面试技巧

基础概念

- 对象与变量
  - a='object'
  - 其中a为变量，指向对象'object'
  - 字符串'object'为对象，创建对象，分配内存地址，每个对象都包含了两个头部信息，一个是类型标志符，另一个是引用计数器，类型属于对象，而非变量
- is 与 == 的区别
  - is 比较的是两个实例对象是不是完全相同，它们是不是同一个对象，占用的内存地址是否相同。即is比较两个条件：1.内容相同。2.内存中地址相同
  - == 是比较两个对象的内容是否相等，即两个对象的“值”是否相等，不管两者在内存中的引用地址是否一
- 鸭子类型
- monkey patch
- 自省
- 列表或者字典推导式
- GIL解释器锁
  - 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.对于io密集型任务，python的多线程起到作用，但对于cpu密集型任务，python的多线程几乎占不到任何优势，还有可能因为争夺资源而变慢
- 高频考点
  - 作用域

```
#以下输出没有问题
  b = 6
  def f2(a):
      print(a)
      print(b)
      # b = 9
  f2(3)

# 编译时由于b赋值，会认为b为局部变量，在局部变量中查找b，而在print(b)之前没有定义b,所以会报错
b = 6
def f2(a):
    print(a)
    print(b)
    b = 9
f2(3)
```

  - 全局变量与非局部变量
  - global
  - nonlocal
  - 闭包
  - 概念：闭包指延伸了作用域的函数，其中包含函数定义体中引用、但是不在定义体中定义的非全局变量，关键是它能访问定义体之外定义的非全局变量
  - 代码

```
  def make_averager():
      series = []

     def averager(new_value):
         series.append(new_value)
         total = sum(series)
         return total / len(series)

     return averager


  avg = make_averager()
  avg(10)
  avg(11)
  print(avg(12))
```

  - 装饰器
  - 不带参数的装饰器
  - 异常处理

```
  # 异常处理
  try:
      print()
  except Exception as e:
      print(e)
  else:
      print('chen gong')
  finally:
      print('finally')

  #自定义异常
  class MyException(Exception):
    pass


  try:
      raise MyException('my exception')
  except Exception as e:
      print(e)
```

- 上下文管理
- 自定义上下文管理器

```
class MyTest(object):
    def __init__(self, x, y):
        print('初始化赋值')
        self._x = x
        self._y = y

    def __enter__(self):
        print('进入上下文管理器')
        # 返回值会赋值给as后面的变量
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if not exc_type:
            print('程序没问题')
        else:
            print('程序有问题，如果你能你看懂，问题如下：')
            print('Type: ', exc_type)
            print('Value:', exc_val)
            print('TreacBack:', exc_tb)

        print('退出上下文管理器')
        # 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

    def div(self):
        print('执行功能函数')
        return self._x / self._y


with MyTest(4, 0) as f:
    print(f.div())</code></pre>
```

- 使用contextmanager

```
from contextlib import contextmanager

def tes():
    print('contextmanager')

class Call(object):
    def __init__(self, func):
        self.__func = func

    def do(self):
        self.__func()

@contextmanager
def wrapper(func):
    print('Call %s()' % func.__name__)
    obj = Call(func)
    yield obj
    print('%s() End' % func.__name__)

'''
执行流程：
1, with语句先执行wrapper函数里面yield之前的语句
2. yield调用会执行with语句内部的所有语句c.do()
3. 最后执行yield之后的语句
'''

with wrapper(tes) as c:
    c.do()</code></pre>
```

- 迭代器(Iterator)：
- 可以被next()函数调用并不断返回下一个值的对象称为迭代器
- Iterator 类必须实现 **next** 方法，Iterator.**iter** 方法直接返回实例本身
- 代码：

```
class IterTest(object):


    def __init__(self, lis):
        self.lis = lis
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        try:
            word = self.lis[self.index]
        except StopIteration as e:
            print('end')
        self.index += 1
        return word



te = IterTest(lis)
print(next(te))
print(next(te))
```

