能，提供了对该函数功能的扩展。

深浅拷贝 答：一层两层的说。

* 直接赋值：其实就是对象的引用（别名）。
* 浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。
* 深拷贝(deepcopy)： copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。
* 拷贝就是一个变量的值传给另外一个变量。在python 中 id()方法可以查看存放变量的内存地址，这为我们下面理解深浅copy提供了便利。

浅拷贝是指把存放变量的地址值传给被赋值，最后两个变量引用了同一份地址；

深拷贝是指被赋值的变量开辟了另一块地址用来存放要赋值的变量的值（内容）。在python中引用copy模块，copy模块中有deepcopy()方法，调用它完成变量的深copy，观察变量地址。

Python当中对于拷贝，分为两种类型。一种是数字和字符串，另一种就是列表、元组、字典等其他类型了。

# **一、数字和字符串的拷贝**

## **1、赋值**

**结论：对于通过用 = 号赋值，数字和字符串在内存当中用的都是同一块地址。对于赋值，字典、列表、元组等其他类型用的内存地址不会变化。**

## 2、浅拷贝

**结论：对于浅拷贝，数字和字符串在内存当中用的也是同一块地址。所以对于浅拷贝，字典、列表、元组等类型，它们只拷贝第一层地址。**

## 3、深拷贝

**所以综上所述，对于数字和字符串的赋值、浅拷贝、深拷贝在内存当中用的都是同一块地址。对于深拷贝，字典、列表、元组等类型，它里面嵌套多少层，就会拷贝多少层出来，但是最底层的数字和字符串地址不变。**

一个函数超过五秒就退出 答：不记得了。

最近写bug时遇到一个关于函数执行超时的问题，就是某一步执行时间过长，或者程序直接卡死了，找到了三种解决方法，现在叙述下，帮助需要的人。（1）使用eventlet。**（2）使用signal设置装饰器。（3）使用threading（**使用threading的timer定时器**）。**

linux怎么查看端口 答：支支吾吾

linux怎么查看内存占用情况 答：支支吾吾

ping这个命令的底层协议是什么 答：tcp。

使用的是[ICMP](http://www.so.com/s?q=ICMP&ie=utf-8&src=internal_wenda_recommend_text)协议，是“Internet Control Message Protocol”（Internet控制消息协议）的缩写，是[TCP/IP协议族](http://www.so.com/s?q=TCP%2FIP%E5%8D%8F%E8%AE%AE%E6%97%8F&ie=utf-8&src=internal_wenda_recommend_text)的一个子协议，用于在IP主机、[路由器](http://www.so.com/s?q=%E8%B7%AF%E7%94%B1%E5%99%A8&ie=utf-8&src=internal_wenda_recommend_text)之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。

它是用来检查网络是否通畅或者网络连接速度的命令。它所利用的原理是这样的：利用网络上机器IP地址的唯一性，给目标IP地址发送一个数据包，再要求对方返回一个同样大小的数据包来确定两台网络机器是否连接相通，时延是多少。

tcp建立连接 答：三次握手四次挥手

tcp的回执是什么包 答：不记得了。

传输控制协议（Transmission Control Protocol, TCP）

TCP协议主为了在主机间实现高可靠性的包交换传输协议。本文将描述协议标准和实现的一些方法。因为计算机网络在现代社会中已经是不可缺少的了，TCP协议主要在网络不可靠的时候完成通信，对军方可能特别有用，但是对于政府和商用部门也适用。

  TCP是面向连接的端到端的可靠协议。它支持多种网络应用程序。TCP对下层服务没有多少要求，它假定下层只能提供不可靠的数据报服务，它可以在多种硬件构成的网络上运行。下面的图是TCP在层次式结构中的位置，它的下层是IP协议，TCP可以根据IP协议提供的服务传送大小不定的数据，IP协议负责对数据进行分段，重组，在多种网络中传送。

TCP的上面就是应用程序，下面是IP协议，上层接口包括一系列类似于操作系统中断的调用。对于上层应用程序来说，TCP应该能够异步传送数据。下层接口我们假定为IP协议接口。为了在并不可靠的网络上实现面向连接的可靠的传送数据，TCP必须解决可靠性，流量控制的问题，必须能够为上层应用程序提供多个接口，同时为多个应用程序提供数据，同时TCP必须解决连接问题，这样TCP才能称得上是面向连接的，最后，TCP也必须能够解决通信安全性的问题。

网络环境包括由网关（或其它设备）连接的网络，网络可以是局域网也可以是一些城域网或广域网，但无论它们是什么，它们必须是基于包交换的。主机上不同的协议有不同的端口号，一对进程通过这个端口号进行通信。这个通信不包括计算机内的I/O操作，只包括在网络上进行的操作。

  网络上的计算机被看作包传送的源和目的结点。特别应该注意的是：计算机中的不同进程可能同时进行通信，这时它们会用端口号进行区别，不会把发向A进程的数据由B进程接收的。

进程为了传送数据会调用TCP，将数据和相应的参数传送给TCP，于是TCP会将数据传送到目的TCP那里，当然这是通过将TCP包打包在IP包内在网络上传送达到的。

  接收方TCP在接收到数据后会通信上层应用程序，TCP会保证接收数据顺序的正确性。虽然下层协议可能不会保证顺序是正确的。这里需要说明的是网关在接收到这个包后，会将包解开，看看是不是已经到目的地了，如果没有到，应该走什么路由达到目的地，在决定后，网关会根据下一个网络内的协议情况再次将TCP包打包传送，如果需要，还要把这个包再次分成几段再传送。

  这个落地检查的过程是一个耗时的过程。从上面，我们可以看出TCP传送的基本过程，当然具体过程可能要复杂得多。

在实现TCP的主机上，TCP可以被看成是一个模块，和文件系统区别不大，TCP也可以调用一些操作系统的功能，TCP不直接和网络打交道，控制网络的任务由专门的设备驱动模块完成。

  TCP只是调用IP接口，IP向TCP提供所有TCP需要的服务。通过下图我们可以更清楚地看到TCP协议的结构。

上面已经说过了，TCP连接是可靠的，而且保证了传送数据包的顺序，保证顺序是用一个序号来保证的。响应包内也包括一个序列号，表示接收方准备好这个序号的包。

  在TCP传送一个数据包时，它同时把这个数据包放入重发队列中，同时启动记数器，如果收到了关于这个包的确认信息，将此包从队列中删除，如果计时超时则需要重新发送此包。请注意，从TCP返回的确认信息并不保证最终接收者接收到数据，这个责任由接收方负责。

每个用于传送TCP的通道都有一个端口标记，因为这个标记是由每个TCP终端确定的，因此TCP可能不唯一，为了保证这个数值的唯一，要使用网络地址和端口号的组合达到唯一标识的目的，我们称这个为了套接字（Socket），一个连接由连接两端的套接字标识，本地的套接字可能和不同的外部套接字通信，这种通信是全双工的。

通过向本地端口发送OPEN命令及外部套接字参数建立连接，TCP返回一个标记这个连接的名称，以后如果用户需要使用这个名称标记这个连接。为了保存这个连接的信息，我们假设有一个称为传输控制块（Transmission Control Block，TCB）的东西来保存。

  OPEN命令还指定这个连接的建立是主动请求还是被动等待请求。下面我们要涉及具体的功能了，TCP段以internet数据报的形式传送。IP包头传送不同的信息域，包括源地址和目的地址。TCP头跟在internet包头后面，提供了一些专用于TCP协议的信息。

  下图是TCP包头格式图：

源端口：16位；

目的端口：16位

序列码：32位，当SYN出现，序列码实际上是初始序列码（ISN），而第一个数据字节是ISN+1；

确认码：32位，如果设置了ACK控制位，这个值表示一个准备接收的包的序列码；

数据偏移量：4位，指示何处数据开始；

保留：6位，这些位必须是0；

控制位：6位；

窗口：16位；

校验位：16位；

优先指针：16位，指向后面是优先数据的字节；

选项：长度不定；但长度必须以字节记；选项的具体内容我们结合具体命令来看；

填充：不定长，填充的内容必须为0，它是为了保证包头的结合和数据的开始处偏移量能够被32整除；

我们前面已经说过有一个TCB的东西了，TCB里有存储了包括发送方，接收方的套接字，用户的发送和接收的缓冲区指针等变量。

  除了这些还有一些变量和发送接收序列号有关：

发送序列变量

SND。UNA - 发送未确认

SND。NXT - 发送下一个

SND。WND - 发送窗口

SND。UP - 发送优先指针

SND。WL1 - 用于最后窗口更新的段序列号

SND。

  WL2 - 用于最后窗口更新的段确认号

ISS - 初始发送序列号

接收序列号

RCV。NXT - 接收下一个

RCV。WND - 接收下一个

RCV。UP - 接收优先指针

IRS - 初始接收序列号

下图会帮助您了解发送序列变量间的关系：

当前段变量

SEG。

  SEQ - 段序列号

SEG。ACK - 段确认标记

SEG。LEN - 段长

SEG。WND - 段窗口

SEG。UP - 段紧急指针

SEG。PRC - 段优先级

连接进程是通过一系列状态表示的，这些状态有：LISTEN，SYN-SENT，SYN-RECEIVED，ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT和 CLOSED。

  CLOSED表示没有连接，各个状态的意义如下：

LISTEN - 侦听来自远方TCP端口的连接请求；

SYN-SENT - 在发送连接请求后等待匹配的连接请求；

SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；

ESTABLISHED - 代表一个打开的连接，数据可以传送给用户；

FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；

FIN-WAIT-2 - 从远程TCP等待连接中断请求；

CLOSE-WAIT - 等待从本地用户发来的连接中断请求；

CLOSING - 等待远程TCP对连接中断的确认；

LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；

TIME-WAIT - 等待足够的时间以确保远程TCP接收到连接中断请求的确认；

CLOSED - 没有任何连接状态；

TCP连接过程是状态的转换，促使发生状态转换的是用户调用：OPEN，SEND，RECEIVE，CLOSE，ABORT和STATUS；传送过来的数据段，特别那些包括以下标记的数据段SYN，ACK，RST和FIN；还有超时，上面所说的都会时TCP状态发生变化。

下面的图表示了TCP状态的转换，但这图中没有包括错误的情况和错误处理，不要把这幅图看成是总说明了。

3。3。 序列号

请注意，我们在TCP连接中发送的字节都有一个序列号。因为编了号，所以可以确认它们的收到。对序列号的确认是累积性的，也就是说，如果用户收到对X的确认信息，这表示在X以前的数据（不包括X）都收到了。

  在每个段中字节是这样安排的：第一个字节在包头后面，按这个顺序排列。我们需要认记实际的序列空间是有限的，虽然很大，但是还是有限的，它的范围是0到2的32次方减1。我想熟悉编程的一定知道为什么要在计算两个段是不是相继的时候要使用2的32次方为模了。

  TCP必须进行的序列号比较操作种类包括以下几种：

(a) 决定一些发送了的但未确认的序列号；

(b) 决定所有的序列号都已经收到了；

(c) 决定下一个段中应该包括的序列号。

对于发送的数据TCP要接收确认，处理确认时必须进行下面的比较操作：

SND。

  UNA = 最老的确认了的序列号；

SND。NXT = 下一个要发送的序列号；

SEG。ACK = 接收TCP的确认，接收TCP期待的下一个序列号；

SEG。SEQ = 一个数据段的第一个序列号；

SEG。LEN = 数据段中包括的字节数；

SEG。

  SEQ+SEG。LEN-1 = 数据段的最后一个序列号。

请注意下面的关系：

SND。UNA 0

RCV。NXT =0

0

不可接受

>0

>0

RCV。NXT = B SYN 本方序列号是X

2) A B ACK 确认对方序列号

上面的第2步和第3步可以合并，这时可以成为3阶段，所以我们可以称它为三消息握手。

  这个过程是必须的，因为序列号不和全局时钟关联，TCP也可以有不同的机制选择ISN。接收到第一个SYN的接收方不可能知道这个数据段是不是被延时，除非它记住了在连接上使用的最近的序列号（这通常是不可能的），因此它必须要求发送者确认。

为了保证TCP获得的确认是刚才发送的段产生的，而不是仍然在网络中的老数据段产生的，因此TCP必须在MSL时间之内保持沉默。

  在本文中，我们假设MSL=2小时，这是出于工程的需要，如果用户觉得可以，他可以改变MSL。请注意如果TCP重新初始化，而内存中的序列号正在使用，不需要等待，但必须确认使用的序列号比当前使用的要大。

如果一台主机在未保留任何序列号的情况下失败，那么它应该在MSL时间之内不发出任何数据段。

  下面将会这一情况进行说明。TCP的实现可以不遵守这个规定，但是这会造成老数据被当成新数据接收，而新数据被当成老数据拒绝的情况。

每当数据段形成并进入输出队列，TCP会为它指定序列空间中的一个值。TCP中多复本检测和序列算法都依赖于这个地址空间，在对方发送或接收之前不会超过2的32次方个包存在于输出队列中。

  所有多余的数据段都会被删除。如果没有这个规定，会出现多个数据段被指定同一个序列号的情况，会造成混乱。数据段中序列号的多少和数据段中的字节数一样多。

在通常情况下，TCP保留下一个要发送的序列号和还未确认的最老的序列号，不要在没有确认的时候就再次使用，这样会有些风险，也正是因为这样的目的，所以序列空间很大。

  对于2M的网络，要4。5小时来耗尽序列空间，因为一个数据段可能的最大生存时间也不过十几分之一秒，这就留下了足够的空间；而在100M的网络上需要5。4分钟，虽然少了点，但也可以了。

如果在实现TCP时没有为保存序列号留下空间，那清除多余的包可能就不能实现了，因此推荐这种类型的TCP实现最好在失败后等待MSL时间，这样保证多余的包被删除。

  这种情况有时候也可能会出现在保留序列号的TCP实现中。如果TCP在选择一个另一个TCP连接正在使用的序列号时，这台主机突然失败了，这就产生了问题。这个问题的实质在于主机不知道它失败了多久，也不知道多余的复本是不是还在网络中。

处理这种问题的方法是等待MSL时间，如果不这样就要冒着对方错误接收数据的危险，要等待的时间也就称为“沉默时间”。

  实现者可以让用户选择是不是等待，但是无论用户如何也不见得非要等待MSL时间。

3。4。 建立一个连接

建立连接应用的是三消息握手。如果双方同时都发送SYN也没有关系，双方会发现这个SYN中没有确认，于是就知道了这种情况，通常来说，应该发送一个"reset"段来解决这种情况。

  三消息握手减少了连接失败的可能性。下面就是一个例子，在尖括号是的就是数据段中的内容和标记。其它的就不多说了。

在第2行，TCP A发送SYN初始化序列号，表示它要使用序列号100；第3行中，TCP B给出确认，并且期待着A的带有序列号101的数据段；第4行，TCP A给出确认，而在第5行，它也给出确认，并发送了一些数据，注意第4行的序列号与第5号的一样，因为ACK信息不占用序列号空间内的序列号。

  同时产生请求的情况如下图所示，只复杂一点。

使用三消息握手的主要原因是为了防止使用过期的数据段。为了这个目的，必须引入新的控制消息，RESET。如果接收TCP处理非同步状态，在接收到RESET后返回到LISTEN状态。如果TCP处理下面几种状态ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT时，放弃连接并通过用户。

  我们下面就详细说明后一种情况。

通过上面的例子，我们可以看出TCP连接是如何从过期数据段的干扰下恢复的。请注意第4行和第5行中的RST（RESET信号）。

半开连接和其它非正常状态

如果一方在未通过另一方的情况下关闭连接，或双方虽然失败而不同步的情况我们称为半开连接状态。

  在一方试图发送数据时连接会自动RESET。然而这种情况毕竟属于不正常情况。应该做出相应的处理。如果A处的连接已经关闭，B处并不知道。当B希望发送数据到A时，就会收到RESET信号，表示这个TCP连接有误，要中止当前连接。

假设A和B两个进程相互通信的时候A的TCP发生了失败，A依靠操作系统支持TCP的存在，通常这种情况下会有恢复机制起作用，当TCP重新恢复的时候，A可能希望从恢复点开始工作。

  这样A可能会试图OPEN连接，然后在这个它认为还是打开的连接上传送数据，这时A会从本地（也就是A的）TCP上获得错误消息“未打开连接”。A的TCP将发送包括SYN的数据段。下面的例子将显示这一过程：

上面这个例子中，A方收到的信息并没有确认任何东西，这时候A发现出了问题，于是发送了RST控制信息。

  另一种情况是发生在A失败，而B方仍然试图发送数据时，下面的例子可以表示这种情况，请注意第2行中A对B发送来的信息不知所云。

在下面的例子中，A方和B方进行的被动连接，它们都在等待SYN信息。过期的包传送到B方使B回应了，而收到回应的A却发现不对头，传送RST控制信息，B方返回被动LISTEN状态。

现实中的情况太多了，我们列举一些产生RST控制信息的规则如下：通常情况下，RST在收到的信息不是期待的信息时产生。如果在不能确定时不要轻易发送RST控制信息。下面有三类情况：

如果连接已经不存在，而发送来的消息又不是RST，那么要返回RST。

  如果想拒绝对不存在的连接进行SYN，可以使用这种办法。如果到达的信息有一个ACK域，返回的RST信息可以从ACK域中取得序列号，如果没有这个域，就把RST的序列号设置为0，ACK域被设备为序列号和到达段长度之和。连接仍然处于CLOSE状态。

如果连接处于非同步状态（LISTEN，SYN-SENT，SYN-RECEIVED），而且收到的确认是对未发出包的确认或是接收到数据段的安全级别与不能连接要求的相一一致时，就发送RST。

  如果SYN未被确认时，而且收到的数据段的优先级比要求的优先级要高，那么要么提高本地优先级（得事先征得用户和系统的许可）要么发送RST；如果接收数据段的优先级比要求的优先级低，就算是匹配了，当然如果对方发现优先级不对提高了优先级，在下一个包中提高了优先级，这就不算是匹配了。

  如果连接已经进入SYN，那么接收到数据段的优先级必须和本地优先级一样，否则发送RST。如果到达的信息有一个ACK域，返回的RST信息可以从ACK域中取得序列号，如果没有这个域，就把RST的序列号设置为0，ACK域被设备为序列号和到达段长度之和。

  连接仍然处于与原来相同的状态。

如果连接处于同步状态（ESTABLISHED，FIN-WAIT-1，FIN-WAIT-2，CLOSE-WAIT，CLOSING，LAST-ACK，TIME-WAIT），任何超出接收窗口的序列号的数据段都产生如下结果：发出一个空确认数据段，此段中包括当前发送序列号，另外还包括一个确认指出希望接收的下一个数据段的序列号，连接仍然保存在原来的状态。

  如果因为安全级，优先级之类的问题，那就发送RST信号然后进入CLOSED状态

django请求生命周期 答：讲了一下生命周期

wsgiref和uWSGI的区别 答：并发量不同。

wsgiref则是官方给出的一个实现了WSGI标准用于演示用的简单Python内置库，它实现了一个简单的WSGI Server和WSGI Application（在simple\_server模块中），主要分为五个模块：simple\_server， util， headers， handlers， validate。本质上就是编写一个socket服务端，用于接收用户请求(django)。（django框架实现服务端与客户端的通信(基于wsgiref)）

wsgiref源码地址：<https://pypi.python.org/pypi/wsgiref>

werkzeug 不是一个web服务器，也不是一个web框架，而是一个工具包，官方的介绍说是一个 WSGI 工具包，它可以作为一个 Web 框架的底层库，因为它封装好了很多 Web 框架的东西，werkzeug,本质上就是编写一个socket服务端，用于接收用户请求(flask)，和django中的wsgiref是类似的。（flask框架实现服务端与客户端的通信(基于werzeug)）

**uwsgi：**是一种线路协议，是**uWSGI服务器**的独占协议，用于定义传输信息的类型（type of information），每一个uwsgi packet 前4byte为传输信息类型的描述，与WSGI协议是两种东西。

uWSGI是一个Web服务器，它实现了WSGI协议、uwsgi、http等协议。Nginx中HttpUwsgiModule的作用是与uWSGI服务器进行交换。django项目部署实际上是uWSGI，他才是web服务器，而不是uwsgi。

有了uWSGI为什么还需要nginx？因为nginx具备优秀的静态内容处理能力，然后将动态内容转发给uWSGI服务器，这样可以达到很好的客户端响应。支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能。这时候nginx和uWSGI之间的沟通就要用到uwsgi协议。

redis五大数据类型 答：巴拉巴拉

redis如果存多了要怎么办 答：持久化存储。1. 增加内存，2. 使用淘汰策略，3. 使用集群。

重点介绍下23；

第2点：

我们知道，redis设置配置文件的maxmemory参数，可以控制其最大可用内存大小（字节）。

那么当所需内存，超过maxmemory怎么办？

这个时候就该配置文件中的maxmemory-policy出场了。

其默认值是noeviction。

下面我将列出当可用内存不足时，删除redis键具有的淘汰规则。

规则名称

规则说明

volatile-lru

使用LRU算法删除一个键（只对设置了生存时间的键）

allkeys-lru

使用LRU算法删除一个键

volatile-random

随机删除一个键（只对设置了生存时间的键）

allkeys-random

随机删除一个键

volatile-ttl

删除生存时间最近的一个键

noeviction

不删除键，只返回错误

LRU算法，least RecentlyUsed，最近最少使用算法。也就是说默认删除最近最少使用的键。

但是一定要注意一点！redis中并不会准确的删除所有键中最近最少使用的键，而是随机抽取3个键，删除这三个键中最近最少使用的键。

那么3这个数字也是可以设置的，对应位置是配置文件中的maxmeory-samples.

 第3点：

3.集群怎么做

Redis仅支持单实例，内存一般最多10~20GB。对于内存动辄100~200GB的系统，就需要通过集群来支持了。

Redis集群有三种方式：客户端分片、代理分片、RedisCluster(在之后一篇文章详细说一下。)

·        客户端分片

通过业务代码自己实现路由

优势：可以自己控制分片算法、性能比代理的好

劣势：维护成本高、扩容/缩容等运维操作都需要自己研发

·        代理分片

代理程序接收到来自业务程序的数据请求，根据路由规则，将这些请求分发给正确的Redis实例并返回给业务程序。使用类似Twemproxy、Codis等中间件实现。

优势：运维方便、程序不用关心如何链接Redis实例

劣势：会带来性能消耗（大概20%）、无法平滑扩容/缩容，需要执行脚本迁移数据，不方便(Codis在Twemproxy基础上优化并实现了预分片来达到Auto Rebalance)。

·        Redis Cluster

优势：官方集群解决方案、无中心节点，和客户端直连，性能较好

劣势：方案太重、无法平滑扩容/缩容，需要执行相应的脚本，不方便、太新，没有相应成熟的解决案例

celery讲一下 答：异步框架。

# Celery介绍

Celery是一个功能完备即插即用的任务队列。它使得我们不需要考虑复杂的问题，使用非常简单。celery看起来似乎很庞大，先对其进行简单的了解，然后再去学习其他一些高级特性。 celery适用异步处理问题，当发送邮件、或者文件上传, 图像处理等等一些比较耗时的操作，我们可将其异步执行，这样用户不需要等待很久，提高用户体验。 celery的特点是：

* 简单，易于使用和维护，有丰富的文档。
* 高效，单个celery进程每分钟可以处理数百万个任务。
* 灵活，celery中几乎每个部分都可以自定义扩展。

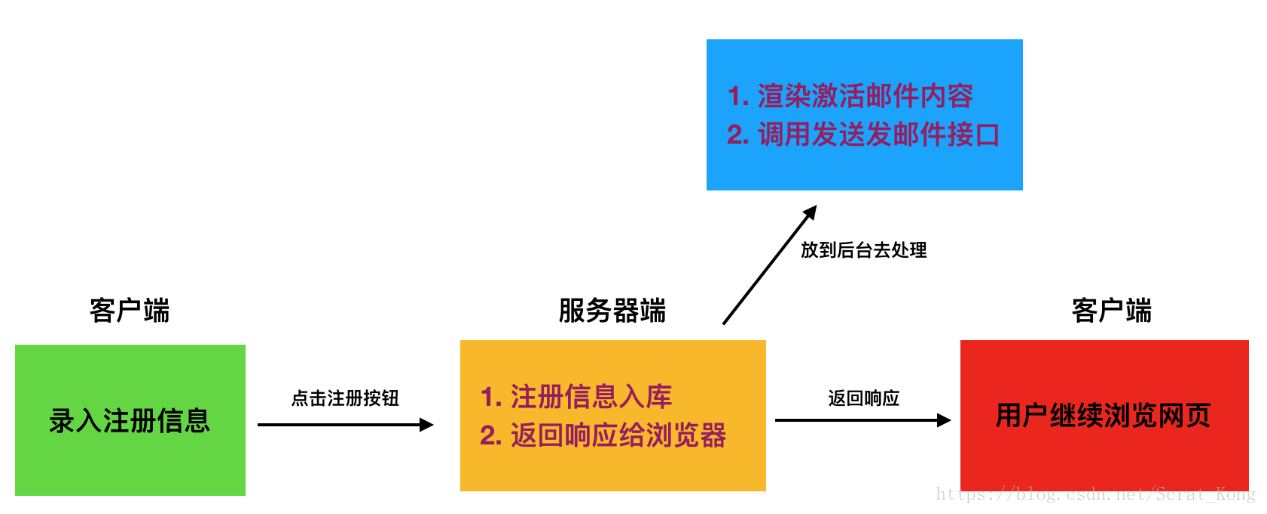
Celery非常易于集成到一些web开发框架中.

为什么要用消息队列？

我们在做网站后端程序开发时，会碰到这样的需求：用户需要在我们的网站填写注册信息，我们发给用户一封注册激活邮件到用户邮箱，如果由于各种原因，这封邮件发送所需时间较长，那么客户端将会等待很久，造成不好的用户体验.



我们将耗时任务放到后台异步执行。不会影响用户其他操作。除了注册功能，例如上传，图形处理等等耗时的任务，都可以按照这种思路来解决。 如何实现异步执行任务呢？我们可使用celery. celery除了刚才所涉及到的异步执行任务之外，还可以实现定时处理某些任务。



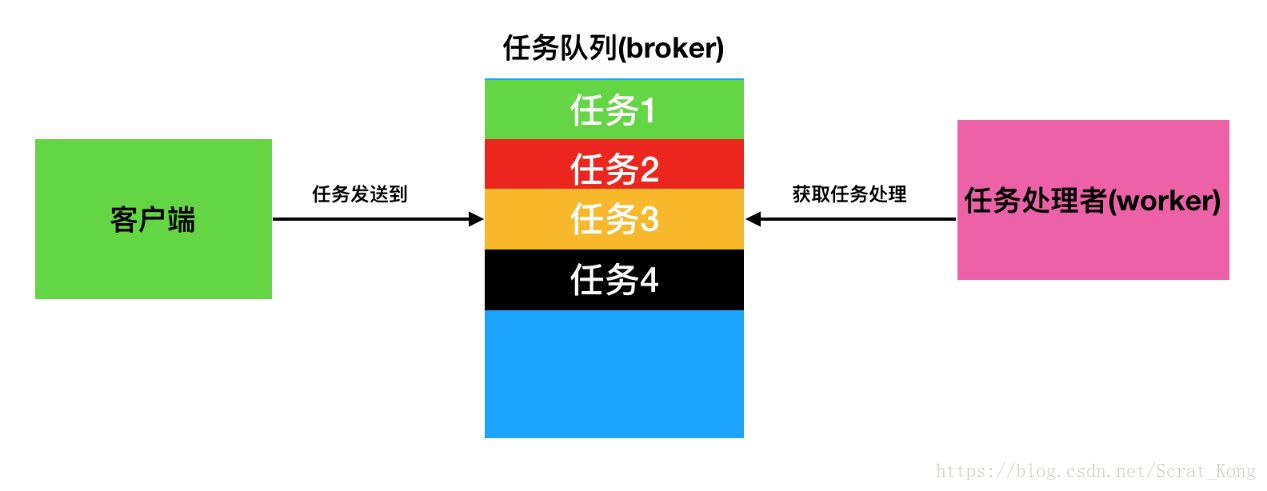
任务队列

任务队列是一种跨线程、跨机器工作的一种机制.

  任务队列中包含称作任务的工作单元。有专门的工作进程持续不断的监视任务队列，并从中获得新的任务并处理.

  celery通过消息进行通信，通常使用一个叫Broker(中间人)来协client(任务的发出者)和worker(任务的处理者). clients发出消息到队列中，broker将队列中的信息派发给worker来处理。

  一个celery系统可以包含很多的worker和broker，可增强横向扩展性和高可用性能。



多进程和多线程的区别 答：巴拉巴拉

进程之间怎么通信 答：队列或管道

索引有哪些 答：只回答了主键索引，唯一索引。

 MySQL支持以下几种类型的索引。

      （1）B-Tree索引

      （2）哈希索引

      （3）空间数据索引（R-Tree）

      （4）全文索引

      （5）其他索引类别

二、索引类型

Mysql目前主要有以下几种索引类型：FULLTEXT，HASH，BTREE，RTREE。

1. FULLTEXT

即为全文索引，目前只有MyISAM引擎支持。其可以在CREATE TABLE ，ALTER TABLE ，CREATE INDEX 使用，不过目前只有 CHAR、VARCHAR ，TEXT 列上可以创建全文索引。

全文索引并不是和MyISAM一起诞生的，它的出现是为了解决WHERE name LIKE “%word%"这类针对文本的模糊查询效率较低的问题。

2. HASH

由于HASH的唯一（几乎100%的唯一）及类似键值对的形式，很适合作为索引。

HASH索引可以一次定位，不需要像树形索引那样逐层查找,因此具有极高的效率。但是，这种高效是有条件的，即只在“=”和“in”条件下高效，对于范围查询、排序及组合索引仍然效率不高。

3. BTREE

BTREE索引就是一种将索引值按一定的算法，存入一个树形的数据结构中（二叉树），每次查询都是从树的入口root开始，依次遍历node，获取leaf。这是MySQL里默认和最常用的索引类型。

4. RTREE

RTREE在MySQL很少使用，仅支持geometry数据类型，支持该类型的存储引擎只有MyISAM、BDb、InnoDb、NDb、Archive几种。

相对于BTREE，RTREE的优势在于范围查找。

ps. 此段详细内容见此片博文：Mysql几种索引类型的区别及适用情况

三、索引种类

普通索引：仅加速查询

唯一索引：加速查询 + 列值唯一（可以有null）

主键索引：加速查询 + 列值唯一（不可以有null）+ 表中只有一个

组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并

全文索引：对文本的内容进行分词，进行搜索

ps.

索引合并，使用多个单列索引组合搜索

覆盖索引，select的数据列只用从索引中就能够取得，不必读取数据行，换句话说查询列要被所建的索引覆盖

从物理存储角度

1、聚集索引（clustered index）

2、非聚集索引（non-clustered index）

索引是怎么加快查询速度的 答：支支吾吾，它是一种新的数据结构。

打个来比方，索引就是书的目录，有了目录，查询起来当然更方便。可以分为聚自族,非聚族索引 聚族索引一个表只能有一个，它查询速度快，占有空间小非聚族索百引可以有多个，但查询速度慢，占有空间大。所有的素度引都有一个缺点，如果表里面的数据频繁更新问的话，那建索引就要慎重，索引提答高查询速度，但降低更新速度

为什么能够提高查询速度？

索引就是通过事先排好序，从而在查找时可以应用二分查找等高效率的算法。

一般的顺序查找，复杂度为O(n)，而二分查找复杂度为O(log2n)。当n很大时，二者的效率相差及其悬殊。

举个例子：

表中有一百万条数据，需要在其中寻找一条特定id的数据。如果顺序查找，平均需要查找50万条数据。而用二分法，至多不超过20次就能找到。二者的效率差了2.5万倍！

在一个或者一些字段需要频繁用作查询条件，并且表数据较多的时候，创建索引会明显提高查询速度，因为可由全表扫描改成索引扫描。

（无索引时全表扫描也就是要逐条扫描全部记录，直到找完符合条件的，索引扫描可以直接定位）

不管数据表有无索引，首先在SGA的[数据缓冲区](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E7%BC%93%E5%86%B2%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPh7-Pj6znWT1Phf3rAnk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0drHf4PWbd)中查找所需要的数据，如果[数据缓冲区](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E7%BC%93%E5%86%B2%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPh7-Pj6znWT1Phf3rAnk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0drHf4PWbd)中没有需要的数据时，服务器进程才去读磁盘。

1、无索引，直接去读表数据存放的磁盘块，读到[数据缓冲区](http://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E7%BC%93%E5%86%B2%E5%8C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPh7-Pj6znWT1Phf3rAnk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPH0drHf4PWbd)中再查找需要的数据。

2、有索引，先读入索引表，通过索引表直接找到所需数据的物理地址，并把数据读入数据缓冲区中。

索引有什么副作用吗？

（1）索引是有大量数据的时候才建立的，没有大量数据反而会浪费时间，因为索引是使用二叉树建立.

