## 面试知识点简单汇总（自测）

### 招聘网站的一般要求汇总（面试造火箭系列）：

具备开发高并发引擎能力  
熟悉HTTP，TCP/IP等常用协议原理  
熟悉Docker相关理念及核心技术  
精通SQL和NoSQL数据库体系架构设计及高可用性解决方案；  
熟悉mysql，redis，mongo等常用数据库，具有数据库开发和设计能力；  
熟悉linux系统，会熟练编写多进程多线程的程序，熟悉网络编程；  
熟悉python异步IO  
熟练使用Linux系统，掌握基本命令，可编写简单的shell脚本  
了解异步框架、集群与负载均衡，消息中间件，容灾备份等技术；  
完成系统 API 接口等开发工作；

### Python知识点:

写一个简单的python socket编程  
python2.x 与python3.x的主要区别  
python新式类和旧式类的区别  
请简述线程＼进程＼协程的特性  
什么是闭包?  
Python的实例方法,类方法,静态方法之间的区别及调用关系  
Python内存管理和垃圾回收机制  
Python的装饰器内部实现原理  
描述你知道的设计模式及各模式特点  
常用算法(冒泡,二叉树,快排, 堆排序等)  
线程池的原理和实现  
解释一下 Django 和 Tornado 的关系、差别  
解释下Http协议的特点  
model之F/Q操作, 写一个Q示例  
simple\_tag用法  
写一个字典推导式  
Python中单下划线和双下划线  
什么是迭代器  
分别用\_\_new\_\_方法和装饰器实现单例模式  
简述Python的作用域以及Python搜索变量的顺序  
GIL线程全局锁  
Python-copy()与deepcopy()区别  
select,poll和epoll  
可变与不可变类型；  
浅拷贝与深拷贝的实现方式、区别；deepcopy如果你来设计，如何实现；  
**new**() 与 **init**()的区别；  
你知道几种设计模式；  
编码和解码你了解过么；  
列表推导list comprehension和生成器的优劣；  
什么是装饰器；如果想在函数之后进行装饰，应该怎么做；  
手写个使用装饰器实现的单例模式；  
使用装饰器的单例和使用其他方法的单例，在后续使用中，有何区别；  
手写：正则邮箱地址；  
介绍下垃圾回收：引用计数/分代回收/孤立引用环；  
多进程与多线程的区别；CPU密集型适合用什么；  
进程通信的方式有几种；  
介绍下协程，为何比线程还快；  
range和xrange的区别（他妹的我学的py3…）；  
将IP地址字符串（比如“172.0.0.1”）转为32位二进制数的函数。

### 算法排序部分

手写快排；堆排；几种常用排序的算法复杂度是多少；快排平均复杂度多少，最坏情况如何优化；  
手写：已知一个长度n的无序列表，元素均是数字，要求把所有间隔为d的组合找出来，你写的解法算法复杂度多少；  
手写：一个列表A=[A1，A2，…,An]，要求把列表中所有的组合情况打印出来；  
手写：用一行python写出1+2+3+…+10\*\*8；  
手写python：用递归的方式判断字符串是否为回文；  
单向链表长度未知，如何判断其中是否有环；  
单向链表如何使用快速排序算法进行排序；  
手写：一个长度n的无序数字元素列表，如何求中位数，如何尽快的估算中位数，你的算法复杂度是多少；  
如何遍历一个内部未知的文件夹（两种树的优先遍历方式）

### 网络基础部分

TCP/IP分别在模型的哪一层；  
socket长连接是什么意思；  
select和epoll你了解么，区别在哪；  
TCP UDP区别；三次握手四次挥手讲一下；  
TIME\_WAIT过多是因为什么；  
http一次连接的全过程：你来说下从用户发起request——到用户接收到response；  
http连接方式。get和post的区别，你还了解其他的方式么；  
restful你知道么；  
状态码你知道多少，比如200/403/404/504等等；

### 数据库部分

MySQL锁有几种；死锁是怎么产生的；  
为何，以及如何分区、分表；  
MySQL的char varchar text的区别；  
了解join么，有几种，有何区别，A LEFT JOIN B，查询的结果中，B没有的那部分是如何显示的（NULL）；  
索引类型有几种，BTree索引和hash索引的区别（我没答上来这俩在磁盘结构上的区别）；  
手写：如何对查询命令进行优化；  
NoSQL了解么，和关系数据库的区别；redis有几种常用存储类型；  
说说慢查询  
说说数据库的优化

### django项目部分

都是让简单的介绍下你在公司的项目，不管是不是后端相关的，主要是要体现出你干了什么；  
你在项目中遇到最难的部分是什么，你是怎么解决的；  
你看过django的admin源码么；看过flask的源码么；你如何理解开源；  
MVC / MTV；  
缓存怎么用；  
中间件是干嘛的；  
CSRF是什么，django是如何避免的；XSS呢；  
如果你来设计login，简单的说一下思路；  
session和cookie的联系与区别；session为什么说是安全的；  
uWSGI和Nginx的作用；

扩展阅读: [关于Python的面试题](https://github.com/taizilongxu/interview_python)

## Python基础知识

### 将一个字符串逆序，不能使用反转函数

str = "abcdegfgijlk"   
str[::-1]   
'kljigfgedcba'

### 求从10到100中能被3或5整除的数的和

l = []   
for i in range(10, 100):   
 if i % 3 == 0 or i % 5 == 0:   
 l.append(i)   
print(l, sum(l))

### 什么是Python

Python是一种解释型语言, 动态语言, python中一切都是对象, python目前在大数据/人工智能/机器学习/自动化应用十分广泛

a=1, b=2, 不用中间变量交换a和b的值

a,b = b, a

### 请用自己的算法, 按升序合并如下两个list, 并去除重复的元素

list1 = [2, 3, 8, 4, 9, 5, 6]   
list2 = [5, 6, 10, 17, 11, 2]   
s = sorted(set(list1 + list2))   
print(s)+

### 请描述set的用途并举例说明

去重, 计算交并叉集

### python2.x 与python3.x的主要区别

编码/数据类型 3没有long 增加了bytes xrange / dict的.keys()、.items 和.values()方法返回迭代器 / 3有抽象基类 / 异常

### Python里面search()和match()的区别

match()函数只检测RE是不是在string的开始位置匹配, search()会扫描整个string查找匹配

### 用Python匹配HTML tag的时候，<.\*>和<.\*?>有什么区别

贪婪和非贪婪 \*号是一个量词 量词后面加? 号表示 非贪婪,也就是尽可能少的匹配

### 什么是闭包?

简单说,闭包就是根据不同的配置信息得到不同的结果, 装饰器就是一种闭包, 闭包有效的减少了函数所需定义的参数数目。

def line\_conf(a, b):   
 def line(x):   
 return a\*x + b   
 return line   
  
 line1 = line\_conf(1, 1)   
line2 = line\_conf(4, 5)   
print(line1(5), line2(5))   
# (6, 25)

例子中, 如果没有闭包，我们需要每次创建直线函数的时候同时说明a,b,x。这样，我们就需要更多的参数传递，也减少了代码的可移植性。利用闭包，我们实际上创建了泛函。  
[python闭包的理解说明](https://www.cnblogs.com/lirunzhou/p/5881812.html)

### C++/C/JAVA/Python之间的区别？

* python： 快速开发应用程序
* java： 健壮的大型软件
* C++： 需求效率的软件
* C： 操作系统及驱动

### 内存管理和垃圾回收机制

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）以空间换时间的方法提高垃圾回收效率。

**1. 内存管理&引用计数**  
python创建新对象都是在内存上开辟一个块, 每个对象只存有一份数据, 赋值和复制都是创建了新的引用, 使用的是对象和引用分离策略  
在Python中，每个对象都有存有指向该对象的引用总数，即引用计数, 如果引用计数为0, 那这个对象就会被python垃圾回收机制回收

**2. 标记-清除机制**  
基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。  
同时为了保证效率, Python只会在垃圾达到一定阈值时，垃圾回收才会启动。

**3. 分代回收策略**  
这一策略的基本假设是，存活时间越久的对象，越不可能在后面的程序中变成垃圾。Python默认定义了三代对象集合，索引数越大，对象存活时间越长。

### 动态语言和静态语言的区别

动态语言（弱类型语言）是一类在运行时可以改变其结构的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化  
比如一个类中只定义了一个对象的名字和性别, 可以动态为其加入年龄属性  
静态语言（强类型语言）是在编译时变量的数据类型即可确定的语言，多数静态类型语言要求在使用变量之前必须声明数据类型  
强类型语言是一旦变量的类型被确定，就不能转化的语言。  
弱类型语言则反之，一个变量的类型是由其应用上下文确定的。  
静态语言的优势  
由于类型的强制声明，使得IDE有很强的代码感知能力，故在实现复杂的业务逻辑、大型系统, 错误更容易被发现, 调试难度相对低；  
由于静态语言相对比较封闭，使得第三方开发包对代码的侵害性可以降到最低；  
动态语言的优势  
思维不受束缚，可以任意发挥，把更多的精力放在产品本身上；  
集中思考业务逻辑实现，思考过程即实现过程；但代码中的坑可能不被IDE发现, 后期调试相对费力

### 字典推导式

d = {key: value for (key, value) in iterable}

### Python中单下划线和双下划线

\_\_foo\_\_:一种约定,Python内部的名字,用来区别其他用户自定义的命名，以防冲突。  
\_foo:一种约定,用来指定变量私有，程序员用来指定私有变量的一种方式  
\_\_foo:这个有真正的意义:解析器用\_\_classname、\_\_foo来代替这个名字,以区别和其他类相同的命名。  
### 迭代器协议  
1.迭代器协议是指：对象必须提供一个next方法，执行该方法要么返回迭代中的下一项，要么就引起一个StopIteration异常，以终止迭代 （只能往后走不能往前退）  
2.可迭代对象：实现了迭代器协议的对象（如何实现：对象内部定义一个\_\_iter\_\_()方法）  
3.协议是一种约定，可迭代对象实现了迭代器协议，python的内部工具（如for循环，sum，min，max函数等)使用迭代器协议访问对象。

for 循环机制  
for循环的本质：循环所有对象，全都是使用迭代器协议。  
for循环就是基于迭代器协议提供了一个统一的可以遍历所有对象的方法，即在遍历之前，先调用对象的\_\_iter\_\_方法将其转换成一个迭代器，然后使用迭代器协议去实现循环访问，这样所有的对象就都可以通过for循环来遍历了，  
列表，字符串，元组，字典，集合，文件对象等本质上来说都不是可迭代对象，在使用for循环的时候内部是先调用他们内部的*iter*方法，使他们变成了可迭代对象，然后在使用可迭代对象的*next*方法依次循环元素，当元素循环完时，会触发StopIteration异常，for循环会捕捉到这种异常，终止迭代