- 可迭代对象(Iterable)
- 具体的 Iterable.**iter** 方法应该返回一个 Iterator 实例
- 直接作用于for循环的数据类型有以下几种：

  a. 一类是集合数据类型，如list、tuple、dict、set、str等；

  b. 一类是generator，包括生成器和带yield的generator function。
- 集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象
- 可以直接作用于for循环的对象统称为可迭代对象
- 生成器

```
def gen():
i = 0
while i < 3:
    yield i
    i += 1
```

```
do = gen()  # 这一步不可缺少，不可以写成next(gen())
print(next(do))
print(next(do))
print(next(do))
print(next(do))</code></pre>
```

- 协程
- 先说明生成器
  - 含yield的生成器中使用.send() 方法允许在客户代码和生成器之间双向交换数据。
  - .throw()前者的作用是让调用方抛出异常
  - .close()后者的作用是终止生成器
- 生成器实现生产者与消费者模型

```
def consumer():
    print('调用')
    r = 10
    while True:
        print('[CONSUMER] Consuming yield r %s...' % r)
        n = yield r
        print(n)
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        r +=10

def produce(c):
    c.send(None)
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return r: %s' % r)
    c.close()

c = consumer()
produce(c)
```

- 原生协程 yield from
  - 子生成器
  - 委派生成器
  - 调用方
  - 代码

```
# 子生成器
def average():
    total = 0
    while True:
        term = yield
        if term is None:
            break
        total += term
    return total


# 委派生成器
def grouper(results,key):
    results[key]=0
    # 只循环一次
    while True:
        print(results[key])
        # 子生成器调用结束后将结果赋值给results[key]
        results[key] = yield from average()


data = [1, 2, 3, 4, 5]

# 客户端代码，即调用方
def main(data):
    results = dict()
    group = grouper(results,'to')
    next(group)
    for value in data:
        print(value)
        group.send(value)
    group.send(None)
    print(results)


main(data)</code></pre></li>
```

- collections等内置模块
- CPython解释器
- GIL
- 内存管理

- 不可重复的数据结构
- 深拷贝与浅拷贝
  - 浅拷贝
  - 直接赋值,默认浅拷贝传递对象的引用而已,原始列表改变，被赋值的b也会做相同的改变
  - copy.copy()浅拷贝，没有拷贝子对象，所以原始数据改变，子对象会改变
  - 深拷贝
  - 深拷贝，包含对象里面的自对象的拷贝，所以原始对象的改变不会造成深拷贝里任何子元素的改变
- 如何传参

    - 传递的是对象，而不是变量或者指针
    - 对象有两种,“可更改”（mutable）与“不可更改”（immutable）对象。在python中，strings, tuples, 和numbers是不可更改的对象，而 list, dict, set 等则是可以修改的对象。(这就是这个问题的重点)
    - 当一个引用传递给函数的时候,函数自动复制一份引用,这个函数里的引用和外边的引用没有半毛关系了.所以第一个例子里函数把引用指向了一个不可变对象,当函数返回的时候,外面的引用没半毛感觉.而第二个例子就不一样了,函数内的引用指向的是可变对象,对它的操作就和定位了指针地址一样,在内存里进行修改.

- 判断对象是否存在属性或者方法
    - hasattr
    - getattr
    - setattr

- 单例模式 了解 super指定起始查找点，顺着MRO查找第二个参数 new 拦截类的实例化 init创建实例后初始化值 call实现可调用

- 回调 w eb start_response

- 函数式编程
    - map
    - filter
    - reduce
    - 闭包
    - 绑定外部作用域的变量的函数
    - 即使程序离开外部作用域，如果闭包仍然可见，绑定变量不会销毁
    - 每次运行外部函数都会重新创建闭包

- Python2与Python3的区别

数据结构与算法

- 常考内容 要求手写
    - 常用内置结构
        - list
        - tuple
        - set
        - dict
        - collections orderdict()

- 常考数据结构
    - 连表
    - 二叉树
    - 栈
    - 队列

- 常考算法
    - 二分查找
    - 哈希查找
    - 分块查找
    - 顺序查找
    - 堆排序 重点
    - 插入排序
    - 冒泡排序
    - shell排序
    - 快速排序 重点
    - 基数排序
    - 二路并归排序 重点