（2）当一个系统查询比较频繁，而新建，修改等操作比较少时，可以创建索引，这样查询的速度会比以前快很多，同时也带来弊端，就是新建或修改等操作时，比没有索引或没有建立覆盖索引时的要慢。

（3）索引并不是越多越好，太多索引会占用很多的索引表空间，甚至比存储一条记录更多。

对于需要频繁新增记录的表，最好不要创建索引，没有索引的表，执行insert、append都很快，有了索引以后，会多一个维护索引的操作，一些大表可能导致insert 速度非常慢。

索引优点中，最重要的就是加快查询速度。

索引有如下3个特点，这3点虽然看上去，句句都是废话，但仔细想想就发现，这3个特点包含的东西，远远超过了3句话所涵盖的。

（1）索引有序

创建索引后，之所以查询速度会快，是由于创建索引时，会按照字段进行排序，如果有多个字段，那么会按照多个字段逐个排序。

在查找的时候，可以用类似二分查找的算法，快速判断要找的值在哪个B树的分支。

（2）索引高度不会太高

在数据量比较少的情况下，建不建索引，效果不大。

但当数据量越来越大，比如达到百万、甚至亿级的时候，查找数据就会越来越慢，为什么会那么慢呢？

在没有索引的情况下，要从头到尾的找，比如表里有1亿条数据，那么就得扫描这1亿条数据，最后可能只找到了10条满足要求的数据。

在这个过程中，大部分都是做的无用功。。。

而创建索引后，生成了B树，虽然有1亿条数据，但是这个B树不会提高，一般不超过7层，那么这个层数是如何影响查找效率的呢？

因为索引存储的值，都是在叶子节点上，所以要找到值，就必须要从B树的根开始找，如果有6层，那么相当于根开始到叶子，一共要进过6次的判断，最后找到想要的值。

所以，层数越少，比较次数少，需要读取的数据叶子数，也少，IO次数就少，那就会更快。

（3）索引中包含字段的值

给某个字段建索引，那么这个索引中就会包含这个字段的值，这句话看上去像废话，但很有用。

在某些情况下，索引不起过滤的所用，貌似这个索引好像就没什么用。

其实这个时候，对索引的访问方式，就不是原来的索引查找，而是索引扫描，也就是说sql要访问的数据，都可以直接从索引中出来，那么就没必要去查表的数据，索引起到的是覆盖查询的目的，相对于查整个表来说，索引页数更少，IO就少，所以查询时间就会小很多。

索引底层是什么数据结构 答：不知道。

索引是帮助MySQL高效获取数据的排好序的数据结构（容易忽略的点：排好序）（形象点就是教科书的目录）

索引存储在文件里（也就是说有IO操作）

索引结构：

这里说说在几种数据结构中，mysql为什么选择hash，B+Tree

1. 二叉树
2. 红黑树
3. hash
4. BTree
5. B+Tree

首先，如果数据没有索引，那么我们读取数据是这样的

MySQL官方对索引的定义为:索引(Index)是帮助MySQL高效获取数据的数据结构

大家使用索引有没有想过这个问题？为什么索引能够帮助mysql高效获取数据？我一一给大家道来！在给大家讲之前，先更大家分享一些计算机基础知识，有助于理解

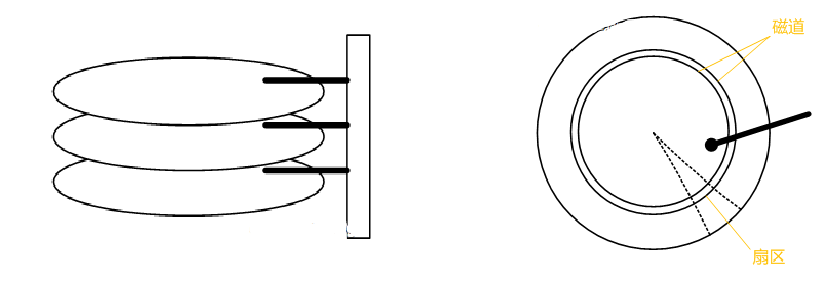
我们都知道mysql数据是已文件的形式存储在磁盘上的。

那磁盘是有一圈一圈的磁道组成的

磁头移动到不同磁道，磁盘旋转，这样就可以读取到数据

磁盘存取原理 1.寻道时间(速度慢，费时) 2.旋转时间(速度较快)

cpu读取数据都是从内存去读，内存去磁盘读取数据，内存读取磁盘数据大小都是已页的大小单位，一页=10kb



总结：

通过上面的一些知识，我们知道当磁头移动到另一个磁道读取诗句就是我们常说的一次I/O操作，但是我们知道mysql数据是分布到不同的磁道上的，每次读取数据都要把所有磁道读取一遍，那我们进行I/O次数就很多了，查询效率就很低

那索引就是把索引字段数据的地址保存起来，来帮助mysql直接定位到哪个磁道的哪个扇区，这样就减少I/O操作了，自然查询效率就提高了

B-Tree和B+Tree

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构，在本文的下一节会结合存储器原理及计算机存取原理讨论为什么B-Tree和B+Tree在被如此广泛用于索引，这一节先单纯从数据结构角度描述它们。

a.二叉树

二叉树是n(n>=0)个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树组成。

不使用原因：会出现极端情况，一个节点只有一度，就是只有一个子节点，那读取树的一层就是一次I/O，那性能也不好

b.红黑树

红黑树即为平衡二叉树的一种

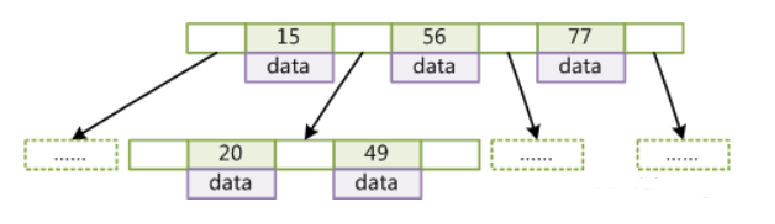
不使用原因：极端情况下，一个节点有2个子节点，那就出现一层只有2个节点的情况，这种性能也不好

c.Hash

不使用原因：Hash是把索引数据进行Hash算法对应一个地址，我们会发现这个好像性能很好啊，直接找到，但是我们想想，它能满足我们日常开发大部分情况吗？比如通过大于或者小于去筛选数据，所以说也不合适，当然mysql还是提供了Hash索引，毕竟有些场合还是用起来也不错

d.B Tree

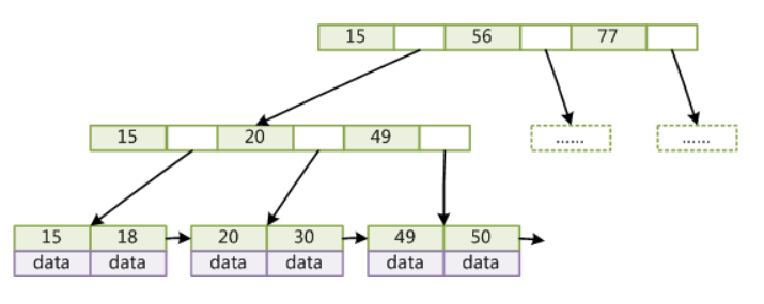
1.度(Degree)-节点的数据存储个数 2.叶节点具有相同的深度 3.叶节点的指针为空 4.节点中的数据key从左到右递增排列



不使用原因：虽然解决了每一层的节点数的极端情况下，但是我们会发现每个节点存储了索引和数据，那一层能存储的数据太多也不好,毕竟内存能读取的数据大小就是10kb

e.B+Tree

1.B+Tree(B-Tree变种) 2.非叶子节点不存储data，只存储key，可以增大度 3.叶子节点不存储指针 4.顺序访问指针，提高区间访问的性能



使用原因：设计有几个方面

1.非叶子节点不存储data，只存储key，可以增大度

2.叶子节点不存储指针

3.顺序访问指针，提高区间访问的性能

3.B+Tree索引的性能分析

一般使用磁盘I/O次数评价索引结构的优劣

预读：磁盘一般会顺序向后读取一定长度的数据(页的整数倍)放入内存

局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用

B+Tree节点的大小设为等于一个页，每次新建节点直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，就实现了一个节点的载入只需一次I/O

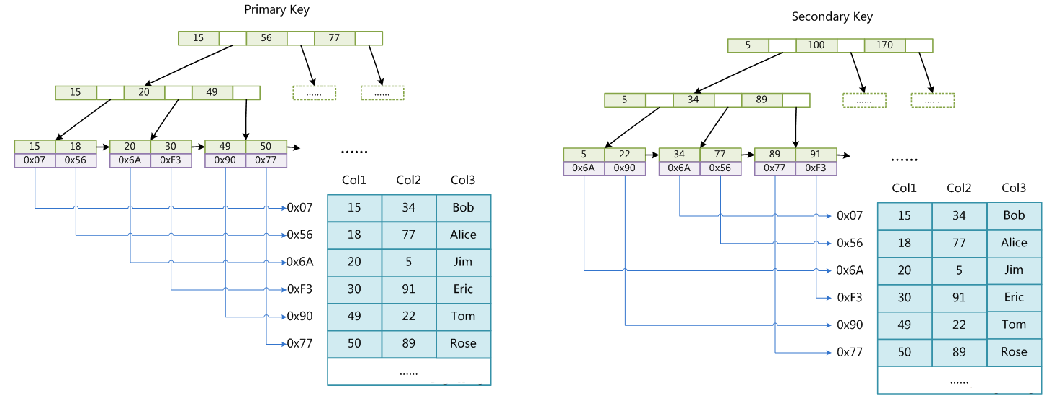
B+Tree的度d一般会超过100，因此h非常小(一般为3到5之间)

4.不同的存储引擎有不同的索引实现

1.MyISAM索引实现(非聚集) 2.InnoDB索引实现(聚集)

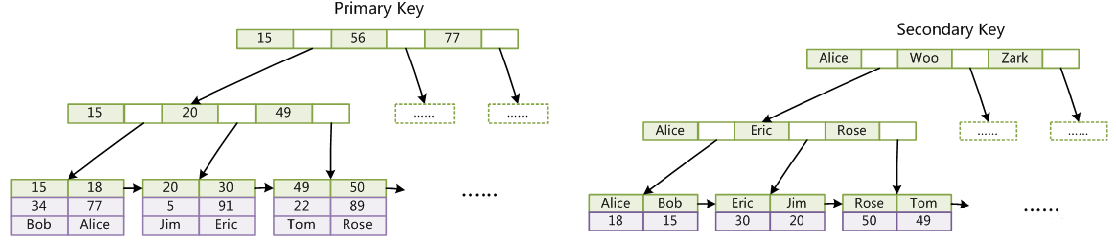
a.MyISAM索引实现(非聚集)

--->MyISAM索引文件和数据文件是分离的



b.InnoDB索引实现(聚集)

1.数据文件本身就是索引文件 2.表数据文件本身就是按B+Tree组织的一个索引结构文件 3.聚集索引-叶节点包含了完整的数据记录 4.为什么InnoDB表必须有主键，并且推荐使用整型的自增主键？ 5.为什么非主键索引结构叶子节点存储的是主键值？(一致性和节省存储空间)



5.联合索引结构

---->联合索引的底层存储结构长什么样？

6.索引最左前缀原理

你在orm里写原生sql，怎么判断他走了索引或者是全局查了一次 答：不知道。

django中的ORM提供的操作功能有限，在模型提供的API不能满足实际工作需要时，可以在ORM中直接执行原生sql语句。

Django提供两种方法使用原生SQL进行查询：一种是使用raw（）方法，进行原生SQL查询并返回模型实例：另一种是完全避开模型层，直接咨询自定义的SQL语句。

你们之前的项目上线了是怎么看他死了没 答：不清楚，这是运维做的。

讲一个项目 答：cmdb

怎么起一个容器 答：docker start 容器。

进入Docker容器比较常见的几种做法如下：

1.使用docker attach

2.使用SSH

3.使用nsenter

4.使用exec

怎么新建一个容器 答：。。。。

容器的概念：

独立运行的一个应用 和一组应用

以及这些应用对应的环境。

docker启动有两种方式。

a 基于镜像

docker run centos cal

启动这个centos 打印日历表。

也可以使用

docker run -it centos /bin/bash

会看到一个交互式的客户端

将终止状态下的容器重新启动

docker start centos

在守护状态下运行

docker run -d centos /bin/bash -c “hello wordl;sleep 1;done’

-d 在后台运行一个容器。

docker logs 容器ID

查看某个容器的日志输出。

如何终止一个容器的运行呢？

docker ps -a 查看所有的容器

可以根据容器的ID 进行 stop restart 等操作

终止之后 可以使用 docker ps -a 查看 是否存在呢

全局异常捕获 答：。。。。。

Python全局未知异常捕获

log日志监测程序运行：

可以

1⃣️Linux重定向将python运行报错记录到log日志中，

2⃣️或者用python内置的logging模块。

但是在用logging模块时，try...except...可以捕获到可预见/考虑到的异常，用logging记录到log日志文件中，但是在对于未知异常或导致程序退出时的情形无法记录到log日志中，用Linux的重定向可以捕捉到。

现在到问题时我不想用Linux系统提供的重定向方法，我需要捕获python运行中未知的异常并记录到log日志中。

推荐使用sys.excepthook。详细参考这篇文章，简洁明了。

也可以用try...except...包裹全部代码，但我觉得这样不够优雅。

当要捕获所有异常的时候，推荐使用：

#推荐写法 可以捕获除与程序退出sys.exit()相关之外的所有异常。

try:

:

except Exception as e:

# error occurred, log 'e', etc.

#不推荐使用以下这种

会捕获所有异常，包括键盘中断和程序退出请求（用sys.exit()就无法退出程序了，因为异常被捕获了），因此慎用。

try:

:

except:

# error occurred, log 'e', etc.

老版本的Python，except语句写作"except Exception, e"，

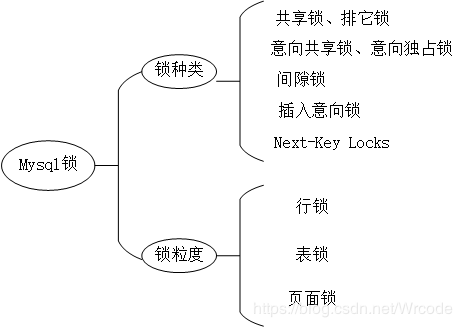
Python 2.6后应写作"except Exception as e"。

装饰器 答：。。。。

http状态码 答：2 3 4 5

秒杀用什么锁 答：悲观锁，其实应该用乐观锁

简单的秒杀实现方式比较多，常见的有可重入锁、redis分布式锁、线程同步锁、数据库锁等，其中最简单也最容易实现的就是数据库乐观锁了，下面的demo以springboot+Data JPA 框架为基础，利用mysql乐观锁实现了一个简单的秒杀场景，乐观锁本质并不会加锁，只是在进行数据修改做版本校验，校验通过更新数据，否则不更新。本demo仅供了解并发学习之用，还有很多未完善的地方，请大家多多包涵。



背景：有库存数要求的秒杀活动。

基于redis解决方案

1、基于watch的乐观锁方案

watch用于监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。这种方案跟mysql中的乐观锁方案类似，具体表现也是一样的。

2、基于list的队列方案

基于队列的方案利用了redis出队操作的原子性，抢购开始之前首先将商品编号放入响应的队列中，在抢购时依次从队列中弹出操作，这样可以保证每个商品只能被一个进程获取并操作，不存在超发的情况。该方案的优点是理解和实现起来都比较简单，缺点是当商品数量较多是，需要将大量的数据存入到队列中，并且不同的商品需要存入到不同的消息队列中

讲项目 答：cmdb

写题目给你写

装饰器怎么写

摘要

装饰器是在不修改原来函数的情况下，对函数增新的功能的一种方法

装饰器内可带参数

一个函数可以有多个装饰器

实现方法

1.原始方法

import time

def deco(func):

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

startTime = time.time()

func(\*args, \*\*kwargs)

endTime = time.time()

msecs = (endTime - startTime)\*1000

print("time is %d ms" %msecs)

return wrapper

@deco

def func(a,b):

print("hello，here is a func for add :")

time.sleep(1)

print("result is %d" %(a+b))

2.类方法

class logging(object):

def \_\_init\_\_(self, func):

self.func = func

def \_\_call\_\_(self, \*args, \*\*kwargs):

print "[DEBUG]: enter function {func}()".format(

func=self.func.\_\_name\_\_)

return self.func(\*args, \*\*kwargs)

@logging

def say(something):

print "say {}!".format(something)

上一篇文章介绍了 装饰器的概念。现在讲一下在程序中怎么来写装饰器。上代码：

1 def X(fun):

2 def Y(b):

3 print(b)

4 fun()

5 return Y

6

7 def test():

8 print('OK')

9

10 test = X(test)

11 test(1)

前五行是一个闭包，因为内层函数的参数是外层函数的变量，而外层函数返回了内存函数的引用。

第10行，在调用函数X时，将函数test的引用（注意不是test(),没有小括号）作为参数传入，此时X(test）返回的是 Y函数的引用。所以第十行的结果就是test指向了Y函数的引用。所以，第11行的test()就是调用的Y函数的引用，同时Y函数中的fun()指向了第7行的test()函数。

第二种写法：

1 def X(fun):

2 def Y(b):

3 print(b)

4 fun()

5 return Y

6

7 @X #相当于 test = X(test)

8 def test():

9 print('OK')

10

11 test(1)

@在python中叫做语法糖，而@X 与test = X(test)等价，所以上面两种方法是等价的，只是第一种方法比较直观的展示了装饰器，而第二种写法比较美观简洁。

**045、with管理上下文的源码怎么写，类似的就行？**

Python with上下文管理及自定义上下文管理

一、上下文管理器

上下文管理器是对Context Manager的翻译 ,上下文是 context 直译的叫法,在程序中用来表示代码执行过程中所处的前后环境.

在文件操作时，需要打开，关闭文件，而在文件在进行读写操作时，就是处在文件操作的上下文中，也就是文件操作环境中.

语法: with

with 语句是 Pyhton 提供的一种简化语法,从Python 2.5 开始引入的一种与异常处理相关的功能,适用于对资源进行访问的场合,确保不管使用过程中是否发生异常都会执行必要的“扫尾”操作,释放资源.

比如文件使用后自动关闭、数据库连接的自动关闭等.

二、with上下文管理器示例

with open('test.txt', 'a') as f:

f.write('上下文管理\n')

运行以上代码,会在代码同级目录下创建一个叫test.txt的文本文件,并在文件中追加内容"上下文管理",并换行.

在执行 with 语句时，首先执行 with 后面的 open 代码(如果已经有test.txt文件则打开,如果没有test.txt文件则创建),然后通过as将代码的结果保存到 f 中(相当于给test.txt起个别名f,f只是一个变量名,您可以自定义)

在with下面是对test.txt执行的操作,如示例中的操作是写入内容.

在操作后，不需要对文件test.txt进行关闭操作f.close()，with上下文管理器会在文件使用完后帮我们关闭test.txt文件.

这么做即可以简化代码,又可以避免因粗心忘记执行关闭操作而出现异常,因为在实际开发中,打开一个文件后,进行的操作可能非常复杂,这种情况是很可能忘记做"扫尾"操作的.

三、with上下文管理的原理

在使用with上下文管理器时,并不是不需要关闭文件,而是文件的关闭操作在 with 的上下文管理器中已经实现了.当文件操作执行完成后,with语句会自动调用上下文管理器里的关闭语句来关闭文件资源.

我们的应用场景并不是一成必变,with是怎么实现帮我们做"扫尾"操作的呢?

with语句在执行时,调用上下文管理器中的 \_\_enter\_\_ 和 \_\_exit\_\_ 两个方法,这两个方法就是上下文管理器中实现的方法.

\_\_enter\_\_ 方法会在执行with后面的语句时执行,一般用来处理操作前的内容.比如打开文件,创建对象,初始化等.

\_\_exit\_\_ 方法会在with内的代码执行完毕后执行,一般用来处理一些善后收尾工作,比如文件的关闭,数据库的关闭等.

四、自定义上下文管理器

根据上下文管理的原理,上下文管理器的原理是实现了\_\_enter\_\_和\_\_exit\_\_这两个方法,所以我们可以根据此原理来自定义自己的上下文管理器.

在自定义上下文管理器时，只要在类中实现 \_\_enter\_\_ 和 \_\_exit\_\_ 两个方法即可.

class OpenFile(object):

"""自定义上下文管理类"""

def \_\_init\_\_(self, file, mode):

self.\_file = file

self.\_mode = mode

def \_\_enter\_\_(self):

print('\_\_enter\_\_ 打开文件')

self.\_handle = open(self.\_file, self.\_mode)

return self.\_handle

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

print('\_\_exit\_\_ 关闭文件')

self.\_handle.close()

with OpenFile('test01.txt', 'w') as f:

f.write('自定义上下文管理\n')

上面的代码中,我们不是使用Python实现好的open来打开文件,而是通过自定义的OpenFile类来完成文件的操作.

代码运行过程和结果与用open类似,会先执行OpenFile后的代码(自定义类OpenFile中的代码),如果存在文件test01.txt,打开文件并在其中写入"自定义上下文管理",如果不存在文件test01.txt,则创建文件并写入"自定义上下文管理".

说明我们成功实现了自定义上下文管理.

五、上下文管理中的自定义异常处理

我们看到,\_\_exit\_\_ 方法中有三个参数exc\_type ,exc\_val 和exc\_tb ,这三个参数是用来接收异常信息的，如果代码在运行时发生异常，异常信息会被保存到这三个参数中.

class OpenFile(object):

"""自定义上下文管理类"""

def \_\_init\_\_(self, file, mode):

self.\_file = file

self.\_mode = mode

def \_\_enter\_\_(self):

self.\_handle = open(self.\_file, self.\_mode)

return self.\_handle

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

print('Type: ', exc\_type)

print('Value:', exc\_val)

print('TreacBack:', exc\_tb)

self.\_handle.close()

with OpenFile('test01.txt', 'r') as f:

f.write('自定义上下文管理\n')

上面的代码中,传入的'w'换成了'r',会出现异常,因为以读的模式打开文件,不能进行写操作.运行结果为:

Traceback (most recent call last):

Type: <class 'io.UnsupportedOperation'>

File "python\_demo/with\_demo.py", line 45, in <module>

Value: not writable

f.write('自定义上下文管理\n')

TreacBack: <traceback object at 0x0000024EA5E11508>

io.UnsupportedOperation: not writable

可以看到我们打印的三个参数exc\_type ,exc\_val 和exc\_tb的值.分别存储了异常的以下信息:

exc\_type : 异常类型

exc\_val : 异常值

exc\_tb : 异常回溯追踪

当with中执行的语句发生异常时,异常信息会被发送到 \_\_exit\_\_ 方法的参数中,这时可以根据情况选择如何处理异常.

因为在 \_\_exit\_\_函数执行异常处理时,会根据函数的返回值决定是否将系统抛出的异常继续向外抛出.如果返回值为 False 就会向外抛出,用户就会看到.如果返回值为 True 不会向外抛出,而是显示我们自定义的信息.

我们可以根据这个原理自定义如何处理异常,只要将返回值设置为True就行了.

class OpenFile(object):

"""自定义上下文管理类"""

def \_\_init\_\_(self, file, mode):

self.\_file = file

self.\_mode = mode

def \_\_enter\_\_(self):

self.\_handle = open(self.\_file, self.\_mode)

return self.\_handle

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.\_handle.close()

# 通过exc\_type参数接收到的值，来判断程序执行是否出现异常

# 如果是None,说明没有异常

if exc\_type == "None":

print('正常执行')

else:

# 否则出现异常，可以选择怎么处理异常

print(exc\_type, exc\_val)

# 返回值决定了捕获的异常是否继续向外抛出

# 如果是False那么就会继续向外抛出，程序会看到系统提示的异常信息

# 如果是True不会向外抛出,程序看不到系统提示信息，只能看到else中的输出

return True

with OpenFile('test01.txt', 'r') as f:

f.write('自定义上下文管理\n')

通过设置返回值为True,上下文管理器不会向外抛出异常,此时我们在根据exc\_type(异常类型)的值是否为"None"来自定义输出的异常信息,就可以实现上下文管理中的自定义异常了.

**047、property是什么？**

# Python property() 函数

property() 函数的作用是在新式类中返回属性值。

### 语法

以下是 property() 方法的语法:

class property([fget[, fset[, fdel[, doc]]]])

### 参数

* fget -- 获取属性值的函数
* fset -- 设置属性值的函数
* fdel -- 删除属性值函数
* doc -- 属性描述信息

### 返回值

返回新式类属性。

## 1. 基本认识

property属性可以用来给属性添加约束，比如温度属性，我们不允许低于-273度；成绩属性，我们不允许0分以下等等。而且使用property属性，将来修改约束条件的时候也很方便，可以在代码的调用方式不变的情况下改变结果。

python中使用property属性有两种方法。使用@property装饰器和使用property()函数。

我们通过廖雪峰官方网站的实例来对此加深认识。

## 2. @property装饰器

@property装饰器就是负责把一个方法变成属性调用的。如下实例就可以通过s.score来获得成绩，并且对score赋值之前做出了数据检查。

3. property()函数

python中关于property()函数的介绍如下，在jupyter notebook中输入property??，即可查看用法：

\_\_setattr\_\_

[python 魔法方法补充（\_\_setattr\_\_,\_\_getattr\_\_,\_\_getattribute\_\_）](https://www.cnblogs.com/big-handsome-guy/p/8618078.html)

\_\_getattr\_\_方法：

拦截点号运算。对象.属性：如果找不到属性，就会用属性名作为字符串，调用该方法；如果继承树可以找到该属性，则不会调用该方法

\_\_setattr\_\_方法：

会拦截所有属性的的赋值语句。如果定义了这个方法，self.arrt = value 就会变成self.\_\_setattr\_\_("attr", value).

这个需要注意。当在\_\_setattr\_\_方法内对属性进行赋值是，不可使用self.attr = value,因为他会再次调用self.\_\_setattr\_\_("attr", value),则会形成无穷递归循环，最后导致堆栈溢出异常。应该通过对属性字典做索引运算来赋值任何实例属性，也就是使用self.\_\_dict\_\_['name'] = value.

**057、一个字符串，让你倒序输出，例子 my name is >>> is name my？**

1. #python解决字符串倒序输出
2. def string\_reverse(m):
3. num=len(m)
4. a=[]
5. for i in range(num):
6. a.append(m[num-1-i])#从最后一位的元素开始往新list内添加元素
7. return a

对于一个给定的字符串，逆序输出，这个任务对于python来说是一种很简单的操作，毕竟强大的列表和字符串处理的一些列函数足以应付这些问题 了，今天总结了一下python中对于字符串的逆序输出的几种常用的方法，一共总结出来了六种方法，个人认为比较全面了，如有其他方法欢迎前来补充

方法一：

直接使用字符串切片功能逆转字符串

方法二：

将字符串转换为列表使用reverse函数

方法三：

新建一个列表，从后往前添加元素

方法四：

借助于collections模块现成的方法extendleft

方法五：

递归实现

方法六：

借助基本的Swap操作，以中间为基准交换对称位置的字符

    下面是具体的实现：

#!usr/bin/env python

#encoding:utf-8

'''''

\_\_Author\_\_:沂水寒城

功能：输入一个字符串，倒叙输出

'''

import collections

def func1(one\_str):

'''

直接使用字符串切片功能逆转字符串

'''

return one\_str[::-1]

def func2(one\_str):

'''

将字符串转换为列表使用reverse函数

'''

one\_str\_list = list(one\_str)

one\_str\_list.reverse()

return ''.join(one\_str\_list)

def func3(one\_str):

'''

新建一个列表，从后往前添加元素

'''

one\_list=[]

for i in range(len(one\_str)-1,-1,-1):

one\_list.append(one\_str[i])

return ''.join(one\_list)

def func4(one\_str):

'''

借助于collections模块现成的方法extendleft

'''

deque1=collections.deque(one\_str)

deque2=collections.deque()

for one\_char in deque1:

deque2.extendleft(one\_char)

return ''.join(deque2)

def func5(one\_str):

'''

递归实现

'''

if len(one\_str)<=1:

return one\_str

return one\_str[-1]+func5(one\_str[:-1])

def func6(one\_str):

'''

借助基本的Swap操作，以中间为基准交换对称位置的字符

'''

one\_str\_list=list(one\_str)

if len(one\_str\_list)==0 or len(one\_str\_list)==1:

return one\_str\_list

i=0

length=len(one\_str\_list)

while i < length/2:

one\_str\_list[i], one\_str\_list[length-i-1]=one\_str\_list[length-i-1], one\_str\_list[i]

i+=1

return ''.join(one\_str\_list)

def main\_func(str\_list):

'''

主调用函数

'''

for one\_str in str\_list:

one\_list=[]

one=[]

one\_list.append(func1(one\_str))

one\_list.append(func2(one\_str))

one\_list.append(func3(one\_str))

one\_list.append(func4(one\_str))

one\_list.append(func5(one\_str))

one\_list.append(func6(one\_str))

print '字符串{0}逆序为：'.format(one\_str)

print one\_list

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

str\_list=['123456','abcdefg','zyxvuw','Together\_CZ']

main\_func(str\_list)

结果如下：

字符串123456逆序为：['654321', '654321', '654321', '654321', '654321', '654321']

字符串abcdefg逆序为：['gfedcba', 'gfedcba', 'gfedcba', 'gfedcba', 'gfedcba', 'gfedcba']

字符串zyxvuw逆序为：['wuvxyz', 'wuvxyz', 'wuvxyz', 'wuvxyz', 'wuvxyz', 'wuvxyz']

字符串Together\_CZ逆序为：['ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT', 'ZC\_rehtegoT']

判断括号是否合法，类似于leetcode上的，[](){}，每个括号都要成对

python判断括号是否成对出现且顺序合法

第一种方法

def isBracketPair(a):

bracketAll = {")": "(", "]": "[", "}": "{"}

b = []

for i in a:

if i in bracketAll.values():

b.append(i)

elif len(b) > 0 and b[-1] == bracketAll.get(i):

b.pop()

else:

return "bad"

if len(b) == 0:

return "ok"

else:

return "bad"

第二种方法

class Solution(object):

def isValid(self, s):

"""

:type s: str

:rtype: bool

"""

# 判断是否是奇数或空字符

if len(s)<2 or len(s)%2!=0:

if s=='':

return True

else:

return False

count = 0

length = len(s)

# 将其中的(){}[] 都换掉，然后判断是否有剩余

while(count<length/2):

s = s.replace("{}","").replace("[]","").replace("()","")

count+=1

if len(s)>0:

return False

else:

return True

第三种方法

{}[]（) 是合法的括号。

   {}[} 是不合法的括号。

# 判断是否为合法的匹配括号。

class Solution():

def validBracket(self, nums):

dic2 = {"(":")","{":"}","[":"]"}

stack = []

for i in range(len(nums)):

if nums[i] in dic2:

stack.append(nums[i])

elif len(stack)!=0 and dic2[stack[-1]]==nums[i]:

stack.pop()

else:

return False

return stack==[]

if \_\_name\_\_ == "\_\_main\_\_":

s = Solution()

str2 = "(){}"

print(s.validBracket(str2))

get-post定义的规范

**get post 区别restful规范**

在网站开发的时候通常会对POST和GET产生混淆，让人混淆的主要原因是基本上POST能解决的问题GET都能解决，反之亦然。今天就来说说者两者的区别。

GET：字面理解就是获取资源

* GET请求标准上是幂等的（用户应该认为请求是安全的-资源不会被修改，这里所以说应该是服务器端并不保证资源不会被修改）
* GET请求可以被浏览器缓存；响应也可以被缓存（根据缓存头信息来处理）
* GET请求可以保存在浏览器历史记录中，也可以作为链接分发或分享，可以收藏为书签
* GET请求的数据都在URL中，可以方便都从浏览器中获取数据（因此不能携带诸如密码的明文数据）
* GET请求的长度会有限制（比如IE的路径总长度需小于2048个字符）
* GET请求的数据只能包含ASCII字符

POST：字面理解就是发布新资源

* POST请求标准上不是幂等的（用户应该认为请求是有副作用的-可能会导致资源修改）
* POST请求URL可以被浏览器缓存，但是POST数据不会被缓存；响应可以被缓存（根据缓存头信息来处理）
* POST请求不便于分发或分享，因为POST数据会丢失，不能收藏为书签。
* POST请求没有长度限制，可以用来处理“请求数据”很大的场景（只要不超过服务器端的处理能力）
* POST请求的数据不限于ASCII字符，可以包含二进制数据

上面两者区别的解释中幂等可能不太好理解，幂等（idempotent、idempotence）其实是一个数学或计算机学概念，常见于抽象代数中。幂等具体表现为：

* 对于单目运算，如果一个运算对于在范围内的所有的一个数多次进行该运算所得的结果和进行一次该运算所得的结果是一样的，那么我们就称该运算是幂等的。比如绝对值运算就是一个例子，在实数集中，有abs(a)=abs(abs(a))。
* 对于双目运算，则要求当参与运算的两个值是等值的情况下，如果满足运算结果与参与运算的两个值相等，则称该运算幂等，如求两个数的最大值的函数，即max(x,x) = x。

通俗的讲幂等的意味着对同一URL的多个请求应该返回同样的结果。但其实也不不是非常的严格，比如新闻站点的头版不断更新。虽然第二次请求会返回不同的一批新闻，该操作仍然被认为是和幂等的，因为它总是返回当前的新闻。从根本上说，如果目标是当用户打开一个链接时，他可以确信从自身的角度来看没有改变资源即可。

事实上Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查，改，增，删4个操作。也就是说GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。所以GET在信息修改层面，GET比POST安全。GET 请求一般不应产生副作用。它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

在说过了GET、POST实际用法后，我们发现很多人都没有按照HTTP规范（<http://www.ietf.org/rfc/rfc2616.txt>）去做。导致这个问题的原因有很多，比如说：

1. 很多人贪方便，更新资源时用了GET，因为用POST必须要到FORM（表单），这样会麻烦一点。
2. 对资源的增，删，改，查操作，其实都可以通过GET/POST完成，不需要用到PUT和DELETE。
3. 早期的Web MVC框架设计者们并没有有意识地将URL当作抽象的资源来看待和设计，所以导致一个比较严重的问题是传统的Web MVC框架基本上都只支持GET和POST两种HTTP方法，而不支持PUT和DELETE方法。

以上3点都是没有严格遵守HTTP规范，随着架构的发展，出现了REST(Representational State Transfer)，一套支持HTTP规范的RESTful架构。

REST这个词，是Roy Thomas Fielding在他2000年的博士论文中提出的。REST即Representational State Transfer的缩写。直接翻译是”表现层状态转化”。具体请查看：<http://zh.wikipedia.org/wiki/REST>

在使用CXF开发Web Service，尤其是在RESTful时，时常会遇到如下的格式报错：

No operation matching request path "xxxxxxx" is found, Relative Path: /, HTTP Method: POST, ContentType: application/x-www-form-urlencoded;charset=UTF-8, Accept: \*/\*,. Please enable FINE/TRACE log level for more details.