访问方式常见的有下标方式访问、迭代器协议访问、for循环访问

### 迭代器

迭代器是访问集合元素的一种方式。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退  
不像列表把所有元素一次性加载到内存，迭代器是以一种延迟计算（lazy evaluation）方式返回元素，不要求事先准备好整个迭代过程中所有的元素。迭代器仅仅在迭代到某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合  
迭代器有两个基本的方法 next方法：返回迭代器的下一个元素 \_\_ 方法：返回迭代器对象本身

### 生成器

语法上和函数类似：生成器函数和常规函数几乎是一样的。它们都是使用def语句进行定义，差别在于，生成器使用yield语句返回一个值，而常规函数使用return语句返回一个值（只要一个函数内出现了yield，那它就是一个生成器函数，执行这个函数就得到一个生成器）  
自动实现迭代器协议：对于生成器，Python会自动实现迭代器协议，以便应用到迭代背景中（如for循环，sum函数）。由于生成器自动实现了迭代器协议，所以，我们可以调用它的next方法，并且，在没有值可以返回的时候，生成器自动产生StopIteration异常  
状态挂起：生成器使用yield语句返回一个值。yield语句挂起该生成器函数的状态，保留足够的信息，以便之后从它离开的地方继续执行  
生成器的唯一注意事项就是：生成器只能遍历一次。

### python语法糖

语法糖指那些没有给计算机语言添加新功能，而只是对人类来说更“甜蜜”的语法, 语法糖往往给程序员提供了更实用的编码方式，有益于更好的编码风格，更易读  
举些例子吧：  
1. c = [b,a][a>b] 取两个中的最大值  
2. lambda、filter、map、reduce函数  
3. `list1=[2\\*x+1 for x in range(10)]  
4. 对列表lst = [1, -2, 10, -12, -4, -5, 9, 2] 实现排序，按照正的放前面，负的放后面，并且分别按绝对值从小到大。即输出：[1, 2, 9, 10, -2, -4, -5, -12]方法是：lst.sort(key=lambda x: (x < 0, abs(x)))等同于：lst.sort(key=lambda x: abs(x))--->lst.sort(key=lambda x: x < 0)  
5. 装饰器

### Python的装饰器内部实现原理

装饰器本质上是一个Python函数，是闭包的一种实现, 它的作用是让其他函数在不需要做任何代码变动的前提下增加额外功能。  
使用装饰器的时候, 解析器把被装饰的函数作为参数传递给装饰器, 然后再返回一个函数对象, 装饰器内部实现需要额外增加的功能和被装饰函数的功能,虽然被装饰函数的调用方法没有改变, 但实际上已经不是原来函数, 而变成了装饰器返回的函数对象  
它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。  
什么是面向切面编程AOP?  
这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

### 简述Python的作用域以及Python搜索变量的顺序

Python作用域简单说就是一个变量的命名空间。代码中变量被赋值的位置，就决定了哪些范围的对象可以访问这个变量，这个范围就是变量的作用域。  
在Python中，只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域。  
Python的变量名解析机制也称为 LEGB 法则：本地作用域（Local）→当前作用域被嵌入的本地作用域（Enclosing locals）→全局/模块作用域（Global）→内置作用域（Built-in）

### GIL线程全局锁

线程全局锁(Global Interpreter Lock),即Python为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程

### 死锁

原因:  
\* 竞争资源  
\* 程序推进顺序不当  
必要条件:  
\* 互斥条件  
\* 请求和保持条件  
\* 不剥夺条件  
\* 环路等待条件  
处理死锁基本方法:  
\* 预防死锁(摒弃除1以外的条件)  
\* 避免死锁(银行家算法)  
\* 检测死锁(资源分配图)  
\* 解除死锁  
\* 剥夺资源  
\* 撤销进程

### 调度算法

先来先服务(FCFS, First Come First Serve)  
短作业优先(SJF, Shortest Job First)  
最高优先权调度(Priority Scheduling)  
时间片轮转(RR, Round Robin)  
多级反馈队列调度(multilevel feedback queue scheduling)  
实时调度算法:  
最早截至时间优先 EDF  
最低松弛度优先 LLF

### 什么lambda函数(匿名函数),匿名函数有什么局限性

匿名函数，也就是lambda函数，通常用在函数体比较简单的函数上。匿名函数顾名思义就是函数没有名字，因此不用担心函数名冲突。  
不过Python对匿名函数的支持有限，只有一些简单的情况下可以使用匿名函数。

### KISS原则

KISS原则是英语 Keep It Simple, Stupid 的首字母缩写。  
KISS原则是指在设计当中应当注重简约的原则。

### 简述函数式编程

在函数式编程中，函数是基本单位，变量只是一个名称，而不是一个存储单元。  
除了匿名函数外，Python还使用fliter(),map(),reduce(),apply()函数来支持函数式编程

### Python-copy()与deepcopy()区别

—–我们寻常意义的复制就是深复制，即将被复制对象完全再复制一遍作为独立的新个体单独存在。所以改变原有被复制对象不会对已经复制出来的新对象产生影响。  
—–而浅复制并不会产生一个独立的对象单独存在，他只是将原有的数据块打上一个新标签，所以当其中一个标签被改变的时候，数据块就会发生变化，另一个标签也会随之改变。  
浅拷贝只拷贝一层, 深拷贝拷贝全部  
第一：非容器类型（不可变对象, 比如数字，字符串和其他原子类型的对象，例如代码，类型和range对象等）没有拷贝一说，浅拷贝是完全用切片操作来完成的。  
第二：如果元组变量只包含原子类型对象，那么深拷贝将不会进行。

### 如何捕获异常，常用的异常机制有哪些？

如果我们没有对异常进行任何预防，那么在程序执行的过程中发生异常，就会中断程序，调用python默认的异常处理器，并在终端输出异常信息。  
try...except...finally语句:当try语句执行时发生异常，回到try语句层，寻找后面是否有except语句。  
找到except语句后，会调用这个自定义的异常处理器。except将异常处理完毕后，程序继续往下执行。finally语句表示，无论异常发生与否，finally中的语句都要执行。  
assert语句：判断assert后面紧跟的语句是True还是False，如果是True则继续执行print，如果是False则中断程序，调用默认的异常处理器，同时输出assert语句逗号后面的提示信息。  
with语句：如果with语句或语句块中发生异常，会调用默认的异常处理器处理，但文件还是会正常关闭。

### 有用过with statement吗？它的好处是什么？具体如何实现？

with语句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“清理”操作，释放资源，  
比如文件使用后自动关闭、线程中锁的自动获取和释放等。

## 面向对象

### 基本概念

多态 迭代器和生成器

### Python的面向对象和Java面向对象的区别

python可面向对象编程，可以不按照面向对象编程。java就必须面向对象编程了，必须建个类

### Python的实例方法,类方法,静态方法之间的区别及调用关系

在类里面定义的函数就是方法,类方法需要@classmethod修饰并且有个隐藏参数 cls,实例方法必须有个参数 self, 静态方法必须有@staticmethod修饰,类和实例都可以访问静态方法,实例可以访问实例方法也可以访问类方法,类可以访问类方法也可以访问实例方法,访问实例方法  
必须要带参数 self, 可以理解为类其实也是一个实例,类访问实例方法不带参数会报错的。类本身可以访问函数,实例却不行  
详细: [Python的实例方法,类方法,静态方法之间的区别及调用关系](https://www.cnblogs.com/funfunny/p/5892212.html)

### python新式类和旧式类的区别

主要区别是多继承中，新式类采用广度优先搜索，而旧式类是采用深度优先搜索

### \_\_new\_\_和\_\_init\_\_的区别

创建一个新实例时调用\_\_new\_\_,初始化一个实例时用\_\_init\_\_,这是它们最本质的区别。  
\_\_new\_\_是一个静态方法,而\_\_init\_\_是一个实例方法。  
\_\_new\_\_方法会返回一个创建的实例,而\_\_init\_\_什么都不返回。  
只有在\_\_new\_\_返回一个cls的实例时后面的\_\_init\_\_才能被调用。  
单例模式的实现可以使用\_\_new\_\_方法

### 单例模式\_\_new\_\_方法版本

class Singleton(object):   
 def \_\_new\_\_(cls, \*args, \*\*kw):   
 if not hasattr(cls, '\_instance'):   
 orig = super(Singleton, cls)   
 cls.\_instance = orig.\_\_new\_\_(cls, \*args, \*\*kw)   
 return cls.\_instance   
   
class MyClass(Singleton):   
 # ...

### 单例模式装饰器版本

def singleton(cls, \*args, \*\*kw):   
 instances = {}   
 def getinstance():   
 if cls not in instances:   
 instances[cls] = cls(\*args, \*\*kw)   
 return instances[cls]   
 return getinstance   
  
 @singleton   
class MyClass:   
 # ...

## MySQL数据库相关

### MySQL练习题

[MySQL练习题参考答案](http://www.cnblogs.com/wupeiqi/articles/5748496.html)

### 内联，左外联，右外联，全连接，交叉连接 的区别

典型多对多联表查询:

select man\_to\_women.nid,man.name as mname, women.name as wname from man\_to\_women   
left join man on man\_to\_women.man\_id = man.nid   
left join women on man\_to\_women.women\_id = women.nid   
where man.name = 'alex'

左联:显示左边表的所有数据,只显示右边表与左边表有关联的数据

### 什么是视图？以及视图的使用场景有哪些？

视图是一种虚拟的表，具有和物理表相同的功能  
只暴露部分字段给访问者，所以就建一个虚表，就是视图。  
查询的数据来源于不同的表，而查询者希望以统一的方式查询，这样也可以建立一个视图，把多个表查询结果联合起来，查询者只需要直接从视图中获取数据，不必考虑数据来源于不同表所带来的差异

### 视图作用

数据库视图隐藏了数据的复杂性。  
数据库视图有利于控制用户对表中某些列的访问。  
数据库视图使用户查询变得简单。

### 说一下事务的特性？

**原子性(Atomicity)**：事务中的全部操作在数据库中是不可分割的，要么全部完成，要么均不执行。  
**一致性(Consistency)**：几个并行执行的事务，其执行结果必须与按某一顺序串行执行的结果相一致。  
**隔离性(Isolation)**：事务的执行不受其他事务的干扰，事务执行的中间结果对其他事务必须是透明的。  
**持久性(Durability)**：对于任意已交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出

### 什么是存储过程

[简述存储过程](https://www.ilwid.net/posts/3b0cd4ca.html#%E5%AD%98%E5%82%A8%E8%BF%87%E7%A8%8B)

### 创建一个完整的存储过程示例

delimiter $$ # 修改结束符为$$, 以免程序把begin...end之间的;作为结束符   
drop procedure if exitsts proc\_p1 $$ # 如果已经存在proc\_p1就先删除   
create procedure proc\_p1(   
 in i1 int # 需要一个int类型的参数   
)   
begin # sql逻辑代码要放在begin...end之间   
 declare d1 int; # 声明一个d1变量   
 declare d2 int default 3; # 声明一个默认值为3的d2变量   
 set d1 = i1 + i2;   
 select \*from man\_to\_women where nid > d1;   
end $$   
deliniter ; # 修改结束附为默认的分号,以免影响其他语句   
  