- 编程范式
    - 面向对象
    - 装饰器
    - 单例模式

Linux

- 常用命令
    - 帮助
      man tar
      tar --help
      tldr

```
chown chmod chgrp
ls/rm/cd/cp/mv/touch/rename/ln(软连接与硬链接)
```

```
lcate find grep
find . -name '*.pyc' -delete
```

vi/nano
cat/head/tail
more/less

进程相关

ps
kill 执行的原理 发送信号

top/htop

内存
free

网络工具
ifconfig
lsof/netstat 查看端口
ssh/scp tcpdump抓包

useradd/usermod
groupadd/groupmod

线程 进程 并发 并行 协程 线程安全

- 线程安全
  - GIL解释器锁
  - 线程同步 如何保证线程同步
  - 互斥索
  - 信号量 参数value为1
  - 事件信号
- 进程间的通信
  - 管道pipe
  - 信号
  - 消息队列
  - 共享内存
  - 信号量
  - socket：web应用这种方式
- 多线程使用
- 多进程使用
  - uwsgi

内存管理

- 分页地址
- 分段机制
- 虚拟内存
- 垃圾回收
  - 引用计算为主 标记清除和分代回收为辅 解决循环引用

网络编程 TCP UDP HTTP

- 网络协议TCP和UDP
  - 浏览器输入一个url中间经历的过程:
    i. DNS查询
    ii. TCP握手 wireshark抓包更直观 重点有两点(状态与发包内容) 手画图
    iii. HTTP请求 携带的信息包
    iv. 反向代理 Nginx
    v. uwsgi 服务器
    vi. web app （flask框架）
    vii. TCP挥手 为什么是四次而不是三次或者五次
  - TCP与UDP的区别
  - 面向连接 可靠地 基于字节流
  - 无连接 不可靠 面向报文
- DNS查询
  - 查询过程：缓存查询与域名服务器查询
    - 浏览器缓存:当用户通过浏览器访问某域名时，浏览器首先会在自己的缓存中查找是否有该域名对应的IP地址（若曾经访问过该域名且没有清空缓存便存在）；
    - 系统缓存:当浏览器缓存中无域名对应IP则会自动检查用户计算机系统Hosts文件DNS缓存是否有该域名对应IP;
    - 路由器缓存:当浏览器及系统缓存中均无域名对应IP则进入路由器缓存中检查，以上三步均为客服端的DNS缓存；
    - ISP（互联网服务提供商）DNS缓存:当在用户客服端查找不到域名对应IP地址，则将进入ISP DNS缓存中进行查询。比如你用的是电信的网络，则会进入电信的DNS缓存服务器中进行查找；
    - 根域名服务器:当以上均未完成，则进入根服务器进行查询。全球仅有13台根域名服务器，1个主根域名服务器，其余12为辅根域名服务器。根域名收到请求后会查看区域文件记录，若无则将其管辖范围内顶级域名（如.com）服务器IP告诉本
    - 顶级域名服务器:顶级域名服务器收到请求后查看区域文件记录，若无则将其管辖范围内主域名服务器的IP地址告诉本地DNS服务器；
    - 主域名服务器:主域名服务器接受到请求后查询自己的缓存，如果没有则进入下一级域名服务器进行查找，并重复该步骤直至找到正确纪录；

        - 保存结果至缓存:本地域名服务器把返回的结果保存到缓存，以备下一次使用，同时将该结果反馈给客户端，客户端通过这个IP地址与web服务器建立链接。
    - 为什么使用UDP协议进行DNS查询
    - 网络资源角度：基于UDP的DNS协议只要一个请求、一个应答就可以了，而使用基于

- TCP协议

    - 面向连接 可靠地 基于字节流
    - TCP三次握手的详细过程：
    - 位码即tcp标志位，有6种标示：SYN(synchronous建立联机) ACK(acknowledgement 确认) PSH(push传送) FIN(finish结束) RST(reset重置) URG(urgent紧急)Sequence number(顺序号码) Acknowledge number(确认号码)
    - 不要将确认序号Ack与标志位中的ACK搞混了。
    - 确认方Ack=发起方Req+1，两端配对
    - 手绘握手连接图