崩不崩溃，刺不刺激 ; )

总结一下啊！

HTTP-GET的处理特征如下：

1、将数据添加到URL

2、利用一个问号（”?”）代表URL地址的结尾与数据的开端。

3、每一个数据的元素以 名称/值 (name/value)　的形式出现。

4、利用一个分号(“;”)来区分多个数据元素。

HTTP-POST的处理特征如下：

1、将数据包括在HTTP主体中。

2、同样的，数据的元素以 名称/值 (name/value)　的形式出现。

3、但是每一个数据元素分别占用主体的一行。

[【前端基础系列】理解GET与POST请求区别](https://www.cnblogs.com/GeniusLyzh/p/9327319.html)

语义区别

* GET请求用于获取数据
* POST请求用于提交数据

缓存

* GET请求能被缓存，以相同的URL再去GET请求会返回304
* POST请求不能缓存

数据长度

HTTP协议从未规定过GET/POST请求长度是多少，所谓的请求长度限制由浏览器和Web服务器决定的，各种浏览器和 web 服务器的设定均不一样，这依赖于各个浏览器厂家的规定或者可以根据 web 服务器的处理能力来设定。传统IE中URL的最大可用长度为2048字符，其他浏览器对URL长度限制实现上有所不同，POST请求无长度限制（目前理论上是这样的）。

数据包

多数浏览器对于POST采用两阶段发送数据的，先发送请求头，再发送请求体，即使参数再少再短，也会被分成两个步骤来发送（相对于GET），也就是第一步发送header数据，第二步再发送body部分。HTTP是应用层的协议，而在传输层有些情况TCP会出现两次连接的过程，HTTP协议本身不保存状态信息，一次请求一次响应。对于TCP而言，通信次数越多反而靠性越低，能在一次连结中传输完需要的消息是最可靠的，尽量使用GET请求来减少网络耗时。如果通信时间增加，这段时间客户端与服务器端一直保持连接状态，在服务器侧负载可能会增加，可靠性会下降。

安全

这里的「安全」和通常理解的「安全」意义不同，如果一个方法的语义在本质上是「只读」的，那么这个方法就是安全的。客户端向服务端的资源发起的请求如果使用了是安全的方法，就不应该引起服务端任何的状态变化，因此也是无害的。 此RFC定义，GET, HEAD, OPTIONS 和 TRACE 这几个方法是安全的。

但是这个定义只是规范，并不能保证方法的实现也是安全的。

幂等

幂等的概念是指同一个请求方法执行多次和仅执行一次的效果完全相同。按照RFC规范，PUT，DELETE和安全方法都是幂等的。同样，这也仅仅是规范，服务端实现是否幂等是无法确保的。引入幂等主要是为了处理同一个请求重复发送的情况，比如在请求响应前失去连接，如果方法是幂等的，就可以放心地重发一次请求。这也是浏览器在后退/刷新时遇到POST会给用户提示的原因：POST语义不是幂等的，重复请求可能会带来意想不到的后果。

[**HTTP:Post和Get详解**](https://www.cnblogs.com/flysnail/archive/2011/06/19/2084677.html)

* 一 原理区别

　　 一般在浏览器中输入网址访问资源都是通过GET方式；在FORM提交中，可以通过Method指定提交方式为GET或者POST，默认为GET提交Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE

　　URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查 ，改 ，增 ，删 4个操作。GET一般用于获取/查询 资源信息，而POST一般用于更新资源信息。

　　根据HTTP规范，GET用于信息获取，而且应该是安全的和幂等的 。

　　1.所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

　　\* 注意：这里安全的含义仅仅是指是非修改信息。

　　2.幂等的意味着对同一URL的多个请求应该返回同样的结果。这里我再解释一下幂等 这个概念：

　　幂等 （idempotent、idempotence）是一个数学或计算机学概念，常见于抽象代数中。

　　幂等有以下几种定义：

　　对于单目运算，如果一个运算对于在范围内的所有的一个数多次进行该运算所得的结果和进行一次该运算所得的结果是一样的，那么我们就称该运算是幂等的。比如绝对值运算就是一个例子，在实数集中，有abs(a) = abs(abs(a)) 。

　　对于双目运算，则要求当参与运算的两个值是等值的情况下，如果满足运算结果与参与运算的两个值相等，则称该运算幂等，如求两个数的最大值的函数，有在在实数集中幂等，即max(x,x) = x 。

看完上述解释后，应该可以理解GET幂等的含义了。

　　但在实际应用中，以上2条规定并没有这么严格。引用别人文章的例子：比如，新闻站点的头版不断更新。虽然第二次请求会返回不同的一批新闻，该操作仍然被认为是安全的和幂等的，因为它总是返回当前的新闻。从根本上说，如果目标是当用户打开一个链接时，他可以确信从自身的角度来看没有改变资源即可。

　　根据HTTP规范，POST表示可能修改变服务器上的资源的请求 。继续引用上面的例子：还是新闻以网站为例，读者对新闻发表自己的评论应该通过POST实现，因为在评论提交后站点的资源已经不同了，或者说资源被修改了。

　　上面大概说了一下HTTP规范中，GET和POST的一些原理性的问题。但在实际的做的时候，很多人却没有按照HTTP规范去做，导致这个问题的原因有很多，比如说：

　　1.很多人贪方便，更新资源时用了GET，因为用POST必须要到FORM（表单），这样会麻烦一点。

　　2.对资源的增，删，改，查操作，其实都可以通过GET/POST完成，不需要用到PUT和DELETE。

　　3.另外一个是，早期的但是Web MVC框架设计者们并没有有意识地将URL当作抽象的资源来看待和设计 。还有一个较为严重的问题是传统的Web MVC框架基本上都只支持GET和POST两种HTTP方法，而不支持PUT和DELETE方法。

　　以上3点典型地描述了老一套的风格（没有严格遵守HTTP规范），随着架构的发展，现在出现REST(Representational State Transfer)，一套支持HTTP规范的新风格，这里不多说了，可以参考《RESTful Web Services》。

* 二 表现形式区别

  搞清了两者的原理区别，我们再来看一下他们实际应用中的区别：

  为了理解两者在传输过程中的不同，我们先看一下HTTP协议的格式：

  HTTP请求：

|  |
| --- |
| <request line>        <headers>        <blank line>  <request-body> |

在HTTP请求中，第一行必须是一个请求行（request line），用来说明请求类型、要访问的资源以及使用的HTTP版本。紧接着是一个首部（header）小节，用来说明服务器要使用的附加信息。在首部之后是一个空行，再此之后可以添加任意的其他数据[称之为主体（body）]。

**GET实例：**

|  |
| --- |
| GET /books/?sex=man&name=Professional HTTP/1.1  Host: www.wrox.com  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)  Gecko/20050225 Firefox/1.0.1  Connection: Keep-Alive |

**POST实例：**

|  |
| --- |
| POST / HTTP/1.1  Host: www.wrox.com  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)  Gecko/20050225 Firefox/1.0.1  Content-Type: application/x-www-form-urlencoded  Content-Length: 40  Connection: Keep-Alive <br>  （----此处空一行----）  <br>name=Professional%20Ajax&publisher=Wiley |

 有了以上对HTTP请求的了解和示例，我们再来看两种提交方式的区别：

　　（1）GET提交，请求的数据会附在**URL之后（就是把数据放置在HTTP协议头中*），*以?分割URL和传输数据，多个参数用&连接；**例如：

|  |
| --- |
| login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0 %E5%A5%BD。 |

如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如： %E4%BD%A0%E5%A5%BD，其中％XX中的XX为该符号以16进制表示的ASCII。

 　　POST提交：把提交的数据放置在是**HTTP包的包体中**。上文示例中红色字体标明的就是实际的传输数据。因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变。

　　（2）传输数据的大小：

　　首先声明：HTTP协议没有对传输的数据大小进行限制，HTTP协议规范也没有对URL长度进行限制。而在实际开发中存在的限制主要有：

GET:特定浏览器和服务器对URL长度有限制，例如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。因此对于GET提交时，传输数据就会受到URL长度的限制。

POST:由于不是通过URL传值，理论上数据不受限。但实际各个WEB服务器会规定对post提交数据大小进行限制，Apache、IIS6都有各自的配置。

　　（3）安全性：

　　POST的安全性要比GET的安全性高。注意：这里所说的安全性和上面GET提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的Security的含义，比如：通过GET提交数据，用户名和密码将明文出现在URL上，因为(1)登录页面有可能被浏览器缓存， (2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用GET提交数据还可能会造成Cross-site request forgery攻击

　　（4）Http get,post,soap协议都是在http上运行的

　　　　1）get：请求参数是作为一个key/value对的序列（查询字符串）附加到URL上的查询字符串的长度受到web浏览器和web服务器的限制（如IE最多支持2048个字符），不适合传输大型数据集同时，它很不安全。

　　　　2）post：请求参数是在http标题的一个不同部分（名为entity body）传输的，这一部分用来传输表单信息，因此必须将Content-type设置: application/x-www-form-urlencoded。post设计用来支持web窗体上的用户字段，其参数也是作为key/value对传输。

　　　　但是：它不支持复杂数据类型，因为post没有定义传输数据结构的语义和规则。

　　　　3）soap：是http post的一个专用版本，遵循一种特殊的xml消息格式Content-type设置为: text/xml 任何数据都可以xml化。

* 三 HTTP响应

1．HTTP响应格式：

|  |
| --- |
| <status line>  <headers>  <blank line>  [<response-body>] |

在响应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行（status line）通过提供一个状态码来说明所请求的资源情况。

HTTP响应实例：

|  |
| --- |
| HTTP/1.1 200 OK  Date: Sat, 31 Dec 2005 23:59:59 GMT  Content-Type: text/html;charset=ISO-8859-1  Content-Length: 122  ＜html＞  ＜head＞  ＜title＞Wrox Homepage＜/title＞  ＜/head＞  ＜body＞  ＜!-- body goes here --＞  ＜/body＞  ＜/html＞ |

2．最常用的状态码有：

◆200 (OK): 找到了该资源，并且一切正常。

◆304 (NOT MODIFIED): 该资源在上次请求之后没有任何修改。这通常用于浏览器的缓存机制。

◆401 (UNAUTHORIZED): 客户端无权访问该资源。这通常会使得浏览器要求用户输入用户名和密码，以登录到服务器。

◆403 (FORBIDDEN): 客户端未能获得授权。这通常是在401之后输入了不正确的用户名或密码。

◆404 (NOT FOUND): 在指定的位置不存在所申请的资源。

* 四 完整示例：

**HTTP GET**

 发送

|  |
| --- |
| GET /DEMOWebServices2.8/Service.asmx/CancelOrder?UserID=string&PWD=string&OrderConfirmation=string HTTP/1.1  Host: api.efxnow.com |

 回复

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type: text/xml; charset=utf-8  Content-Length: length    <?xml version="1.0" encoding="utf-8"?>  <objPlaceOrderResponse xmlns="https://api.efxnow.com/webservices2.3">  <Success>boolean</Success>  <ErrorDescription>string</ErrorDescription>  <ErrorNumber>int</ErrorNumber>  <CustomerOrderReference>long</CustomerOrderReference>  <OrderConfirmation>string</OrderConfirmation>  <CustomerDealRef>string</CustomerDealRef>  </objPlaceOrderResponse> |

**HTTP POST**

发送

|  |
| --- |
| POST /DEMOWebServices2.8/Service.asmx/CancelOrder HTTP/1.1  Host: api.efxnow.com  Content-Type: application/x-www-form-urlencoded  Content-Length: length    UserID=string&PWD=string&OrderConfirmation=string |

回复

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type: text/xml; charset=utf-8  Content-Length: length    <?xml version="1.0" encoding="utf-8"?>  <objPlaceOrderResponse xmlns="https://api.efxnow.com/webservices2.3">  <Success>boolean</Success>  <ErrorDescription>string</ErrorDescription>  <ErrorNumber>int</ErrorNumber>  <CustomerOrderReference>long</CustomerOrderReference>  <OrderConfirmation>string</OrderConfirmation>  <CustomerDealRef>string</CustomerDealRef>  </objPlaceOrderResponse> |

**SOAP 1.2**

发送

|  |
| --- |
| POST /DEMOWebServices2.8/Service.asmx HTTP/1.1  Host: api.efxnow.com  Content-Type: application/soap+xml; charset=utf-8  Content-Length: length    <?xml version="1.0" encoding="utf-8"?>  <soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">  <soap12:Body>    <CancelOrder xmlns="https://api.efxnow.com/webservices2.3">    <UserID>string</UserID>    <PWD>string</PWD>    <OrderConfirmation>string</OrderConfirmation>    </CancelOrder>  </soap12:Body>  </soap12:Envelope> |

回复

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type: application/soap+xml; charset=utf-8  Content-Length: length    <?xml version="1.0" encoding="utf-8"?>  <soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">  <soap12:Body>    <CancelOrderResponse xmlns="https://api.efxnow.com/webservices2.3">    <CancelOrderResult>    <Success>boolean</Success>    <ErrorDescription>string</ErrorDescription>    <ErrorNumber>int</ErrorNumber>    <CustomerOrderReference>long</CustomerOrderReference>    <OrderConfirmation>string</OrderConfirmation>    <CustomerDealRef>string</CustomerDealRef>    </CancelOrderResult>    </CancelOrderResponse>  </soap12:Body>  </soap12:Envelope> |

 在写代码过程中，get与post是两种不同的提交方式。下面，列举出两种方式的不同。

方法/步骤

1. get是从服务器上获取数据，post是向服务器传送数据。
2. get是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTP post机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。
3. 对于get方式，服务器端用Request.QueryString获取变量的值，对于post方式，服务器端用Request.Form获取提交的数据。
4. get传送的数据量较小，不能大于2KB。post传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。
5. get安全性非常低，post安全性较高。
6. HTTP 定义了与服务器交互的不同方法，最基本的方法是 GET 和 POST。事实上 GET 适用于多数请求，而保留 POST 仅用于更新站点。根据 HTTP 规范，GET 用于信息获取，而且应该是 安全的和幂等的。所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。幂等的意味着对同一 URL 的多个请求应该返回同样的结果。完整的定义并不像看起来那样严格。从根本上讲，其目标是当用户打开一个链接时，她可以确信从自身的角度来看没有改变资源。 比如，新闻站点的头版不断更新。虽然第二次请求会返回不同的一批新闻，该操作仍然被认为是安全的和幂等的，因为它总是返回当前的新闻。反之亦然。POST 请求就不那么轻松了。POST 表示可能改变服务器上的资源的请求。仍然以新闻站点为例，读者对文章的注解应该通过 POST 请求实现，因为在注解提交之后站点已经不同了
7. 在FORM提交的时候，如果不指定Method，则默认为GET请求，Form中提交的数据将会附加在url之后，以?分开与url分开。字母数字字符原 样发送，但空格转换为“+“号，其它符号转换为%XX,其中XX为该符号以16进制表示的ASCII（或ISO Latin-1）值。GET请求请提交的数据放置在HTTP请求协议头中，而POST提交的数据则放在实体数据中；GET方式提交的数据最多只能有1024字节，而POST则没有此限制。

END

经验内容仅供参考，如果您需解决具体问题(尤其法律、医学等领域)，建议您详细咨询相关领域专业人士。

[举报](javascript:void(0))作者声明：本篇经验系本人依照真实经历原创，未经许可，谢绝转载。

[**GET、POST请求的区别分析**](https://segmentfault.com/a/1190000019635464)

1 前言

众所周知，在我们开发项目的过程中，关于POST与GET请求是我们不得不掌握的知识，那么它们两者之间又有什么区别呢？接下来，我们一起从HTTP报文等角度来探讨学习一下关于两者的不同

2 HTTP请求方法

HTTP 定义了一组请求方法, 以表明要对给定资源执行的操作。指示针对给定资源要执行的期望动作. 虽然他们也可以是名词, 但这些请求方法有时被称为HTTP动词. 每一个请求方法都实现了不同的语义, 但一些共同的特征由一组共享：: 例如一个请求方法可以是 safe, idempotent, 或 cacheable.

| 方法名称 | 用法 |
| --- | --- |
| GET | GET方法请求一个指定资源的表示形式. 使用GET的请求应该只被用于获取数据. |
| HEAD | HEAD方法请求一个与GET请求的响应相同的响应，但没有响应体. |
| POST | 用于将实体提交到指定的资源，通常导致在服务器上的状态变化或副作用. |
| PUT | PUT方法用请求有效载荷替换目标资源的所有当前表示。 |
| DELETE | DELETE方法删除指定的资源。 |
| CONNECT | CONNECT方法建立一个到由目标资源标识的服务器的隧道。 |
| OPTIONS | OPTIONS方法用于描述目标资源的通信选项。 |
| TRACE | TRACE方法沿着到目标资源的路径执行一个消息环回测试。 |
| PATCH | PATCH方法用于对资源应用部分修改。 |

3 标准参考

W3school:[http://www.w3school.com.cn/ta...](http://www.w3school.com.cn/tags/html_ref_httpmethods.asp)

| 项目 | GET | POST |
| --- | --- | --- |
| 后退按钮/刷新 | 无害 | 数据会被重新提交（浏览器应该告知用户数据会被重新提交）。 |
| 书签 | 可收藏为书签 | 不可收藏为书签 |
| 缓存 | 能被缓存 | 不能缓存 |
| 编码类型 | application/x-www-form-urlencoded | application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。 |
| 历史 | 参数保留在浏览器历史中。 | 参数保留在浏览器历史中。 |
| 对数据长度的限制 | 是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。 | 无限制 |
| 对数据类型的限制 | 只允许 ASCII 字符。 | 没有限制。也允许二进制数据。 |
| 安全性 | 与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET ！ | POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。 |
| 可见性 | 数据在 URL 中对所有人都是可见的。 | 数据不会显示在 URL 中。 |

4 副作用和幂等的概念

副作用指对服务器上的资源做改变，搜索是无副作用的，注册是副作用的。

幂等指发送 M 和 N 次请求（两者不相同且都大于 1），服务器上资源的状态一致，比如注册 10 个和 11 个帐号是不幂等的，对文章进行更改 10 次和 11 次是幂等的。因为前者是多了一个账号（资源），后者只是更新同一个资源。

在规范的应用场景上说，Get 多用于无副作用，幂等的场景，例如搜索关键字。Post 多用于副作用，不幂等的场景，例如注册。

5 技术上的不同

* Get 请求能缓存，Post 不能
* Post 相对 Get 安全一点点，因为Get 请求都包含在 URL 里（当然你想写到 body 里也是可以的），且会被浏览器保存历史纪录。Post 不会，但是在抓包的情况下都是一样的。
* URL有长度限制，会影响 Get 请求，但是这个长度限制是浏览器规定的，不是 RFC 规定的
* Post 支持更多的编码类型且不对数据类型限制

6 首部

首部分为请求首部和响应首部，并且部分首部两种通用，接下来我们就来学习一部分的常用首部。

1. 通用首部

| 通用字段 | 作用 |
| --- | --- |
| Cache-Control | 控制缓存的行为 |
| Connection | 浏览器想要优先使用的连接类型，比如 keep-alive |
| Date | 创建报文时间 |
| Pragma | 报文指令 |
| Via | 代理服务器相关信息 |
| Transfer-Encoding | 传输编码方式 |
| Upgrade | 要求客户端升级协议 |
| Warning | 在内容中可能存在错误 |

1. 请求首部

| 请求首部 | 作用 |
| --- | --- |
| Accept | 能正确接收的媒体类型 |
| Accept-Charset | 能正确接收的字符集 |
| Accept-Encoding | 能正确接收的编码格式列表 |
| Accept-Language | 能正确接收的语言列表 |
| Expect | 期待服务端的指定行为 |
| From | 请求方邮箱地址 |
| Host | 服务器的域名 |
| If-Match | 两端资源标记比较 |
| If-Modified-Since | 本地资源未修改返回 304（比较时间） |
| If-None-Match | 本地资源未修改返回 304（比较标记） |
| User-Agent | 客户端信息 |
| Max-Forwards | 限制可被代理及网关转发的次数 |
| Proxy-Authorization | 向代理服务器发送验证信息 |
| Range | 请求某个内容的一部分 |
| Referer | 表示浏览器所访问的前一个页面 |
| TE | 传输编码方式 |

1. 响应首部

| 响应首部 | 作用 |
| --- | --- |
| Accept-Ranges | 是否支持某些种类的范围 |
| Age | 资源在代理缓存中存在的时间 |
| ETag | 资源标识 |
| Location | 客户端重定向到某个 URL |
| Proxy-Authenticate | 向代理服务器发送验证信息 |
| Server | 服务器名字 |
| WWW-Authenticate | 获取资源需要的验证信息 |

1. 实体首部

| 实体首部 | 作用 |
| --- | --- |
| Allow | 资源的正确请求方式 |
| Content-Encoding | 内容的编码格式 |
| Content-Language | 内容使用的语言 |
| Content-Length | request body 长度 |
| Content-Location | 返回数据的备用地址 |
| Content-MD5 | Base64加密格式的内容 MD5检验值 |
| Content-Range | 内容的位置范围 |
| Content-Type | 内容的媒体类型 |
| Expires | 内容的过期时间 |
| Last\_modified | 内容的最后修改时间 |

7 GET 和 POST 报文上的区别

结论：GET 和 POST 方法没有实质区别，只是报文格式不同。

GET 和 POST 只是 HTTP 协议中两种请求方式，而 HTTP 协议是基于 TCP/IP 的应用层协议，无论 GET 还是 POST，用的都是同一个传输层协议，所以在传输上，没有区别。

报文格式上，不带参数时，最大区别就是第一行方法名不同

POST方法请求报文第一行是这样的POST /products/create HTTP/1.1

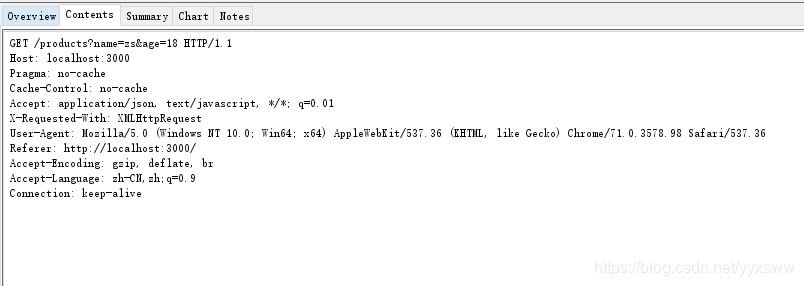
GET方法请求报文第一行是这样的 GET /products?name=zs&age=18 HTTP/1.1

是的，不带参数时他们的区别就仅仅是报文的前几个字符不同而已

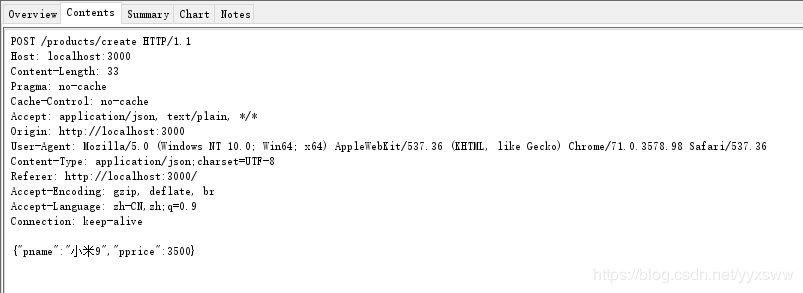
带参数时报文的区别呢？ 在约定中，GET 方法的参数应该放在 url 中，POST 方法参数应该放在 body 中

举个例子，如果参数是 pname='小米9', pprice=3500

GET 请求方法的报文如下：



POST 请求方法的报文如下：



**现在我们知道了两种方法本质上是 TCP 连接，没有差别，也就是说，如果我不按规范来也是可以的。我们可以在 URL 上写参数，然后方法使用 POST；也可以在 Body 写参数，然后方法使用 GET。当然，这需要服务端支持。**

接口评审：明确出需求 然后开会评审 要什么接口 接口参数、返回json内容、格式 协定好 再做

**第三方接口评审规范**

**接口测试&Python实现接口测试**

一、接口测试是什么

1、接口测试是软件测试

2、接口测试是集成测试的一部分

3、接口测试不等同于接口自动化测试

4、大家常说的接口测试大部分是指功能性的接口测试，实际还包括性能的接口测试和安全性的接口测试等

5、需要做接口测试的情况大致为：

[1]系统与系统之间的调用。例如：淘宝的订单系统和支付系统；

[2]上层服务对下层服务的调用。例如：服务层对数据层的调用；

[3]服务之间的调用。

二、接口测试的意义

1、稳：接口相对UI稳定，当接口自动化创建后，相对稳定的运行

2、低（成本）：因为接口相对稳定，所以一旦建立，不需要大量的维护成本

3、快：执行快、响应快

三、接口测试的流程

流程：

接口文档 — 接口测试计划、方案 — 接口测试用例（评审）— 执行 — 集成到Jenkins — 接口反馈

1、接口文档

[1]接口文档五要素：接口地址、接口请求的方式、是否有请求参数（参数相关属性）、返回参数说明（参数相关属性）、返回结果样例。



[2]如果没有接口文档，到功能测试阶段，需要自己抓包，抓包工具如Fiddler等

2、设计接口测试用例的原则：看测试的目的是什么，如果目的是调通，那么力度可以小点；如果是业务层面，那么需要结合需求文档，用例需要覆盖全面。

3、设计接口测试用例的目的：

[1]检查返回数据类型与接口文档是否一致；

[2]检查返回字段值与数据库值是否一致；

4、测试点

[1]单一接口功能的测试主要测试返回的数据结构是否和接口文档给出的一致

[2]接口的正常功能是否完成

[3]接口的参数检查测试，接口的异常测试

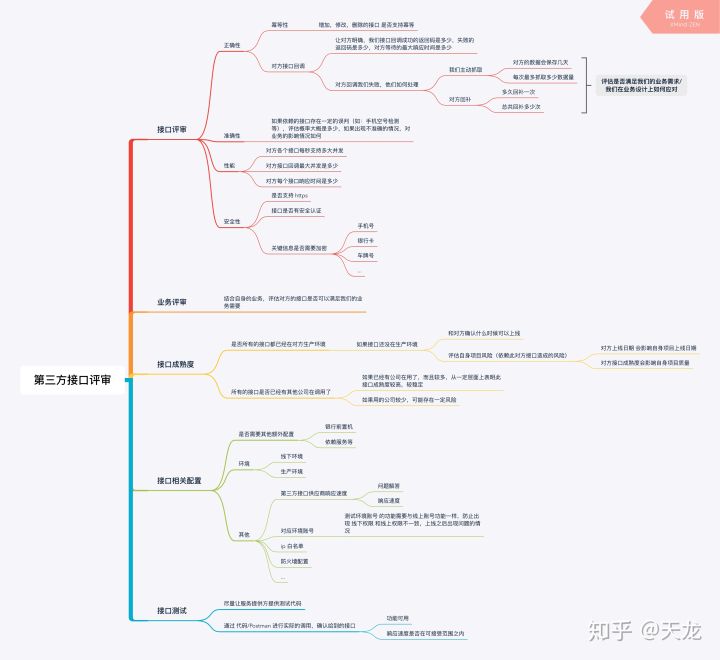
[4]多接口组合测试，实际上是在测试一个业务流。

[5]在测试过程中一次调用多个接口。

四、代码实现

****

****



# 第三方接口评审

## 接口评审

### 正确性

- 幂等性

- 增加，修改，删除的接口 是否支持幂等

- 对方接口回调

- 让对方明确，我们接口回调成功的返回码是多少，失败的返回码是多少，对方等待的最大响应时间是多少

- 对方回调我们失败，他们如何处理

- 我们主动抓取

- 对方的数据会保存几天

- 每次最多抓取多少数据量

- 对方回补

- 多久回补一次

- 总共回补多少次

### 准确性

- 如果依赖的接口存在一定的误判（如：手机空号检测等），评估概率大概是多少，如果出现不准确的情况，对业务的影响情况如何

### 性能

- 对方各个接口每秒支持多大并发

- 对方接口回调最大并发是多少

- 对方每个接口响应时间是多少

### 安全性

- 是否支持 https

- 接口是否有安全认证

- 关键信息是否需要加密

- 手机号

- 银行卡

- 车牌号

- ...

## 业务评审

### 结合自身的业务，评估对方的接口是否可以满足我们的业务需要

## 接口成熟度

### 是否所有的接口都已经在对方生产环境

- 如果接口还没在生产环境

- 和对方确认什么时候可以上线

- 评估自身项目风险（依赖此对方接口造成的风险）

- 对方上线日期 会影响自身项目上线日期

- 对方接口成熟度会影响自身项目质量

### 所有的接口是否已经有其他公司在调用了

- 如果已经有公司在用了，而且较多，从一定层面上表明此接口成熟度较高，较稳定

- 如果用的公司较少，可能存在一定风险

## 接口相关配置

### 是否需要其他额外配置

- 银行前置机

- 依赖服务等

### 环境

- 线下环境

- 生产环境

### 其他

- 第三方接口供应商响应速度

- 问题解答

- 响应速度

- 对应环境账号

- 测试环境账号 的功能需要与线上账号功能一样，防止出现 线下权限 和线上权限不一致，上线之后出现问题的情况

- ip 白名单

- 防火墙配置

- ...

## 接口测试

### 尽量让服务提供方提供测试代码

### 通过 代码/Postman 进行实际的调用，确认给到的接口

- 功能可用

- 响应速度是否在可接受范围之内

[**盗亦有道：如何来评审接口文档**](https://www.cnblogs.com/xiashiwendao/p/3492100.html)

评审一项东西，首先是要你自己心理有谱，需要评审那些点，所以评审本质是评审两样东西：第一，该有的点是否都有；第二，这些点是否达到了要求。

在评审接口文档的时候，需要评审：

1.接口参数是否足够，要知道参数本质就是描摹接口，所以参数应该是对一个接口方法的一个很好的说明；场景是否都满足；参数的描述是否足够具体，便于他人了解；

2.返回值是否足够；返回参数的描述是否具体，便于他人了解；

3.错误返回是否全面；

4.示例是否准确，清晰。

**接口管理规范：**

**如何写出完美的接口：接口规范定义、接口管理工具推荐**

无规矩不成方圆，为了开发人员间更好的配合，我特意整理了这么一篇文档供大家参考学习，如有意见、见解，请在评论区留言探讨。

     接口规范说起来大，其实也就那么几个部分，接口规范、接口管理工具、接口文档编写、开发文档编写。

接口规范定义

一、协议规范

    为了确保不同系统/模块间的数据交互，需要事先约定好通讯协议，如：TCP、HTTP、HTTPS协议。为了确保数据交互安全，建议使用HTTPS协议。

二、接口路径规范

    作为接口路径，为了方便清晰的区分来自不同的系统，可以采用不同系统/模块名作为接口路径前缀。

格式规范如下：

    支付模块   /pay/xx

    订单模块  /order/xx

三、版本控制规范

    为了便于后期接口的升级和维护，建议在接口路径中加入版本号，便于管理，实现接口多版本的可维护性。如果你细心留意过的话，你会发现好多框架对外提供的API接口中(如：Eureka)，都带有版本号的。如：接口路径中添加类似"v1"、"v2"等版本号。

格式规范如下：

      /xx/v1/xx

更新版本后可以使用v2、v3等、依次递加。

四、接口命名规范

    和Java命名规范一样，好的、统一的接口命名规范，不仅可以增强其可读性，而且还会减少很多不必要的口头/书面上的解释。

    可结合【接口路径规范】、【版本控制规范】，外加具体接口命名(路径中可包含请求数据，如：id等)，建议具体接口命名也要规范些，可使用"驼峰命名法"按照实现接口的业务类型、业务场景等命名，有必要时可采取多级目录命名，但目录不宜过长，两级目录较为适宜。

格式规范如下：

    /user/v1/sys/login     用户服务/模块的系统登录接口

   /zoo/v1/zoos/{ID}        动物园服务/模块中，获取id为ID的动物

具体接口命名，通常有以下两种方式：

接口名称动词前/后缀化

    接口名称以接口数据操作的动词为前/后缀，常见动词有：add、delete、update、query、get、send、save、detail、list等，如：新建用户addUser、查询订单详情queryOrderDetail。

接口名称动词+请求方式

     接口路径中包含具体接口名称的名词，接口数据操作动作以HTTP请求方式来区分。常用的HTTP请求方式有：

    GET：从服务器取出资源（一项或多项）。

    POST：在服务器新建一个资源。

    PUT：在服务器更新资源（客户端提供改变后的完整资源）。

    PATCH：在服务器更新资源（客户端提供改变的属性）。

    DELETE：从服务器删除资源。

如：

    GET /zoo/v1/zoos：列出所有动物园

    POST /zoo/v1/zoos：新建一个动物园

    GET /zoo/v1/zoos/{ID}：获取某个指定动物园的信息

    PUT /zoo/v1/zoos/{ID}：更新某个指定动物园的信息（提供该动物园的全部信息）

    PATCH /zoo/v1/zoos/{ID}：更新某个指定动物园的信息（提供该动物园的部分信息）

    DELETE /zoo/v1/zoos/{ID}：删除某个动物园

    GET /zoo/v1/zoos/{ID}/animals：列出某个指定动物园的所有动物

    DELETE /zoo/v1/zoos/ID/animals/ID：删除某个指定动物园的指定动物

五、请求参数规范

请求方式：

按照GET、POST、PUT等含义定义，避免出现不一致现象，对人造成误解、歧义。

请求头：

请求头根据项目需求添加配置参数。如：请求数据格式，accept=‘application/json’等。如有需要，请求头可根据项目需求要求传入用户token、唯一验签码等加密数据。

请求参数/请求体：

        请求参数字段，尽可能与数据库表字段、对象属性名等保持一致，因为保持一致最省事，最舒服的一件事。

六、返回数据规范

    统一规范返回数据的格式，对己对彼都有好处，此处以json格式为例。返回数据应包含：返回状态码、返回状态信息、具体数据。

格式规范如下：

{

    "status":"000000",

    "msg":"success",

    "data": {

        //json格式的具体数据

    }

}

        返回数据中的状态码、状态信息，常指具体的业务状态，不建议和HTTP状态码混在一起。HTTP状态，是用来体现 HTTP链路状态情况，如：404-Not Found。HTTP状态码和json结果中的状态码，并存尚可，用于体现不同维度的状态。

接口管理工具推荐

    接口开发完后，最终的目的是提供给其他系统/模块来使用的，因此，接口的管理是必不可少的。

接口管理的痛点

    接口的管理常常面临很多的痛苦，这里就列举几个常见的，看看你是否也遇到过。

系统/模块太多、接口太多，没有系统统一管理所有接口。

代码修改后，接口文档没有及时更新，造成接口文档和实际接口不一致的现象。

接口管理系统自主研开发成本高。

接口管理缺少接口mock功能。

接口管理工具推荐

       在日常工作过程中用过、接触过的接口管理工具也是不尽其数，下面介绍你可能使用过、没有使用过的接口管理工具，同时也介绍这些接口管理工具的优缺点。

word

      相信大家之前用来管理接口比较多的应该是word吧，开发人员将系统的接口维护在word文档里，不管是组内沟通还是和其他团队的接口沟通都离不开这些接口文档，每次修改文档和代码都要同步修改。相信使用word的缺点大家应该也很清楚，就是维护和管理很麻烦，我们经常会遇到文档和代码不一致的情况，大部分不一致都是因为接口因为种种原因修改了，开发人员大部分都是只改了代码里的接口实现，而没有去修改接口文档。而且word文档搜索接口也很麻烦，没办法建全局索引，只能一个个文档点开查看，想想就很痛苦。但不可否认的是，word对于一些小团队用起来还是挺方便的，不用搭建系统，给谁一看就明白。

自建接口管理系统

       对于一些有一定规模的企业，在各项工程管理活动上都非常正规，各种ISO标准要遵守，自然对接口管理的要求也非常高，之前在国有银行，我们就是自建了接口管理系统，自建还是很消耗人力成本的，从开发到后续运维，都要消耗人力，但是自建的好处就是，可以根据公司的要求进行各种花样的定制，我们之前在接口管理系统中加入了很多好用的定制功能，例如接口被哪些系统调用、接口是在哪个批次投产又在哪个批次做过变更等等，这对于架构师来说非常好用，用于分析接口影响范围非常方便。目前开源的接口管理系统还没有能做到这些定制化功能的。

wiki

      之前在小团队的时候还用过一段时间的私有wiki，wiki特别适合于小团队高速线性迭代开发，在wiki上看到的就是最新的接口，团队内所有成员看到的都是一样的，如果接口有变化，相关开发人员修改后立即生效，保证了顺畅的接口沟通。但是wiki的缺点也很多，接口文档只是静态页面，无法实现一些动态效果，无法实现追溯等等缺点。

RAP

       相信很多互联网公司都在使用RAP，RAP是阿里开源的一套接口管理系统，RAP可以比较方便的管理公司所有系统的接口，同时还有比较完善的权限管理，还可以做接口mock，方便开发人员在接口功能还没有完成的时候能够及时发布出去，给调用方去使用。但是RAP的缺点就是每个接口都需要维护进去，接口修改后也需要及时维护，当时我们在使用的时候遇到的最大的问题也是经常碰到接口没有及时维护的问题。

swagger



      上面说的那些接口管理工具，其实都有一个很大的问题就是修改代码后需要同步维护接口文档，但是让程序员去修改文档是很难的，大部分程序员都比较讨厌维护各类文档。当我第一次了解到swagger的时候，发现这简直就是为程序员定制的接口管理工具，swagger定义了很多注解，在对接口加上swagger相关的注解，当接口代码修改后，swagger在工程启动后会根据代码自动生成最新的接口html文档，同时swagger提供了mock接口模拟的功能，也能够更加方便的模拟接口，并且还能够在swagger界面上直接发起接口调用，可以方便调用方在还没写代码的时候就能够尝试下接口调用后的结果。

      看了那么多swagger的优点，下面也说说swagger的缺点，那就是swagger是跟随着每个工程一起启动的，这就导致每个工程都有一个swagger的访问地址，如果公司系统很多的话，那就会导致查看不同系统的接口都要到不同的地址去查看，每个开发都要自己收藏好各个系统的swagger地址。有些公司也自己开发了统一网关，将所有swagger的接口地址聚合起来，但是多少还是涉及到一些开发工作的，而且做的还不一定很完善。

Easy Mock

        官网的这张图基本上介绍清楚了easymock的核心功能，这其中我最看重的功能有两块，一个是能够集成swagger接口并集中管理所有接口，另一个就是响应式数据。

       EasyMock能够根据swagger接口的地址自动导入所有swagger接口，非常方便，对于非swagger的接口也可以手工维护进去，这样可以很方便的做到全公司接口统一维护，而且也有比较完善的接口权限管理，方便分组管理。但缺点就是过于庞大，可能太适合小一点项目或团队。



      上面提及到接口管理工具，大家可根据自己项目的规模、需求，进行实际选择，切记生搬硬套。

**接口规范文档总结、接口管理工具推荐、如何写出完美的接口**

写在前面：这是我最近整理的接口规范文档，无规矩不成方圆，为了app开发人员与后台接口开发人员更好的配合，我特意整理了这么一篇文档供大家参考学习，如有意见请在评论区留言谢谢。因部分内容涉及公司代码，我对本文档略有删减。