 # 调用存储过程   
call proc\_p1(2)

### sql注入原理

sql注入漏洞产生的原因最常见的就是字符串拼接SQL语句,这种漏洞可以利用注释语句绕过验证  
如 select name from userinfo where name='alex' and password = '888'  
用户如果在name字段输入 alex' or 1=1 --f 就可以成功绕过验证。  
要解决这个问题就不能在程序中拼接sql语句,例如使用pymysql的execute方法,这个方法会自动对用户输入的引号特殊字符做转义

### 简单说一说drop、delete与truncate的区别

delete和truncate只删除表的数据不删除表的结构  
速度,一般来说: drop> truncate >delete  
delete语句是del,这个操作会放到rollback segement中,事务提交之后才生效;  
如果有相应的trigger,执行的时候将被触发。  
truncate,drop是ddl, 操作立即生效,原数据不放到rollback segment中,不能回滚. 操作不触发trigger。  
使用场景:  
不再需要一张表的时候，用drop  
想删除部分数据行时候，用delete，并且带上where子句  
保留表而删除所有数据的时候用truncate

### 数据库怎么优化查询效率？

通常会在WHERE、JOIN ON和ORDER BY使用到字段上加上索引。  
避免查询时判断NULL，否则可能会导致全表扫描。  
避免使用OR来连接查询条件，否则可能导致全表扫描，可以改用UNION或UNION ALL。  
避免LIKE查询，否则可能导致全表扫描。  
不使用SELECT \*，只查询必须的字段，避免加载无用数据。  
能用UNION ALL的时候就不用UNION，UNION过滤重复数据要耗费更多的cpu资源。  
避免Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大 量日志

总结如下：  
1、避免模糊查询，如OR、LIKE等，因为会导致全表扫描；  
2、在常用字段加索引，例如WHERE、JOIN ON和ORDER BY使用到字段上应该加索引  
3、尽量避免全局性的读写操作，例如SELECT \* 、Update全部字段

### 数据库优化方案？

1.优化索引、SQL 语句、分析慢查询;  
2.设计表的时候严格根据数据库的设计范式来设计数据库;  
3.使用缓存，把经常访问到的数据而且不需要经常变化的 数据放在缓存中，能节约磁盘 IO;  
4.优化硬件;采用 SSD，使用磁盘队列技术 (RAID0,RAID1,RDID5)等;  
5.采用 MySQL 内部自带的表分区技术，把数据分层不同 的文件，能够提高磁盘的读取效率;  
6.垂直分表;把一些不经常读的数据放在一张表里，节约 磁盘 I/O;  
7.主从分离读写;采用主从复制把数据库的读操作和写入操作分离开来;  
8.分库分表分机器(数据量特别大)，主要的的原理就是数据路由;  
9.选择合适的表引擎，参数上的优化;  
10.进行架构级别的缓存，静态化和分布式;  
11.不采用全文索引;  
12.采用更快的存储方式，例如 NoSQL 存储经常访问的数

### Mysql 几种锁的区别

* 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。
* 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。
* 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

三种锁各有各的特点，若仅从锁的角度来说，表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如WEB应用；行级锁更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLTP）系统。

### 参考资料

[常见面试题整理--数据库篇](http://blog.csdn.net/weinierzui/article/details/71054964)

## Redis基础及高可用、高并发、集群相关知识

### 什么是redis

redis是一个key-value存储系统.和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(列表)、set(集合)、zset(sorted set --有序集合)和hashs（哈希类型）  
这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的.  
在此基础上，redis支持各种不同方式的排序.与memcached一样，为了保证效率，数据都是缓存在内存中.  
区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步.

### 使用redis有哪些好处？

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O(1)  
支持丰富数据类型，支持string，list，set，sorted set，hash  
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  
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

### Redis解决秒杀/抢红包等高并发事务活动

秒杀开始前30分钟把秒杀库存从数据库同步到Redis Sorted Set  
用户秒杀库存放入秒杀限制数长度的Sorted Set  
秒杀到指定秒杀数后，Sorted Set不在接受秒杀请求，并显示返回标识  
秒杀活动完全结束后，同步Redis数据到数据库，秒杀正式结束

### redis数据超过可用内存的处理方式

1、修改redis.conf中的maxmemory-policy选项,设置删除redis键的淘汰规则  
2、加内存  
3、缩短（或设置）数据过期时间，以释放内存  
4、redis集群  
如果redis数据中，存的是关键数据，怕丢失的数据，那么，redis数据占用的内存，不要超过内存大小的70%（保险起见不要超过55%），以免内存不足！不要期望什么内存淘汰策略！它所做的居然是删数据！  
如果数据是不怕丢的缓存数据，那么可以在redis.conf里，配置如下两项，进行内存数据淘汰（其实是删除）

### 什么是高可用架构

在介绍高可用架构的方案之前，先说一下什么是高可用架构，高可用架构应具备但不限于以下特征：  
主从切换  
很好理解，当其中一台机器的服务宕机后，对于服务调用者来说，能够迅速的切换到其他可用服务，从服务升级为主服务，这种切换速度应当控制在秒级别（几秒钟）。  
当宕机的服务恢复之后，自动变为从服务，主从服务角色切换。主从切换一定是要付出代价的，所以当主服务恢复之后，也就不再替换现有的主服务。  
负载均衡  
当服务的请求量比较高的时候，一台服务不能满足需求，这时候需要多台机器提供同样的服务，将所有请求分发到不同机器上。  
高可用架构中应该具有丰富的负载均衡策略和易调节负载的方式。  
甚至可以自动化智能调节，例如由于机器性能的原因，响应时间可能不一样，这时候可以向性能差的机器少一点分发量，保证各个机器响应时间的均衡。  
易横向扩展  
当用户量越来越多，已有服务不能承载更多的用户的时候，便需要对服务进行扩展，扩展的方式最好是不触动原有服务，对于服务的调用者是透明的。

### Redis 高可用

高可用架构一般都要具备主从复制、负载均衡、易于扩展等特性，如果节点较少可以直接使用Redis集群，Redis集群具备这些特性，只要有一半以上的节点正常就能提供服务；如果规模较大的时候，就可以引入sentinel“哨兵”组件，它能提供监控、提醒、自动故障迁移等服务  
[部署高可用的Redis集群架构](http://blog.csdn.net/u013063153/article/details/71191138)

### Redis 高并发

高并发下主要考虑的是Redis集群方案，把流量分散到不同的Redis机器中去。  
Redis集群可以从横向和纵向两个方面考虑，增强整体的可用性。  
横向可以从分库剥离，负载均衡，一致性算法方面进行考虑  
纵向可以从副本集，持久化，灾备等方面进行考虑  
（即横向集群方案做高并发，纵向做高可用）

### redis应用之利用redis高效统计访问量及访问的用户（计数）

原理：利用setbit方法，设置一个二进制每一位都为0的变量如usercount，通过算法给每一个访客一个唯一的数字ID标识，然后给usercount设置该用户ID位为1即可，通过bitcount usercount命令获取usercount有多少个1即为访客总数。通过这个方法保存1亿个用户的访问数据（总访问量和来访的用户）只需要不足10m的空间，而且非常高效。 使用方法如下：

set usercount 0   
>>> OK   
bitcount usercount   
>>> 2 # 0的二进制值为0110000，所以为2   
setbit usercount 2 0   
setbit usercount 3 0 # 以上两句设置usercount的第2、3位为0，此时二进制值为 00000000   
setbit usercount 4999 1 # 假设ID为4999的用户来访，设置usercount 的第4999位为1   
setbit usercount 3582 1 # 假设ID为3582的用户来访，设置usercount 的第3582位为1   
bitcount usercount   
>>> 2 # 此时共有4999、3582两个访客，因此总数为2   
getbit usercount 4999   
>>> 1 # 获取ID为4999的用户的来访状态，如果值为1，则是访客，如果是0则是非访客

### redis 和 mysql 的区别？

redis 是内存数据库，数据保存在内存中，速度快。  
mysql 是关系型数据库，持久化存储，存放在磁盘里面，功能强大。检索的话，会涉及到一定的 IO，数据访问也就慢。  
一般需要永久存储的数据使用MySQL,比如用户信息,文章等  
而临时数据或者经常访问的数据就会使用redis,比如用户的session/排行榜/访问计数/Pub Sub 构建实时消息系统/缓存等

### Memcache与Redis的区别都有哪些？

1)、存储方式 Memecache把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。Redis有部份存在硬盘上，这样能保证数据的持久性。  
2)、数据支持类型 Memcache对数据类型支持相对简单。 Redis有复杂的数据类型。  
3)、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。

扩展阅读  
[Redis、Memcache和MongoDB的区别](https://www.cnblogs.com/tuyile006/p/6382062.html)

### Redis常见的性能问题和解决方法

1.Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化。  
2.如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。  
3.为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内。  
4.尽量避免在压力较大的主库上增加从库  
5.为了Master的稳定性，主从复制不要用图状结构，用单向链表结构更稳定，即主从关系为：Master<--Slave1<--Slave2<--Slave3.......，这样的结构也方便解决单点故障问题，实现Slave对Master的替换，也即，如果Master挂了，可以立马启用Slave1做Master，其他不变。

### mySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略（回收策略）。redis 提供 6种数据淘汰策略：  
volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰  
volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰  
volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰  
allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰  
allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰  
no-enviction（驱逐）：禁止驱逐数据

### redis的并发竞争问题如何解决?

Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有2种解决方法：  
1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。  
2.服务器角度，利用setnx实现锁。  
注：对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

### redis持久化

[redis持久化的几种方式（RDB和AOF持久化的优劣比较）](https://www.cnblogs.com/chenliangcl/p/7240350.html)

### Redis如果运行过程中崩溃了怎么办。

这个问题面试官想问的是Redis的数据保障的方法，不然崩溃了除了重启还能怎么办？  
Redis有提供数据持久化的功能，一种是快照，一种是AOF。  
快照是在某一个时间点将所有数据写入到磁盘中，AOF是将被执行的命令复制到硬盘中，快照的文件体积要比AOF的文件体积小。前者在恢复时速度比后者快，但是因为是间隔持久化，所以会有一定量的数据丢失。后者因为是实时写入的，所以数据的完整性比较好，如果丢失的话一般也就丢失一秒的数据。  
其次需要做主从复制，这样一份数据可以保存在多台服务器上，且可以避免Redis崩溃到重启完成这段时间内无法提供正常服务，同时从服务器可以分担主服务器的读压力。