    i. 为什么是三次握手而不是两次或者四次？

        - 为什么不是两次握手：
        - 关键点是TCP是全双工通信，即客户端在给服务器端发送信息的同时，服务器端也可以给客户端发送信息，简化模型为：
            - 第一次握手： A给B打电话说，你可以听到我说话吗？(把SYN比作询问)
            - 第二次握手： B收到了A的信息，然后对A说： 我可以听得到你说话啊(把ACK比作回答)，你能听得到我说话吗？
            - 第三次握手： A收到了B的信息，然后说可以的，我要给你发信息啦
        - 补充说明：而半双工的意思是A可以给B发，B也可以给A发，但是A在给B发的时候，B不能给A发，即不同时，为半双工。单工为只能A给B发，B不能给A发； 或者是只能B给A发，不能A给B发。
        - 为什么不是四次握手：
        - 三次可以完成的事情，四次浪费网络资源

    ii. TCP为什么是四次挥手，而不是三次挥手或者五次挥手？

    - 关键点还是在于TCP是全双工通信，简化模型为：
        - 第一次挥手：A:“喂，我不说了 (FIN)。”，此时A->FIN_WAIT1，即A处于等待结束状态，需要B的回复
        - 第二次挥手：B:“我知道了(ACK)。等下，上一句还没说完。Balabala.....（传输数据）”，B->CLOSE_WAIT | A->FIN_WAIT2，此时B在传输数据等待结束，同时回复A已收信息，
        - 第三次挥手： B:"好了，说完了，我也不说了（FIN）。"B->LAST_ACK
        - 第四次挥手：A:"我知道了（ACK）。"A->TIME_WAIT | B->CLOSED
        - A等待2MSL,保证B收到了消息,否则重说一次"我知道了",A->CLOSED
    - 补充说明：MSL 是Maximum Segment Lifetime英文的缩写，中文可以译为“报文最大生存时间”，他是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。RFC 793中规定了MSL为2分钟，实际应用中常用的是30秒，1分钟和2分钟等

    i. 四次挥手释放连接时，等待2MSL的意义？

        - 第一，为了保证A发送的最有一个ACK报文段能够到达B。这个ACK报文段有可能丢失，因而使处在LAST-ACK状态的B收不到对已发送的FIN和ACK 报文段的确认。B会超时重传这个FIN和ACK报文段，而A就能在2MSL时间内收到这个重传的ACK+FIN报文段。接着A重传一次确认。
        - 第二，就是防止上面提到的已失效的连接请求报文段出现在本连接中，A在发送完最有一个ACK报文段后，再经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失

    ii. 为什么在四次挥手的过程中一般都是客户端先发起呢?

    - 关键点在于：TCP挥手时要等2MSL
    - 现象：在调试客户端和服务器端（使用TCP套接字）的代码时我发现，如果先结束服务器端后结束客户端，紧接着再重启服务器端就会出现绑定失败的错误 OSError: [Errno 98] Address already in use 等待一段时间后大概一分钟左右就能正常重启服务器端
    - 分析：
    - 因为TCP是全双工的，所以客户端和服务端都可以先进行挥手。在socket编程中哪一方先执行close()操作，哪一方则先进行挥手（发送FIN包）。
    - 以客户端先挥手为例，在TCP处于TIME_WAIT状态时，客户端从TIME_WAIT状态到CLOSED状态需要2MSL，因为客户端要确定服务端收到了ACK，如果服务端没收到ACK，客户端则一定会在2MSL时间内再收到一次FIN。而在socket编程中客户端一般不需要绑定，而服务器端一般都要绑定，如果先结束服务器端则是服务器端先进行挥手操作，那么服务器端从TIME_WAIT到CLOSED状态则需要2MSL。这段时间服务器端绑定的端口号被占用了，套接字不会释放。所以这段时间重启了服务器之后，会出现绑定失败的错误OSError: [Errno 98] Address already in use

    i. tcp长连接和短连接

    - TCP长/短连接的优点和缺点:
    - 长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。
    - client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损,如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。
    - 短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。
    - 但如果客户请求频繁，将在TCP的建立和关闭操作上浪费时间和带宽。
    - TCP长/短连接的应用场景：
    - 数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费
    - 而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源
    - 如何区分：请求头中有：Connection: keep-alive