接口规范说起来大，其实也就那么几个部分，接口规范、接口管理工具、接口文档编写、开发文档编写。以下将详细介绍，下面进入正文：

接口规范文档

具体内容如下：

一：协议规范

二：域名规范

三：版本控制规范

四：API路径规范

五：API命名规范

六：请求参数规范

七：列表请求特殊规范

八：返回数据规范

九：接口文档规范

十：接口管理工具使用教程

参与编写

更新日期

版本

备注

AbyssKitty

2018/04/06

V1.1.0

无

V1.1.0更新日志：

1. 新增协议规范、域名规范、版本控制规范、列表特殊规范。

2. 更新接口管理工具使用教程。

3. 美化排版。

正文：

一：协议规范

为进一步确保数据交互安全。正式地址（生产地址）必须遵循HTTPS协议。

二：域名规范

每个项目要有且仅有一个自己唯一的域名+端口。在项目配置文件中要添加静态变量专门进行存储。

如果一个域名满足不了要求，那么就需要再添加一个。

格式规范如下：

（java）public static final String URL\_BASE = “https://127.0.0.1:8080/”;

（java）public static final String URL\_BASE\_SUB = “https://192.168.0.1:8080/”;

必须以https开头，并以“/”结尾。

三：API路径规范

作为接口路径，为了和其他路径完美区分，必须在路径中添加api目录

格式规范如下：

（java）public static final String URL\_API = “api/”;

（PHP）php目录是加index.php/api/

必须以字母开头，并以“/”结尾。

四：版本控制规范

项目正式上线后，正式版本要确定接口版本、并备份接口代码。

为方便管理，需要在接口路径中加入版本号信息。

格式规范如下：

（java）public static final String URL\_VERSION =”v1/”;

必须以字母开头，并以“/”结尾。

更新版本后可以使用v2 v3等、依次递加。

五：API命名规范

根据二：域名规范、三：API路径规范、四：版本控制规范。项目中必须在配置文件中增加BaseUrl静态常量。值=三个相加。

格式规范如下：

（java）public static final String BASEURL=URL\_BASE+URL\_API+URL\_VERSION;

具体代码如下：

BASEURL = [“https://127.0.0.1:8080/api/v1/”]

BASEURL = [“https://127.0.0.1:8080/api/v1/”]

BASEURL = [“https://127.0.0.1:8080/api/v1/”]

重要的事情说三遍。

根据业务需求，可以在v1版本文件夹里创建，一个或者多个接口文件。

一个的规范：

https://127.0.0.1:8080/api/v1/getBanner

这就是一个获取banner的接口。

多个的规范是根据业务需求来区分：

https://127.0.0.1:8080/api/v1/home/getBanner

https://127.0.0.1:8080/api/v1/user/userLogin

新建user文件，里面存放用户级别的操作：如登陆、注册、修改密码等等。

新建sms文件，里面存放对短信的接口操作：如发送验证码、验证手机号等等。

所以，接口方法文件必须要有自己的规范，命名必须统一使用驼峰命名法或者下划线拼接命名法。举个栗子：（upperCamelCase）（upper\_camel\_case）。所有接口命名方式，必须遵循如下规范。

（1）新增方法：如新增一个地址、新增一个联系人。

命名规范：

必须以“add”为前缀。例如addAddress

事例地址：https://127.0.0.1:8080/api/v1/addAddress

（2）删除方法：如删除一个地址。

命名规范：

必须以“delete”为前缀。例如deleteAddress

事例地址：https://127.0.0.1:8080/api/v1/deleteAddress

（3）修改方法：如修改一个地址。

命名规范：

必须以“updata”为前缀。例如updataAddress

事例地址：https://127.0.0.1:8080/api/v1/updataAddress

（4）获取方法：如获取一个地址。

命名规范：

必须以“get”为前缀。例如getAddress

事例地址：https://127.0.0.1:8080/api/v1/getAddress

（5）获取列表方法：如获取一个地址列表。

命名规范：

必须以“get”为前缀、“List”为后缀。例如getAddressList

事例地址：https://127.0.0.1:8080/api/v1/getAddressList

其他规范：

发送验证码使用‘send’为前缀、保存一个数据以‘save’为前缀、上传图片以‘uploadImage’为名称等等。

具体地址就等于（BASEURL+“address/getAddressList”）

目的：一目了然、降低维护成本。

六：请求参数规范

请求方式：公共数据使用get方式请求，私有数据使用post方式请求。尽量全部是用post。

请求头：请求头根据项目需求添加配置参数。如：accept=‘application/json

’等。请求头根据项目需求可以要求传入用户token、app名称版本、唯一验签码等加密数据。

    请求参数：

根据数据库字段进行命名、保持一致最省事。

七：列表请求特殊规范

列表请求，请求参数规范，必须传参：页数和每页个数的字段。并可包含查询等信息。

1. 列表接口必传字段（分页、使用小写字母）。

page ：页数，从1开始。例如：{ “page”: 1  }

subnumber : 每页数量。

2. 列表接口选传字段。

只要是列表接口、一般情况下都会存在检索条件，例如淘宝商品检索。检索条件为选填。

后台需进行非空非null判断，选传字段为空为null默认查询全部。有值则需要接收，并进行sql查询。

规范事例：

普通列表接口：https://127.0.0.1:8080/api/v1/getBannerList

（不传page、后台默认返回全部数据）

（banner接口不需要使用检索条件）

需检索列表接口：https://127.0.0.1:8080/api/v1/getOrderList

（不传page、后台默认返回全部数据）

（Order订单接口需要检索条件，不传就不检索，只进行分页查询）

（如果有 time price等检索条件，不传就不检索，传了就进行条件查询，并返回相应数据）

八：返回数据规范

注：列表数据返回，没有特殊情况的条件下，必须最新数据在上面，依次排序。

返回事例：

{

"list":[],

"object":{}, //"object":""

"status":"SUCCESS",

"message":"我是提示消息",

 ...

"page":1,

"subnumber":10,

}

必选-命名规范：驼峰命名法。

必选-新增键值规则：名字对应固定的格式（list就是数组[]）。

举个栗子：比如一个"list":[]满足不了需求，那么可以新增一个"map":[]。

比如一个"object":{"name":"小明"}满足不了需求，那么可以新增一个"details":{"name":"小红"}。名字对应固定的格式，数组就是数组、实体类就是实体类、字段就是字段。不能data在这个接口返回的是实体类、另一个接口又返回数组了。需要特别注意。

必选-list：list列表（数组）为空时显示[]。

必选-object：实体数据，json键值对。

必选-status：状态信息=SUCCESS、ERROR等静态变量。

必选-message：提示消息。（加载成功、）

可选-page：页数（分页查新时使用、显示第几页从一开始）。

可选-subnumber：每页的格式（分页查询时使用、显示当前页的个数）。

九：接口文档规范

接口文档需要包含以下部分：

文档名称。

版本号。

编写人。

编写、修改日期。

baseUrl地址。

更新日志。

接口详情。（详情规范如下）

接口详情编辑规范：

一个完整的接口需要由以下几部分组成

1.请求地址 例如：https://127.0.0.1:8080/xxx/xxx/xxx

2.请求方式 例如：POST、GET等

3.请求参数 例如：传 id：“1”，name：“小明”

4.返回参数 例如：{ json... } 【参考上面的接口规范】

5.返回事例 例如：{ json... }

十：接口管理工具使用教程

接口管理工具有很多，例如RAP、eolinker等等。

接口管理工具基本的作用都是用来管理接口的。这里简单介绍eolinker的使用方法。

使用方法步骤：

创建接口管理项目。

邀请开发者同事加入。

编写接口（接口地址、请求参数及备注、请求方式、返回参数及备注、返回事例、在线测试接口）。

开发者使用接口。

过程中灵活配合，接口可以灵活更新。

完成项目后可以导出接口文档。

附件：XXX接口管理工具使用教程点击进入eolinker使用教程

RAP的特色：

RAP是一个GUI的WEB接口管理工具。在RAP中，您可定义接口的URL、请求&响应细节格式等等。通过分析这些数据，RAP提供MOCK服务、测试服务等自动化工具。RAP同时提供大量企业级功能，帮助企业和团队高效的工作。

在前后端分离的开发模式下，我们通常需要定义一份接口文档来规范接口的具体信息。如一个请求的地址、有几个参数、参数名称及类型含义等等。RAP 首先方便团队录入、查看和管理这些接口文档，并通过分析结构化的文档数据，重复利用并生成自测数据、提供自测控制台等等... 大幅度提升开发效率。

强大的GUI工具 给力的用户体验，你将会爱上使用RAP来管理您的API文档。

完善的MOCK服务 文档定义好的瞬间，所有接口已经准备就绪。有了MockJS，无论您的业务模型有多复杂，它都能很好的满足。

庞大的用户群 RAP在阿里巴巴有200多个大型项目在使用，也有许多著名的公司、开源人士在使用。RAP跟随这些业务的成行而成长，专注细节，把握质量，经得住考验。

免费 + 专业的技术支持 RAP是免费的，而且你的技术咨询都将在24小时内得到答复。大多数情况，在1小时内会得到答复。

RAP是一个可视化接口管理工具 通过分析接口结构，动态生成模拟数据，校验真实接口正确性， 围绕接口定义，通过一系列自动化工具提升我们的协作效率。

完本。

[**API接口规范(试行版)**](https://www.cnblogs.com/jackgeek/p/ju-miAPI-jie-kou-gui-fan-shi-xing-ban.html)

1.协议

API与用户的通信协议，总是使用HTTPS协议，确保交互数据的传输安全。

2.安全

为了保证接口接收到的数据不是被篡改以及防止信息泄露造成损失，对敏感数据进行加密及签名。

数据加密

api接口请求参数一律采用RSA进行加解密，在客户端使用公钥对请求参数进行加密，在服务端使用对数私钥据进行解密，防止信息泄露。

签名

为了防止请求数据在网络传输过程中被恶意篡改，对所有非查询接口增加数字签名，签名原串为对请求参数进行自然排序，通过私钥加签后放入sign参数中。

时间戳

api接口中增加时间戳timestamp字段，作用：固定时间范围内，减少同一请求被暴力调用的次数。

3.API版本控制

API的版本号统一放入URL。

https://api.example.com/v{n}/

v{n}n代表版本号，分为整形和浮点型

整形版本号：大功能版本发布形式；具有当前版本状态下的所有API接口，例如：v1,v2

浮点型：为小版本号，只具备补充api的功能，其他api都默认调用大版本号的API，例如v1.1,v1.2

4.API路径规则

在RESTful架构中，每个网址代表一种资源(resource)，所以网址中不能有动词，只能有名词。名词尽量与数据库表格名对应。

例子:

https://api.example.com/v1/products

https://api.example.com/v1/users

https://api.example.com/v1/employees

5.HTTP请求方式

对于资源的具体操作类型，由HTTP动词表示。

常用的HTTP动词由下面四个(括号里是对应的SQL命令)。

GET(SELECT):从服务器取出资源。

POST(CREATE):在服务器新建一个资源。

PUT(UPDATE):在服务器更新资源。

DELETE(DELETE):从服务器删除资源。

例子:

GET/product:列出所有商品

POST/product:新建一个商品

GET/product/ID:获取某个指定商品的信息

PUT/product/ID:更新某个指定商品的信息

DELETE/product/ID:删除某个商品

GET/product/ID/purchase:列出某个指定商品的所有投资者

GET/product/ID/purchase/ID:获取某个指定商品的指定投资者信息

6.请求数据

公共请求参数

| 参数名称 | 参数类型 | 是否必填 | 最大长度 | 描述 | 示例 |
| --- | --- | --- | --- | --- | --- |
| charset | String | 是 | 10 | 请求使用的编码格式如:utf-8,gbk | utf-8 |
| sign\_type | String | 是 | 10 | 生成签名字符串所使用的算法类型 | RSA |
| sign | String | 是 | 344 | 请求参数签名串 | djdu7dusufiusgfu |
| timestamp | String | 是 | 14 | 发送请求的时间,格式:yyyyMMddHHmmss | 20180505121212 |
| requestSource | String | 否 | 8 | 客户端请求来源APP WAP PC | APP |
| token | String | 否 |  | 鉴权标识，用于登录判断 |  |
| biz\_content | String | 是 |  | 请求参数集合，除公共参数外所有请求参数 |  |

请求参数(biz\_content)

| 参数名称 | 参数类型 | 是否必填 | 最大长度 | 描述 | 示例 |
| --- | --- | --- | --- | --- | --- |
| phone | String | 是 | 10 | 登录手机号 | 15088890908 |
| phoneCode | String | 是 | 6 | 验证码 | 234567 |

7.返回数据

为了保障前后端的数据交互的顺畅，统一接口返回模板如下:

{

code:0000,

data:{},

msg:''

}

code:接口的执行状态

0000：表示成功

其他：不太异常

Data接口的主数据

返回JSON对象

Msg信息

当code!=0000都应该有错误信息

8.非RESTful API需求

由于实际业务开展过程中，可能会出现各种的api不是简单的restful规范能实现的，因此需要一些api突破restful规范原则。

8.1页面级API

把当前页面中需要用到的所有数据通过一个接口一次性返回全部数据。

例子:

api/v1/get-home-data返回首页用到的所有数据

此类API存在缺陷:只要业务需求变动，该api就需要跟着变更。

8.2自定义组合API

把当前用户需要在第一时间内容加载的多个接口合并成一个请求发送到服务端，服务端根据请求内容，一次性把所有数据合并返回，相比于页面级API，具备更高的灵活性，同时又能很容易实现页面级API功能。

规范

地址:api/v1/testApi

传入参数:

data:[

{url:'api1',type:'get',data:{}},

{url:'api2',type:'get',data:{}},

{url:'api3',type:'get',data:{}},

]

返回数据

{

code:0000,

msg:'',

data:[

{code:0000,msg:'',data:[]},

{code:0000,msg:'',data:[]},

{code:0000,msg:'',data:[]},

]

}

9.API共建平台

RAP是一个GUI的WEB接口管理工具。在RAP中，可定义接口的URL、请求&响应格式等等。通过分析这些数据，RAP提供MOCK服务、测试服务等自动化工具。

9.1什么是RAP?

在前后端分离的开发模式下，我们通常需要定义一份接口文档来规范接口的具体信息。如请求地址、有几个参数、参数名称及类型含义等等。RAP首先方便团队录入、查看和管理这些接口文档，并通过分析结构化的文档数据，重复利用并生成自测数据、提供自测控制台等等。

9.2RAP的特色

通过RAP来管理API文档。

提供MOCK服务，通过MockJS创建mock测试数据。

[参考文献](https://www.cnblogs.com/bndong/p/6139598.html)

架构管理规范

**软件架构管理过程**

上一篇博客谈到架构的起源，也就是Dewayne E. Perry 和 Alexander L. Wolf在1992年发表的“Foundations for the Study of Software Architecture”，也说到了目前架构已经到了3.0版本，即架构 = 一系列的架构设计决策 + 这些决策背后的原理。这次来聊聊软件架构管理过程。

我认为目前来说，大部分的架构设计或者方法都有其生存的环境，比如项目环境、企业环境、行业环境等，适合你的不一定适合就适合我，很难找到一套通用的解决方案来应对所有变化，所以目前的架构解决方案呈现多样化。但是在各种各样的架构解决方案背后，却有着相同或者相似的架构管理过程。

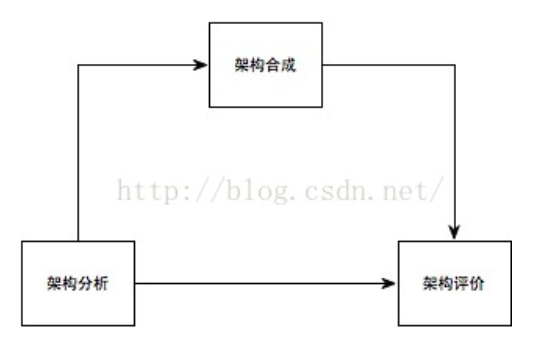
架构管理过程的元素是活动，最初只有三个，由Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, 以及 Pierre America提出：

Architectural Analysis（架构分析活动）：该活动用于定义一个软件架构必须解决的问题。

Architectural Synthesis（架构合成活动）：该活动用于建议能解决目标问题的潜在解决方案。

Architectural Evaluation（架构评价活动）：该活动用于评估潜在的解决方案，并确保架构决策的正确性。

这三个活动构成了最初的软件架构管理过程：

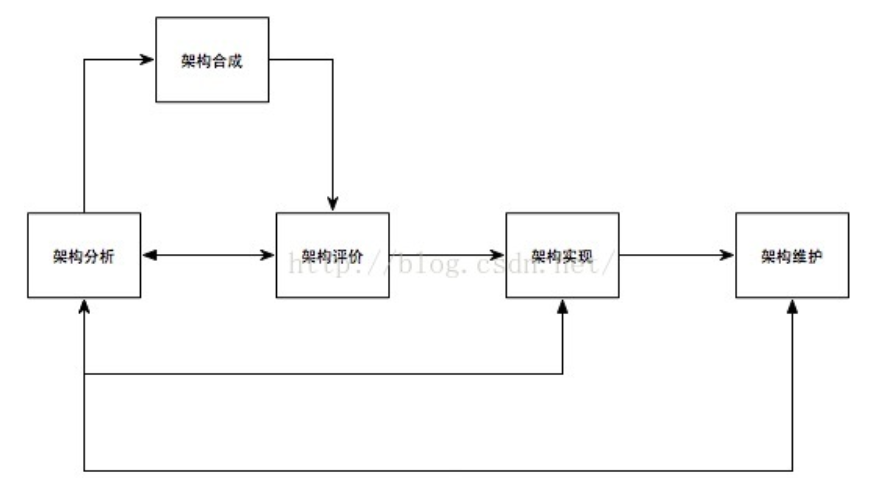


在2010年，Antony Tang, Paris Avgeriou, Anton Jansen, Rafael Capilla, 以及 Muhammad Ali Babar，扩展了软件架构活动到五个。Architectural Analysis， Architectural Synthesis， Architectural Evaluation仍然在列，同时他们增加了：

Architectural Implementation（架构实现）：该活动用于实现设计的架构，是一个从架构设计到详细设计的过程。

Architectural Maintenance（架构维护）：该活动用于维护架构，比如通过修改架构来解决发现的错误等。

于是软件架构过程演化为：



随着架构的发展，我们发现架构活动还可以包括架构演化、架构恢复、架构描述、架构理解、架构影响分析、架构复用以及架构重构，但是以上五个活动我认为是架构的核心活动，而其他活动则是一般性活动，这些一般性活动可以出现在整个架构生命周期的任意阶段，用以支持以上五个核心活动。

最有成就感的项目，学到经验，让自己比较踏实，让自己得到启发的项目

接口文档怎么维护，如何维护三端（小程序，安卓，网页等）连接，以一套接口为原则，接口冲突怎么解决：利用apidoc维护api接口文档

[接口中解决默认方法冲突](https://www.cnblogs.com/huangzefeng/p/10492942.html)

如果先在一个接口中将一个方法定义为默认方法，然后又在超类或者另一个接口中定义了同样的方法，那么同时实现这两个接口的类或者既继承了超类又实现了接口的类就会发生冲突。但Java提供相应的规则：

1.超类优先。如果超类提供一个具体方法，同名而且有相同参数类型的默认方法会被忽略。

2.接口冲突。如果一个超接口提供了一个默认方法，另一个接口提供了一个同名而且参数类型相同的方法，就必须解决冲突。

例子

1.假设Person是一个类（有getName方法），Named是一个接口

interface Named{

default String getName(){

return getClass().getName()

}

}

有一个类Student继承了Person和实现了Named接口：

class Student extends Person implements Named{...}

在这种情况下，只会考虑超类方法，接口的所有默认方法都会被忽略。接口可添加默认方法在Java SE 8中才有，使用“类优先”原则可以确保Java SE 7的兼容性，如果为一个接口添加默认方法，这对于有这个默认方法之前能正常工作的代码不会有任何影响。

2.假设Person是一个接口（默认实现了getName方法），Named依旧是一个接口，类Student实现了这两个接口：

class Student implements Person,Named{...}

类会继承Person和Named接口中提供的两个不一致的getName方法，并不是从中选择一个，这样Java编译器就会报告一个错误，我们需要解决这个二义性。只需要在Student类中提供一个getName方法，在这个方法中，可以选择两个冲突方法中的一个：

class Student implement Person,Named{

public String getName(){

return Person.super.getName();

}

}

**mysql存储过程：**

MySQL PROCEDURE

存储过程（PROCEDURE）是存储在服务器中的一组指定的 SQL 语句，客户机无需不断重新发出单个语句，而可以引用存储过程。存储过程类型分为存储过程 (PROCEDURE) 和存储函数 (FUNCTION) 。

Ø   存储过程：通过 CALL 语句调用过程。这些过程可以使用输出变量或结果集传回值。

Ø   存储函数：在语句中调用函数。这些函数可返回标量值。

l   使用存储过程的好处：

1)          客户机功能

通过存储过程，您可以在数据库中集中创建一个语句或一系列语句，以供使用不同编程语言编写或在不同平台上运行的多个客户机应用程序使用。

2)          安全性

存储过程为需要最高安全级别的应用程序提供了一个解决方法。例如，银行针对所有常用操作均使用存储过程和函数。这提供了一致、安全的环境。可对存储过程进行编码，以确保正确记录了每个操作。在此类设置中，应用程序和用户无法直接访问数据库表，只能执行特定的存储过程。

3)          性能

因为服务器和客户机之间需要发送的信息变少了，所以存储过程可提升性能。客户机按名称调用存储过程，而不是传递该存储过程所包含的所有语句。

4)          函数库

通过存储过程，可以在数据库服务器中使用函数库。这些库用作数据库的 API 。

l   使用存储过程的问题：

1)          增加了服务器负载

在数据库自身中执行存储过程可增加服务器负载并降低应用程序的性能。可以运行测试并运用常识来确保在数据库本身中包含逻辑所带来的方便比可能引发的性能问题更为显著。

2)          开发工具有限

MySQL 中支持存储过程的开发工具不像在更通用的编程语言中那样成熟和明确。此局限性会使存储过程的编写和调试过程更加困难，在决策过程中需要加以考虑。

3)          语言功能和速度有限

虽然在许多情况下在数据库本身中包含逻辑具有很大的优势，但是与其他编程语言相比，在可实现的内容方面仍有局限。存储过程在数据库上下文中执行，与客户机应用程序中的存储过程相比，在处理大量数据时性能较好，但是客户机应用程序语言可能具有更强大、更通用的处理、集成或其他库功能。您必须考虑所需功能的范围，以确保针对每个存储过程使用最佳的可能解决方案。

4)          调试和概要分析功能有限

1.1.         执行存储过程

用于调用存储过程的命令与 MySQL 中的其他命令非常类似。使用 CALL 语句来调用存储过程（ procedure ）。存储过程使用输出变量或结果集传回值。执行 FUNCTION ，像其他任何函数一样，从语句内部调用函数（即，通过调用相应函数的名称），函数返回标量值。

每个存储过程均与特定数据库相关联。这有多重含义：

Ø   USE <database> ：调用存储过程时， MySQL 会在该存储过程运行期间执行隐式 USE <database> 。不能在存储过程内发出 USE 语句。

Ø   限定名称：可使用存储过程的数据库名称限定存储过程名称。执行此操作可引用当前数据库以外的存储过程。例如，要调用与 test 数据库相关联的存储过程 p 或函数 f ，请使用 CALL test.p() 或 test.f() 。

Ø   数据库删除：删除数据库时，也会删除与其关联的所有存储过程。

MySQL 允许在存储过程内使用常规 SELECT 语句。此类查询的结果集将直接发送到客户机。

1)          存储过程示例

mysql> DELIMITER //

mysql> CREATE PROCEDURE record\_count ()

-> BEGIN

-> SELECT 'Country count ', COUNT(\*) FROM Country;

-> SELECT 'City count ', COUNT(\*) FROM City;

-> SELECT 'CountryLanguage count', COUNT(\*) FROM CountryLanguage;

-> END//

mysql> DELIMITER ;

Ø   复合语句

通过在存储过程中使用 BEGIN … END 语法并使用触发器，可以创建复合语句。 BEGIN … END 块可包含零个或多个语句。空复合语句是合法的，而且复合语句中的语句数量没有上限。

Ø   分隔符

在 BEGIN … END 语法中，必须使用分号 (;) 终止每个语句。由于 mysql 客户机使用分号作为 SQL 语句的默认终止字符，在以交互方式或针对批处理使用 mysql 命令行客户机时，必须使用 DELIMITER 语句更改此设置。

示例中，第一个 DELIMITER 语句用于将 SQL 语句终止字符更改为两个正斜杠 (//) 。此更改可确保客户机不会将复合语句中的分号解释为语句分隔符，并确保客户机不会过早地将 CREATE PROCEDURE 语句发送到服务器。当创建存储过程的语句以 // 终止时，客户机会先将该语句发送到服务器，然后再发出第二个 DELIMITER 语句将语句分隔符重置为分号。

2)          存储函数：示例

mysql> DELIMITER //

mysql> CREATE FUNCTION pay\_check (gross\_pay FLOAT(9,2), tax\_rate FLOAT (3,2))

-> RETURNS FLOAT(9,2)

-> NO SQL

-> BEGIN

-> DECLARE net\_pay FLOAT(9,2)

-> DEFAULT 0;

-> SET net\_pay=gross\_pay - gross\_pay \* tax\_rate;

-> RETURN net\_pay;

-> END//

mysql> DELIMITER ;

Ø   RETURNS 子句

RETURNS 子句用于确定此函数要返回的值的类型。

Ø   特征

通过多个特征，可确定有关存储函数所使用的数据的性质。在 MySQL 中，这些特征仅供参考。服务器不会使用这些特征来限制允许存储函数执行的语句种类。

l   CONTAINS SQL 表示存储函数包含用于读取或写入数据的语句。如果未显式提供以上任何特征，则此为默认值。

l   NO SQL 表示存储函数不包含任何 SQL 语句。

l   READS SQL DATA 表示存储函数包含用于读取数据的语句（例如， SELECT ）而不包含用于写入数据的语句。

l   MODIFIES SQL DATA 表示存储过程包含用于写入数据的语句（例如， INSERT 或 DELETE ）。

注：在启用了二进制日志记录后，如果创建函数时未指定以下项之一，则 MySQL 会产生一个错误： NO SQL 、 READS SQL DATA 或 DETERMINISTIC 。

Ø   DECLARE 语句

在存储过程中使用 DECLARE 语句来声明本地变量并初始化用户变量。可将 DEFAULT 子句添加到 DECLARE 语句的结尾，以便为用户变量指定初始值。如果省去 DEFAULT 子句，则用户变量的初始值为 NULL 。

Ø   SET 语句

通过 SET 语句，您可以使用 = 或 := 作为赋值运算符来向定义的变量赋值。

Ø   RETURN 语句

RETURN 语句用于终止存储函数的执行，并将值表达式返回给函数调用方。

1.2.         检查存储过程

Ø   SHOW CREATE PROCEDURE 和 SHOW CREATE FUNCTION

这些语句为 MySQL 扩展，类似于 SHOW CREATE TABLE 。这些语句返回可用于重新创建指定存储过程的具体字符串。这些语句的主要限制之一是您必须知道过程或函数的名称，并且必须确定其为过程或函数，然后才能尝试查看相应信息。

Ø   SHOW PROCEDURE STATUS 和 SHOW FUNCTION STATUS

这些语句特定于 MySQL 。它们可返回存储过程的特征，如数据库、名称、类型、创建者以及创建和修改日期。这些语句有一个优点：可基于 LIKE 模式显示特定存储过程。如果未指定任何模式，则会根据所使用的语句，列出所有存储过程或所有存储函数的信息。例如，以下语句显示名称以“ film ”开头的过程的相关信息：

SHOW PROCEDURE STATUS LIKE 'film%'\G

Ø   INFORMATION\_SCHEMA.ROUTINES

INFORMATION\_SCHEMA.ROUTINES 表包含存储过程（过程和函数）的相关信息，并返回可同时在 SHOW CREATE … 和 SHOW … STATUS 语句中找到的大部分详细信息，以包含用于创建存储过程的实际语法。在这三个选项中，此表可完整地呈现数据库中的可用存储过程。

示例：

mysql> SELECT routine\_name, routine\_schema, routine\_type, definer

> FROM INFORMATION\_SCHEMA.ROUTINES

> WHERE routine\_name LIKE 'film%';

+-------------------+----------------+--------------+----------------+

| routine\_name | routine\_schema | routine\_type | definer |

+-------------------+----------------+--------------+----------------+

| film\_in\_stock | sakila | PROCEDURE | root@localhost |

| film\_not\_in\_stock | sakila | PROCEDURE | root@localhost |

+-------------------+----------------+--------------+----------------+

2 rows in set (0.00 sec)

Ø   mysql 系统数据库中与编程组件关联的表

mysql 系统数据库中包含的一些表可提供与 MySQL 存储过程功能相关的信息。这些表包括：

l   mysql.event 表，包含 MySQL 服务器中所存储事件的相关信息；

l   mysql.proc 表，包含 MySQL 服务器中的存储过程和函数的相关信息；

l   mysql.procs\_priv 表，为引用存储过程的用户提供访问控制授予详细信息；

1.3.         存储过程和执行安全性

存储过程和函数的使用涉及多个权限。

默认操作：创建存储过程时， MySQL 会自动向您的帐户授予对该存储过程的 EXECUTE 和 ALTER ROUTINE 权限。拥有撤消权限以及 GRANT OPTION 权限的用户稍后可撤消或删除这些权限。在创建存储过程后，可以通过发出 SHOW GRANTS 语句来验证这些权限。

授予权限：当在全局级别或数据库级别授予所有权限时， GRANT ALL 语句包括除 GRANT OPTION 之外的所有存储过程权限。要授予 GRANT OPTION 权限，请在该语句结尾包含 WITH GRANT OPTION 子句。您可以在单个存储过程级别授予 EXECUTE 、 ALTER ROUTINE 和 GRANT OPTION 权限，但仅限于已经存在的存储过程。要授予对单个存储过程的权限，可使用其数据库名称限定存储过程，并提供关键字 PROCEDURE 或 FUNCTION 以指示存储过程类型，如以下示例中所示：

mysql> GRANT EXECUTE, ALTER ROUTINE ON PROCEDURE world\_innodb.record\_count TO 'magellan'@'localhost' WITH GRANT OPTION;

mysql> GRANT ALL ON world\_innodb.\* TO 'magellan'@'localhost';

mysql> GRANT EXECUTE, ALTER ROUTINE ON PROCEDURE world\_innodb.record\_count TO 'magellan'@'localhost';

权限对应允许的操作

CREATE ROUTINE ：创建存储过程。

ALTER ROUTINE ：更改或删除存储过程。

EXECUTE ：执行存储过程。

GRANT OPTION ：将权限授予其他帐户。

## MySQL存储过程详解  mysql 存储过程

mysql存储过程详解

1.     存储过程简介

我们常用的操作数据库语言SQL语句在执行的时候需要要先编译，然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

一个存储过程是一个可编程的函数，它在数据库中创建并保存。它可以有SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟。它允许控制数据的访问方式。

存储过程通常有以下优点：

(1).存储过程增强了SQL语言的功能和灵活性。存储过程可以用流控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2).存储过程允许标准组件是编程。存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

(3).存储过程能实现较快的执行速度。如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

(4).存储过程能过减少网络流量。针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织程存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大增加了网络流量并降低了网络负载。

(5).存储过程可被作为一种安全机制来充分利用。系统管理员通过执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

2.     关于MySQL的存储过程

存储过程是数据库存储的一个重要的功能，但是MySQL在5.0以前并不支持存储过程，这使得MySQL在应用上大打折扣。好在MySQL 5.0终于开始已经支持存储过程，这样即可以大大提高数据库的处理速度，同时也可以提高数据库编程的灵活性。

3.     MySQL存储过程的创建

(1). 格式

MySQL存储过程创建的格式：CREATE PROCEDURE 过程名 ([过程参数[,...]])

[特性 ...] 过程体

这里先举个例子：

1. mysql> DELIMITER //
2. mysql> CREATE PROCEDURE proc1(OUT s int)
3. -> BEGIN
4. -> SELECT COUNT(\*) INTO s FROM user;
5. -> END
6. -> //
7. mysql> DELIMITER ;

注：

（1）这里需要注意的是DELIMITER //和DELIMITER ;两句，DELIMITER是分割符的意思，因为MySQL默认以";"为分隔符，如果我们没有声明分割符，那么编译器会把存储过程当成SQL语句进行处理，则存储过程的编译过程会报错，所以要事先用DELIMITER关键字申明当前段分隔符，这样MySQL才会将";"当做存储过程中的代码，不会执行这些代码，用完了之后要把分隔符还原。

（2）存储过程根据需要可能会有输入、输出、输入输出参数，这里有一个输出参数s，类型是int型，如果有多个参数用","分割开。

（3）过程体的开始与结束使用BEGIN与END进行标识。

这样，我们的一个MySQL存储过程就完成了，是不是很容易呢?看不懂也没关系，接下来，我们详细的讲解。

(2). 声明分割符

其实，关于声明分割符，上面的注解已经写得很清楚，不需要多说，只是稍微要注意一点的是：如果是用MySQL的Administrator管理工具时，可以直接创建，不再需要声明。

(3). 参数

MySQL存储过程的参数用在存储过程的定义，共有三种参数类型,IN,OUT,INOUT,形式如：

CREATE PROCEDURE([[IN |OUT |INOUT ] 参数名 数据类形...])

IN 输入参数:表示该参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值

OUT 输出参数:该值可在存储过程内部被改变，并可返回

INOUT 输入输出参数:调用时指定，并且可被改变和返回

Ⅰ. IN参数例子

创建:

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE demo\_in\_parameter(IN p\_in int)
3. -> BEGIN
4. -> SELECT p\_in;
5. -> SET p\_in=2;
6. -> SELECT p\_in;
7. -> END;
8. -> //
9. mysql > DELIMITER ;
10. 执行结果:
11. mysql > SET @p\_in=1;
12. mysql > CALL demo\_in\_parameter(@p\_in);
13. +------+
14. | p\_in |
15. +------+
16. |   1  |
17. +------+
19. +------+
20. | p\_in |
21. +------+
22. |   2  |
23. +------+
25. mysql> SELECT @p\_in;
26. +-------+
27. | @p\_in |
28. +-------+
29. |  1    |
30. +-------+
31. 以上可以看出，p\_in虽然在存储过程中被修改，但并不影响@p\_id的值

Ⅱ.OUT参数例子

创建:

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE demo\_out\_parameter(OUT p\_out int)
3. -> BEGIN
4. -> SELECT p\_out;
5. -> SET p\_out=2;
6. -> SELECT p\_out;
7. -> END;
8. -> //
9. mysql > DELIMITER ;
10. 执行结果:
11. mysql > SET @p\_out=1;
12. mysql > CALL sp\_demo\_out\_parameter(@p\_out);
13. +-------+
14. | p\_out |
15. +-------+
16. | NULL  |
17. +-------+
19. +-------+
20. | p\_out |
21. +-------+
22. |   2   |
23. +-------+
25. mysql> SELECT @p\_out;
26. +-------+
27. | p\_out |
28. +-------+
29. |   2   |
30. +-------+
31. Ⅲ. INOUT参数例子

创建:

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE demo\_inout\_parameter(INOUT p\_inout int)
3. -> BEGIN
4. -> SELECT p\_inout;
5. -> SET p\_inout=2;
6. -> SELECT p\_inout;
7. -> END;
8. -> //
9. mysql > DELIMITER ;

执行结果:

1. mysql > SET @p\_inout=1;
2. mysql > CALL demo\_inout\_parameter(@p\_inout) ;
3. +---------+
4. | p\_inout |
5. +---------+
6. |    1    |
7. +---------+
9. +---------+
10. | p\_inout |
11. +---------+
12. |    2    |
13. +---------+
15. mysql > SELECT @p\_inout;
16. +----------+
17. | @p\_inout |
18. +----------+
19. |    2     |
20. +----------+

(4). 变量

Ⅰ. 变量定义

DECLARE variable\_name [,variable\_name...] datatype [DEFAULT value];

其中，datatype为MySQL的数据类型，如:int, float, date, varchar(length)

例如:

1. DECLARE l\_int int unsigned default 4000000;
2. DECLARE l\_numeric number(8,2) DEFAULT 9.95;
3. DECLARE l\_date date DEFAULT '1999-12-31';
4. DECLARE l\_datetime datetime DEFAULT '1999-12-31 23:59:59';
5. DECLARE l\_varchar varchar(255) DEFAULT 'This will not be padded';

Ⅱ. 变量赋值

 SET 变量名 = 表达式值 [,variable\_name = expression ...]