### Redis集群

Redis集群创建步骤是  
1、在至少3台服务器上安装redis服务  
2、分别修改这些服务器上redis的配置文件并确保它们可以互相访问，然后启动它们  
3、使用Redis 官方提供的redis-trib.rb工具的create命令创建集群

### 简单说一下Redis集群的工作原理

redis cluster在设计的时候，就考虑到了去中心化，去中间件，也就是说，集群中的每个节点都是平等的关系，都是对等的，每个节点都保存各自的数据和整个集群的状态。每个节点都和其他所有节点连接，而且这些连接保持活跃，这样就保证了我们只需要连接集群中的任意一个节点，就可以获取到其他节点的数据。  
Redis 集群没有并使用传统的一致性哈希来分配数据，而是采用另外一种叫做哈希槽 (hash slot)的方式来分配的。redis cluster 默认分配了 16384 个slot，当我们set一个key 时，会用CRC16算法来取模得到所属的slot，然后将这个key 分到哈希槽区间的节点上，具体算法就是：CRC16(key) % 16384。所以我们在测试的时候看到set 和 get 的时候，直接跳转到了7000端口的节点。  
Redis 集群会把数据存在一个 master 节点，然后在这个 master 和其对应的salve 之间进行数据同步。当读取数据时，也根据一致性哈希算法到对应的 master 节点获取数据。只有当一个master 挂掉之后，才会启动一个对应的 salve 节点，充当 master 。  
需要注意的是：必须要3个或以上的主节点，否则在创建集群时会失败，并且当存活的主节点数小于总节点数的一半时，整个集群就无法提供服务了。

### 悲观锁和乐观锁区别，乐观锁适用于什么情况

悲观锁，每次访问数据的时候都觉得会有别人修改它，所以每次拿数据时都会上锁，确保在自己使用的过程中不会被他人访问。乐观锁每次拿数据的时候都不会上锁，只在更新数据的时候去判断该数据是否被别人修改过。  
大多数的关系数据库写入操作都是基于悲观锁，缺点在于如果持有锁的客户端运行的很慢，那么等待解锁的客户端被阻塞的时间就越长。Redis的事务是基于乐观锁的机制，不会在执行WATCH命令时对数据进行加锁，只是会在数据已经被其他客户端抢先修改了的情况下，通知执行WATCH命令的客户端。  
乐观锁适用于读多写少的情况，因为在写操作比较频繁的时候，会不断地retry，从而降低性能。

### MVCC

在并发读写数据库时，读操作可能会不一致的数据（脏读）。为了避免这种情况，需要实现数据库的并发访问控制，最简单的方式就是加锁访问。  
由于，加锁会将读写操作串行化，所以不会出现不一致的状态。但是，读操作会被写操作阻塞，大幅降低读性能。  
而其优化的手段就是，在进行写操作时，将数据copy一份，不会影响原有数据，然后进行修改，修改完成后原子替换掉旧的数据，而读操作只会读取原有数据。  
通过这种方式实现写操作不会阻塞读操作，从而优化读效率。  
MVCC全称是Multi-Version Concurrent Control，即多版本并发控制。  
在MVCC协议下每个读操作会看到一个一致性的snapshot，并且可以实现非阻塞的读。MVCC允许数据具有多个版本，这个版本可以是时间戳或者是全局递增的事务ID，  
在同一个时间点，不同的事务看到的数据是不同的。

### 参考资料

[数据库SQL优化大总结之 百万级数据库优化方案](https://www.ilwid.net/posts/1f8fcfbf.html)

## 多线程、多进程、协程、异步编程、I/O多路复用

### 请简述线程、进程、协程的特性

进程是系统进行资源分配和调度的最小单位,进程是线程的容器,一个进程可以包含多个线程,进程之间数据不能互相访问  
线程是CPU调度的最小单位,线程是程序执行流的最小单元, 每一个进程都至少有一个线程, 线程之间数据可以共享  
进程线程都是由操作系统调度,python中多线程和多进程都是通过切换上下文来实现,都会耗费额外的系统资源  
协程由程序员调度, 由代码切换来控制, 系统并不知道协程的存在, 各种多并发异步非阻塞模块都是基于协程来实现的  
进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间，而协程就是用户自己控制切换的时机，不再需要陷入系统的内核态。  
Python里最常见的yield就是协程的思想

### 线程有几种状态?生命周期是怎样的?

线程有五种状态:创建、就绪、运行、阻塞、死亡。  
调用start方法时，线程就会进入就绪状态。  
在线程得到cpu时间片时进入运行状态。  
线程调用yield方法可以让出cpu时间回到就绪状态。  
线程运行时可能由于IO、调用sleep、wait、join方法或者无法获得同步锁等原因进入阻塞状态。  
当线程获得到等待的资源资源或者引起阻塞的条件得到满足时，会从阻塞状态进入就绪状态。  
当线程的run方法执行结束时，线程就进入死亡状态。

### 进程间通讯的方式有哪些，各有什么优缺点：

1）管道：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程之间使用。进程的亲缘关系通常是指父子进程关系。  
2）有名管道（FIFO）：有名管道也是半双工的通信方式，但是允许在没有亲缘关系的进程之间使用，管道是先进先出的通信方式。  
3）信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。  
4）消息队列：消息队列是有消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。  
5）信号 ( sinal ) ：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。  
6）共享内存( shared memory ) ：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。  
7）套接字( socket ) ：套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信。  
8）文件

### 线程池的原理：

既然是线程池（Thread pool），其实名字很形象，就是把指定数量的可用子线程放进一个"池里"，有任务时取出一个线程执行，任务执行完后，  
并不立即销毁线程，而是放进线程池中，等待接收下一个任务。这样内存和cpu的开销也比较小，并且我们可以控制线程的数量。

### 线程池的实现：

线程池有很多种实现方式，在python中，已经给我们提供了一个很好的实现方式：Queue-队列。因为python中Queue本身就是同步的，所以也就是线程安全的，所以我们可以放心的让多个线程共享一个Queue。  
那么说到线程池，那么理应也得有一个任务池，任务池中存放着待执行的任务，各个线程到任务池中取任务执行，那么用Queue来实现任务池是最好不过的。

线程安全和非线程安全  
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。如Queue、logging  
非线程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。如list、set、dict

### 异步IO / IO多路复用的原理

[IO多路复用的原理详解](http://www.cnblogs.com/alex3714/articles/5248247.html)  
[IO多路复用(番外篇)](http://www.cnblogs.com/alex3714/articles/5876749.html)  
[Python Select epoll详解](http://www.cnblogs.com/alex3714/articles/4372426.html)

### 简述事件驱动

1. 有一个事件（消息）队列；
2. 鼠标按下时，往这个队列中增加一个点击事件（消息）；
3. 有个循环，不断从队列取出事件，根据不同的事件，调用不同的函数，如onClick()、onKeyDown()等；
4. 事件（消息）一般都各自保存各自的处理函数指针，这样，每个消息都有独立的处理函数；

### 简述I/O多路复用

I/O多路复用是指单个线程，通过记录跟踪每个I/O流(sock)的状态，来同时管理多个I/O流。即系统内核缓冲I/O数据，当某个I/O准备好后，系统通知应用程序该I/O可读或可写，这样应用程序可以马上完成相应的I/O操作，而不需要等待系统完成相应I/O操作，从而应用程序不必因等待I/O操作而阻塞。  
与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。比如高并发应用nginx就是用I/O多路复用,它的性能是极佳的。  
目前支持I/O多路复用的系统调用有select，epoll，它们本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

### I/O多路复用和异步I/O的区别

I/O多路复用的本质是同步IO，是阻塞的，而异步I/O是非阻塞的。  
阻塞、非阻塞、IO多路复用，都是同步IO，需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的；异步是非阻塞的，真正的异步IO需要CPU的深度参与。  
使用 select，epoll 的IO多路复用的模型中，虽然进程大部分时间都不会阻塞，但是它仍然要求进程去主动的轮询，并且当数据准备完成以后，也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存。  
而异步I/O则完全不同。用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。  
nginx、tornado、twised等应用，虽然我们习惯性地叫它们为异步IO，但是它们实际上是IO多路复用。因为它们的底层是基于select和epoll实现的。  
真正的异步现在内核支持不完美，实际应用比较少，python中真正支持异步IO的模块是asyncio

### select,poll和epoll之间的异同

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。  
所有的I/O都是轮询的方法,只不过实现的层面不同罢了。  
Windows底层只支持select, Linux很早就支持epoll了,tornado底层使用的就是epoll的。  
基本上select有3个缺点:  
\* 连接数受限（1024）  
\* 查找配对速度慢  
\* 数据由内核拷贝到用户态  
poll改善了第一个缺点  
epoll改了三个缺点。

### epoll中et和lt的区别与实现原理

LT：水平触发，效率会低于ET触发，尤其在大并发，大流量的情况下。但是LT对代码编写要求比较低，不容易出现问题。LT模式服务编写上的表现是：只要有数据没有被获取，内核就不断通知你，因此不用担心事件丢失的情况。  
ET：边缘触发，效率非常高，在并发，大流量的情况下，会比LT少很多epoll的系统调用，因此效率高。但是对编程要求高，需要细致的处理每个请求，否则容易发生丢失事件的情况。

## 网络编程基础知识

### 解释下Http协议

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。  
HTTP协议的主要特点可概括如下：  
1. 支持客户/服务器模式。  
2. 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。  
3. 灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。  
4. 无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。  
5. 无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

[关于HTTP协议，一篇就够了](https://www.cnblogs.com/ranyonsue/p/5984001.html)

### 在浏览器地址栏键入URL，按下回车之后会经历以下流程：

1、浏览器向 DNS 服务器请求解析该 URL 中的域名所对应的 IP 地址;  
2、解析出 IP 地址后，根据该 IP 地址和默认端口 80，和服务器建立TCP连接;  
3、浏览器发出读取文件(URL 中域名后面部分对应的文件)的HTTP 请求，该请求报文作为 TCP 三次握手的第三个报文的数据发送给服务器;  
4、服务器对浏览器请求作出响应，并把对应的 html 文本发送给浏览器;  
5、释放 TCP连接;  
6、浏览器将该 html 文本并显示内容;

### HTTP之状态码

状态代码有三位数字组成，第一个数字定义了响应的类别，共分五种类别:  
1xx：指示信息--表示请求已接收，继续处理  
2xx：成功--表示请求已被成功接收、理解、接受  
3xx：重定向--要完成请求必须进行更进一步的操作  
4xx：客户端错误--请求有语法错误或请求无法实现  
5xx：服务器端错误--服务器未能实现合法的请求

### 常见状态码：

200 OK //客户端请求成功   
400 Bad Request //客户端请求有语法错误，不能被服务器所理解   
401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用   
403 Forbidden //服务器收到请求，但是拒绝提供服务   
404 Not Found //请求资源不存在，eg：输入了错误的URL   
500 Internal Server Error //服务器发生不可预期的错误   
503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