- HTTP协议

    - pip install httpie
    - curl
    - HTTP协议有哪些部分组成 使用抓包工具去查看和理解
    - HTTP请求
        - 状态行
        - 请求头
        - 消息主体
    - HTTP响应
        - 状态行
        - 响应头
        - 响应正文
    - HTTP响应状态码 五中类型
    - 幂等性 GET
    - 长连接
    - 为什么需要长连接
    - 如何区分短还是长连接

        - 1. content-Length 携带报文长度
        - 2. Transfer-Encoding:chunked
    - cookie与session的区别，识别用户
    - 前后端分离如何识别用户状态
    - 本节重点内容：
    - 请求和响应的组成
    - 常用HTTP方法和幂等性
    - 长连接 session cookie

- socket编程原理
    - TCP编程

- 并发编程IO多路复用
    - IO多路复用

- 并发网络库

- 异步框架

- WSGI与web框架 重点
    - WSGI 是什么 解决什么问题
    - uWSGI uWSGI是一个web服务器，实现了WSGI协议，uwsgi协议，http协议等。
    - uwsgi 是一种协议
    - uwsgi与nginx之间是通过套接字socket就行通信的，他们之间的交互可以理解为是两个进程之间的交互，而它通过实现wsgi协议，可以与python写的应用程序进行交互
    - 无论多么复杂的Web应用程序，入口都是一个WSGI处理函数。HTTP请求的所有输入信息都可以通过environ获得，HTTP响应的输出都可以通过start_response()加上函数返回值作为Body
    - web框架的组成(基础重点) 尝试自己编写一个web服务器
    - 中间件 用于请求之前和请求之后做一些处理
    - 路由 表单验证 权限认证 ORM 视图函数 模板渲染 序列化
    - redis RESTful

- RESTful
    - 前后端分离意义及方式
    - 什么是RESTful 怎么设计RESTfu
    - 表现层状态转移 其中表现层指资源的表现形式，图片 文本；
    - 资源(Resource) 使用URI指向的一个实体
    - 一种以资源为中心的web软件架构风格
    - 如何理解资源

- RESTful API
    - 三部分组成
    - HTTP方法GET、DELETE、POST和PUT操作资源
    - json
    - RESTful API插件

- 如何设计RESTful API
    - 使用名字而不是动词，名词使用复数而不是单数
    - 同一资源可以有一组操作
        - GET http://[hostname]/api/users 检索用户列表
        - GET http://[hostname]/api/users/[id] 查询单个用户
        - POST http://[hostname]/api/users 创建用户
        - PUT http://[hostname]/api/users/[id] 修改用户
        - DELETE http://[hostname]/api/users/[id] 删除用户
    - 定义返回状态码

- cookie
    i. 是一个非常具体的东西，指的就是浏览器里面能永久存储的一种数据,cookie由服务器生成，发送给浏览器，浏览器把cookie以kv形式保存到某个目录下的文本文件内，下一次请求同一网站时会把该cookie发送给服务器。由于cookie是存在客户端上的，所以浏览器加入了一些限制确保cookie不会被恶意使用，同时不会占据太多磁盘空间，所以每个域的cookie数量是有限的。

- session
    i. 服务器使用session把用户的信息临时保存在了服务器上，用户离开网站后session会被销毁。这种用户信息存储方式相对cookie来说更安全
    - 缺点：
    i. Seesion：每次认证用户发起请求时，服务器需要去创建一个记录来存储信息。当越来越多的用户发请求时，内存的开销也会不断增加。
    ii. 可扩展性：在服务端的内存中使用Seesion存储登录信息，伴随而来的是可扩展性问题，如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失

- token 签名加密
    i. Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。
    ii. Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。
    iii. 使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