Ⅲ. 用户变量

ⅰ. 在MySQL客户端使用用户变量

1. mysql > SELECT 'Hello World' into @x;
2. mysql > SELECT @x;
3. +-------------+
4. |   @x        |
5. +-------------+
6. | [Hello World](http://www.chengxusheji.com/) |
7. +-------------+
8. mysql > SET @y='Goodbye Cruel World';
9. mysql > SELECT @y;
10. +---------------------+
11. |     @y              |
12. +---------------------+
13. | Goodbye Cruel World |
14. +---------------------+
16. mysql > SET @z=1+2+3;
17. mysql > SELECT @z;
18. +------+
19. | @z   |
20. +------+
21. |  6   |
22. +------+

ⅱ. 在存储过程中使用用户变量

1. mysql > CREATE PROCEDURE GreetWorld( ) SELECT CONCAT(@greeting,' World');
2. mysql > SET @greeting='Hello';
3. mysql > CALL GreetWorld( );
4. +----------------------------+
5. | CONCAT(@greeting,' World') |
6. +----------------------------+
7. |  Hello World               |
8. +----------------------------+

ⅲ. 在存储过程间传递全局范围的用户变量

1. mysql> CREATE PROCEDURE p1()   SET @last\_procedure='p1';
2. mysql> CREATE PROCEDURE p2() SELECT CONCAT('Last procedure was ',@last\_proc);
3. mysql> CALL p1( );
4. mysql> CALL p2( );
5. +-----------------------------------------------+
6. | CONCAT('Last procedure was ',@last\_proc |
7. +-----------------------------------------------+
8. | Last procedure was p1                         |
9. +-----------------------------------------------+

注意:

①用户变量名一般以@开头

②滥用用户变量会导致程序难以理解及管理

(5). 注释

MySQL存储过程可使用两种风格的注释

双模杠：--

该风格一般用于单行注释

c风格： 一般用于多行注释

例如：

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc1 --name存储过程名
3. -> (IN parameter1 INTEGER)
4. -> BEGIN
5. -> DECLARE variable1 CHAR(10);
6. -> IF parameter1 = 17 THEN
7. -> SET variable1 = 'birds';
8. -> ELSE
9. -> SET variable1 = 'beasts';
10. -> END IF;
11. -> INSERT INTO table1 VALUES (variable1);
12. -> END
13. -> //
14. mysql > DELIMITER ;

4.     MySQL存储过程的调用

用call和你过程名以及一个括号，括号里面根据需要，加入参数，参数包括输入参数、输出参数、输入输出参数。具体的调用方法可以参看上面的例子。

5.     MySQL存储过程的查询

我们像知道一个数据库下面有那些表，我们一般采用show tables;进行查看。那么我们要查看某个数据库下面的存储过程，是否也可以采用呢？答案是，我们可以查看某个数据库下面的存储过程，但是是令一钟方式。

我们可以用

select name from mysql.proc where db=’数据库名’;

或者

select routine\_name from information\_schema.routines where routine\_schema='数据库名';

或者

show procedure status where db='[数据库](http://www.redisfans.com/)名';

进行查询。

如果我们想知道，某个存储过程的详细，那我们又该怎么做呢？是不是也可以像操作表一样用describe 表名进行查看呢？

答案是：我们可以查看存储过程的详细，但是需要用另一种方法：

SHOW CREATE PROCEDURE 数据库.存储过程名;

就可以查看当前存储过程的详细。

6.     MySQL存储过程的修改

ALTER PROCEDURE

更改用CREATE PROCEDURE 建立的预先指定的存储过程，其不会影响相关存储过程或存储功能。

7.     MySQL存储过程的删除

删除一个存储过程比较简单，和删除表一样：

DROP PROCEDURE

从MySQL的表格中删除一个或多个存储过程。

8.     MySQL存储过程的控制语句

(1). 变量作用域

内部的变量在其作用域范围内享有更高的优先权，当执行到end。变量时，内部变量消失，此时已经在其作用域外，变量不再可见了，应为在存储

过程外再也不能找到这个申明的变量，但是你可以通过out参数或者将其值指派

给会话变量来保存其值。

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc3()
3. -> begin
4. -> declare x1 varchar(5) default 'outer';
5. -> begin
6. -> declare x1 varchar(5) default 'inner';
7. -> select x1;
8. -> end;
9. -> select x1;
10. -> end;
11. -> //
12. mysql > DELIMITER ;

 (2). 条件语句

Ⅰ. if-then -else语句

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc2(IN parameter int)
3. -> begin
4. -> declare var int;
5. -> set var=parameter+1;
6. -> if var=0 then
7. -> insert into t values(17);
8. -> end if;
9. -> if parameter=0 then
10. -> update t set s1=s1+1;
11. -> else
12. -> update t set s1=s1+2;
13. -> end if;
14. -> end;
15. -> //
16. mysql > DELIMITER ;
17. Ⅱ. case语句：
18. mysql > DELIMITER //
19. mysql > CREATE PROCEDURE proc3 (in parameter int)
20. -> begin
21. -> declare var int;
22. -> set var=parameter+1;
23. -> case var
24. -> when 0 then
25. -> insert into t values(17);
26. -> when 1 then
27. -> insert into t values(18);
28. -> else
29. -> insert into t values(19);
30. -> end case;
31. -> end;
32. -> //
33. mysql > DELIMITER ;

(3). 循环语句

Ⅰ. while ···· end while：

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc4()
3. -> begin
4. -> declare var int;
5. -> set var=0;
6. -> while var<6 do
7. -> insert into t values(var);
8. -> set var=var+1;
9. -> end while;
10. -> end;
11. -> //
12. mysql > DELIMITER ;

Ⅱ. repeat···· end repeat：

它在执行操作后检查结果，而while则是执行前进行检查。

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc5 ()
3. -> begin
4. -> declare v int;
5. -> set v=0;
6. -> repeat
7. -> insert into t values(v);
8. -> set v=v+1;
9. -> until v>=5
10. -> end repeat;
11. -> end;
12. -> //
13. mysql > DELIMITER ;
14. Ⅲ. loop ·····end loop:

loop循环不需要初始条件，这点和while 循环相似，同时和repeat循环一样不需要结束条件, leave语句的意义是离开循环。

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc6 ()
3. -> begin
4. -> declare v int;
5. -> set v=0;
6. -> LOOP\_LABLE:loop
7. -> insert into t values(v);
8. -> set v=v+1;
9. -> if v >=5 then
10. -> leave LOOP\_LABLE;
11. -> end if;
12. -> end loop;
13. -> end;
14. -> //
15. mysql > DELIMITER ;

Ⅳ. LABLES 标号：

标号可以用在begin repeat while 或者loop 语句前，语句标号只能在合法的语句前面使用。可以跳出循环，使运行指令达到复合语句的最后一步。

(4). ITERATE迭代

Ⅰ. ITERATE:

通过引用复合语句的标号,来从新开始复合语句

1. mysql > DELIMITER //
2. mysql > CREATE PROCEDURE proc10 ()
3. -> begin
4. -> declare v int;
5. -> set v=0;
6. -> LOOP\_LABLE:loop
7. -> if v=3 then
8. -> set v=v+1;
9. -> ITERATE LOOP\_LABLE;
10. -> end if;
11. -> insert into t values(v);
12. -> set v=v+1;
13. -> if v>=5 then
14. -> leave LOOP\_LABLE;
15. -> end if;
16. -> end loop;
17. -> end;
18. -> //
19. mysql > DELIMITER ;

9.     MySQL存储过程的基本函数

(1).字符串类

CHARSET(str) //返回字串字符集

CONCAT (string2 [,... ]) //连接字串

INSTR (string ,substring ) //返回substring首次在string中出现的位置,不存在返回0

LCASE (string2 ) //转换成小写

LEFT (string2 ,length ) //从string2中的左边起取length个字符

LENGTH (string ) //string长度

LOAD\_FILE (file\_name ) //从文件读取内容

LOCATE (substring , string [,start\_position ] ) 同INSTR,但可指定开始位置

LPAD (string2 ,length ,pad ) //重复用pad加在string开头,直到字串长度为length

LTRIM (string2 ) //去除前端空格

REPEAT (string2 ,count ) //重复count次

REPLACE (str ,search\_str ,replace\_str ) //在str中用replace\_str替换search\_str

RPAD (string2 ,length ,pad) //在str后用pad补充,直到长度为length

RTRIM (string2 ) //去除后端空格

STRCMP (string1 ,string2 ) //逐字符比较两字串大小,

SUBSTRING (str , position [,length ]) //从str的position开始,取length个字符,

注：mysql中处理字符串时，默认第一个字符下标为1，即参数position必须大于等于1

1. mysql> select substring('abcd',0,2);
2. +-----------------------+
3. | substring('abcd',0,2) |
4. +-----------------------+
5. |                       |
6. +-----------------------+
7. 1 row in set (0.00 sec)
9. mysql> select substring('abcd',1,2);
10. +-----------------------+
11. | substring('abcd',1,2) |
12. +-----------------------+
13. |     ab                |
14. +-----------------------+
15. 1 row in set (0.02 sec)

TRIM([[BOTH|LEADING|TRAILING] [padding] FROM]string2) //去除指定位置的指定字符

UCASE (string2 ) //转换成大写

RIGHT(string2,length) //取string2最后length个字符

SPACE(count) //生成count个空格

(2).数学类

ABS (number2 ) //绝对值

BIN (decimal\_number ) //十进制转二进制

CEILING (number2 ) //向上取整

CONV(number2,from\_base,to\_base) //进制转换

FLOOR (number2 ) //向下取整

FORMAT (number,decimal\_places ) //保留小数位数

HEX (DecimalNumber ) //转十六进制

注：HEX()中可传入字符串，则返回其ASC-11码，如HEX('DEF')返回4142143

也可以传入十进制整数，返回其十六进制编码，如HEX(25)返回19

LEAST (number , number2 [,..]) //求最小值

MOD (numerator ,denominator ) //求余

POWER (number ,power ) //求指数

RAND([seed]) //随机数

ROUND (number [,decimals ]) //四舍五入,decimals为小数位数]

注：返回类型并非均为整数，如：

(1)默认变为整形值

1. mysql> select round(1.23);
2. +-------------+
3. | round(1.23) |
4. +-------------+
5. |           1 |
6. +-------------+
7. 1 row in set (0.00 sec)
9. mysql> select round(1.56);
10. +-------------+
11. | round(1.56) |
12. +-------------+
13. |           2 |
14. +-------------+
15. 1 row in set (0.00 sec)

(2)可以设定小数位数，返回浮点型数据

1. mysql> select round(1.567,2);
2. +----------------+
3. | round(1.567,2) |
4. +----------------+
5. |           1.57 |
6. +----------------+
7. 1 row in set (0.00 sec)

SIGN (number2 ) //

(3).日期时间类

ADDTIME (date2 ,time\_interval ) //将time\_interval加到date2

CONVERT\_TZ (datetime2 ,fromTZ ,toTZ ) //转换时区

CURRENT\_DATE ( ) //当前日期

CURRENT\_TIME ( ) //当前时间

CURRENT\_TIMESTAMP ( ) //当前时间戳

DATE (datetime ) //返回datetime的日期部分

DATE\_ADD (date2 , INTERVAL d\_value d\_type ) //在date2中加上日期或时间

DATE\_FORMAT (datetime ,FormatCodes ) //使用formatcodes格式显示datetime

DATE\_SUB (date2 , INTERVAL d\_value d\_type ) //在date2上减去一个时间

DATEDIFF (date1 ,date2 ) //两个日期差

DAY (date ) //返回日期的天

DAYNAME (date ) //英文星期

DAYOFWEEK (date ) //星期(1-7) ,1为星期天

DAYOFYEAR (date ) //一年中的第几天

EXTRACT (interval\_name FROM date ) //从date中提取日期的指定部分

MAKEDATE (year ,day ) //给出年及年中的第几天,生成日期串

MAKETIME (hour ,minute ,second ) //生成时间串

MONTHNAME (date ) //英文月份名

NOW ( ) //当前时间

SEC\_TO\_TIME (seconds ) //秒数转成时间

STR\_TO\_DATE (string ,format ) //字串转成时间,以format格式显示

TIMEDIFF (datetime1 ,datetime2 ) //两个时间差

TIME\_TO\_SEC (time ) //时间转秒数]

WEEK (date\_time [,start\_of\_week ]) //第几周

YEAR (datetime ) //年份

DAYOFMONTH(datetime) //月的第几天

HOUR(datetime) //小时

LAST\_DAY(date) //date的月的最后日期

MICROSECOND(datetime) //微秒

MONTH(datetime) //月

MINUTE(datetime) //分返回符号,正负或0

SQRT(number2) //开平方

本文主要记录了本人学习mysql存储过程时的笔记，文档中用到的表结构在文档结尾处有说明。

一、定义

       存储过程（Stored Procedure）是在大型数据库系统中，一组为了完成特定功能的SQL 语句集，存储在数据库中，经过第一次编译后调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。

二、存储过程的特点

    1、能完成较复杂的判断和运算

    2、可编程行强，灵活

    3、SQL编程的代码可重复使用

    4、执行的速度相对快一些

    5、减少网络之间的数据传输，节省开销

**MySQL 存储过程**

MySQL 5.0 版本开始支持存储过程。

存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。

存储过程是为了完成特定功能的SQL语句集，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数(需要时)来调用执行。

存储过程思想上很简单，就是数据库 SQL 语言层面的代码封装与重用。

**优点**

* 存储过程可封装，并隐藏复杂的商业逻辑。
* 存储过程可以回传值，并可以接受参数。
* 存储过程无法使用 SELECT 指令来运行，因为它是子程序，与查看表，数据表或用户定义函数不同。
* 存储过程可以用在数据检验，强制实行商业逻辑等。

**缺点**

* 存储过程，往往定制化于特定的数据库上，因为支持的编程语言不同。当切换到其他厂商的数据库系统时，需要重写原有的存储过程。
* 存储过程的性能调校与撰写，受限于各种数据库系统。

**一、存储过程的创建和调用**

* 存储过程就是具有名字的一段代码，用来完成一个特定的功能。
* 创建的存储过程保存在数据库的数据字典中。

**创建存储过程**

CREATE [DEFINER = { user | CURRENT\_USER }] 　PROCEDURE sp\_name ([proc\_parameter[,...]]) [characteristic ...] routine\_body proc\_parameter: [ IN | OUT | INOUT ] param\_name type characteristic: COMMENT 'string' | LANGUAGE SQL | [NOT] DETERMINISTIC | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } | SQL SECURITY { DEFINER | INVOKER } routine\_body: 　　Valid SQL routine statement [begin\_label:] BEGIN 　　[statement\_list] 　　　　…… END [end\_label]

**MYSQL 存储过程中的关键语法**

声明语句结束符，可以自定义:

DELIMITER $$

或

DELIMITER //

声明存储过程:

CREATE PROCEDURE demo\_in\_parameter(IN p\_in int)

存储过程开始和结束符号:

BEGIN .... END

变量赋值:

SET @p\_in=1

变量定义:

DECLARE l\_int int unsigned default 4000000;

创建mysql存储过程、存储函数:

create procedure 存储过程名(参数)

存储过程体:

create function 存储函数名(参数)

**实例**

创建数据库，备份数据表用于示例操作：

mysql> create database db1; mysql> use db1; mysql> create table PLAYERS as select \* from TENNIS.PLAYERS; mysql> create table MATCHES as select \* from TENNIS.MATCHES;

下面是存储过程的例子，删除给定球员参加的所有比赛：

mysql> delimiter $$　　#将语句的结束符号从分号;临时改为两个$$(可以是自定义) mysql> CREATE PROCEDURE delete\_matches(IN p\_playerno INTEGER) -> BEGIN -> 　　DELETE FROM MATCHES -> WHERE playerno = p\_playerno; -> END$$ Query OK, 0 rows affected (0.01 sec) mysql> delimiter;　　#将语句的结束符号恢复为分号

**解析：**默认情况下，存储过程和默认数据库相关联，如果想指定存储过程创建在某个特定的数据库下，那么在过程名前面加数据库名做前缀。 在定义过程时，使用 **DELIMITER $$** 命令将语句的结束符号从分号 **;** 临时改为两个 **$$**，使得过程体中使用的分号被直接传递到服务器，而不会被客户端（如mysql）解释。

调用存储过程：

call sp\_name[(传参)];

mysql> select \* from MATCHES; +---------+--------+----------+-----+------+ | MATCHNO | TEAMNO | PLAYERNO | WON | LOST | +---------+--------+----------+-----+------+ | 1 | 1 | 6 | 3 | 1 | | 7 | 1 | 57 | 3 | 0 | | 8 | 1 | 8 | 0 | 3 | | 9 | 2 | 27 | 3 | 2 | | 11 | 2 | 112 | 2 | 3 | +---------+--------+----------+-----+------+ 5 rows in set (0.00 sec) mysql> call delete\_matches(57); Query OK, 1 row affected (0.03 sec) mysql> select \* from MATCHES; +---------+--------+----------+-----+------+ | MATCHNO | TEAMNO | PLAYERNO | WON | LOST | +---------+--------+----------+-----+------+ | 1 | 1 | 6 | 3 | 1 | | 8 | 1 | 8 | 0 | 3 | | 9 | 2 | 27 | 3 | 2 | | 11 | 2 | 112 | 2 | 3 | +---------+--------+----------+-----+------+ 4 rows in set (0.00 sec)

**解析：**在存储过程中设置了需要传参的变量p\_playerno，调用存储过程的时候，通过传参将57赋值给p\_playerno，然后进行存储过程里的SQL操作。

**存储过程体**

* 存储过程体包含了在过程调用时必须执行的语句，例如：dml、ddl语句，if-then-else和while-do语句、声明变量的declare语句等
* 过程体格式：以begin开始，以end结束(可嵌套)

BEGIN

　　BEGIN

　　　　BEGIN

　　　　　　statements;

　　　　END

　　END

END

**注意：**每个嵌套块及其中的每条语句，必须以分号结束，表示过程体结束的begin-end块(又叫做复合语句compound statement)，则不需要分号。

为语句块贴标签:

[begin\_label:] BEGIN

　　[statement\_list]

END [end\_label]

例如：

label1: BEGIN 　　label2: BEGIN 　　　　label3: BEGIN 　　　　　　statements; 　　　　END label3 ; 　　END label2; END label1

标签有两个作用：

* 1、增强代码的可读性
* 2、在某些语句(例如:leave和iterate语句)，需要用到标签

**二、存储过程的参数**

MySQL存储过程的参数用在存储过程的定义，共有三种参数类型,IN,OUT,INOUT,形式如：

CREATEPROCEDURE 存储过程名([[IN |OUT |INOUT ] 参数名 数据类形...])

* IN 输入参数：表示调用者向过程传入值（传入值可以是字面量或变量）
* OUT 输出参数：表示过程向调用者传出值(可以返回多个值)（传出值只能是变量）
* INOUT 输入输出参数：既表示调用者向过程传入值，又表示过程向调用者传出值（值只能是变量）

**1、in 输入参数**

mysql> delimiter $$ mysql> create procedure in\_param(in p\_in int) -> begin -> 　　select p\_in; -> 　　set p\_in=2; -> select P\_in; -> end$$ mysql> delimiter ; mysql> set @p\_in=1; mysql> call in\_param(@p\_in); +------+ | p\_in | +------+ | 1 | +------+ +------+ | P\_in | +------+ | 2 | +------+ mysql> select @p\_in; +-------+ | @p\_in | +-------+ | 1 | +-------+

以上可以看出，p\_in 在存储过程中被修改，但并不影响 @p\_id 的值，因为前者为局部变量、后者为全局变量。

**2、out输出参数**

mysql> delimiter // mysql> create procedure out\_param(out p\_out int) -> begin -> select p\_out; -> set p\_out=2; -> select p\_out; -> end -> // mysql> delimiter ; mysql> set @p\_out=1; mysql> call out\_param(@p\_out); +-------+ | p\_out | +-------+ | NULL | +-------+ 　　#因为out是向调用者输出参数，不接收输入的参数，所以存储过程里的p\_out为null +-------+ | p\_out | +-------+ | 2 | +-------+ mysql> select @p\_out; +--------+ | @p\_out | +--------+ | 2 | +--------+ 　　#调用了out\_param存储过程，输出参数，改变了p\_out变量的值

**3、inout输入参数**

mysql> delimiter $$ mysql> create procedure inout\_param(inout p\_inout int) -> begin -> select p\_inout; -> set p\_inout=2; -> select p\_inout; -> end -> $$ mysql> delimiter ; mysql> set @p\_inout=1; mysql> call inout\_param(@p\_inout); +---------+ | p\_inout | +---------+ | 1 | +---------+ +---------+ | p\_inout | +---------+ | 2 | +---------+ mysql> select @p\_inout; +----------+ | @p\_inout | +----------+ | 2 | +----------+ #调用了inout\_param存储过程，接受了输入的参数，也输出参数，改变了变量

**注意：**

1、如果过程没有参数，也必须在过程名后面写上小括号例：

CREATE PROCEDURE sp\_name ([proc\_parameter[,...]]) ……

2、确保参数的名字不等于列的名字，否则在过程体中，参数名被当做列名来处理

**建议：**

* 输入值使用in参数。
* 返回值使用out参数。
* inout参数就尽量的少用。

**三、变量**

**1. 变量定义**

局部变量声明一定要放在存储过程体的开始：

DECLAREvariable\_name [,variable\_name...] datatype [DEFAULT value];

其中，datatype 为 MySQL 的数据类型，如: int, float, date,varchar(length)

例如:

DECLARE l\_int int unsigned default 4000000; DECLARE l\_numeric number(8,2) DEFAULT 9.95; DECLARE l\_date date DEFAULT '1999-12-31'; DECLARE l\_datetime datetime DEFAULT '1999-12-31 23:59:59'; DECLARE l\_varchar varchar(255) DEFAULT 'This will not be padded';

**2. 变量赋值**

SET 变量名 = 表达式值 [,variable\_name = expression ...]

**3. 用户变量**

在MySQL客户端使用用户变量:

mysql > SELECT 'Hello World' into @x; mysql > SELECT @x; +-------------+ | @x | +-------------+ | Hello World | +-------------+ mysql > SET @y='Goodbye Cruel World'; mysql > SELECT @y; +---------------------+ | @y | +---------------------+ | Goodbye Cruel World | +---------------------+ mysql > SET @z=1+2+3; mysql > SELECT @z; +------+ | @z | +------+ | 6 | +------+

**在存储过程中使用用户变量**

mysql > CREATE PROCEDURE GreetWorld( ) SELECT CONCAT(@greeting,' World'); mysql > SET @greeting='Hello'; mysql > CALL GreetWorld( ); +----------------------------+ | CONCAT(@greeting,' World') | +----------------------------+ | Hello World | +----------------------------+

**在存储过程间传递全局范围的用户变量**

mysql> CREATE PROCEDURE p1() SET @last\_procedure='p1'; mysql> CREATE PROCEDURE p2() SELECT CONCAT('Last procedure was ',@last\_procedure); mysql> CALL p1( ); mysql> CALL p2( ); +-----------------------------------------------+ | CONCAT('Last procedure was ',@last\_proc | +-----------------------------------------------+ | Last procedure was p1 | +-----------------------------------------------+

**注意:**

* 1、用户变量名一般以@开头
* 2、滥用用户变量会导致程序难以理解及管理

**四、注释**

MySQL 存储过程可使用两种风格的注释

两个横杆**--**：该风格一般用于单行注释。

**c 风格**： 一般用于多行注释。

例如：

mysql > DELIMITER // mysql > CREATE PROCEDURE proc1 --name存储过程名 -> (IN parameter1 INTEGER) -> BEGIN -> DECLARE variable1 CHAR(10); -> IF parameter1 = 17 THEN -> SET variable1 = 'birds'; -> ELSE -> SET variable1 = 'beasts'; -> END IF; -> INSERT INTO table1 VALUES (variable1); -> END -> // mysql > DELIMITER ;

**MySQL存储过程的调用**

用call和你过程名以及一个括号，括号里面根据需要，加入参数，参数包括输入参数、输出参数、输入输出参数。具体的调用方法可以参看上面的例子。

**MySQL存储过程的查询**

我们像知道一个数据库下面有那些表，我们一般采用 **showtables;** 进行查看。那么我们要查看某个数据库下面的存储过程，是否也可以采用呢？答案是，我们可以查看某个数据库下面的存储过程，但是是另一钟方式。

我们可以用以下语句进行查询：

selectname from mysql.proc where db='数据库名';

或者

selectroutine\_name from information\_schema.routines where routine\_schema='数据库名';

或者

showprocedure status where db='数据库名';

**如果我们想知道，某个存储过程的详细，那我们又该怎么做呢？是不是也可以像操作表一样用describe 表名进行查看呢？**

**答案是：**我们可以查看存储过程的详细，但是需要用另一种方法：

SHOWCREATE PROCEDURE 数据库.存储过程名;

就可以查看当前存储过程的详细。

**MySQL存储过程的修改**

ALTER PROCEDURE

更改用 CREATE PROCEDURE 建立的预先指定的存储过程，其不会影响相关存储过程或存储功能。

**MySQL存储过程的删除**

删除一个存储过程比较简单，和删除表一样：

DROPPROCEDURE

从 MySQL 的表格中删除一个或多个存储过程。

**MySQL存储过程的控制语句**

**(1). 变量作用域**

内部的变量在其作用域范围内享有更高的优先权，当执行到 end。变量时，内部变量消失，此时已经在其作用域外，变量不再可见了，应为在存储过程外再也不能找到这个申明的变量，但是你可以通过 out 参数或者将其值指派给会话变量来保存其值。

mysql > DELIMITER // mysql > CREATE PROCEDURE proc3() -> begin -> declare x1 varchar(5) default 'outer'; -> begin -> declare x1 varchar(5) default 'inner'; -> select x1; -> end; -> select x1; -> end; -> // mysql > DELIMITER ;

**(2). 条件语句**

1. if-then-else 语句

mysql > DELIMITER // mysql > CREATE PROCEDURE proc2(IN parameter int) -> begin -> declare var int; -> set var=parameter+1; -> if var=0 then -> insert into t values(17); -> end if; -> if parameter=0 then -> update t set s1=s1+1; -> else -> update t set s1=s1+2; -> end if; -> end; -> // mysql > DELIMITER ;

2. case语句：

mysql > DELIMITER // mysql > CREATE PROCEDURE proc3 (in parameter int) -> begin -> declare var int; -> set var=parameter+1; -> case var -> when 0 then -> insert into t values(17); -> when 1 then -> insert into t values(18); -> else -> insert into t values(19); -> end case; -> end; -> // mysql > DELIMITER ; case when var=0 then insert into t values(30); when var>0 then when var<0 then else end case

**(3). 循环语句**

1. while ···· end while

mysql > DELIMITER // mysql > CREATE PROCEDURE proc4() -> begin -> declare var int; -> set var=0; -> while var<6 do -> insert into t values(var); -> set var=var+1; -> end while; -> end; -> // mysql > DELIMITER ;

while 条件 do

--循环体

endwhile

2. repeat···· end repea

它在执行操作后检查结果，而 while 则是执行前进行检查。

mysql > DELIMITER // mysql > CREATE PROCEDURE proc5 () -> begin -> declare v int; -> set v=0; -> repeat -> insert into t values(v); -> set v=v+1; -> until v>=5 -> end repeat; -> end; -> // mysql > DELIMITER ;

repeat

--循环体

until 循环条件

end repeat;

3. loop ·····endloop

loop 循环不需要初始条件，这点和 while 循环相似，同时和 repeat 循环一样不需要结束条件, leave 语句的意义是离开循环。

mysql > DELIMITER // mysql > CREATE PROCEDURE proc6 () -> begin -> declare v int; -> set v=0; -> LOOP\_LABLE:loop -> insert into t values(v); -> set v=v+1; -> if v >=5 then -> leave LOOP\_LABLE; -> end if; -> end loop; -> end; -> // mysql > DELIMITER ;

4. LABLES 标号：

标号可以用在 begin repeat while 或者 loop 语句前，语句标号只能在合法的语句前面使用。可以跳出循环，使运行指令达到复合语句的最后一步。

**(4). ITERATE迭代**

ITERATE 通过引用复合语句的标号,来从新开始复合语句:

mysql > DELIMITER // mysql > CREATE PROCEDURE proc10 () -> begin -> declare v int; -> set v=0; -> LOOP\_LABLE:loop -> if v=3 then -> set v=v+1; -> ITERATE LOOP\_LABLE; -> end if; -> insert into t values(v); -> set v=v+1; -> if v>=5 then -> leave LOOP\_LABLE; -> end if; -> end loop; -> end; -> // mysql > DELIMITER ;

*参考文章：*

*https://www.cnblogs.com/geaozhang/p/6797357.html*

*http://blog.sina.com.cn/s/blog\_86fe5b440100wdyt.html*

mysql 存储过程 教程

mysql是开源的关系型数据库，由于其体积小、开源、有条件免费，所以，使用mysql的用户还是很多的。

本文就从创建表、插入测试数据、创建存储过程、执行存储过程、修改存储过程、删除存储过程等方面，完整的演示mysql存储过程的使用

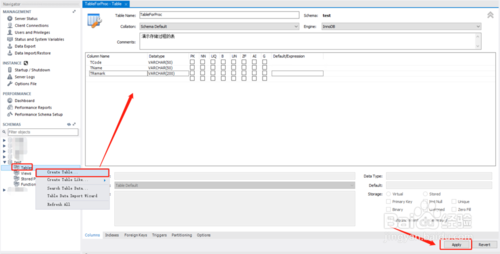
方法/步骤

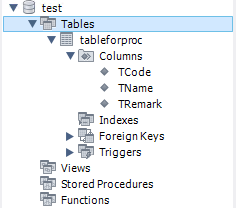
1. 创建表

     1）打开mysql的图形化管理工具【MySQL Workbench 6.3 CE】，登录mysql数据库

     2）在【test】测试数据库（自己创建的空数据库，实际场景使用具体数据库即可）的【Tables】菜单上，点击右键选择【Create Table】

     3）在新建表界面，输入表名称、备注、添加列，点击【Apply】按钮完成创建表

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=1)

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=2)

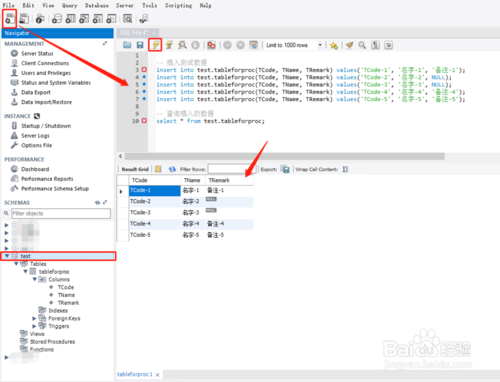
1. 插入测试数据

     1）选中test数据库，此步可选（因为mysql的sql语句都是要带数据库名称前缀的）

     2）点击菜单顶部的第一个图标菜单【Create a new tab ...】

     3）在sql语句窗口，输入插入语句与查询语句

     4）点击sql窗口上面的 闪电 图标，执行sql语句

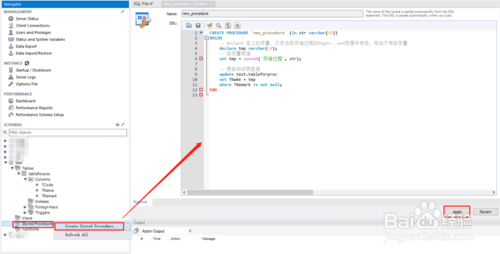
[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=3)

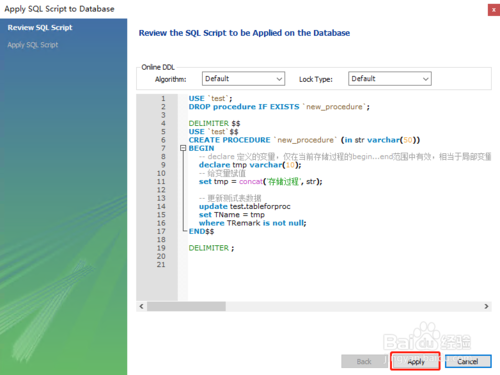
1. 创建存储过程

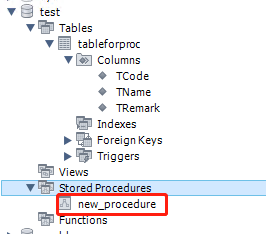
    1）在test数据库的【Stored Procedures】菜单中点击右键【Create Stored Procedures】

    2）在创建存储过程界面，输入如图所示的代码（虽然有一个输入参数str，但是没有使用，在存储过程中可以直接使用这个变量）

    3）点击【Apply】创建存储过程，在弹窗中都点击【Apply】即可

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=4)

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=5)

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=6)

1. 执行存储过程

     1）在SQL窗口中调用存储过格式【call new\_procedure('haha');】

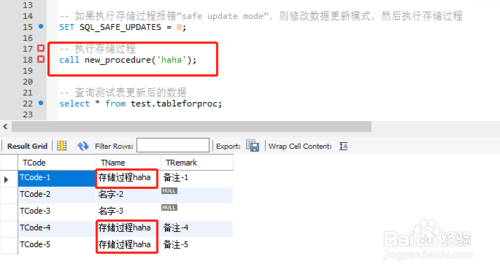
          A）call是调用存储过程的必须命令

          B）new\_procedure是要调用的存储过程名字

          C）('haha')是传入存储过程的参数

     2）如果执行存储过程报错“safe update mode”，则修改数据更新模式，然后执行存储过程

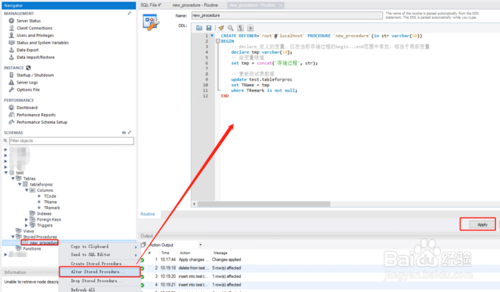
          SET SQL\_SAFE\_UPDATES = 0;

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=7)

1. **修改存储过程**

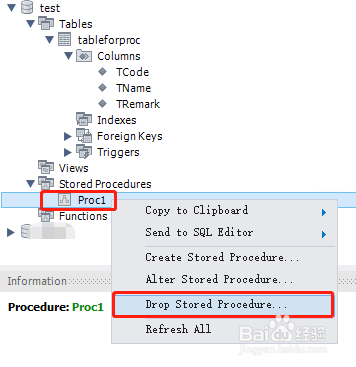
     1）在要修改的存储过程名字上点击右键，选择【Alter Stored Procedure...】

     2）在修改存储过程界面，直接修改后，Apply即可

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=8)

1. **删除存储过程**

     在要删除的存储过程名字上点击右键，选择【Drop Stored Procedure...】即可

[](http://jingyan.baidu.com/album/851fbc3778faf53e1e15ab43.html?picindex=9)

END

经验内容仅供参考，如果您需解决具体问题(尤其法律、医学等领域)，建议您详细咨询相关领域专业人士。

[举报](javascript:void(0))作者声明：本篇经验系本人依照真实经历原创，未经许可，谢绝转载。

小程序的类型，什么类型的项目

mysql的主从复制，主从服务

**mysql复制--主从复制配置**

当单台 MYSQL 服务器无法满足当前网站流量时的优化方案。需要搭建 mysql 集群技术。

复制功能：

数据分布

负载均衡（读）

备份

高可用和故障切换

MySQL升级测试

复制方式：

主–从复制

主–主复制

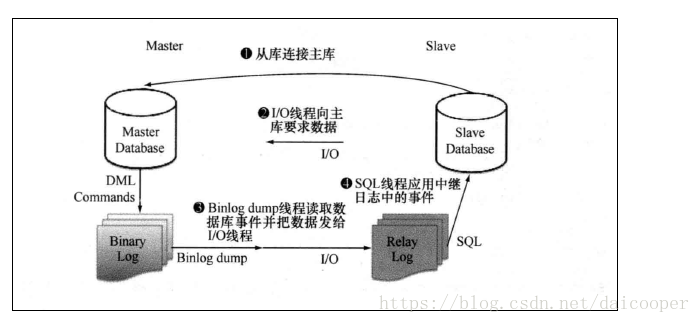
半同步复制

复制原理：

Mysql 中有一种日志叫做 bin 日志（二进制日志）。这个日志会记录下所有修改了数据库的SQL 语句（insert,update,delete,create/alter/drop table, grant 等等）。

主从复制的原理其实就是把主服务器上的 bin 日志复制到从服务器上执行一遍，这样从服务器上的数据就和主服务器上的数据相同了。

复制流程图：



复制过程：

主节点必须启用二进制日志，记录任何修改了数据库数据的事件。

从节点开启一个线程（I/O Thread)把自己扮演成 mysql 的客户端，通过 mysql 协议，请求主节点的二进制日志文件中的事件

主节点启动一个线程（dump Thread），检查自己二进制日志中的事件，跟对方请求的位置对比，如果不带请求位置参数，则主节点就会从第一个日志文件中的第一个事件一个一个发送给从节点。

从节点接收到主节点发送过来的数据把它放置到中继日志（Relay log）文件中。并记录该次请求到主节点的具体哪一个二进制日志文件内部的哪一个位置（主节点中的二进制文件会有多个，在后面详细讲解）。

从节点启动另外一个线程（sql Thread ），把 Relay log 中的事件读取出来，并在本地再执行一次。

复制中线程的作用：

从节点：

I/O Thread: 从 Master 节点请求二进制日志事件，并保存于中继日志中。

Sql Thread: 从Relay log 中读取日志事件并在本地完成重放。

主节点：

Dump Thread:为每个 Slave 的 I/O Thread 启动一个 dump 线程，用于向从节点发送二进制事件。

\*\*思考：\*\*从节点需要建立二进制日志文件吗？

看情况，如果从节点需要作为其他节点的主节点时，是需要开启二进制日志文件的。这种情况叫做级联复制。如果只是作为从节点，则不需要创建二进制文件。

Mysql复制特点：

异步复制：主节点中一个用户请求一个写操作时，主节点不需要把写的数据在本地操作完成同时发送给从服务器并等待从服务器反馈写入完成，再响应用户。主节点只需要把写入操作在本地完成，就响应用户。但是，从节点中的数据有可能会落后主节点，可以使用（很多软件来检查是否落后）

主从数据不一致。

主从复制配置过程：

主节点：

启用二进制日志。

为当前节点设置一个全局唯一的server\_id。

创建有复制权限的用户账号 REPLIACTION SLAVE ,REPLIATION CLIENT。

从节点：

启动中继日志。

为当前节点设置一个全局唯一的server\_id。

使用有复制权限的用户账号连接至主节点，并启动复制线程。

配置演示：

测试环境：

主节点：192.168.0.196

从节点：192.168.0.198

mysql版本：mysql-5.5.17

linux：centos6.5

1：编辑主节点配置文件

2：启用二进制日志文件

添加：log-bin = mysql-bin

添加 :server-id =1

添加：innodb-file-per-table =ON

添加：skip\_name\_resolve=ON

3：启动mysql服务，并连接mysql

4：查看日志信息

1:)查看二进制日志是否开启

2:)查看主节点二进制日志列表

3:)查看主节点的server id

5：在主节点上创建有复制权限的用户。REPLIACTION SLAVE ，REOPLIATION CLIENT

6：刷新

主服务器配置完成。

注意：确保主服务器的iptables没有阻断3306的访问端口。

7：编辑从节点配置文件

8：开启中继日志

添加：relay-log=relay-log

添加：relay-log-index=relay-log.index

添加：server-id=2

添加：innodb\_file\_per\_table=ON

添加：skip\_name\_resolve=ON

9:启动从节点mysql服务，并登陆mysql

10:查看从节点日志信息

1:)查看中继日志是否开启

2:)产看server 相关信息

11：在从节点配置访问主节点的参数信息

添加 主节点主机，访问主节点的用户名及密码，主节点二进制文件信息。

注意：主节点的二进制文件一定要是二进制列表中的最后一个二进制文件。

CHANGE MASTER TO MASTER\_HOST='192.168.0.196',MASTER\_USER='repluser',MASTER\_PASSWORD='replpass', MASTER\_LOG\_FILE='mysql-bin.000042',MASTER\_LOG\_POS=362;

12：查看从节点的状态信息

因为没有启动 从节点的复制线程，IO线程 和 SQL 线程都为NO.