### TCP（传输控制协议）与UDP（用户数据包协议）的区别；

1）TCP提供面向连接的传输，通信前要先建立连接（三次握手机制）；UDP提供无连接的传输，通信前不需要建立连接。  
2）TCP提供可靠的传输（有序，无差错，不丢失，不重复）；UDP提供不可靠的传输。  
3）TCP面向字节流的传输，因此它能将信息分割成组，并在接收端将其重组；UDP是面向数据报的传输，没有分组开销。  
4）TCP提供拥塞控制和流量控制机制；UDP不提供拥塞控制和流量控制机制。

### 什么时候使用TCP协议，什么时候使用UDP协议；

1. 对数据可靠性的要求。对数据要求高可靠性的应用需选择TCP协议，如验证，密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择UDP传送；
2. 应用的实时性。TCP协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用TCP协议会有较大的延迟，因此不适合对实时性要求较高的应用，如VOIP、视频监控等。相反，UDP协议则在这些应用中能发挥很好的作用。
3. 网络的可靠性。由于TCP协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率，因此，在网络状况不是很好的情况下需选用TCP协议（如在广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要采用TCP协议，而建议选择UDP协议来减少网络负荷。

为什么早期QQ使用UDP协议  
原因是因为当时没有epoll这种可以支持成千上万tcp并发连接的技术,所以他们使用了udp,然后在udp上面封装了一下,模拟了一下tcp,解决了大并发的问题,之后因为做的很nb了,虽然epoll这种技术出现了,还是没有改回使用tcp了.现在再做类似的东西就不需要使用udp了

### 三次握手，四次挥手的详细过程以及作用；

**三次握手**：  
第一次握手是客户端connect连接到server，server accept client的请求之后，向client端发送一个消息，相当于说我都准备好了，你连接上我了，这是第二次握手，第3次握手就是client向server发送的，就是对第二次握手消息的确认。之后client和server就开始通讯了。

**四次挥手**  
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。  
收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。CP的连接的拆除需要发送四个包，因此称为四次挥手(four-way handshake)。客户端或服务器均可主动发起挥手动作，在socket编程中，任何一方执行close()操作即可产生挥手操作。  
断开连接的一端发送close请求是第一次挥手，另外一端接收到断开连接的请求之后需要对close进行确认，发送一个消息，这是第二次挥手，发送了确认消息之后还要向对端发送close消息，要关闭对对端的连接，这是第3次挥手，而在最初发送断开连接的一端接收到消息之后，进入到一个很重要的状态time\_wait状态，这个状态也是面试官经常问道的问题，最后一次挥手是最初发送断开连接的一端接收到消息之后。对消息的确认。

### TCP为什么不是两次连接？而是三次握手？

如果A与B两个进程通信，如果仅是两次连接。可能出现的一种情况就是：A发送完请报文以后，由于网络情况不好，出现了网络拥塞，即B延时很长时间后收到报文，即此时A将此报文认定为失效的报文。B收到报文后，会向A发起连接。此时两次握手完毕，B会认为已经建立了连接可以通信，B会一直等到A发送的连接请求，而A对失效的报文回复自然不会处理。依次会陷入B忙等的僵局，造成资源的浪费。

### 网络编程的一般步骤

对于TCP连接：  
1.服务器端： 1）创建套接字create；2）绑定端口号bind；3）监听连接listen；4）接受连接请求accept，并返回新的套接字；5）用新返回的套接字recv/send；6）关闭套接字。  
2.客户端： 1）创建套接字create; 2）发起建立连接请求connect; 3）发送/接收数据send/recv；4）关闭套接字。  
TCP总结：  
Server端：create -- bind -- listen-- accept-- recv/send-- close  
Client端：create------- conncet------send/recv------close.

对于UDP连接：  
1.服务器端:1）创建套接字create；2）绑定端口号bind；3）接收/发送消息recvfrom/sendto；4）关闭套接字。  
2.客户端:1）创建套接字create；2）发送/接收消息sendto/recvfrom；3）关闭套接字.  
UDP总结:  
Server端：create----bind ----recvfrom/sendto----close  
Client端：create---- sendto/recvfrom----close.

### 写一个简单的python socket

import socket   
# 服务端   
s = socket.socket()   
s.bind(('192.168.1.11', '1000'))   
s.listen()   
conn, addr = s.accept()   
data = s.recv(1024)   
print(data)   
s.send('ok')   
s.close()   
  
 # 客户端   
c = socket.socket()   
c.connect(('192.168.1.11', '1000'))   
c.send(b'hello world')   
data = c.recv(1024)   
print('recv:',data)   
c.close()

### GET和POST，有什么区别

根据HTTP协议,他们根本区别是，简单点儿说，GET用于获取数据，POST用于修改数据。  
另外用法上, 一般GET使用URL或Cookie传参, 而POST将数据放在BODY中, 但HTTP协议没有这样的要求, 技术上说GET和POST与数据如何传递没有关系

### apache和nginx的区别

nginx 相对 apache 的优点：  
+ 轻量级，同样起web 服务，比apache 占用更少的内存及资源  
+ 抗并发，nginx 处理请求是异步非阻塞的，支持更多的并发连接，而apache 则是阻塞型的，在高并发下nginx 能保持低资源低消耗高性能  
+ 配置简洁  
+ 高度模块化的设计，编写模块相对简单  
+ 社区活跃  
apache 相对nginx 的优点：  
+ rewrite ，比nginx 的rewrite 强大  
+ 模块超多，基本想到的都可以找到  
+ 少bug ，nginx 的bug 相对较多  
+ 超稳定

### 怎么保证密码保存和传输是安全的

MD5+Salt+时间戳+sign，这个salt可以随机值(传递给服务端，服务端使用预先约定到的协议进行解密)

### CSRF和XSS区别

CSRF重点在请求,XSS重点在脚本  
CSRF(Cross-site request forgery)跨站请求伪造  
XSS(Cross Site Scripting)跨站脚本攻击

### 幂等 Idempotence

HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。(注意是副作用)  
幂等与你是不是分布式高并发还有JavaEE都没有关系。关键是你的操作是不是幂等的。  
一个幂等的操作典型如：把编号为5的记录的A字段设置为0这种操作不管执行多少次都是幂等的。  
一个非幂等的操作典型如：把编号为5的记录的A字段增加1这种操作显然就不是幂等的。  
要做到幂等性，从接口设计上来说不设计任何非幂等的操作即可。  
譬如说需求是：当用户点击赞同时，将答案的赞同数量+1。  
改为：当用户点击赞同时，确保答案赞同表中存在一条记录，用户、答案。赞同数量由答案赞同表统计出来。

### RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。  
RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。  
RPC使得开发包括网络分布式多程序在内的应用程序更加容易。  
总结:服务提供的两大流派.传统意义以方法调用为导向通称RPC。为了企业SOA,若干厂商联合推出webservice,制定了wsdl接口定义,传输soap.当互联网时代,臃肿SOA被简化为http+xml/json.但是简化出现各种混乱。以资源为导向,任何操作无非是对资源的增删改查，于是统一的REST出现了.  
进化的顺序: RPC -> SOAP -> RESTful

### WSGI

WSGI, Web Server Gateway Interface，是Python应用程序或框架和Web服务器之间的一种接口，  
WSGI的其中一个目的就是让用户可以用统一的语言(Python)编写前后端。

### 中间人攻击

在GFW里屡见不鲜的中间人攻击（Man-in-the-middle attack，通常缩写为MITM）是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制。

## Web 框架之 Django、Tornado

### Django请求生命周期

1. 当用户在浏览器中输入url时，浏览器会生成请求头和请求体发给服务端。请求头和请求体中会包含浏览器的动作(action)，这个动作通常为get或者post，体现在url之中。
2. url经过Django中的wsgi，再经过Django的中间件,最后url到过路由映射表,在路由中一条一条进行匹配。一旦其中一条匹配成功就执行对应的视图函数,后面的路由就不再继续匹配了。
3. 视图函数根据客户端的请求查询相应的数据。返回给Django，然后Django把客户端想要的数据做为一个字符串返回给客户端。
4. 客户端浏览器接收到返回的数据，经过渲染后显示给用户。

### 解释下django-debug-toolbar的使用

使用django开发站点时，可以使用django-debug-toolbar来进行调试。  
在settings.py中添加debug\_toolbar.middleware.DebugToolbarMiddleware到项目的MIDDLEWARE\_CLASSES 内。

### 如何进行Django单元测试

单元测试存在的意义就是，可以让我们放心的重构代码，可以在重构代码的时候省下测试重构的代码能否正确运行的时间  
[Django 单元测试](https://code.ziqiangxuetang.com/django/django-test.html)  
[Django单元测试基础知识](https://www.jianshu.com/p/34267dd79ad6)

### model之F/Q操作

F操作,使用查询条件的值,F()允许Django在未实际链接数据的情况下具有对数据库字段的值的引用, Python都不需获取F内的值, 直接操作后保存, 用法场景例如 字段+1(加减乘除运算) 字段比较如日期比较

from django.db.models import F, Q   
models.UserInfo.objects.filter().update(salary=F('salary')+500)

### Q操作,构造搜索条件, 用于构造复杂的查询条件如各种筛选过滤

con = Q()   
q1 = Q()   
q1.connector = 'OR'   
q1.children.append(('id', 1))   
q1.children.append(('id', 2))   
q2 = Q()   
q2.connector = 'OR'   
q2.children.append(('status', '在线'))   
con.add(q1, 'AND')   
con.add(q2, 'AND')   
models.Tb1.objects.filter(con)

### simple\_tag用法

from django import template   
register = template.Library()   
  
 @register.simple\_tag   
def foo()...