13：启动复制线程

START SLAVE 可以指定线程类型：IO\_THREAD ,SQL\_THREAD, 如果不指定，则两个都启动。

14:再次查看从节点状态信息

至此，mysql的主----从复制配置完成。

15：测试

1：）在主节点创建数据库，并查看主节点二进制日志信息

2：）在从节点查找二进制日志信息，并查看mydb数据库是否复制成功

主从复制架构中应注意的问题：

从节点要设置某些限定使得它不能进行写操作,才能保证复制当中的数据一致。

1：限制从服务器为只读

在从服务器上设置：

read\_only = ON,但是此限制对拥有SUPER权限 的用户均无效。

阻止所有用户：

mysq>FLUSH TABLES WITH READ LOCK;

2:如何保证主从复制时的事物安全？

1：在主节点设置参数

sync\_binlog=1： Mysql开启bin-log日志使用bin-log时，默认情况下，并不是每次执行写入就与硬盘同步，这样在服务器崩溃时，就可能导致bin-log最后的语句丢失。可以通过这个参数来调节，sync\_binlog=N，使执行N次写入后，与硬盘同步。1是最安全的，但是也是最慢的。

如果用到innode 存储引擎:

innodb\_flush\_logs\_at\_trx\_commit=ON（刷写日志：在事务提交时，要将内存中跟事务相关的数据立即刷写到事务日志中去。）

innodb\_support\_xa=ON (分布式事务：基于它来做两段式提交功能)

sync\_master\_info=1：每次给从节点dump一些事件信息之后，主节点的master info 信息会立即同步到磁盘上。让从服务器中的 master\_info 及时更新。

2：在每个slave节点

skip\_slave\_start =ON (跳过自动启动，使用手动启动。)

relay\_log也会在内从中先缓存，然后在同步到relay\_log中去，可以使用下面参数使其立即同步。

sync\_relay\_log =1 ，默认为10000，即每10000次sync\_relay\_log事件会刷新到磁盘。为0则表示不刷新，交由OS的cache控制。

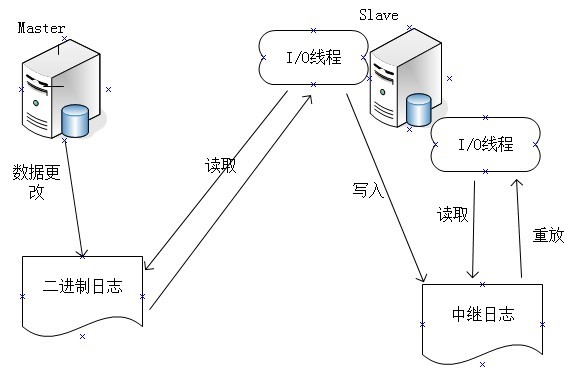
sync\_relay\_log\_info=1每间隔多少事务刷新relay-log.info，如果是table（innodb）设置无效，每个事务都会更新

注： 在从节点中 master.info是记录在主节点复制位置的文件。

relylog\_info: 本地将来至于主节点的哪一个二进制文件中position并且保存文本地哪一个中继日志中的哪一个postion. 从节点启动时也需要根据relay-log.info定位本地relay-log.

Mysql主从复制

工作原理图：



**主从复制的原理：**

分为同步复制和异步复制，实际复制架构中大部分为异步复制。 复制的基本过程如下：

1).Slave上面的IO进程连接上Master，并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；

2).Master接收到来自Slave的IO进程的请求后，通过负责复制的IO进程根据请求信息读取制定日志指定位置之后的日志信息，返回给Slave 的IO进程。返回信息中除了日志所包含的信息之外，还包括本次返回的信息已经到Master端的bin-log文件的名称以及bin-log的位置；

3).Slave的IO进程接收到信息后，将接收到的日志内容依次添加到Slave端的relay-log文件的最末端，并将读取到的Master端的 bin-log的文件名和位置记录到master-info文件中，以便在下一次读取的时候能够清楚的告诉Master“我需要从某个bin-log的哪个位置开始往后的日志内容，请发给我”；

4).Slave的Sql进程检测到relay-log中新增加了内容后，会马上解析relay-log的内容成为在Master端真实执行时候的那些可执行的内容，并在自身执行。

**环境描述**

操作系统：CentOS6.3\_x64

主服务器master：192.168.0.202

从服务器slave：192.168.0.203

一、mysql主从复制

**1、主从安装mysql，版本一致**

我们装的是mysql-5.5.30.tar.gz这里省略...请参考<http://going.blog.51cto.com/7876557/1290440>

**2、修改master，slave服务器**

master服务器配置：

vi /usr/local/mysql/etc/my.cnf

[mysqld]

server-id=202 #设置服务器唯一的id，默认是1，我们设置ip最后一段，slave设置203

log-bin=mysql-bin # 启用二进制日志

#binlog-ignore-db = mysql,information\_schema #忽略写入binlog的库

slave服务器配置：

vi /usr/local/mysql/etc/my.cnf

[mysqld]

server-id=203

replicate-do-db = abc #只同步abc库

slave-skip-errors = all #忽略因复制出现的所有错误

**3、重启主从服务器mysql**

/etc/init.d/mysqld restart

**4、在主服务器上建立帐户并授权slave**

mysql> mysql -u root -p123.com

mysql> GRANT REPLICATION SLAVE ON \*.\* to ‘sync’@‘192.168.1.2’ identified by ‘1234.com’; #replication:复制

**5、查看主数据库状态**

mysql> show master status;

+------------------+----------+--------------+------------------+

| File | Position | Binlog\_Do\_DB | Binlog\_Ignore\_DB |

+------------------+----------+--------------+------------------+

| mysql-bin.000002 | 263 | | |

+------------------+----------+--------------+------------------+

**6、配置从数据库**

mysql> change master to

 -> master\_host='192.168.0.202',

 -> master\_user='sync',

 -> master\_password='1234.com',

 -> master\_log\_file='mysql-bin.000002',

 -> master\_log\_pos=263;

#Log和pos是master上随机获取的。这段也可以写到my.cnf里面。

**7、启动slave同步进程并查看状态**

mysql> start slave;

[](https://s4.51cto.com/attachment/201310/103403849.jpg)

其中Slave\_IO\_Running 与 Slave\_SQL\_Running 的值都必须为YES，才表明状态正常。

**8、验证主从同步**

在主mysql创建数据库abc，再从mysql查看已经同步成功！

mysql> create database abc;

mysql> show databases;

+--------------------+

| Database |

+--------------------+

| information\_schema |

| abc |

| mysql |

| performance\_schema |

| test |

+--------------------+

在slave启动报错：

“Last\_IO\_Error: Got fatal error 1236 from master when reading data from binary log: ‘Could not find first log file name in binary log index file’”

解决：报错的原因主要是slave设置master的二进制文件名或pos值不对应！

先flush logs；清空日志，在查看下主数据库的状态 show master status;看下日志文件名字和position值；

再在slave中，执行：CHANGE MASTER TO MASTER\_LOG\_FILE=‘二进制日志名’,MASTER\_LOG\_POS=值;

最后启动同步进程：start slave；

# 123、python模块之psutil详解

目录

* [python模块之psutil详解](https://www.cnblogs.com/iamjianghao/p/11894623.html#python%E6%A8%A1%E5%9D%97%E4%B9%8Bpsutil%E8%AF%A6%E8%A7%A3)
  + [简介](https://www.cnblogs.com/iamjianghao/p/11894623.html#%E7%AE%80%E4%BB%8B)
* [功能函数](https://www.cnblogs.com/iamjianghao/p/11894623.html#%E5%8A%9F%E8%83%BD%E5%87%BD%E6%95%B0)
  + [CPU相关：](https://www.cnblogs.com/iamjianghao/p/11894623.html#cpu%E7%9B%B8%E5%85%B3%EF%BC%9A)
  + [Memory内存相关:](https://www.cnblogs.com/iamjianghao/p/11894623.html#memory%E5%86%85%E5%AD%98%E7%9B%B8%E5%85%B3)
  + [Disk相关：](https://www.cnblogs.com/iamjianghao/p/11894623.html#disk%E7%9B%B8%E5%85%B3%EF%BC%9A)
  + [Network相关：](https://www.cnblogs.com/iamjianghao/p/11894623.html#network%E7%9B%B8%E5%85%B3%EF%BC%9A)
  + [进程管理](https://www.cnblogs.com/iamjianghao/p/11894623.html#%E8%BF%9B%E7%A8%8B%E7%AE%A1%E7%90%86)

## 简介

psutil是一个开源切跨平台的库，其提供了便利的函数用来获取才做系统的信息，比如CPU，内存，磁盘，网络等。此外，psutil还可以用来进行进程管理，包括判断进程是否存在、获取进程列表、获取进程详细信息等。而且psutil还提供了许多命令行工具提供的功能，包括：ps，top，lsof，netstat，ifconfig， who，df，kill，free，nice，ionice，iostat，iotop，uptime，pidof，tty，taskset，pmap。

psutil是一个跨平台的库，在官方网站上查到其支持如下操作系统。

Linux

Windows

OSX

FreeBSD

OpenBSD

NetBSD

Sun Solaris

AIX

Works with Python versions from 2.6 to 3.X.

psutil包含了异常、类、功能函数和常量，其中功能函数用来获取系统的信息，如CPU、磁盘、内存、网络等。类用来实现进程的管理功能

## 功能函数

根据函数的功能，主要分为CPU、磁盘、内存、网络几类，下面将会总几个方面来介绍psutil提供的功能函数。

## CPU相关：

| **函数** | **描述** |
| --- | --- |
| psutil.cpu\_count() | cpu\_count(,[logical]):默认返回逻辑CPU的个数,当设置logical的参数为False时，返回物理CPU的个数。 |
| psutil.cpu\_percent() | cpu\_percent(,[percpu],[interval])：返回CPU的利用率,percpu为True时显示所有物理核心的利用率,interval不为0时,则阻塞时显示interval执行的时间内的平均利用率 |
| psutil.cpu\_times() | cpu\_times(,[percpu])：以命名元组(namedtuple)的形式返回cpu的时间花费,percpu=True表示获取每个CPU的时间花费 |
| psutil.cpu\_times\_percent() | cpu\_times\_percent(,[percpu])：功能和cpu\_times大致相同，看字面意思就能知道，该函数返回的是耗时比例。 |
| psutil.cpu\_stats() | cpu\_stats()以命名元组的形式返回CPU的统计信息，包括上下文切换，中断，软中断和系统调用次数。 |
| psutil.cpu\_freq() | cpu\_freq([percpu])：返回cpu频率 |

**示例**：

# 查看cpu个数

>>> import psutil

>>> psutil.cpu\_count()

2

>>> psutil.cpu\_count(logical=False)

1

>>>

# 查看cpu利用率

>>> psutil.cpu\_percent()

0.2

>>> psutil.cpu\_percent(percpu=True)

[0.1, 0.2]

>>>

# 查看cpu时间花费

>>> psutil.cpu\_times()

scputimes(user=29.09, nice=0.0, system=22.62, idle=24434.77, iowait=1.74, irq=0.0, softirq=0.28, steal=0.27, guest=0.0, guest\_nice=0.0)

>>> psutil.cpu\_times(percpu=True)

[scputimes(user=13.64, nice=0.0, system=12.02, idle=12235.5, iowait=1.0, irq=0.0, softirq=0.16, steal=0.09, guest=0.0, guest\_nice=0.0),

scputimes(user=15.47, nice=0.0, system=10.62, idle=12229.44, iowait=0.74, irq=0.0, softirq=0.12, steal=0.17, guest=0.0, guest\_nice=0.0)]

>>> print(cpu\_time)

scputimes(user=29.24, nice=0.0, system=22.76, idle=24618.94, iowait=1.74, irq=0.0, softirq=0.28, steal=0.27, guest=0.0, guest\_nice=0.0)

>>> cpu\_time.user

29.24

## Memory内存相关:

**virtual\_memory()**：以命名元组的形式返回内存使用情况，包括总内存，可用内存，内存利用率，buffer和cache等。单位为字节。

>>> import psutil

>>> psutil.virtual\_memory()

svmem(total=8071716864, available=6532554752, percent=19.1, used=1258717184, free=6526308352, active=1153519616, inactive=194592768, buffers=2129920, cached=284561408, shared=9011200, slab=39006208)

swap\_memory()：以命名元组的形式返回swap/memory使用情况，包含swap中页的换入和换出。

>>> import psutil

>>> psutil.swap\_memory()

sswap(total=17179865088, used=0, free=17179865088, percent=0.0, sin=0, sout=0)

>>>

单位转换

import psutil

def bytes2human(n):

symbols = ('K','M','G','T','P','E','Z','Y')

prefix = {}

for i,s in enumerate(symbols):

prefix[s] = 1 << (i + 1) \* 10

for s in reversed(symbols):

if n >= prefix[s]:

value = float(n) / prefix[s]

return '%.1f%s' % (value,s)

return '%sB' % n

print(bytes2human(psutil.virtual\_memory().total))

## Disk相关：

| **函数** | **描述** |
| --- | --- |
| psutil.disk\_io\_counters() | disk\_io\_counters([perdisk])：以命名元组的形式返回磁盘io统计信息(汇总的)，包括读、写的次数，读、写的字节数等。  当perdisk的值为True，则分别列出单个磁盘的统计信息(字典：key为磁盘名称，value为统计的namedtuple)。 |
| psutil.disk\_partitions() | disk\_partitions([all=False])：以命名元组的形式返回所有已挂载的磁盘，包含磁盘名称，挂载点，文件系统类型等信息。  当all等于True时，返回包含/proc等特殊文件系统的挂载信息 |
| psutil.disk\_usage() | disk\_usage(path)：以命名元组的形式返回path所在磁盘的使用情况，包括磁盘的容量、已经使用的磁盘容量、磁盘的空间利用率等。 |

# 查看所有已挂在的磁盘

>>> psutil.disk\_partitions()

[sdiskpart(device='/dev/vda3', mountpoint='/', fstype='xfs', opts='rw,relatime,attr2,inode64,noquota'), sdiskpart(device='/dev/vda6', mountpoint='/data1', fstype='xfs', opts='rw,relatime,attr2,inode64,noquota'), sdiskpart(device='/dev/vda2', mountpoint='/boot', fstype='xfs', opts='rw,relatime,attr2,inode64,noquota')]

# 使用列表表达式查询指定挂载点信息

>>> [device for device in psutil.disk\_partitions() if device.mountpoint == '/']

[sdiskpart(device='/dev/vda3', mountpoint='/', fstype='xfs', opts='rw,relatime,attr2,inode64,noquota')]

>>>

# 查看磁盘使用情况

>>> psutil.disk\_usage('/')

sdiskusage(total=85857402880, used=3858100224, free=81999302656, percent=4.5)

# 查看磁盘io统计汇总

>>> psutil.disk\_io\_counters()

sdiskio(read\_count=6828, write\_count=3878, read\_bytes=273637888, write\_bytes=30182912, read\_time=6870, write\_time=2079, read\_merged\_count=7, write\_merged\_count=126, busy\_time=4841)

# 分别列出单个磁盘的统计信息

>>> psutil.disk\_io\_counters(perdisk=True)

{'vda': sdiskio(read\_count=6828, write\_count=3878, read\_bytes=273637888, write\_bytes=30182912, read\_time=6870, write\_time=2079, read\_merged\_count=7, write\_merged\_count=126, busy\_time=4841), 'vda1': sdiskio(read\_count=34, write\_count=0, read\_bytes=139264, write\_bytes=0, read\_time=1, write\_time=0, read\_merged\_count=0, write\_merged\_count=0, busy\_time=1), 'vda2': sdiskio(read\_count=1934, write\_count=2049, read\_bytes=22754816, write\_bytes=2097152, read\_time=570, write\_time=801, read\_merged\_count=0, write\_merged\_count=0, busy\_time=1347), 'vda3': sdiskio(read\_count=4009, write\_count=1729, read\_bytes=187268608, write\_bytes=12412416, read\_time=5302, write\_time=911, read\_merged\_count=4, write\_merged\_count=115, busy\_time=3114), 'vda4': sdiskio(read\_count=6, write\_count=0, read\_bytes=18432, write\_bytes=0, read\_time=1, write\_time=0, read\_merged\_count=0, write\_merged\_count=0, busy\_time=1), 'vda5': sdiskio(read\_count=48, write\_count=0, read\_bytes=2248704, write\_bytes=0, read\_time=13, write\_time=0, read\_merged\_count=0, write\_merged\_count=0, busy\_time=9), 'vda6': sdiskio(read\_count=763, write\_count=100, read\_bytes=60118528, write\_bytes=15673344, read\_time=948, write\_time=367, read\_merged\_count=3, write\_merged\_count=11, busy\_time=459), 'sr0': sdiskio(read\_count=0, write\_count=0, read\_bytes=0, write\_bytes=0, read\_time=0, write\_time=0, read\_merged\_count=0, write\_merged\_count=0, busy\_time=0)}

## Network相关：

| **函数** | **详情** |
| --- | --- |
| psutil.net\_io\_counter([pernic]) | 以命名元组的形式返回当前系统中每块网卡的网络io统计信息，包括收发字节数，收发包的数量、出错的情况和删包情况。当pernic为True时，则列出所有网卡的统计信息。 |
| psutil.net\_connections([kind]) | 以列表的形式返回每个网络连接的详细信息(namedtuple)。命名元组包含fd, family, type, laddr, raddr, status, pid等信息。kind表示过滤的连接类型，支持的值如下：(默认为inet) |
| psutil.net\_if\_addrs() | 以字典的形式返回网卡的配置信息，包括IP地址和mac地址、子网掩码和广播地址。 |
| psutil.net\_if\_stats() | 返回网卡的详细信息，包括是否启动、通信类型、传输速度与mtu。 |
| psutil.users() | 以命名元组的方式返回当前登陆用户的信息，包括用户名，登陆时间，终端，与主机信息 |
| psutil.boot\_time() | 以时间戳的形式返回系统的启动时间 |

# 查看网卡信息统计

>>> psutil.net\_io\_counters()

snetio(bytes\_sent=9699431, bytes\_recv=1895536, packets\_sent=8606, packets\_recv=27354, errin=0, errout=0, dropin=0, dropout=0)

# 查看网卡配置信息

>>> psutil.net\_if\_addrs()

{'lo': [snicaddr(family=<AddressFamily.AF\_INET: 2>, address='127.0.0.1', netmask='255.0.0.0', broadcast=None, ptp=None), snicaddr(family=<AddressFamily.AF\_INET6: 10>, address='::1', netmask='ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff', broadcast=None, ptp=None), snicaddr(family=<AddressFamily.AF\_PACKET: 17>, address='00:00:00:00:00:00', netmask=None, broadcast=None, ptp=None)], 'eth0': [snicaddr(family=<AddressFamily.AF\_INET: 2>, address='172.16.5.12', netmask='255.255.255.0', broadcast='172.16.5.255', ptp=None), snicaddr(family=<AddressFamily.AF\_INET6: 10>, address='fe80::9700:20da:ed33:9f50%eth0', netmask='ffff:ffff:ffff:ffff::', broadcast=None, ptp=None), snicaddr(family=<AddressFamily.AF\_PACKET: 17>, address='52:54:00:3d:ea:06', netmask=None, broadcast='ff:ff:ff:ff:ff:ff', ptp=None)]}

>>> a = psutil.net\_if\_addrs()

>>> a['eth0']

[snicaddr(family=<AddressFamily.AF\_INET: 2>, address='172.16.5.12', netmask='255.255.255.0', broadcast='172.16.5.255', ptp=None), snicaddr(family=<AddressFamily.AF\_INET6: 10>, address='fe80::9700:20da:ed33:9f50%eth0', netmask='ffff:ffff:ffff:ffff::', broadcast=None, ptp=None), snicaddr(family=<AddressFamily.AF\_PACKET: 17>, address='52:54:00:3d:ea:06', netmask=None, broadcast='ff:ff:ff:ff:ff:ff', ptp=None)]

>>> a['eth0'][0]

snicaddr(family=<AddressFamily.AF\_INET: 2>, address='172.16.5.12', netmask='255.255.255.0', broadcast='172.16.5.255', ptp=None)

>>> a['eth0'][0][1]

'172.16.5.12'

# 查看当前登录用户信息

>>> psutil.users()

[suser(name='root', terminal='pts/0', host='172.16.2.66', started=1574151552.0, pid=1437), suser(name='root', terminal='pts/1', host='172.16.2.66', started=1574161536.0, pid=1699)]

>>>

## 进程管理

psutil还提供了作为进程管理的功能函数，包括获取进程列表，判断是否存在。

| **函数** | **详情** |
| --- | --- |
| psutil.pids() | 以列表的形式返回当前正在运行的进程 |
| psutil.pid\_exists(1) | 判断给点定的pid是否存在 |
| psutil.process\_iter() | 迭代当前正在运行的进程，返回的是每个进程的Process对象 |
| psutil.Process() | 对进程进行封装，可以使用该类的方法获取进行的详细信息，或者给进程发送信号。 |

**进程相关信息的方法**：

name：获取进程的名称

cmdline：获取启动进程的命令行参数

create\_time：获取进程的创建时间(时间戳格式)

num\_fds：进程打开的文件个数

num\_threads：进程的子进程个数

is\_running：判断进程是否正在运行

send\_signal：给进程发送信号，类似与os.kill等

kill：发送SIGKILL信号结束进程

terminate：发送SIGTEAM信号结束进程

# 以列表形式查看正在运行的进程

>>> psutil.pids()

[1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 32, 33, 34, 35, 43, 44, 45, 46, 47, 48, 61, 93, 99, 233, 234, 236, 237, 238, 242, 244, 245, 260, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 346, 374, 377, 378, 497, 498, 499, 500, 501, 502, 503, 506, 508, 509, 510, 511, 512, 513, 514, 515, 612, 616, 636, 638, 639, 640, 641, 643, 646, 647, 655, 658, 661, 677, 688, 689, 901, 902, 903, 912, 925, 949, 977, 981, 992, 994, 995, 997, 1001, 1002, 1003, 1004, 1255, 1536, 1840, 1842, 1844, 1861, 1862, 1863, 1864]

# 查看进程运行状态，以布尔形式显示

>>> psutil.pid\_exists(1)

True

# 迭代当前正在运行进程，查看列表中前三个实例的信息

>>> list(psutil.process\_iter())[:3]

[psutil.Process(pid=1, name='systemd', started='16:19:47'), psutil.Process(pid=2, name='kthreadd', started='16:19:47'), psutil.Process(pid=3, name='ksoftirqd/0', started='16:19:47')]

>>>

# 通过进程号实例化对象

>>> process = psutil.Process(1)

>>> print(process)

psutil.Process(pid=1, name='systemd', started='16:19:47')

# 获取进程的名称

>>> process.name()

'systemd'

>>> process.create\_time()

1574151587.05

>>> process.num\_fds()

47

>>> process.num\_threads()

1

>>>

# 125、python subprocess详解

目录

* [python subprocess详解](https://www.cnblogs.com/iamjianghao/p/11896564.html#python-subprocess%E8%AF%A6%E8%A7%A3)
  + [简介：](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E7%AE%80%E4%BB%8B%EF%BC%9A)
  + [运行python的时候，我们都是在创建并运行一个进程。像Linux进程那样，一个进程可以fork一个子进程，并让这个子进程exec另外一个程序。在Python中，我们通过标准库中的subprocess包来fork一个子进程，并运行一个外部的程序。subprocess包中定义有数个创建子进程的函数，这些函数分别以不同的方式创建子进程，所以我们可以根据需要来从中选取一个使用。另外subprocess还提供了一些管理标准流(standard stream)和管道(pipe)的工具，从而在进程间使用文本通信。](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E8%BF%90%E8%A1%8Cpython%E7%9A%84%E6%97%B6%E5%80%99%EF%BC%8C%E6%88%91%E4%BB%AC%E9%83%BD%E6%98%AF%E5%9C%A8%E5%88%9B%E5%BB%BA%E5%B9%B6%E8%BF%90%E8%A1%8C%E4%B8%80%E4%B8%AA%E8%BF%9B%E7%A8%8B%E3%80%82%E5%83%8Flinux%E8%BF%9B%E7%A8%8B%E9%82%A3%E6%A0%B7%EF%BC%8C%E4%B8%80%E4%B8%AA%E8%BF%9B%E7%A8%8B%E5%8F%AF%E4%BB%A5fork%E4%B8%80%E4%B8%AA%E5%AD%90%E8%BF%9B%E7%A8%8B%EF%BC%8C%E5%B9%B6%E8%AE%A9%E8%BF%99%E4%B8%AA%E5%AD%90%E8%BF%9B%E7%A8%8Bexec%E5%8F%A6%E5%A4%96%E4%B8%80%E4%B8%AA%E7%A8%8B%E5%BA%8F%E3%80%82%E5%9C%A8python%E4%B8%AD%EF%BC%8C%E6%88%91%E4%BB%AC%E9%80%9A%E8%BF%87%E6%A0%87%E5%87%86%E5%BA%93%E4%B8%AD%E7%9A%84subprocess%E5%8C%85%E6%9D%A5fork%E4%B8%80%E4%B8%AA%E5%AD%90%E8%BF%9B%E7%A8%8B%EF%BC%8C%E5%B9%B6%E8%BF%90%E8%A1%8C%E4%B8%80%E4%B8%AA%E5%A4%96%E9%83%A8%E7%9A%84%E7%A8%8B%E5%BA%8F%E3%80%82subprocess%E5%8C%85%E4%B8%AD%E5%AE%9A%E4%B9%89%E6%9C%89%E6%95%B0%E4%B8%AA%E5%88%9B%E5%BB%BA%E5%AD%90%E8%BF%9B%E7%A8%8B%E7%9A%84%E5%87%BD%E6%95%B0%EF%BC%8C%E8%BF%99%E4%BA%9B%E5%87%BD%E6%95%B0%E5%88%86%E5%88%AB%E4%BB%A5%E4%B8%8D%E5%90%8C%E7%9A%84%E6%96%B9%E5%BC%8F%E5%88%9B%E5%BB%BA%E5%AD%90%E8%BF%9B%E7%A8%8B%EF%BC%8C%E6%89%80%E4%BB%A5%E6%88%91%E4%BB%AC%E5%8F%AF%E4%BB%A5%E6%A0%B9%E6%8D%AE%E9%9C%80%E8%A6%81%E6%9D%A5%E4%BB%8E%E4%B8%AD%E9%80%89%E5%8F%96%E4%B8%80%E4%B8%AA%E4%BD%BF%E7%94%A8%E3%80%82%E5%8F%A6%E5%A4%96subprocess%E8%BF%98%E6%8F%90%E4%BE%9B%E4%BA%86%E4%B8%80%E4%BA%9B%E7%AE%A1%E7%90%86%E6%A0%87%E5%87%86%E6%B5%81standard-stream%E5%92%8C%E7%AE%A1%E9%81%93pipe%E7%9A%84%E5%B7%A5%E5%85%B7%EF%BC%8C%E4%BB%8E%E8%80%8C%E5%9C%A8%E8%BF%9B%E7%A8%8B%E9%97%B4%E4%BD%BF%E7%94%A8%E6%96%87%E6%9C%AC%E9%80%9A%E4%BF%A1%E3%80%82)
  + [subprocess.call()](https://www.cnblogs.com/iamjianghao/p/11896564.html#subprocesscall)
    - [示例：](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E7%A4%BA%E4%BE%8B%EF%BC%9A)
  + [subprocess.check\_call()](https://www.cnblogs.com/iamjianghao/p/11896564.html#subprocesscheck_call)
    - [示例：](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E7%A4%BA%E4%BE%8B%EF%BC%9A)
  + [subprocess.check\_output()：](https://www.cnblogs.com/iamjianghao/p/11896564.html#subprocesscheck_output%EF%BC%9A)
    - [示例：](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E7%A4%BA%E4%BE%8B%EF%BC%9A)
  + [subprocess.Popen()：](https://www.cnblogs.com/iamjianghao/p/11896564.html#subprocesspopen%EF%BC%9A)
    - [Popen方法：](https://www.cnblogs.com/iamjianghao/p/11896564.html#popen%E6%96%B9%E6%B3%95%EF%BC%9A)
    - [示例：](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E7%A4%BA%E4%BE%8B%EF%BC%9A)
    - [注意！](https://www.cnblogs.com/iamjianghao/p/11896564.html#%E6%B3%A8%E6%84%8F%EF%BC%81)

## 简介：

运行python的时候，我们都是在创建并运行一个进程。像Linux进程那样，一个进程可以fork一个子进程，并让这个子进程exec另外一个程序。在Python中，我们通过标准库中的subprocess包来fork一个子进程，并运行一个外部的程序。

subprocess包中定义有数个创建子进程的函数，这些函数分别以不同的方式创建子进程，所以我们可以根据需要来从中选取一个使用。另外subprocess还提供了一些管理标准流(standard stream)和管道(pipe)的工具，从而在进程间使用文本通信。

## subprocess.call()

父进程等待子进程完成

返回退出信息(returncode，相当于Linux exit code)

示例：

>>> a = subprocess.call(['df','-hT'],shell=False)

文件系统 类型 容量 已用 可用 已用% 挂载点

/dev/vda3 xfs 80G 3.6G 77G 5% /

devtmpfs devtmpfs 7.8G 0 7.8G 0% /dev

tmpfs tmpfs 7.8G 0 7.8G 0% /dev/shm

tmpfs tmpfs 7.8G 8.6M 7.8G 1% /run

tmpfs tmpfs 7.8G 0 7.8G 0% /sys/fs/cgroup

/dev/vda6 xfs 404G 2.2G 402G 1% /data1

/dev/vda2 xfs 497M 128M 370M 26% /boot

tmpfs tmpfs 1.5G 0 1.5G 0% /run/user/0

>>> print(a)

0 #程序返回的代码值

>>> a = subprocess.call('aaaa',shell=True)

/bin/sh: aaaa: 未找到命令

>>>

## subprocess.check\_call()

subprocess.check\_call(args, \*, stdin = None, stdout = None, stderr = None, shell = False)

与call方法类似，不同在于如果命令行执行成功，check\_call返回返回码0，否则抛出subprocess.CalledProcessError异常。

subprocess.CalledProcessError异常包括returncode、cmd、output等属性，其中returncode是子进程的退出码，cmd是子进程的执行命令，output为None。

示例：

>>> import subprocess

>>> a = subprocess.check\_call('df -h',shell=True)

文件系统 容量 已用 可用 已用% 挂载点

/dev/vda3 80G 3.6G 77G 5% /

devtmpfs 7.8G 0 7.8G 0% /dev

tmpfs 7.8G 0 7.8G 0% /dev/shm

tmpfs 7.8G 8.6M 7.8G 1% /run

tmpfs 7.8G 0 7.8G 0% /sys/fs/cgroup

/dev/vda6 404G 2.2G 402G 1% /data1

/dev/vda2 497M 128M 370M 26% /boot

tmpfs 1.2G 0 1.2G 0% /run/user/0

>>> print(a)

0

>>> a = subprocess.check\_call('aaaa',shell=True)

/bin/sh: aaaa: 未找到命令

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/usr/local/python3/lib/python3.6/subprocess.py", line 291, in check\_call

raise CalledProcessError(retcode, cmd)

subprocess.CalledProcessError: Command 'aaaa' returned non-zero exit status 127.