### ORM是什么？

ORM 全称是 Object/Relation Mapping,即对象/关系数据库映射。可以讲ORM理解成一种规范，它概述了这类框架的基本特征，完成面相对象的编程语言到关系数据库的映射。  
ORM可以当成是应用程序和数据的桥梁。

基本映射方式 ORM工具提供了持久化类和数据表之间的映射关系，通过这种映射关系的过度，程序员可以很方便地通过持久化类实现数据表的操作。实际上，所有的ORM工具大致都遵循相同的映射思路。  
映射关系：  
1、数据表映射类  
持久化映射到一个数据表。程序使用这个持久化类来创建实例，修改属性，删除实例时，系统自动对这个表进行操作。  
2、数据表的行映射对象（实例）  
持久化类会生成很多实例，每个实例就对应数据表中的一行记录。  
3、数据的列（字段）映射对象的属性  
当程序修改某个持久化对象的指定属性时，ORM将会将其转换成对应数据表中的指定数据行、指定 列的操作。

### 一般的ORM包括以下四部分：

一个对持久类对象进行CRUD操作的API；  
一个语言或API用来规定与类和类属性相关的查询；  
一个规定mapping metadata的工具；  
一种技术可以让ORM的实现同事务对象一起进行dirty checking, lazy association fetching以及其他的优化操作。

### 使用ORM的好处

应用程序不再直接访问底层数据库，而是以面向对象的操作转换成底层的SQL操作。  
就是把持久化对象的保存、修改、删除等操作，转换成对数据库的操作。

### 解释一下 Django 和 Tornado 的关系、差别

python web框架分两种, 一种是django 一种是其他, 其他里又分两种, 一种是tornado, 一种是其他  
django最大的特点是大而全, Django的主要目的是简便、快速的开发数据库驱动的网站。它强调代码复用,多个组件可以很方便的以“插件”形式服务于整个框架  
Tornado是异步非阻塞式服务器，速度快。底层基于epoll

### Django生产环境部署

Django + Uwsgi（动态请求） + Nginx（静态请求） 实现生产环境部署  
[Django + Uwsgi + Nginx 实现生产环境部署](https://www.cnblogs.com/alex3714/p/6538374.html)

### Tornado 的核心是什么？

Tornado 的核心是 ioloop 和 iostream 这两个模块，前者提供了一个高效的 I/O 事件循环，后者则封装了 一个无阻塞的 socket 。通过向 ioloop 中添加网络 I/O 事件，利用无阻塞的 socket ，再搭配相应的回调 函数，便可达到梦寐以求的高效异步执行。

### 怎么使用Tornado 异步非阻塞

非阻塞这部分代码主要就在ioloop.py里  
Future对象 代表将来执行或没有执行的任务的结果。  
两大功能:  
1,线程可以处理其他请求  
2,当前挂起的请求返回并释放  
事件循环 使用 gen.coroutine 装饰器  
生成器 yield future

异步非阻塞tornado模型中使用select或者epoll监听用户请求,当接收到用户请求的时候,首先生成一个Future对象,再把这个socket和future存入一个字典；hold住这个请求并把该请求交给相应的函数处理求,当该函数处理完成后会通过回调函数把结果和状态交给future对象, 这时tornado会把future的数据返回给用户,在这整个过程中, tornado程序是非阻塞的.

### Tornado 异步模式

当发送GET请求时，由于方法被 @gen.coroutine装饰且yield 一个 Future对象，那么Tornado会等待，  
等待用户向future对象中放置数据或者发送信号，如果获取到数据或信号之后，就开始执行doing方法。  
异步非阻塞体现在当在Tornaod等待用户向future对象中放置数据时，还可以处理其他请求。  
注意：在等待用户向future对象中放置数据或信号时，此连接是不断开的。

### @gen.coroutine的作用

gen.coroutine主要是使用协程的方式实现类似异步的处理效果  
它简化异步代码的编写，避免写回调函数，加快开发效率，提高代码可读性，通过结合 pyhton 的 yield 语句实现协程  
通过 @gen.coroutine 修饰的函数返回值变为 Future, 在调用结束的时候会调用  
Future.set\_result，这样就会调用与 Future 相关联的回调

## 前端知识之 jQuery、Ajax、JsonP等

### rgba()和opacity的透明效果有什么不同？

答案： rgba()和opacity都能实现透明效果，但最大的不同是opacity作用于元素，以及元素内的所有内容的透明度，  
而rgba()只作用于元素的颜色或其背景色。（设置rgba透明的元素的子元素不会继承透明效果！）

### px和em的区别？

px和em都是长度单位，区别是，px的值是固定的，指定是多少就是多少，计算比较容易。em得值不是固定的，并且em会继承父级元素的字体大小。  
浏览器的默认字体高都是16px。所以未经调整的浏览器都符合: 1em=16px。那么12px=0.75em, 10px=0.625em。

### javascript：获取所有的checkbox？

var domList = document.getElementsByTagName(‘input’)   
// checkbox属于input，所以通过getElementsByTagName即标签名获取所有的input数组，包含文本框text，单选按钮radio，复选框checkbox等等   
var checkBoxList = []; // 定义一个存储checkbox的空数组   
var len = domList.length;　　//缓存到局部变量 // 第一步获取的数组的长度   
while (len--) {　　//使用while的效率会比for循环更高 // 开始循环判断   
　　if (domList[len].type == ‘checkbox’) { // 如果类型为checkbox即为题目所需的复选框   
 　　checkBoxList.push(domList[len]); // 就把那个元素加入到上面定义的数组中   
　　}   
}

用js实现随机选取10–100之间的10个且不重复的数字，存入一个数组。

var randoms=[];   
while (true)   
{   
 var isExists = false;   
 // 获取一个10–100范围的数   
 var random = parseInt(10 + (90 - 10) \* (Math.random()))   
 // 判断当前随机数是否已经存在   
 for (var i = 0; i < randoms.length; i++) {   
 if (random === randoms[i]) {   
 isExists = true;   
 break;   
 }   
 }   
 // 如果不存在，则添加进去   
 if (!isExists)   
 randoms.push(random);   
 // 如果有10位随机数了，就跳出   
 if (randoms.length === 10)   
 break;   
}

### window.onload与$(document).ready()异同

原生JS的window.onload与Jquery的$(document).ready(function(){})有什么不同？如何用原生JS实现Jq的ready方法？  
window.onload()方法是必须等到页面内包括图片的所有元素加载完毕后才能执行。  
$(document).ready()是DOM结构绘制完毕后就执行，不必等到加载完毕。$(document).ready(function(){})能够简写成$(function(){})。

### JS作用域

1. JS以函数作为作用域
2. 函数的作用域在未调用之前就已经创建
3. 函数的作用域存在作用域链，并且也是在未调用之前创建
4. 函数内局部变量提前声明

### 以下代码的输出是什么?

function t1(age){   
 console.log(age);   
 var age = 27;   
 console.log(age);   
 function age(){};   
 console.log(age);   
}   
t1(3)   
// 答案 function age() 27 27

### 你为什么要使用jquery？

因为jQuery是轻量级的框架，它有强大的选择器，出色的DOM操作的封装，有可靠的事件处理机制(jQuery在处理事件绑定的时候相当的可靠)，完善的ajax(它的ajax封装的非常的好，不需要考虑复杂浏览器的兼容性和XMLHttpRequest对象的创建和使用的问题。) 出色的浏览器的兼容性。而且支持链式操作，隐式迭代。行为层和结构层的分离，还支持丰富的插件，jquery的文档也非常的丰富。

### 讲一下jquery有哪些选择器？

jQuery中的选择器大致分为:基本选择器，层次选择器，过滤选择器，表单选择器

### 你是如何使用jquery中的ajax的？

如果是一些常规的ajax程序的话，使用load(),$.get(),$.post(),就可以搞定了，一般我会使用的是$.post() 方法。如果需要设定beforeSend(提交前回调函数),error(失败后处理),success(成功后处理)及complete(请求完成后处理)回调函数等，这个时候我会使用$.ajax()

### jquery中$.get()提交和$.post()提交有区别吗？

$.get() 方法使用GET方法来进行异步请求的。$.post() 方法使用POST方法来进行异步请求的。  
get请求会将参数跟在URL后进行传递，而POST请求则是作为HTTP消息的实体内容发送给Web服务器的，这种传递是对用户不可见的。  
get方式传输的数据大小不能超过2KB 而POST要大的多  
GET 方式请求的数据会被浏览器缓存起来，因此有安全问题。

### 在jquery中你是如何去操作样式的？

addClass() 来追加样式 ，removeClass() 来删除样式，toggle() 来切换样式, css直接设置样式  
style设置单个样式

### 你在jquery中使用过哪些插入节点的方法，它们的区别是什么？

append(),appendTo(),prepend(),prependTo(),after(),insertAfter()，before(),insertBefore() 大致可以分为 内部追加和外部追加append() 表式向每个元素内部追加内容。appendTo()表示 讲所有的元素追加到指定的元素中。例$(A)appendTo(B) 是将A追加到B中下面的方法解释类似。

你使用过包裹节点的方法吗，包裹节点有方法有什么好处？  
wrapAll(),wrap(), wrapInner()需要在文档中插入额外的结构化标记的时候可以使用这些包裹的方法应为它不会帛画原始文档的语义

### jquery中如何来获取或和设置属性？

jQuery中可以用attr()方法来获取和设置元素属性removeAttr() 方法来删除元素属性

### 如何来设置和获取HTML 和文本的值？

html()方法 类似于innerHTML属性 可以用来读取或者设置某个元素中的HTML内容  
注意：html() 可以用于xhtml文档 不能用于xml文档text() 类似于innerText属性 可以用来读取或设置某个元素中文本内容。val() 可以用来设置和获取元素的值

### jquery中有哪些方法可以遍历节点

children() 取得匹配元素的子元素集合,只考虑子元素不考虑后代元素 next() 取得匹配元素后面紧邻的同辈元素  
prev() 取得匹配元素前面紧邻的同辈元素  
siblings() 取得匹配元素前后的所有同辈元素  
closest() 取得最近的匹配元素  
find() 取得匹配元素中的元素集合 包括子代和后代

### radio单选组的第二个元素为当前选中值，该怎么去取？

$('input[name=items]').get(1).checked = true;

### 你知道jQuery中的事件冒泡吗，它是怎么执行的，何如来停止冒泡事件

知道,事件冒泡是从里面的往外面开始触发。在jQuery中提供了stopPropagation()方法可以停止冒泡。

### 在jquery中你有没有编写过插件，插件有什么好处？它应该注意那些？

1. 插件的好处：对已有的一系列方法或函数的封装，以便在其他地方重新利用，方便后期维护和提高开发效率插件的分类：封装对象方法插件 、封装全局函数插件、选择器插件
2. 注意的地方：
3. 插件的文件名推荐命名为jquery.[插件名].js，以免和其他的javaScript库插件混淆
4. 所有的对象方法都应当附加到jQuery.fn对象上，而所有的全局函数都应当附加到jQuery对象本身上
5. 插件应该返回一个jQuery对象，以保证插件的可链式操作
6. 避免在插件内部使用$作为jQuery对象的别名,而应使用完整的jQuery来表示，这样可以避免冲突或使用闭包来避免
7. 所有的方法或函数插件，都应当一分好结尾，否则压缩的时候可能出现问题。在插件头部加上分号，这样可以避免他人的不规范代码给插件带来影响
8. 在插件中通过$.extent({})封装全局函数,选择器插件，扩展已有的object对象通过$.fn.extend({})封装对象方法插件

## Ajax相关

### Ajax相关知识点

Iframe伪造Ajax请求  
FormData对象以及Ajax文件上传  
Iframe文件上传  
JSONP实现AJax跨域

### XMLHttpRequest对象发送请求

Ajax和XMLHttpRequest  
从上面的解释中可以知道：ajax是一种技术方案，但并不是一种新技术。它依赖的是现有的CSS/HTML/Javascript，而其中最核心的依赖是浏览器提供的XMLHttpRequest对象，是这个对象使得浏览器可以发出HTTP请求与接收HTTP响应。  
所以我用一句话来总结两者的关系：我们使用XMLHttpRequest对象来发送一个Ajax请求。

### 什么是JSONP？

由于同源策略的限制，XmlHttpRequest只允许请求当前源（域名、协议、端口）的资源，为了实现跨域请求，可以通过在页面动态创建script标签，通过通过src属性实现跨域请求，然后在服务端输出JSON数据并执行回调函数，从而解决了跨域的数据数据传输。

### ajax与jsonp跨域请求的异同再做一些补充说明：

1、ajax和jsonp这两种技术在调用方式上“看起来”很像，目的也一样，都是请求一个url，然后把服务器返回的数据进行处理，因此jquery和ext等框架都把jsonp作为ajax的一种形式进行了封装；  
2、但ajax和jsonp其实本质上是不同的东西。ajax的核心是通过XmlHttpRequest获取非本页内容，而jsonp的核心则是动态添加<script>标签来调用服务器提供的js脚本。  
3、所以说，其实ajax与jsonp的区别不在于是否跨域，ajax通过服务端代理一样可以实现跨域，jsonp本身也不排斥同域的数据的获取。  
4、还有就是，jsonp是一种方式或者说非强制性协议，如同ajax一样，它也不一定非要用json格式来传递数据，如果你愿意，字符串都行，只不过这样不利于用jsonp提供公开服务。  
总而言之，jsonp不是ajax的一个特例，哪怕jquery等巨头把jsonp封装进了ajax，也不能改变着一点！

### 浅谈session实现原理(阿里面试题)

现在一般采用使用服务器端产生的Session结合浏览器的Cookie来识别用户和存储数据,  
一般来说包括以下4个步骤：  
1. 服务器端的产生Session ID（一般可以使用UUID模块）  
2. 服务器端和客户端存储Session ID  
3. 从HTTP Header中提取Session ID(发送的是一个COOKIC值)  
4. 根据Session ID从服务器端的Hash中获取请求者身份信息

### cookie机制和session机制的区别

cookie保存在客户端，而session保存在服务器端  
cookie上的数据需要来回传输到服务器上,session会消耗服务器的性能  
安全性要求不是很高的用户数据一般采用cookies,敏感数据使用session

## 爬虫相关知识

### Scrapy需要了解的类

去重 RepeatFilter()类  
序列化 Pipeline()类  
终止 DropItem()类  
信号 - 预留钩子扩展 sinals

### Scrapy框架数据流（重要！面试官一般问：说说Scrapy原理）

Scrapy中的数据流由执行引擎控制，其过程如下：  
\* 引擎从Spiders中获取到的最初的要爬取的请求(Requests)。  
\* 引擎安排请求(Requests)到调度器中，并向调度器请求下一个要爬取的请求(Requests)。  
\* 调度器返回下一个要爬取的请求(Request)给请求。  
\* 引擎从上步中得到的请求(Requests)通过下载器中间件(Downloader Middlewares)发送给下载器(Downloader),这个过程中下载器中间件(Downloader Middlerwares)中的 process\_request() 函数就会被调用。  
\* 一旦页面下载完毕，下载器生成一个该页面的Response，并将其通过下载中间件(Downloader Middlewares)中的 process\_response() 函数，最后返回给引擎  
\* 引擎从下载器中得到上步中的Response并通过Spider中间件(Spider Middewares)发送给Spider处理，这个过程中Spider中间件(Spider Middlewares)中的 process\_spider\_input() 函数会被调用到。  
\* Spider处理Response并通过Spider中间件(Spider Middlewares)返回爬取到的Item及(跟进的)新的Request给引擎，这个过程中Spider中间件(Spider Middlewares)的 process\_spider\_output() 函数会被调用到。  
\* 上步中Spider处理的结果可能是Item和Request，引擎将其爬取到的Item给Item管道(Piplline),将Requests发送给调度器，并向调度器请求可能存在的下一个要爬取的请求(Requests)  
\* (从第二步)重复知道调度器中没有更多的请求(Requests)。

### 反爬的一般手段

* 遵守robots 协议
* 限制频率
* 图像识别验证码
* 多账号反爬
* 代理池
* 分布式爬虫
* 保存cookies,带上cookie去访问，或某些场景不要带cookies访问
* 一般来说移动端、app的页面反爬手段相对少
* 使用PhantomJS，Selenium模拟浏览器

参考:[如何应对网站反爬虫策略？如何高效地爬大量数据?](https://www.zhihu.com/question/28168585)

### 常见的反爬虫和应对方法？（爬虫与反爬虫的博弈）

#### **通过Headers反爬虫**

从用户请求的Headers反爬虫是最常见的反爬虫策略。很多网站都会对Headers的User-Agent进行检测，还有一部分网站会对Referer进行检测（如一些资源网站的防盗链、知乎）。如果遇到了这类反爬虫机制，可以直接在爬虫中添加Headers，构造随机UA和Refer等参数即可绕过。

#### **基于用户行为反爬虫**

还有一部分网站是通过检测用户行为，例如同一IP短时间内多次访问同一页面，或者同一账户短时间内多次进行相同操作。  
大多数网站都是前一种情况，对于这种情况，使用IP代理就可以解决。可以专门写一个爬虫，爬取网上公开的代理ip，检测后全部保存起来。  
对于第二种情况，可以在每次请求后随机间隔几秒再进行下一次请求。有些有逻辑漏洞的网站，可以通过请求几次，退出登录，重新登录来绕过。也可以采用多账号策略。

#### **动态页面的反爬虫**

上述的几种情况大多都是出现在静态页面，还有一部分网站，我们需要爬取的数据是通过ajax请求得到，或者通过JavaScript生成的。首先用Fiddler对网络请求进行分析。如果能够找到ajax请求返回的目标数据，就直接模拟ajax请求。  
但是有些网站把ajax请求的所有参数全部加密了。这种情况可以初步分析JS文件看是否能找到解密方法进行解密，有的网站会进行JS混淆，这种情况可以尝试单独执行js解密会比较困难，一般做法是使用selenium模拟人为操作爬取数据。  
用这套框架几乎能绕过大多数的反爬虫。利selenium+phantomJS能干很多事情，例如识别点触式（12306）或者滑动式的验证码，对页面表单进行暴力破解等。

#### **通过前端CSS和HTML标签进行干扰混淆进行反爬**

前端通过CSS和HTML标签进行干扰混淆关键数据，防止爬虫轻易获取数据  
**1 . font-face，自定义字体干扰**  
如列子：汽车之家论帖子，猫眼电影电影评分 <http://maoyan.com/films/342601>  
破解思路： 找到ttf字体文件地址，然后下载下来，使用font解析模块包对ttf文件进行解析，可以解析出一个字体编码的集合，与dom里的文字编码进行映射，然后根据编码在ttf里的序号进行映射出中文  
可以使用FontForge/FontCreator工具打开ttf文件进行分析  
**2 . 伪元素隐藏式**  
通过伪元素来显示重要数据内容  
如例子：[汽车之家](https://car.autohome.com.cn/config/series/3170.html)  
破解思路： 找到样式文件，然后根据HTML标签里class名称，匹配出CSS里对应class中content的内容进行替换  
**3 . backgroud-image**  
如例子：美团 价格显示  
通过背景图片的position位置偏移量，显示数字/符号，如：价格，评分等  
根据backgroud-postion值和图片数字进行映射  
**4 . html标签干扰**  
通过在重要数据的标签里加入一些有的没的隐藏内容的标签，干扰数据的获取  
如例子：[全网代理IP](http://www.goubanjia.com/)  
破解思路： 过滤掉干扰混淆的HTML标签，或者只读取有效数据的HTML标签的内容.通过移除干扰标签里有display:none隐藏标签，然后再获取text就不会有干扰的内容了  
**5 . 字体替换**  
去哪儿手机版酒店价格获取，如何通过 css 计算元素的绝对定位  
参考：[去哪儿 m 版酒店价格获取，如何通过 css 计算元素的绝对定位](https://www.v2ex.com/t/354305)  
你会发现一个问题，就是页面展示的数字和html文件里的数字不一致  
1240的价格在html里竟然是4230  
根据css你会发现他们用了一个字体，打开你就发现了一件事  
正常字体是0123456789，在去哪儿官方的字体里被替换成了图片里的  
然后你根据这个做一些对应解析，就可以爬出正确的数据了。

#### **投毒，以及喂毒（蜜罐）**

有些网站搞一些正常的url pattern 的页面 链接做成隐藏链接 正常用户看不到，但是大部分爬虫就爬进去了，然后监控这些页面被访问的ip就可以了，基本都是爬虫  
还有些平台发现爬虫后并不会进行限制封杀，而是给爬虫提供误导的数据，影响他们进行错误的决策，这就是喂毒  
为了防止被投毒喂毒，需要对数据进行抽样校验。另外需要分析其投毒逻辑以尽量避免自己的爬虫中陷阱被识别出来

### 投毒反爬策略

从链接发现下手可以搞一些正常的url pattern 的页面 链接做成隐藏链接 正常用户看不到（比如白底白字，或者被遮盖） 但是大部分爬虫就爬进去了 [然后就对这些ip做惩罚就行了比如www.xxx.com?id=12345](http://xn--ipwww-g31hye02mx7k98nfsggtafa520o6t2bw8rs48c0w7ae83a.xxx.com?id=12345) 大部分id参数是正常页面 把一些id做成只有隐藏链接的假id 监控这些页面被访问的ip 然后收割吧 基本都是爬虫

青铜反爬：请求头上做点基本的检验  
白银反爬：限制访问频率，封 IP，跳图形验证码，infinite redirect loop，需要登录  
黄金反爬：前端 js script 实时计算 parameter 加给请求在后端进行验证  
铂金反爬：翻页使用 ajax ，没有 pagination ，数据以 json 形式在前端异步加载，验证参数不正确随机截断 html  
钻石反爬：异地登录需要手机短信验证，拖动、拼图等各类人机验证码  
王者反爬：我们公司的前端写的代码让你的爬虫在 parse html 阶段的难度上升为 nlp ：）

### 为什么要做分布式爬虫

分布式爬虫通常会有一些教材告诉你，为了爬取效率，需要把爬虫分布式部署到多台机器上。这完全是骗人的。  
分布式作用是：防止对方封IP，以及避免使用代理IP，因为一些实时性要求比较高的爬虫系统，使用代理IP响应时间太长

[干货！分享一个调试 cookies 的东西，只要监控到 cookies 被改写，就自动跳断点](https://github.com/paulirish/break-on-access)

## 算法 排序

### 经典排序算法总结与实现

最简单易懂的Python算法实现，带图示  
[经典排序算法总结与实现](http://wuchong.me/blog/2014/02/09/algorithm-sort-summary/)