>>>

## subprocess.check\_output()：

用法与上面两个方法类似，区别是，如果当返回值为0时，直接返回输出结果，如果返回值不为0，直接抛出异常

示例：

>>> a = subprocess.check\_output(['df','-hT'],shell=False)

>>> print(a)

b'\xe6\x96\x87\xe4\xbb\xb6\xe7\xb3\xbb\xe7\xbb\x9f \xe7\xb1\xbb\xe5\x9e\x8b \xe5\xae\xb9\xe9\x87\x8f \xe5\xb7\xb2\xe7\x94\xa8 \xe5\x8f\xaf\xe7\x94\xa8 \xe5\xb7\xb2\xe7\x94\xa8% \xe6\x8c\x82\xe8\xbd\xbd\xe7\x82\xb9\n/dev/vda3 xfs 80G 3.6G 77G 5% /\ndevtmpfs devtmpfs 7.8G 0 7.8G 0% /dev\ntmpfs tmpfs 7.8G 0 7.8G 0% /dev/shm\ntmpfs tmpfs 7.8G 8.6M 7.8G 1% /run\ntmpfs tmpfs 7.8G 0 7.8G 0% /sys/fs/cgroup\n/dev/vda6 xfs 404G 2.2G 402G 1% /data1\n/dev/vda2 xfs 497M 128M 370M 26% /boot\ntmpfs tmpfs 1.5G 0 1.5G 0% /run/user/0\n'

>>> print(str(a.rstrip(),'utf-8'))

文件系统 类型 容量 已用 可用 已用% 挂载点

/dev/vda3 xfs 80G 3.6G 77G 5% /

devtmpfs devtmpfs 7.8G 0 7.8G 0% /dev

tmpfs tmpfs 7.8G 0 7.8G 0% /dev/shm

tmpfs tmpfs 7.8G 8.6M 7.8G 1% /run

tmpfs tmpfs 7.8G 0 7.8G 0% /sys/fs/cgroup

/dev/vda6 xfs 404G 2.2G 402G 1% /data1

/dev/vda2 xfs 497M 128M 370M 26% /boot

tmpfs tmpfs 1.5G 0 1.5G 0% /run/user/0

>>>

## subprocess.Popen()：

subprocess模块定义了一个Popen类，通过它可以创建进程，并与其进行复杂的交互。

class Popen(args, bufsize=0, executable=None, stdin=None,stdout=None, stderr=None, preexec\_fn=None, close\_fds=False,

shell=False, cwd=None, env=None, universal\_newlines=False, startupinfo=None, creationflags=0)

**在一些复杂场景中，我们需要将一个进程的执行输出作为另一个进程的输入。在另一些场景中，我们需要先进入到某个输入环境，然后再执行一系列的指令等。这个时候我们就需要使用到suprocess的Popen()方法。该方法有以下参数**

| **参数** | **作用** |
| --- | --- |
| args | 一般是一个字符串，是要执行的shell命令内容 |
| bufsize | 设置缓冲，负数表示系统默认缓冲，0表示无缓冲，正数表示自定义缓冲行数 |
| stdin | 程序的标准输入句柄，NONE表示不进行重定向，继承父进程，PIPE表示创建管道 |
| stdout | 程序的标准输出句柄，参数意义同上 |
| stderr | 程序的标准错误句柄，参数意义同上，特殊，可以设置成STDOUT，表示与标准输出一致 |
| shell | 为True时，表示将通过shell来执行 |
| cwd | 用来设置当前子进程的目录 |
| env | 用于指定子进程的环境变量。如果env=None，则默认从父进程继承环境变量 |
| universal\_newlines | 不同系统的的换行符不同，当该参数设定为true时，则表示使用\n作为换行符。 |

### Popen方法：

* 1、Popen.poll()：用于检查子进程是否已经结束。设置并返回returncode属性。
* 2、Popen.wait()：等待子进程结束。设置并返回returncode属性。
* 3、Popen.communicate(input=None)：与子进程进行交互。向stdin发送数据，或从stdout和stderr中读取数据。可选参数input指定发送到子进程的参数。Communicate()返回一个元组：(stdoutdata, stderrdata)。注意：如果希望通过进程的stdin向其发送数据，在创建Popen对象的时候，参数stdin必须被设置为PIPE。同样，如果希望从stdout和stderr获取数据，必须将stdout和stderr设置为PIPE。
* 4、Popen.send\_signal(signal)：向子进程发送信号。
* 5、Popen.terminate()：停止(stop)子进程。在windows平台下，该方法将调用Windows API TerminateProcess（）来结束子进程。
* 6、Popen.kill()：杀死子进程。
* 7、Popen.stdin：如果在创建Popen对象是，参数stdin被设置为PIPE，Popen.stdin将返回一个文件对象用于策子进程发送指令。否则返回None。
* 8、Popen.stdout：如果在创建Popen对象是，参数stdout被设置为PIPE，Popen.stdout将返回一个文件对象用于策子进程发送指令。否则返回None。
* 9、Popen.stderr：如果在创建Popen对象是，参数stdout被设置为PIPE，Popen.stdout将返回一个文件对象用于策子进程发送指令。否则返回None。
* 10、Popen.pid：获取子进程的进程ID。
* 11、Popen.returncode：获取进程的返回值。如果进程还没有结束，返回None。
* 12、subprocess.call(\*popenargs, \*\*kwargs)：运行命令。该函数将一直等待到子进程运行结束，并返回进程的returncode。文章一开始的例子就演示了call函数。如果子进程不需要进行交互,就可以使用该函数来创建。
* 13、subprocess.check\_call(*popenargs, \*\*kwargs)：与subprocess.call(*popenargs, \*\*kwargs)功能一样，只是如果子进程返回的returncode不为0的话，将触发CalledProcessError异常。在异常对象中，包括进程的returncode信息

### 示例：

>>> import subprocess

>>> subprocess.Popen('/apps/apache-tomcat-8.5.29/bin/startup.sh',shell=True)

<subprocess.Popen object at 0x7f3579b2dfd0>

>>> Using CATALINA\_BASE: /apps/apache-tomcat-8.5.29

Using CATALINA\_HOME: /apps/apache-tomcat-8.5.29

Using CATALINA\_TMPDIR: /apps/apache-tomcat-8.5.29/temp

Using JRE\_HOME: /usr/java/jdk1.8.0\_111/

Using CLASSPATH: /apps/apache-tomcat-8.5.29/bin/bootstrap.jar:/apps/apache-tomcat-8.5.29/bin/tomcat-juli.jar

Tomcat started.

>>> popen = subprocess.Popen('tail -f /apps/apache-tomcat-8.5.29/logs/catalina.out',shell=True)

>>> at org.apache.tomcat.util.net.AbstractEndpoint.init(AbstractEndpoint.java:1086)

at org.apache.tomcat.util.net.AbstractJsseEndpoint.init(AbstractJsseEndpoint.java:268)

at org.apache.coyote.AbstractProtocol.init(AbstractProtocol.java:581)

at org.apache.catalina.connector.Connector.initInternal(Connector.java:993)

... 13 more

19-Nov-2019 17:05:05.020 信息 [main] org.apache.catalina.startup.Catalina.load Initialization processed in 650 ms

19-Nov-2019 17:05:05.045 信息 [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]

19-Nov-2019 17:05:05.045 信息 [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet Engine: Apache Tomcat/8.5.29

19-Nov-2019 17:05:05.061 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [/apps/apache-tomcat-8.5.29/webapps/ROOT]

KeyboardInterrupt

>>> popen.terminate()

>>>

注意！

在shell=True这个参数，不写的时候默认是False，shell默认为/bin/sh。如果 args是一个字符串，则该字符串指定要通过shell执行的命令。

当需要设置shell=True时(当False时，arges是列表，第一个参数是shell命令，后面的都是参数',' 隔开)，须把args设为string，空格隔开，如下

>>> a = subprocess.Popen(['tail','-f', '/apps/apache-tomcat-8.5.29/logs/catalina.out'])

>>> a = subprocess.Popen('tail -f /apps/apache-tomcat-8.5.29/logs/catalina.out',shell=True)

# [返回主页](https://www.cnblogs.com/iamjianghao/)126、shutil模块高级的文件操作模块：

目录

* [shutil模块高级的文件操作模块：](https://www.cnblogs.com/iamjianghao/p/11899792.html#shutil%E6%A8%A1%E5%9D%97%E9%AB%98%E7%BA%A7%E7%9A%84%E6%96%87%E4%BB%B6%E6%93%8D%E4%BD%9C%E6%A8%A1%E5%9D%97%EF%BC%9A)
  + [简介：](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E7%AE%80%E4%BB%8B%EF%BC%9A)
  + [shutil功能：](https://www.cnblogs.com/iamjianghao/p/11899792.html#shutil%E5%8A%9F%E8%83%BD%EF%BC%9A)
    - [拷贝/删除/移动：](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E6%8B%B7%E8%B4%9D%E5%88%A0%E9%99%A4%E7%A7%BB%E5%8A%A8%EF%BC%9A)
      * [代码演示：](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E4%BB%A3%E7%A0%81%E6%BC%94%E7%A4%BA%EF%BC%9A)
    - [创建压缩解压](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E5%88%9B%E5%BB%BA%E5%8E%8B%E7%BC%A9%E8%A7%A3%E5%8E%8B)
      * [参数注解：](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E5%8F%82%E6%95%B0%E6%B3%A8%E8%A7%A3%EF%BC%9A)
      * [演示：](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E6%BC%94%E7%A4%BA%EF%BC%9A)
    - [注意： shutil 对压缩包的处理是调用 ZipFile 和 TarFile 两个模块来进行的!](https://www.cnblogs.com/iamjianghao/p/11899792.html#%E6%B3%A8%E6%84%8F%EF%BC%9A-shutil-%E5%AF%B9%E5%8E%8B%E7%BC%A9%E5%8C%85%E7%9A%84%E5%A4%84%E7%90%86%E6%98%AF%E8%B0%83%E7%94%A8-zipfile-%E5%92%8C-tarfile-%E4%B8%A4%E4%B8%AA%E6%A8%A1%E5%9D%97%E6%9D%A5%E8%BF%9B%E8%A1%8C%E7%9A%84)
      * [zipfile压缩解压缩](https://www.cnblogs.com/iamjianghao/p/11899792.html#zipfile%E5%8E%8B%E7%BC%A9%E8%A7%A3%E5%8E%8B%E7%BC%A9)
      * [tarfile压缩解压](https://www.cnblogs.com/iamjianghao/p/11899792.html#tarfile%E5%8E%8B%E7%BC%A9%E8%A7%A3%E5%8E%8B)

### 简介：

os模块提供了对目录或者文件的新建/删除/查看文件属性，还提供了对文件以及目录的路径操作。

比如说：绝对路径，父目录…… 但是，os文件的操作还应该包含移动 复制 打包 压缩 解压等操作，这些os模块都没有提供。

而本章所讲的shutil则就是对os中文件操作的补充。--移动 复制 打包 压缩 解压。

### shutil功能：

### 拷贝/删除/移动：

| **函数** | **介绍** | **示例** |
| --- | --- | --- |
| shutil.copyfileobj(fsrc, fdst[, length=16\*1024]) | 将文件内容拷贝到另一个文件中,可以指定大小的 内容 | import shutil  shutil.copyfileobj(open('old.xml','r'), open('new.xml', 'w')) |
| shutil.copyfile(src, dst) | 拷贝文件 | shutil.copyfile('f1.log', 'f2.log') #目标文件无需存在，copyfile调用了copyfileobj |
| shutil.copymode(src, dst) | 仅拷贝权限。内容、组、用户均不变 | shutil.copymode('f1.log', 'f2.log') #目标文件必须存在 |
| shutil.copystat(src, dst) | 仅拷贝状态的信息，  包括：mode bits, atime, mtime, flags | shutil.copystat('f1.log', 'f2.log') #目标文件必须存在 |
| shutil.copy(src, dst) | 拷贝文件和权限 | import shutil  shutil.copy('f1.log', 'f2.log') |
| shutil.copy2(src, dst) | 拷贝文件和状态信息 | import shutil  shutil.copy2('f1.log', 'f2.log') |
| shutil.ignore\_patterns(\*patterns)  shutil.copytree(src, dst, symlinks=False, ignore=None,ignore=None) | 递归的复制文件内容及状态信息,ignore的意思是排除 | import shutil  shutil.copytree('folder1', 'folder2', ignore=shutil.ignore\_patterns('.pyc', 'tmp'))  #目标目录不能存在，注意对folder2目录父级目录要有可写权限 |
| - | 拷贝软连接 | import shutil  shutil.copytree('f1', 'f2', symlinks=True, ignore=shutil.ignore\_patterns('.pyc', 'tmp')) |
| shutil.rmtree(path[, ignore\_errors[, onerror]]) | 递归的去删除文件 | import shutil  shutil.rmtree('folder1') |
| shutil.move(src, dst) | 递归的去移动文件，它类似mv命令，其实就是重命名。 | import shutil  shutil.move('folder1', 'folder3') |

代码演示：

#注意！ 在其中fsrc，fdst都是文件对象，都需要打开后才能进行复制操作

>>> import shutil

>>> f1=open('name','r')

>>> f2=open('name\_copy','w+')

>>> shutil.copyfileobj(f1,f2,length=16\*1024)

#一句就可以实现复制文件内容

>>> shutil.copyfile('name','name\_copy\_2')

#仅复制权限

>>> shutil.copymode('test1','test2')

#递归复制文件内容及状态信息

>>> shutil.copytree('copytree\_test','copytree\_copy')

### 创建压缩解压

**make\_archive(base\_name, format, root\_dir=None, base\_dir=None, verbose=0,dry\_run=0, owner=None, group=None, logger=None)**

#### 参数注解：

\* base\_name： 压缩包的文件名，也可以是压缩包的路径。只是文件名时，则保存至当前目录，否则保存至指定路径，

\* 如 data\_bak =>保存至当前路径

\* 如：/tmp/data\_bak =>保存至/tmp/

\* format： 压缩包种类，“zip”, “tar”, “bztar”，“gztar”

\* root\_dir：要压缩的文件夹路径（默认当前目录）

\* base\_dir：要进行压缩的源文件或目录

\* owner： 用户，默认当前用户

\* group： 组，默认当前组

\* logger： 用于记录日志，通常是logging.Logger对象

演示：

#将 /data 下的文件打包放置当前程序目录

>>> import shutil

>>> ret = shutil.make\_archive("data\_bak", 'gztar', root\_dir='/data')

#将 /data下的文件打包放置 /tmp/目录

>>> import shutil

>>> ret = shutil.make\_archive("/tmp/data\_bak", 'gztar', root\_dir='/data')

### 注意： shutil 对压缩包的处理是调用 ZipFile 和 TarFile 两个模块来进行的!

#### zipfile压缩解压缩

import zipfile

# 压缩

z = zipfile.ZipFile('laxi.zip', 'w')

z.write('a.log')

z.write('data.data')

z.close()

# 解压

z = zipfile.ZipFile('laxi.zip', 'r')

z.extractall(path='.')

z.close()

tarfile压缩解压

import tarfile

# 压缩

>>> t=tarfile.open('/tmp/egon.tar','w')

>>> t.add('/test1/a.py',arcname='a.bak')

>>> t.add('/test1/b.py',arcname='b.bak')

>>> t.close()

# 解压

>>> t=tarfile.open('/tmp/egon.tar','r')

>>> t.extractall('/egon')

>>> t.close()

[返回主页](https://www.cnblogs.com/itbsl/)**124、**[**MySQL主从复制的几种复制方式**](https://www.cnblogs.com/itbsl/p/13507401.html)

目录

* [异步复制](https://www.cnblogs.com/itbsl/p/13507401.html#%E5%BC%82%E6%AD%A5%E5%A4%8D%E5%88%B6)
* [多线程复制](https://www.cnblogs.com/itbsl/p/13507401.html#%E5%A4%9A%E7%BA%BF%E7%A8%8B%E5%A4%8D%E5%88%B6)
* [增强半同步复制](https://www.cnblogs.com/itbsl/p/13507401.html#%E5%A2%9E%E5%BC%BA%E5%8D%8A%E5%90%8C%E6%AD%A5%E5%A4%8D%E5%88%B6)

### 异步复制

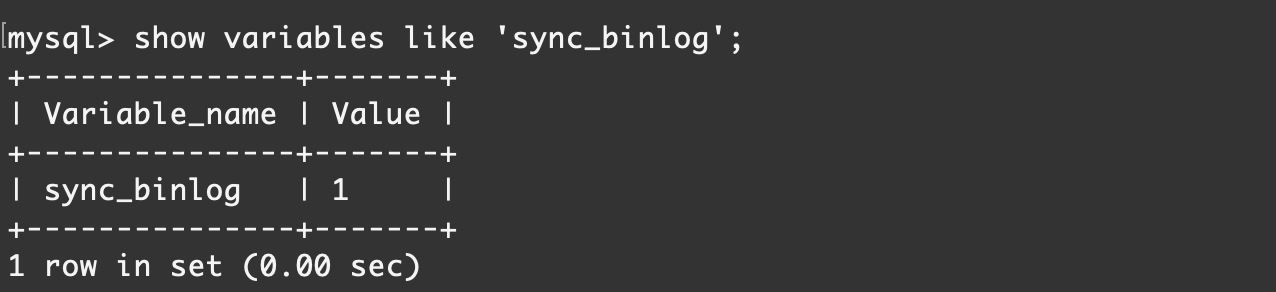
**MySQL的复制默认是异步的**，主从复制至少需要两个MYSQL服务，这些MySQL服务可以分布在不同的服务器上，也可以在同一台服务器上。

MySQL主从异步复制是最常见的复制场景。数据的完整性依赖于主库BINLOG的不丢失，只要主库的BINLOG不丢失，那么就算主库宕机了，我们还可以通过BINLOG把丢失的部分数据通过手工同步到从库上去。

**注意**：主库宕机的情况下，DBA可以通过mysqlbinlog工具手工访问主库binlog，抽取缺失的日志并同步到从库上去；也可以通过配置高可用MHA架构来自动抽取缺失的数据补全从库，或者启用Global Transaction Identifiers（GTID)来自动抽取缺失binlog到从库。

MySQL在BINLOG中记录事务(或SQL语句),也就是说对于支持事务的的引擎(例如InnoDB)来说，每个事务提交时都需要写BINLOG；对于不支持事务的引擎(例如MyISAM)来说，每个SQL语句执行完成时，都需要些BINLOG。为了保证Binlog的安全，MySQL引入sync\_binlog参数来控制BINLOG刷新到磁盘的频率。

show variables like 'sync\_binlog';



* 在默认情况下，sync\_binlog=1，表示事务提交之前，MySQL都需要先把BINLOG刷新到磁盘，这样的话，即使出现数据库主机操作系统崩溃或者主机突然掉电的情况，系统最多损失prepared状态的事务；设置sync\_binlog=1，尽可能保证数据安全。
* sync\_binlog=0,表示MySQL不控制binlog的刷新，由文件系统自己控制文件缓存的刷新。
* sync\_binlog=N,如果N不等于0或者1，刷新方式同sync\_binlog=1类似，只不过此时会延长刷新频率至N次binlog提交组之后。
* 以上是传统的异步复制，在MySQL5.7的并行复制技术(也称多线程复制)到来之前，为人诟病最多的还是效率问题，slave延迟是一个顽疾，虽然之前已经出现了schema级别的并行复制，但实际效果并不好。

### 多线程复制

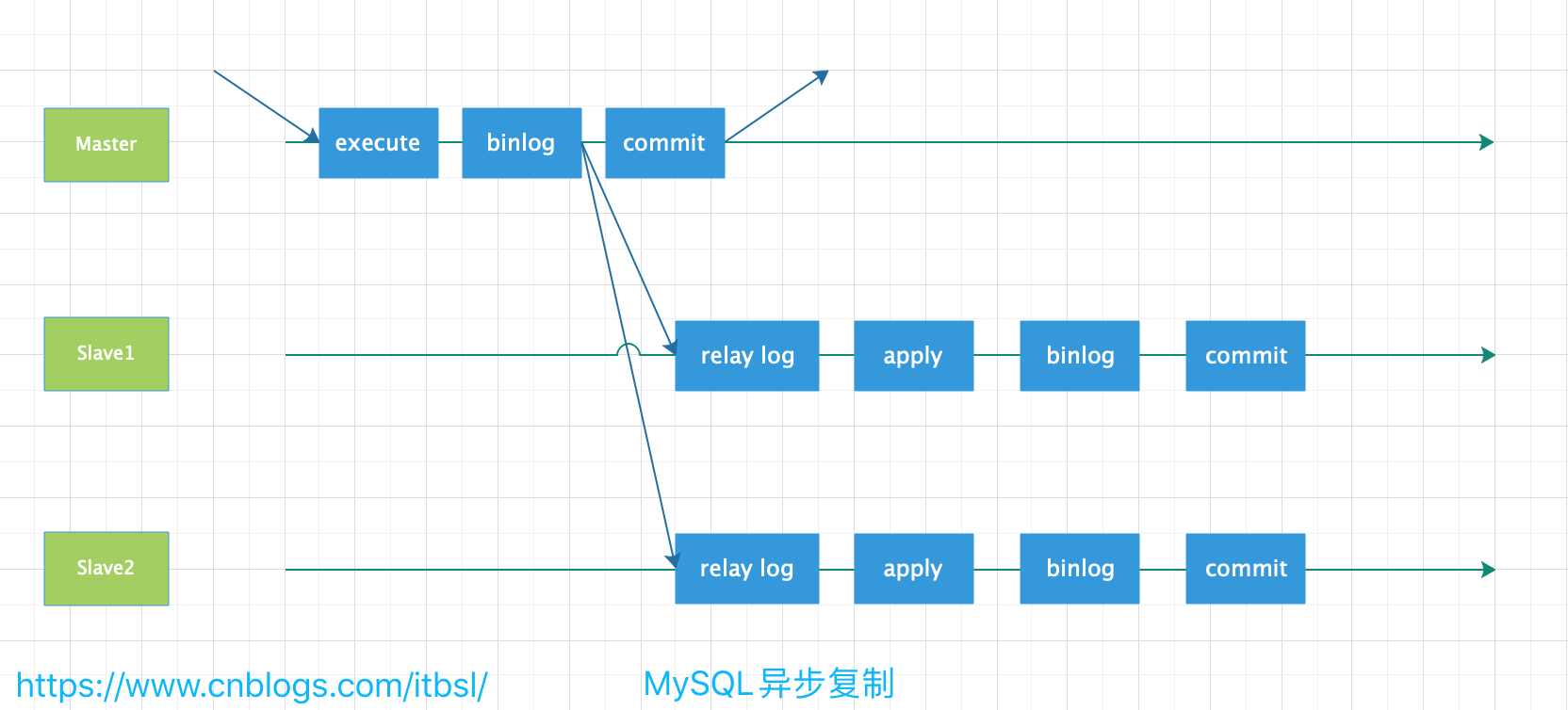
在MySQL5.7中，带来了全新的**多线程复制**技术，解决了当master同一个schema下的数据发生了变更，从库不能并发应用的问题，同时也真正将binlog组提交的优势充分发挥出来，保障了从库并发应用Relay Log的能力。

在MySQL8.0中，多线程复制又进行了技术更新，引入了writeset的概念，而在之前的版本中，如果主库的同一个会话顺序执行多个不同相关对象的事务，例如，先执行了Update A表的数据，又执行了Update B表的数据，那么BINLOG在复制到从库后，这两个事务是不能并行执行的，writeset的到来，突破了这个限制。

### 增强半同步复制

前面介绍的复制是异步操作，主库和从库的数据之间难免会存在一定的延迟，这样存在一个隐患：当在主库上写入一个事务并提交成功，而从库尚未得到主库的BINLOG日志时，主库由于磁盘损坏、内存故障、断电等原因意外宕机，导致主库上该事务BINLOG丢失，此时从库就会损失这个事务，从而造成主从不一致。

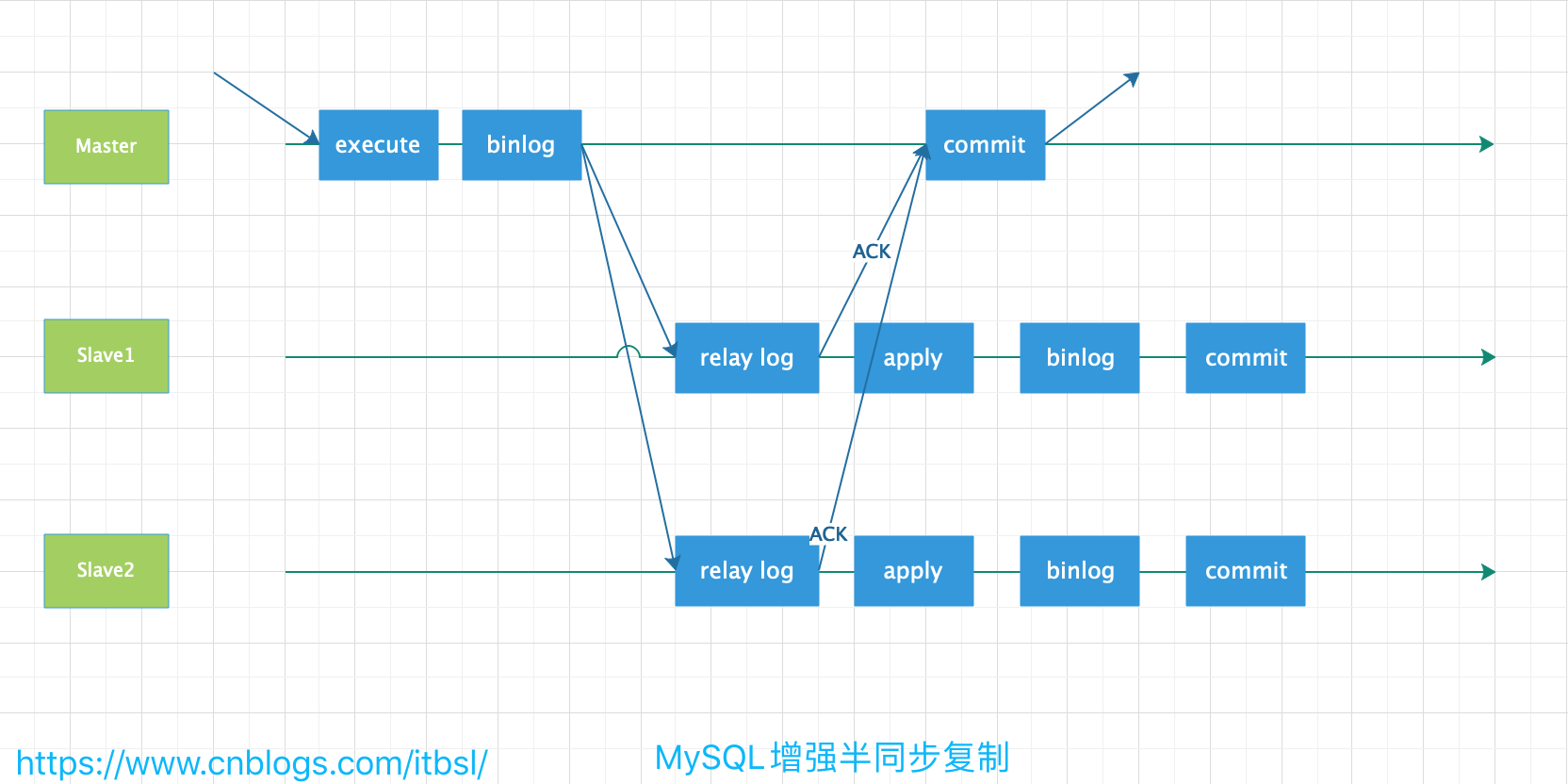
为了解决这个问题，从MySQL5.5开始，引入了半同步复制，此时的技术暂且称之为传统的半同步复制，因该技术发展到MySQL5.7后，已经演变为增强半同步复制(也成为无损复制)。在异步复制时，主库执行Commit提交操作并写入BINLOG日志后即可成功返回客户端，无需等待BINLOG日志传送给从库，如图所示。



而半同步复制时，为了保证主库上的每一个BINLOG事务都能够被可靠地复制到从库上，主库在每次事务成功提交时，并不及时反馈给前端应用用户，而是等待至少一个从库(详见参数rpl\_semi\_sync\_master\_wait\_for\_slave\_count)也接收到BINLOG事务并成功写入中继日志后，主库才返回Commit操作成功给客户端(不管是传统的半同步复制，还是增强的半同步复制，目的都是一样的，只不过两种方式有一个席位地方不同，将在下面说明)

半同步复制保证了事务成功提交后，至少有两份日志记录，一份在主库的BINLOG日志上，另一份在至少一个从库的中继日志Relay Log上，从而更进一步保证了数据的完整性。

在传统的半同步复制中，主库写数据到BINLOG，且执行Commit操作后，会一直等待从库的ACK，即从库写入Relay Log后，并将数据落盘，返回给主库消息，通知主库可以返回前端应用操作成功，这样会出现一个问题，就是实际上主库已经将该事务Commit到了事务引擎层，应用已经可以可以看到数据发生了变化，只是在等待返回而已，如果此时主库宕机，有可能从库还没能写入Relay Log，就会发生主从库不一致。增强半同步复制就是为了解决这个问题，做了微调，即主库写数据到BINLOG后，就开始等待从库的应答ACK，直到至少一个从库写入Relay Log后，并将数据落盘，然后返回给主库消息，通知主库可以执行Commit操作，然后主库开始提交到事务引擎层，应用此时可以看到数据发生了变化。增强半同步复制的大致流程如下图所示。



半同步复制模式下，假如在传送BINLOG日志到从库时，从库宕机或者网络延迟，导致BINLOG并没有即使地传送到从库上，此时主库上的事务会等待一段时间(时间长短由参数rpl\_semi\_sync\_master\_timeout设置的毫秒数决定)，如果BINLOG在这段时间内都无法成功发送到从库上，则MySQL自动调整复制为异步模式，事务正常返回提交结果给客户端。

半同步复制很大程度上取决于主从库之间的网络情况，往返时延RTT越小决定了从库的实时性越好。通俗地说，主从库之间的网络越快，从库约实时。

**注意**：往返时延RTT(Round-Trip Time)在计算机网络中是一个重要的性能指标，它表示从发送端发送数据开始到发送端接收到接收端的确认，总共经历的时长(这里可能有点拗口，我们可以理解为TCP三次握手的前两次握手)。

转载请注明原文链接：<https://www.cnblogs.com/itbsl/p/13507401.html>

如果该文章对您有帮助，请您点击**推荐**，感谢。

[返回主页](https://www.cnblogs.com/itbsl/)**125、**[**MySQL常用的4种主从复制架构**](https://www.cnblogs.com/itbsl/p/13508625.html)

目录

* [一主多从复制架构](https://www.cnblogs.com/itbsl/p/13508625.html#%E4%B8%80%E4%B8%BB%E5%A4%9A%E4%BB%8E%E5%A4%8D%E5%88%B6%E6%9E%B6%E6%9E%84)
* [多级复制架构](https://www.cnblogs.com/itbsl/p/13508625.html#%E5%A4%9A%E7%BA%A7%E5%A4%8D%E5%88%B6%E6%9E%B6%E6%9E%84)
* [双主(Dual Master)复制架构](https://www.cnblogs.com/itbsl/p/13508625.html#%E5%8F%8C%E4%B8%BBdual-master%E5%A4%8D%E5%88%B6%E6%9E%B6%E6%9E%84)
* [多源(Multi-Source)复制架构](https://www.cnblogs.com/itbsl/p/13508625.html#%E5%A4%9A%E6%BA%90multi-source%E5%A4%8D%E5%88%B6%E6%9E%B6%E6%9E%84)
* [如何优化主从延迟问题？](https://www.cnblogs.com/itbsl/p/13508625.html#%E5%A6%82%E4%BD%95%E4%BC%98%E5%8C%96%E4%B8%BB%E4%BB%8E%E5%BB%B6%E8%BF%9F%E9%97%AE%E9%A2%98%EF%BC%9F)

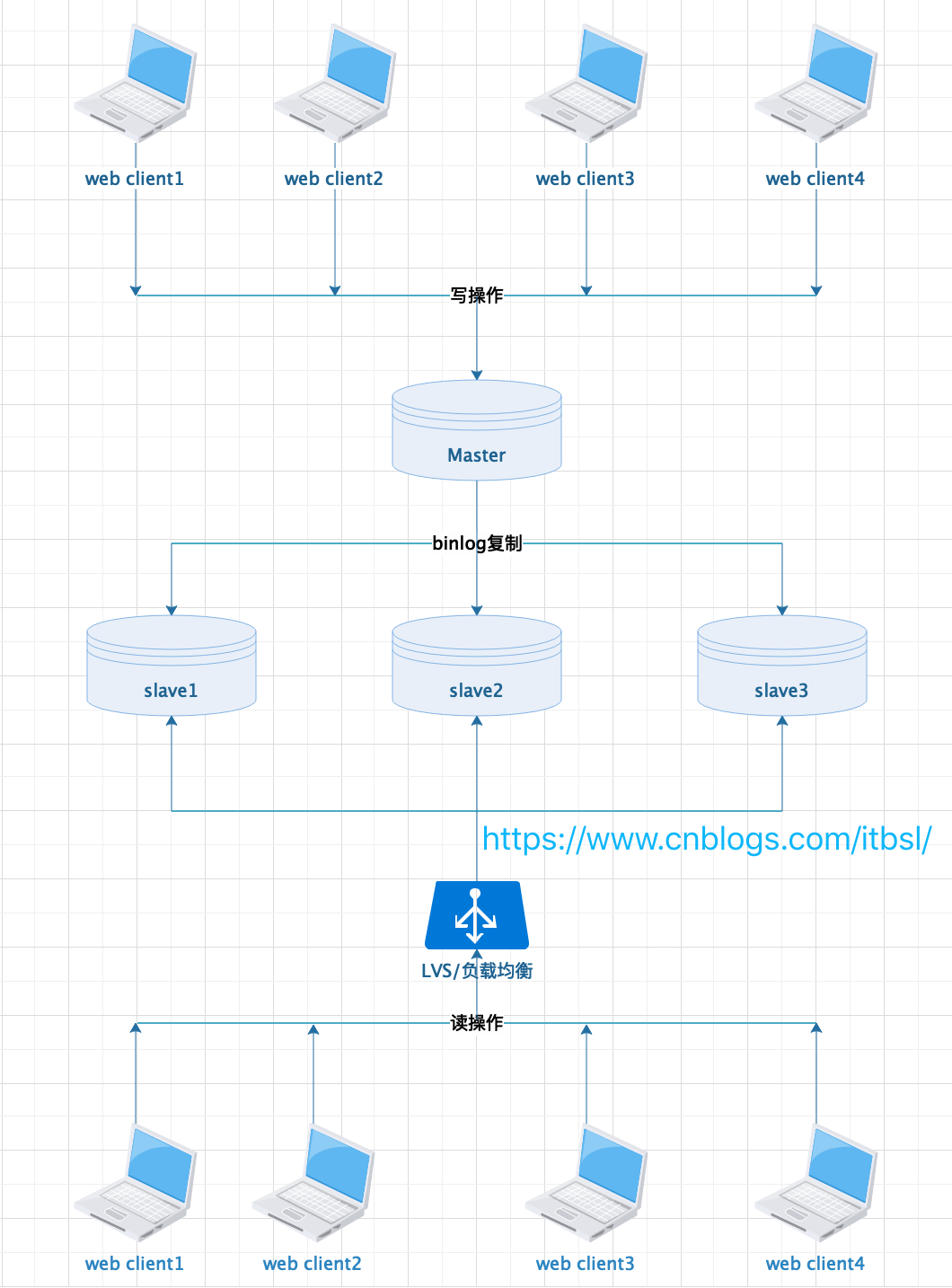
复制的4中常见架构有**一主多从复制架构**、**多级复制架构**、**双主(Dual Master)复制架构**和**多源(Multi-Source)复制架构**。