### 说一下冒泡排序？

回答技巧:回答冒泡原理，最好能手写，拓展一下其他排序？  
冒泡排序的思想: 每次比较两个相邻的元素, 如果他们的顺序错误就把他们交换位置。

### 冒泡排序 BubbleSort

比较相邻的元素。如果第一个比第二个大，就交换他们两个。  
对第0个到第n-1个数据做同样的工作。这时，最大的数就“浮”到了数组最后的位置上。  
针对所有的元素重复以上的步骤，除了最后一个。  
持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

# 10000个随机数据耗时16s   
def maopao(l):   
 for i in range(1, len(l)):   
 for j in range(len(l) - i):   
 if l[j] > l[j + 1]:   
 l[j], l[j + 1] = l[j + 1], l[j]   
 print(l)

### 选择排序 SelectionSort

在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。  
再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。  
以此类推，直到所有元素均排序完毕。

# 10000个随机数据耗时10s   
def xuanze(l):   
 for i in range(len(l)):   
 for j in range(i, len(l)):   
 if l[i] > l[j]:   
 l[i], l[j] = l[j], l[i]

选择排序优化版

# 10000个随机数据耗时5s   
def xuanze\_youhua(l):   
 for i in range(len(l)):   
 small\_index = i   
 for j in range(i, len(l)):   
 if l[small\_index] > l[j]:   
 small\_index = j   
 l[i], l[small\_index] = l[small\_index], l[i]

### 插入排序

从第一个元素开始，该元素可以认为已经被排序  
取出下一个元素，在已经排序的元素序列中从后向前扫描  
如果被扫描的元素（已排序）大于新元素，将该元素后移一位  
重复步骤3，直到找到已排序的元素小于或者等于新元素的位置  
将新元素插入到该位置后  
重复步骤2~5

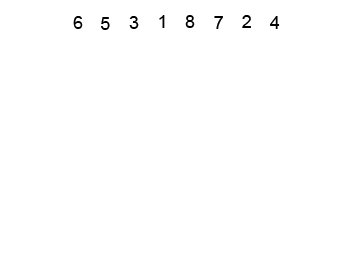
# 是否正确待确认   
# 10000个随机数据耗时10s   
def charu(l):   
 for i in range(1, len(l)):   
 for j in range(i):   
 if l[j] > l[i]:   
 l[j], l[i] = l[i], l[j]   
 print(l)

### 快速排序

快速排序通常明显比同为Ο(n log n)的其他算法更快，因此常被采用，而且快排采用了分治法的思想，所以在很多笔试面试中能经常看到快排的影子。可见掌握快排的重要性。  
从数列中挑出一个元素作为基准数。  
分区过程，将比基准数大的放到右边，小于或等于它的数都放到左边。  
再对左右区间递归执行第二步，直至各区间只有一个数。

# 10000个随机数据排序耗时0.05s   
def quick\_sort(ary):   
 return qsort(ary, 0, len(ary) - 1)   
  
   
 def qsort(ary, left, right):   
 # 快排函数，ary为待排序数组，left为待排序的左边界，right为右边界   
 if left >= right: return ary   
 key = ary[left] # 取最左边的为基准数   
 lp = left # 左指针   
 rp = right # 右指针   
 while lp < rp: # 如果左指针在右指针右边就死循环   
 while ary[rp] >= key and lp < rp: # 拿右指针上的值和key比较, 如果比key大,就向左移指针,直至条件不再成立   
 rp -= 1   
 while ary[lp] <= key and lp < rp:   
 lp += 1   
 ary[lp], ary[rp] = ary[rp], ary[lp] # 对调左右指针上的值   
 # print(ary, 'key:%s lp:%s rp:%s' % (key, lp, rp))   
 ary[left], ary[lp] = ary[lp], ary[left] # 由上条件知道此时左指针值比key小,对调,下次循环基准数更新   
 qsort(ary, left, lp - 1)   
 qsort(ary, rp + 1, right)   
 return ary

### 堆排序 HeapSort



Heapsort-example.gif | center | 350x280

堆排序在 top K 问题中使用比较频繁。堆排序是采用二叉堆的数据结构来实现的，虽然实质上还是一维数组。二叉堆是一个近似完全二叉树 。  
二叉堆具有以下特性：  
父节点的键值总是大于或等于（小于或等于）任何一个子节点的键值。  
每个节点的左右子树都是一个二叉堆（都是最大堆或最小堆）。  
步骤：  
构造最大堆（Build\_Max\_Heap）：若数组下标范围为0~n，考虑到单独一个元素是大根堆，则从下标n/2开始的元素均为大根堆。于是只要从n/2-1开始，向前依次构造大根堆，这样就能保证，构造到某个节点时，它的左右子树都已经是大根堆。  
堆排序（HeapSort）：由于堆是用数组模拟的。得到一个大根堆后，数组内部并不是有序的。因此需要将堆化数组有序化。思想是移除根节点，并做最大堆调整的递归运算。第一次将heap[0]与heap[n-1]交换，再对heap[0...n-2]做最大堆调整。第二次将heap[0]与heap[n-2]交换，再对heap[0...n-3]做最大堆调整。重复该操作直至heap[0]和heap[1]交换。由于每次都是将最大的数并入到后面的有序区间，故操作完后整个数组就是有序的了。  
最大堆调整（Max\_Heapify）：该方法是提供给上述两个过程调用的。目的是将堆的末端子节点作调整，使得子节点永远小于父节点 。

# 堆排序 10000个随机数据排序耗时0.1s   
def heap\_sort(ary):   
 n = len(ary)   
 first = int(n / 2 - 1) # 最后一个非叶子节点   
 for start in range(first, -1, -1): # 倒序遍历,构造大根堆, 整个数组内部子树都已经是大根堆,但并不是有序的   
 max\_heapify(ary, start, n - 1)   
 print(ary)   
 for end in range(n - 1, 0, -1): # 堆排，将大根堆转换成有序数组;思想是移除根节点，并做最大堆调整的递归运算。 每次循环调整边界-1   
 ary[end], ary[0] = ary[0], ary[end] # 用最后一个元素和第一个元素对调   
 max\_heapify(ary, 0, end - 1)   
 print(ary)   
 return ary   
  
 def max\_heapify(ary, start, end):   
 """   
 子树(三个元素)最大堆调整：将堆的末端子节点作调整，使得子节点永远小于父节点   
 :param start: 为当前需要调整最大堆的位置   
 :param end: 调整边界   
 """   
 root = start   
 while True:   
 child = root \* 2 + 1 # 调整节点的子节点   
 if child > end: break   
 if child + 1 <= end and ary[child] < ary[child + 1]:   
 child = child + 1 # 取较大的子节点   
 if ary[root] < ary[child]: # 较大的子节点成为父节点   
 ary[root], ary[child] = ary[child], ary[root] # 交换   
 root = child   
 else:   
 break

### 二分查找

def binary\_search(li,find):   
 low = 0   
 high = len(li)-1 # 需要减一否则会下标越界   
 while low <= high:   
 middle = (low + high) /2   
 if li[middle] == find :   
 return middle   
 elif li[middle] > find:   
 high = middle - 1   
 elif li[middle] < find:   
 low = middle + 1   
 return -1   
if \_\_name\_\_ == '\_\_main\_\_':   
 li = [x for x in range(1,101)]   
 for x in range(102):   
 print binary\_search(li,x)

## 编程题

### and or not运算问题

v = 1 and 2 or 3 and 4   
print(v)   
  
 # >>> 2

参考：[Python：and和or的特殊性质](http://blog.csdn.net/betabin/article/details/8568804)  
总结：对python而言  
其一, 在不加括号时候, and优先级大于or  
其二, x or y 的值只可能是x或y. x为真就是x, x为假就是y  
第三, x and y 的值只可能是x或y. x为真就是y, x为假就是x  
显然,对于, 1 or 5 and 4: 先算5 and 4, 5为真, 值为4. 再算1 or 4, 1 为真,值为1  
对于, (1 or 5) and 4: 先算1 or 5, 1为真, 值为1. 再算1 and 4, 1为真,值为4

### 排序问题

list对象 alist [{'name':'a','age':20},{'name':'b','age':30},{'name':'c','age':25}]，请按alist中元素的 age 由大到小排序；

alist = [{'name': 'a', 'age': 20}, {'name': 'b', 'age': 30}, {'name': 'c', 'age': 25}]   
alist.sort(key=lambda x: -x["age"])   
print(alist)

### 打乱一个排好序的 list 对象 alist；

from random import shuffle   
alist = range(10)   
shuffle(alist)   
print("shuffle", alist)

### 简单实现一个stack；

class Stack:   
 def \_\_init\_\_(self):   
 self.values = []   
 def push(self, o):   
 self.values.append(o)   
 def pop(self):   
 return self.values.pop()   
s = Stack()   
s.push(1)   
s.push(2)   
assert s.pop() == 2   
assert s.pop() == 1   
s.push(3)   
s.push(4)   
assert s.pop() == 4   
assert s.pop() == 3

### 输入某年某月某日，判断这一天是这一年的第几天？

from datetime import datetime   
def day\_of\_year(year, month, day):   
 return (datetime(year, month, day) - datetime(year, 1, 1)).days + 1   
assert day\_of\_year(2014, 1, 10) == 10

### 数据类型转换

将字符串："k:1|k1:2|k2:3|k3:4"，处理成python字典：{k:1, k1:2, ... }

def string\_to\_dict(string):   
 d = {}   
 for kv in string.split("|"):   
 k, v = kv.split(":")   
 if v.isdigit():   
 v = int(v)   
 d[k] = v   
 return d   
string = "k:1|k1:2|k2:3|k3:4"   
print(string\_to\_dict(string))   
string2 = "k:1"   
print(string\_to\_dict(string2))

### 台阶问题/斐波纳挈

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。（用一句代码写斐波那契数列）

fib = lambda n: n if n <= 2 else fib(n - 1) + fib(n - 2)

### 变态台阶问题

一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

fib = lambda n: n if n < 2 else 2 \* fib(n - 1)

### 找零算法（动态规划）

我购物花费28元，支付100元纸币，我最少会被找零多少张纸币？

def coinChange(values, money, coinsUsed):   
# values T[1:n]数组   
# valuesCounts 钱币对应的种类数   
# money 找出来的总钱数   
# coinsUsed 对应于目前钱币总数i所使用的硬币数目   
for cents in range(1, money + 1):   
 minCoins = cents # 从第一个开始到money的所有情况初始   
 for value in values:   
 if value <= cents:   
 temp = coinsUsed[cents - value] + 1   
 if temp < minCoins:   
 minCoins = temp   
 coinsUsed[cents] = minCoins   
 print('面值为：{0} 的最小硬币数目为：{1} '.format(cents, coinsUsed[cents]))   
  
 if \_\_name\_\_ == '\_\_main\_\_':   
values = [25, 21, 10, 5, 1]   
money = 63   
coinsUsed = {i: 0 for i in range(money + 1)}   
coinChange(values, money, coinsUsed)