### 一主多从复制架构

在主库读取请求压力非常大的场景下，可以通过配置**一主多从复制架构**实现读写分离，把大量的对实时性要求不是特别高的读请求通过负载均衡分部到多个从库上(对于实时性要求很高的读请求可以让从主库去读)，降低主库的读取压力，如下图所示。

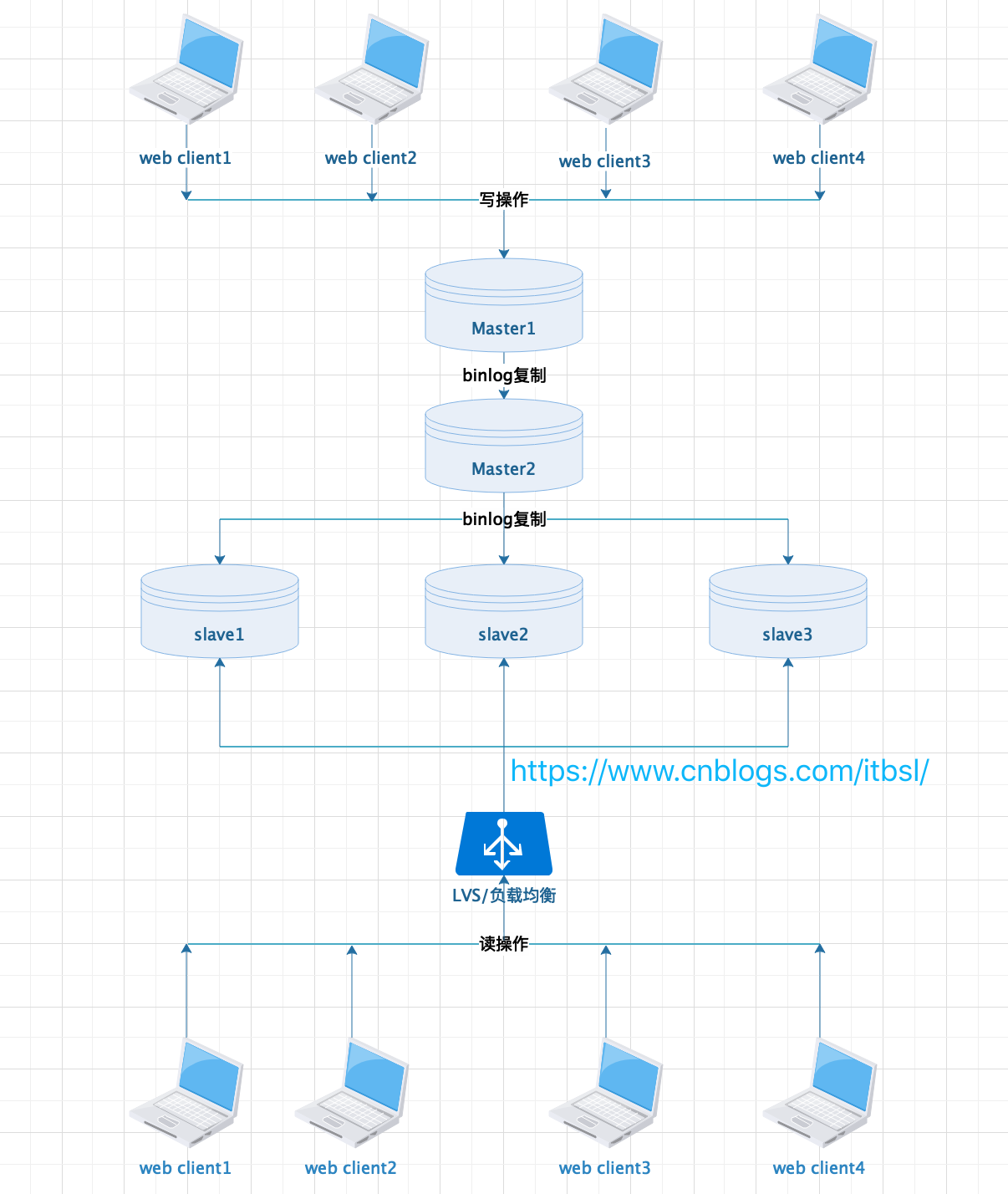
在主库出现异常宕机的情况下，可以把一个从库切换为主库继续提供服务。

在主从复制场景下会出现主从延迟，想想该怎么解决？



### 多级复制架构

一主多从的架构能够解决大部分读请求压力特别大的的场景的需求，考虑到MySQL的复制需要主库发送BINLOG日志到从库的I/O线程，主库的I/O压力和网络压力会随着从库的增加而增长(每个从库都会在主库上有一个独立的BINLOG Dump线程来发送事件)，而**多级复制架构**解决了一主多从场景下的，主库额外的I/O和网络压力。MySQL的多级复制架构如下图所示。



对比一主多从的架构，多级复制仅仅是在主库Master1复制到从库Slave1、Slave2、Slave3的中间增加了一个二级主库Master2，这样，主库Master1只需要给一个从库Master2发送BINLOG日志即可，减轻了主库Master1的压力。二级主库Master2再发送BINLOG日志给所有的从库Slave1、Slave2和Slave3的I/O线程。

多级复制解决了一主多从场景下，主库的I/O负载和网络压力，当然也有缺点：MySQL的传统复制是异步的，多级复制场景下主库的数据是经历两次复制才到达从库Slave1、Slave2、Slave3的，期间的延迟要比一主多从复制场景下只经历一次复制的还大。

**可以通过在二级主库Master2上选择表引擎为BLACKHOLE来降低多级复制的延迟。**顾名思义，BLACKHOLE引擎是一个“黑洞”引擎，写入BLACKHOLE表的数据并不会写会到磁盘上，BLACKHOLE表永远都是空表，INSERT、UPDATE、DELETE操作仅仅在BINLOG中记录事件。

CREATE TABLE `user` (

`id` int NOT NULL AUTO\_INCREMENT PRIMARY KEY,

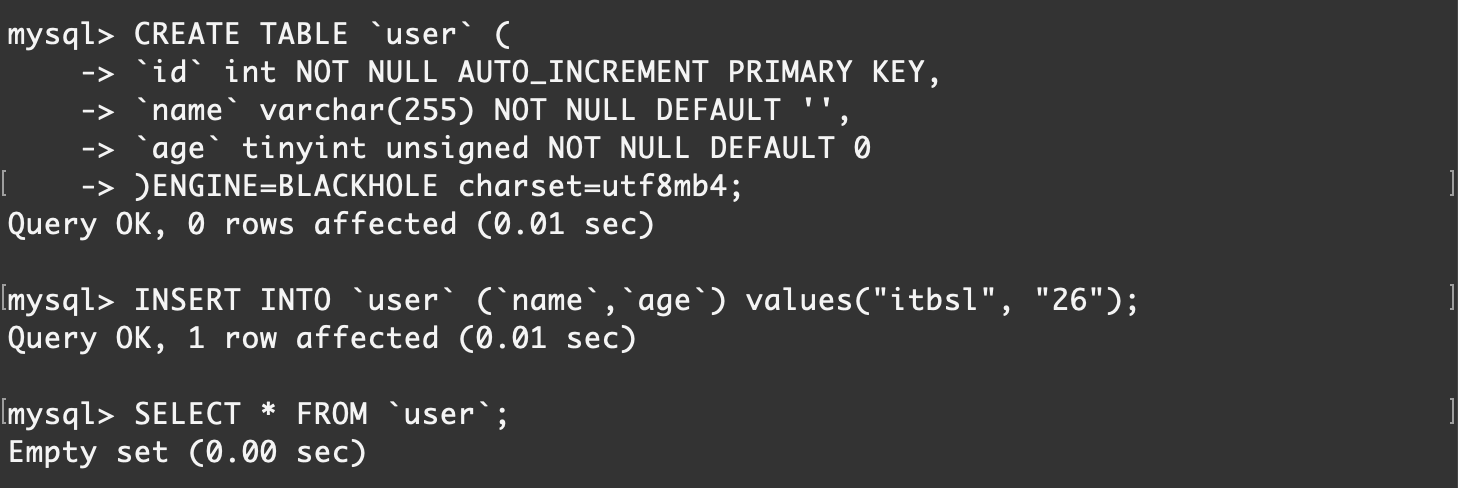
`name` varchar(255) NOT NULL DEFAULT '',

`age` tinyint unsigned NOT NULL DEFAULT 0

)ENGINE=BLACKHOLE charset=utf8mb4;

INSERT INTO `user` (`name`,`age`) values("itbsl", "26");

SELECT \* FROM `user`;

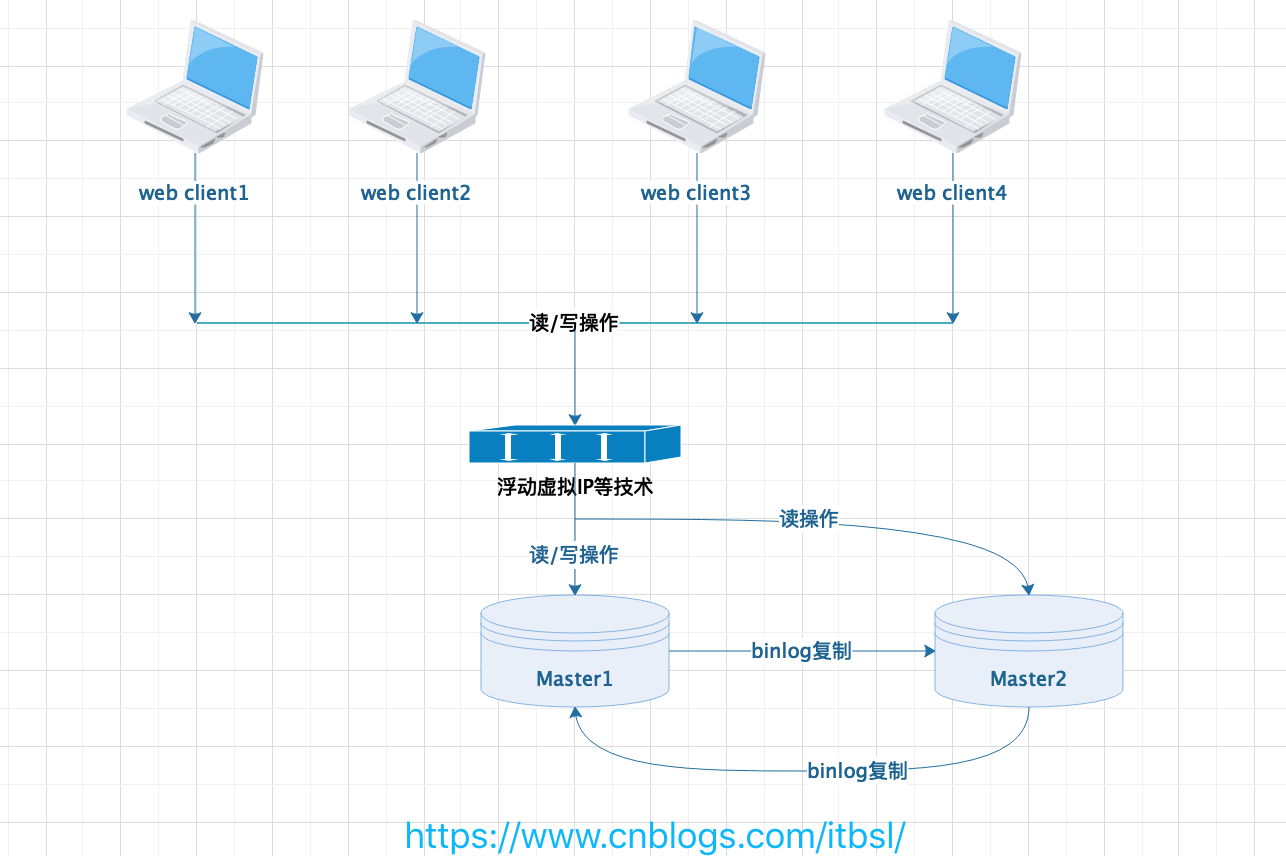


可以看到，存储引擎为BLACKHOLE的user表里没有数据。

BLACKHOLE引擎非常适合二级主库Masger2的场景:Master2并不承担读写请求，仅仅负责将BINLOG日志尽快传送给从库。

### 双主(Dual Master)复制架构

**双主(Dual Master)复制架构**适用于DBA做维护时需要主从切换的场景，通过双主复制架构避免了重复搭建从库的麻烦，双主复制架构如下图所示。



主库Master1和Master互为主从，所有Web Client的写请求都访问主库Master1或Master2。加入，DBA需要做日常维护操作，为了避免影响服务，需进行一下操作。

1、首先，在Master1库上停止Slave线程(STOP SLAVE)，避免后续对Master2库的维护操作操作被实时复制到Master1库上对服务造成影响。

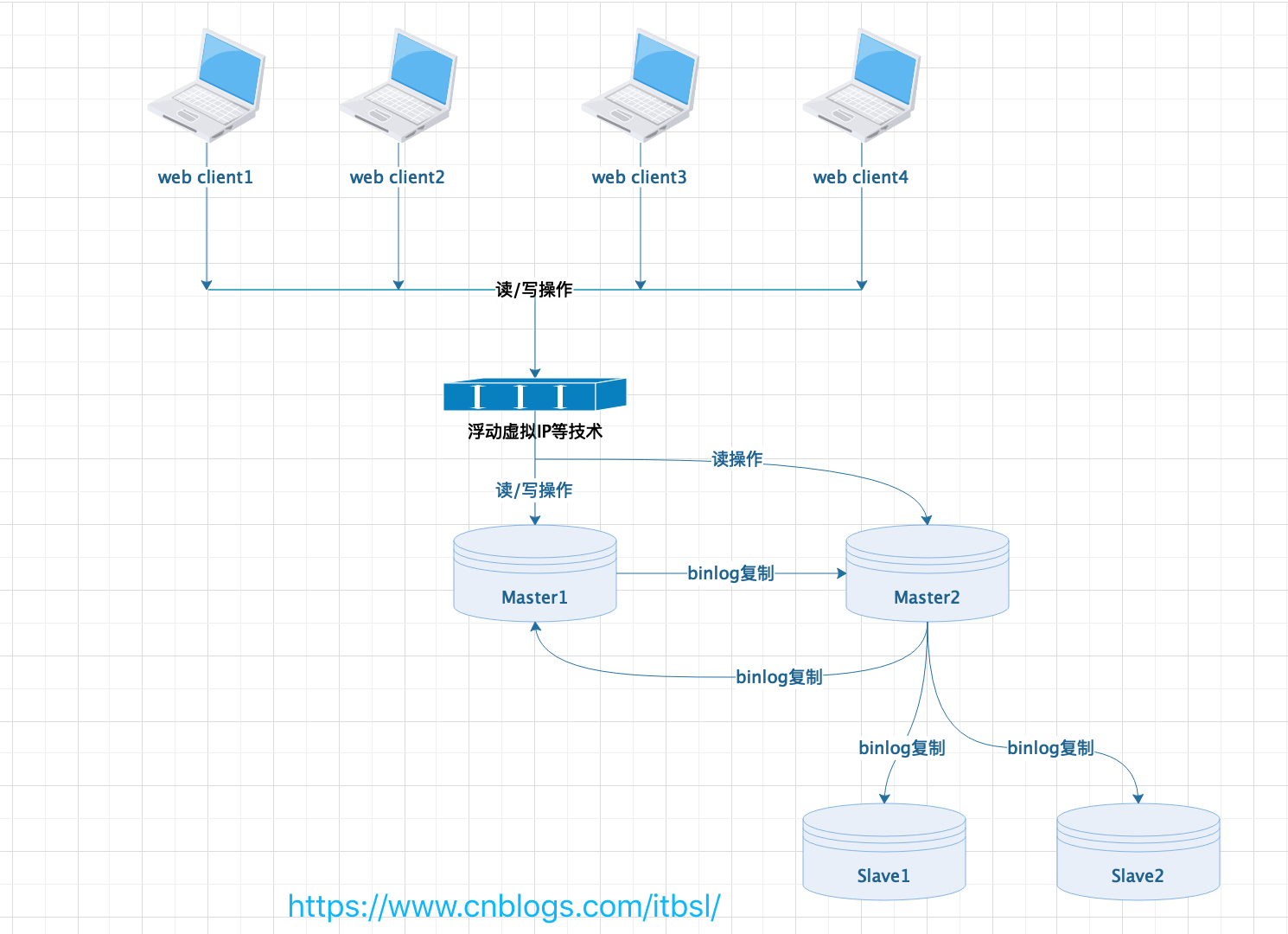
2、其次，在Master2库上停止Slave线程(STOP SLAVE)，开始日常维护操作，例如修改varchar字段从长度10增加到200。

3、然后，在Master2库上完成维护操作之后，打开Master2库上的Slave线程(STRART SLAVE),让Master2的数据和Master1库同步，同步完成后，把应用的写操作切换到Master2库上。

4、最后，确认Master1库上没有应用访问后，打开Master1的Slave线程(START SLAVE)即可。

通过双主复制架构能够大大减轻一主多从架构下对主库进行维护带来的额外搭建从库的工作。

当然双主架构还能和主从复制联合起来使用：在Master2库下配置从库Slave1、Slave2等，这样既可通过从库Slave1等来分担读取压力，同时在DBA做维护的同时，避免了重建从库的额外工作，但需要注意从库的复制延迟。MySQL双主多级复制架构如下所示。



### 多源(Multi-Source)复制架构

**多源(Multi-Source)复制架构**适用于复杂的业务需求，既可以支撑OLTP(联机事务处理),也可以满足OLAP(联机分析处理)。MySQL的多源复制架构我就暂时不画啦,等有空再画好补充上(画图也是个体力活呀)。有兴趣的可以看《深入浅出MySQL数据库开发、优化与管理维护》这本书。

### 如何优化主从延迟问题？

* 优化网络环境
* 在MySQL5.5~MySQL5.6中，使用半同步复制
* 在MySQL5.7中，可以使用增强半同步复制(也称为无损复制)

关于MySQL主从延迟的具体信息，可以看我的另一篇文章[聊聊MySQL主从复制的几种复制方式](https://www.cnblogs.com/itbsl/p/13507401.html)。

转载请注明链接：<https://www.cnblogs.com/itbsl/p/13508625.html>

**126、Ansible 自动化运维工具**

 Ansible是什么？

Ansible是一个“配置管理工具”也是一个“自动化运维工具”

Ansible 作用：

Ansible是新出现的自动化运维工具，基于Python开发，实现批量系统配置，批量程序部署，批量运行命令

### ****Ansible 简介：****

Ansible 是一个能实现批量部署的自动化运维工具，基于python开发，能实现批量系统配置，批量部署程序，批量运行命令。Ansible是基于模块来运行，它本身没有任何批量部署的能力，ansible主要是提供一种批量部署的框架。

Ansible的框架：

1、连接的插件connection plugins： 负责和被监控端的通信

2、Host inventory ： 指定服务器上配置文件定义监控的客户端

3、核心模块： command、shell、yum、copy、ping、rm、自定义模块(第三方模块)等

4、Playbook： 剧本任务，在非必要的情况下可以在一个节点执行多个任务

### ****Ansible 特性：****

 No agent 不需要在被管控主机上安装任何软件

No server 不用单独启用服务，能使用直接运行，使用时直接运行命令

对硬件资源占用小

不需要在被管控的主机上安装任何软件

基于模块工作

基于ssh工作

使用yaml语言

可实现多级指挥

### ****Asible的优势：****

1) 轻量级易操作

2) 分布式，无agent客户端

3) 配置语法是YAML语言

4) 轻量级，无需在客户端安装agent，更新时在操作机更新一次即可

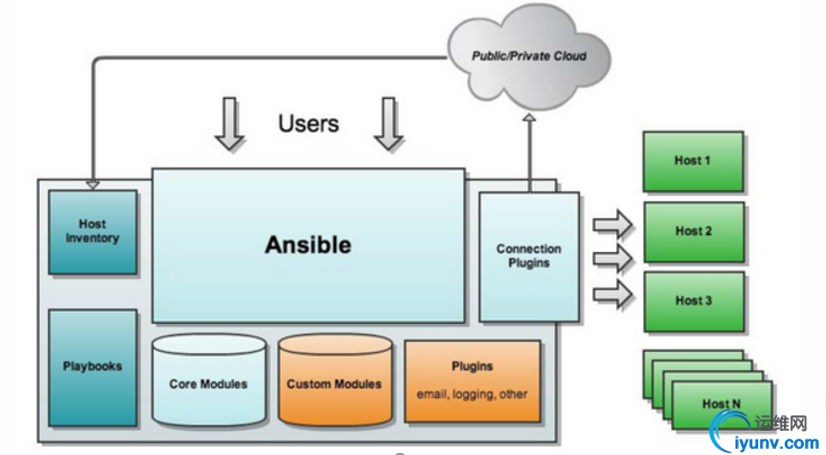
5) 批量任务执行可以写成脚本，不用发送给客户机就可执行

6) 使用python编写，维护简单

7) 支持sudo

### ****Ansible的架构：****

连接其他主机默认使用的ssh协议，也支持local、zeromq的方式连接



核心：ansible

核心模块（core modules）：ansible自带的模块

扩展模块（custom modules）：核心模块不能满足需求，添加扩展模块。

插件（plugins）：完成模块的补充

剧本（playbook）：在ansible配置文件中，可以将多个任务定义在一个剧本中，在ansibel执行剧本。

连接插件（connectior plugins）：ansible默认使用ssh连接的，但是ansible也支持其他的连接方式，其他的连接方式是需要插件的支持，才可以实现通信。

主机群（host inventory）： 定义ansible管理的主机的

### ****Ansible生成的主要文件****

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | /etc/ansible  /etc/ansible/ansible.cfg   #配置文件  /etc/ansible/hosts   #主机库（host inventory）管理被监控的主机  /usr/bin/ansible   #主程序  /usr/bin/ansible-doc   #文档  /usr/bin/ansible-playbook   #剧本 |

### ****Ansible的工作原理：****

1、服务端连接客户端，实现通信

2、通过服务器的节点使用模块实现客户端的需求

3、通过剧本实现多个任务的实现

#### ****Ansible 的7个命令：****

Ansible、Ansible-doc、Ansible-galaxy、Ansible-lint、Ansible-playbook、Ansible-pull、Ansible-vault：

##### ****Ansible：****

Ansible是核心的指令，主要用于执行单条命令，默认跟的是主机和选项部分，默认不指定模块时，使用的是command模块（默认使用的模块可以在ansible.cfg修改）：

Eg：[root@44 ~]# ansible all -a "pwd"

192.168.20.70 | SUCCESS | rc=0 >>

/root

参数：

-a 'Arguments', --args='Arguments' 命令行参数

-m NAME, --module-name=NAME 执行模块的名字，默认使用 command 模块，所以如果是只执行单一命令可以不用 -m参数

-i PATH, --inventory=PATH 指定库存主机文件的路径,默认为/etc/ansible/hosts.

-u Username， --user=Username 执行用户，使用这个远程用户名而不是当前用户

-U --sud-user=SUDO\_User sudo到哪个用户，默认为 root

-k --ask-pass 登录密码，提示输入SSH密码而不是假设基于密钥的验证

-K --ask-sudo-pass 提示密码使用sudo

-s --sudo sudo运行

-S --su 用 su 命令

-l --list 显示所支持的所有模块

-s --snippet 指定模块显示剧本片段

-f --forks=NUM 并行任务数。NUM被指定为一个整数,默认是5。 #ansible testhosts -a "/sbin/reboot" -f 10 重启testhosts组的所有机器，每次重启10台

--private-key=PRIVATE\_KEY\_FILE 私钥路径，使用这个文件来验证连接

-v --verbose 详细信息

all 针对hosts 定义的所有主机执行

-M MODULE\_PATH, --module-path=MODULE\_PATH 要执行的模块的路径，默认为/usr/share/ansible/

--list-hosts 只打印有哪些主机会执行这个 playbook 文件，不是实际执行该 playbook 文件

-o --one-line 压缩输出，摘要输出.尝试一切都在一行上输出。

-t Directory, --tree=Directory 将内容保存在该输出目录,结果保存在一个文件中在每台主机上。

-B 后台运行超时时间

-P 调查后台程序时间

-T Seconds, --timeout=Seconds 时间，单位秒s

-P NUM, --poll=NUM 调查背景工作每隔数秒。需要- b

-c Connection, --connection=Connection 连接类型使用。可能的选项是paramiko(SSH),SSH和地方。当地主要是用于crontab或启动。。。。。

##### ****Ansible-doc：****

Ansible-doc -h 用于查看模块的信息常见的参数-l  -s

Ansible-doc -l  列出已安装的模块

Ansible-doc -s 模块 ： 查查看具体模块的使用方法

##### ****Ansible-galaxy：****

Ansible-galaxy 从站点下载第三方的扩展模块

[root@44 ~]# ansible-galaxy -h

Usage: ansible-galaxy [delete|import|info|init|install|list|login|remove|search|setup] [--help] [options] ...

##### ****Ansible-lint：****

Ansible-lint对playbook的语法进行检查的工具

Ansible-lint 剧本名

注：本人安装的ansible 没有此命令选项所以用ansible-playbook --syntax-check 对剧本进行语法的检查

[root@44 ~]# ansible-playbook --syntax-check  + 剧本名

对剧本语法进行检查但不执行

##### ****Ansible-playbook：****

Ansible-playbook 读取playbook文件执行相应的动作。

执行ansible-playbook的剧本

eg：执行a.yml剧本

[root@44 ~]# ansible-playbook a.yml

##### ****Ansible-pull：****

Ansible-pull 适用于数量多的机器配置，远程执行命令的工具。对运维架构能力比较强

##### ****Ansible-vault：****

Ansible-vault 配置文件有敏感信息，设置加密/解密这个配置文件

## ****常用模块使用****

Ansible语法：ansible 机组名 -m 模块 -a 执行的内容

#### ****1)   Setup模块****

用来查看远程主机的一些基本信息

Ansible 用户 -m setup

#### ****2)        Ping模块****

## 用来测试远程主机的运行状态

Ansible  用户  -m ping

#### ****3)        Copy模块****

复制文件到远程主机

示例：将本地文件“/etc/ansible/ansible.cfg”复制到远程服务器

Ansibles  -m copy -a "src=/etc/ansible/ansible.cfgdest=/tmp/ansible.cfg owner=root group=root mode=0644"

#### ****4)        Command模块****

## 在远程主机上执行命令

相关选项如下：

ansible storm\_cluster-m command -a "uptime"

#### ****5)        Shell模块****

 ## 切换到某个shell执行指定的指令，参数与command相同。

 这个模块与script不同的是，首先要将写好的sh脚本分发到机组，再执行

 与command不同的是，此模块可以支持命令管道，同时还有另一个模块也具备此功能：raw。

示例：

先在本地创建一个SHELL脚本

 vim/tmp/rocketzhang\_test.sh

 将创建的脚本文件分发到远程

ansiblestorm\_cluster -m copy -a "src=/tmp/rocketzhang\_test.shdest=/tmp/rocketzhang\_test.sh owner=root group=root mode=0755"

远程执行

 # ansiblestorm\_cluster -m shell -a "/tmp/rocketzhang\_test.sh"

#### ****6)        Script模块****

 用script：模块可以实现到对象节点上执行本机脚本。有点类似copy+shell+删除copy的脚本的这样一个综合的功能。

 ansible host32-m script -a /tmp/hello.sh

#### ****7)        更多模块****

其他常用模块，比如：service、cron、yum、synchronize就不一一例举，可以结合自身的系统环境进行测试。

service：系统服务管理

cron：计划任务管理

yum：yum软件包安装管理

synchronize：使用rsync同步文件

user：系统用户管理

group：系统用户组管理

 File ：  对文件目录的创建、删除

Path 参数：用来指定要操作的文件或目录

 State 参数： 设定操作的需求

State=directory   创建目录

State=touch     创建文件

State=link       创建硬连接

State=absent     缺席、删除

## ****Ansible-playbook  剧本****

 Ansible-playbook的作用：

 为什么使用ansible-playbook哪？当批量部署的过程中有些监控器需要做出改动的时候，我们不需要通过ansible命令去手动去改动，那样太过于消耗时间，我们可以把它写成脚本，有改动的时候，只需要修改定义下的变量就好。Ansible中的脚本功能就是ansible-playbook剧本。

### ****Ansible-playbook 的使用方法：****

Ansible-playbook的组织格式：YAML语言格式

YAML的简介：

类似于半结构化数据，声明式配置，可读性较高用来表达资料序列的格式，容易与脚本语言交互

YAML的语法格式：

①　任何书记结构用缩进来标识，可以嵌套

 ②　每一行的一个键值key：value ，冒号隔开。若想在一行标识用{}和，分隔

③　列表用-标识

 Playbook 组成的结构：

Host： 被监控的主机

 Tasks ：任务，调用的模块执行操作

Variables： 变量

Templates: 模板

Handlers ：处理器

Roles ：角色

基本结构：

   – host:webservices

            remote\_user:

                – tasks:

                – task1

            module\_name

        YAML文件扩展名通常为.yml，如test.yml。

Eg：

[root@44 ~]# cat a.yml

- hosts: 192.16.20.70

  remote\_user: root

  tasks:

  - name: add group

    group: name=new state=present

   - name: add user

    user: name=yxyfyy group=new

- hosts: 192.168.20.70

  remote\_user: root

  tasks:

  - name: copy

    copy: src=/etc/services dest=/tmp/ owner=nobody group=new

 [root@44 ~]# ansible-playbook --syntax-check a.yml

  [WARNING]: Could not match supplied host pattern, ignoring: 192.16.20.70

playbook: a.yml

编写playbook 剧本（执行拷贝的模板，把主机上的1111.txt文件拷贝到被监控的主机上。）

 [root@44 ~]# cat a.yml

 - hosts: 192.168.20.70

  remote\_user: root

   tasks:

   - name: copy

    copy: src=/root/1111.txt dest=/tmp/ owner=nobody group=root

对playbook剧本进行检测有没有错误

 [root@44 ~]# ansible-playbook --syntax-check a.yml

playbook: a.yml

执行playbook剧本

 [root@44 ~]# ansible-playbook a.yml

 PLAY [192.168.20.70] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 ok: [192.168.20.70]

TASK [copy] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [192.168.20.70]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

192.168.20.70              : ok=2    changed=1    unreachable=0    failed=0

检查结果

[root@44 ~]# cat 1111.txt

ansible

[root@yk3 ~]# cat /tmp/1111.txt

ansible

Ansible-playbook的组成：

Ansible-playbook可以理解为把ansible的使用的模块方法写入ansible-playbook剧本中。

Ansible-playbook文件是以“.yaml”或“.yml”结尾的文件

EG:

[root@44 ~]# cat a.yml

- hosts: 192.168.20.70 （指定被操作主机）

（yaml语法中使用冒号映射键值对，“冒号”后必须加空格）

  remote\_user: root （指定远程操作时使用那个用户）

（注意格式：yaml语法用层级关系，同一层级的是对齐的。（开头对齐））

  tasks: （操作的任务列表，可以有多个任务）

  - name: copy （自定义任务名称）

copy: src=/root/1111.txt dest=/tmp/ owner=nobody group=root

（任务的操作，copy：copy模块，src：源地址，（copy的内置条件） dest：目的地址， owner：所属人  group：所属组）

Hosts：定义要操作的主机，可以是IP地址、主机名（提前域名解析）、定义的分组。  定义多个主机要用，分隔开

## ****Ansible-playbook  剧本****

### ****Ansible-playbook的作用：****

为什么使用ansible-playbook哪？当批量部署的过程中有些监控器需要做出改动的时候，我们不需要通过ansible命令去手动去改动，那样太过于消耗时间，我们可以把它写成脚本，有改动的时候，只需要修改定义下的变量就好。Ansible中的脚本功能就是ansible-playbook剧本。

### ****Ansible-playbook 的使用方法：****

Ansible-playbook的组织格式：YAML语言格式

YAML的简介：

类似于半结构化数据，声明式配置，可读性较高用来表达资料序列的格式，容易与脚本语言交互

YAML的语法格式：

①　任何书记结构用缩进来标识，可以嵌套

②　每一行的一个键值key：value ，冒号隔开。若想在一行标识用{}和，分隔

③　列表用-标识

Playbook 组成的结构：

Host： 被监控的主机

Tasks ：任务，调用的模块执行操作

Variables： 变量

Templates: 模板

Handlers ：处理器

Roles ：角色

基本结构：

  – host:webservices

            remote\_user:

                – tasks:

                – task1

            module\_name

        YAML文件扩展名通常为.yml，如test.yml。

Eg：

[root@44 ~]# cat a.yml

- hosts: 192.16.20.70

  remote\_user: root

  tasks:

  - name: add group

    group: name=new state=present

  - name: add user

    user: name=yxyfyy group=new

- hosts: 192.168.20.70

  remote\_user: root

  tasks:

  - name: copy

    copy: src=/etc/services dest=/tmp/ owner=nobody group=new

 [root@44 ~]# ansible-playbook --syntax-check a.yml

 [WARNING]: Could not match supplied host pattern, ignoring: 192.16.20.70

playbook: a.yml

编写playbook 剧本（执行拷贝的模板，把主机上的1111.txt文件拷贝到被监控的主机上。）

[root@44 ~]# cat a.yml

- hosts: 192.168.20.70

  remote\_user: root

  tasks:

  - name: copy

    copy: src=/root/1111.txt dest=/tmp/ owner=nobody group=root

对playbook剧本进行检测有没有错误

[root@44 ~]# ansible-playbook --syntax-check a.yml

playbook: a.yml

执行playbook剧本

[root@44 ~]# ansible-playbook a.yml

PLAY [192.168.20.70] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [192.168.20.70]

TASK [copy] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [192.168.20.70]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

192.168.20.70              : ok=2    changed=1    unreachable=0    failed=0

检查结果

[root@44 ~]# cat 1111.txt

ansible

[root@yk3 ~]# cat /tmp/1111.txt

ansible

Ansible-playbook的组成：

Ansible-playbook可以理解为把ansible的使用的模块方法写入ansible-playbook剧本中。

Ansible-playbook文件是以“.yaml”或“.yml”结尾的文件

EG:

[root@44 ~]# cat a.yml

- hosts: 192.168.20.70 （指定被操作主机）

（yaml语法中使用冒号映射键值对，“冒号”后必须加空格）

  remote\_user: root （指定远程操作时使用那个用户）

（注意格式：yaml语法用层级关系，同一层级的是对齐的。（开头对齐））

  tasks: （操作的任务列表，可以有多个任务）

  - name: copy （自定义任务名称）

copy: src=/root/1111.txt dest=/tmp/ owner=nobody group=root

（任务的操作，copy：copy模块，src：源地址，（copy的内置条件） dest：目的地址， owner：所属人  group：所属组）

Hosts：定义要操作的主机，可以是IP地址、主机名（提前域名解析）、定义的分组。  定义多个主机要用，分隔开

介绍项目过程中，测试完之后是由运维发布，测试让运维发版

有什么能力竞争这个岗位

如何学习一门新技术，学习方式,举一个类子

接触自动化测试

逻辑能力和算法能力

如果一个评估项目，需要一个月的时间，用什么心态和方法来圆满完成。

讲一讲项目组之间如果出现了进度紧张，应该怎么处理

做web项目不懂的，如何让一个不认识的同事来帮你，如何请教他

如何和前端有冲突，比如一个方法的争议，如何处理。

自己的缺点和优点

简单一句话介绍自己的工作上和生活。讲明自己是一个什么样的人

讲一下自己的俩年到五年的计划

应该做哪些努力来提升自己