设计模式

- 装饰器模式
- 观察者模式
- 单例模式
- 工厂方法

- 抽象工厂
- 代理模式

数据库 MySQL MySQL重点有事物 存储字段

- 事务的原理，特性，事物并发控制
    - 一系列SQL的集合 要么全部成功 要么全部失败
    - ACID四个特性(必须记住)
    - 原子性(Atomicity)：一个事务中所有的操作全部完成或者全部失败
    - 一致性(Consistency)：事务开始和结束之后数据完整性没有被破坏
    - 隔离性(Isolation)：允许多个事务同时对数据库修改和读写
    - 持久性(Durability)：事务结束后，修改时永久的不会丢失
    - 如果不对事务进行并发控制，可能会产生四种异常情况
    - 幻读(phantom read)：一个事务第二次查出第一次没有的结果
    - 非重复读(nonrepeatable read):一个事务重复读两次得到不同的结果
    - 脏读(dirty read)： 一个事务读物到另一个事务没有提交的修改
    - 修改丢失(lost update)：并发写入造成其中一些修改丢失
    - 为了解决并发控制异常，定义了4中事务隔离级别
    - 读未提交(read uncommitted)：别的事务可以读取到未提交改变
    - 读已提交(read uncommit):只能读取已经提交的数据
    - 可重复读(repeatable read)：同一个事务先后查询结果一样 (MySQL innoDB默认实现可重复读级别)
    - 串行化(serializable):事务完成串行化的执行，隔离级别最高，执行化效率最低
    - 如何解决并发场景下，写入数据库会有数据重复问题？
    - 使用数据库的唯一索引
    - 使用队列异步写入
    - 使用redis等实现分布式锁
- MySQL死锁的原因与解决办法
- 悲观锁
    - 悲观锁是先获取锁在进行操作。一锁二查三更新(select for update)
- 乐观锁
    - 乐观锁先修改，更新的时候发现数据已经变了就回滚(check and set)
- 使用根据响应速度 冲突频率 重试代价来判断使用哪种
- MySQL常用数据类型
    - 字符串类型
    - char
    - varchar
    - text
    - 数值
    - tinyint
    - 时间与日期
- innoDB 与MyISAM区别
- 常用字段 含义 区别
- 常用数据库引擎之间的区别
  innoDB支持事物安全与外键
- MySQL索引原理及优化
    - 索引是数据表中一个或者多个列进行排序的数据结构
    - B-Tree
    - 线性查找：一个个找，实现简单；缺点：太慢
    - 二分查找：有序，简单；缺点：要求有序的，插入特别慢
    - HASH：查询快；缺点：占用空间，不太适合存储大规模数据
    - 二叉树查找：插入与查询很快；缺点：无法存超大规模数据，复杂度退化为线性查找
    - 二叉平衡树：解决bst退化的问题，树是平衡的；节点非常多的时候，依然树高很高
    - B Tree多路平衡查找树：多路平衡查找树
    - MySQL 实际使用B +Tree作为索引的数据结构:
        - 只在叶子节点带有指向记录的指针(为什么？ 可以增加树的度？)
        - 叶子节点通过指针相连。为什么？实现范围查询
    - 创建索引类型
    - 普通索引
    - 唯一索引
    - 多列索引
    - 主键索引
    - 全文索引 innoDB不支持
    - 什么时候创建索引？
    i. 经常用作查询条件的字段(where条件)
    ii. 经常用作表连接的字段
    iii. 经常出现在order by, group by之后的字段
    - 创建索引需要注意什么？
    - 非空字段 NOT NULL MySQL很难对空值做查询优化
    - 区分度高，离散度大，作为索引的字段值尽量不要有大量相同值
    - 索引的长度不要太长(比较耗费时间)
    - 索引什么时候失效？
    i. 以%开头的like语句，模糊搜索

    ii. 出现隐士类型转换(Python 这种动态语言查询中需要的注意)
    iii. 没有满足最左前缀原则(想想为什么是最左匹配？ 多列索引时以元组作为索引，缺失时导致无法比较)
    - 总结：以上索引失效的最终原因为key无法比较
    - 聚集索引和非聚集索引
    i. 聚集还是非聚集指的是B+ Tree叶节点存的是指针还是数据记录
    ii. myisam索引和数据分离，使用的是非聚集索引
    iii. innoDB数据文件是索引文件，主键索引就是聚集索引
    - 如何排查慢查询？
    i. slow _query_log_file 开启并查询慢查询日志
    ii. 通过explain排查索引问题
    iii. 调整数据修改索引，业务代码层限制不合理访问
    - 索引的原理 类型 结构
    - 创建索引的注意事项 使用原则
    - 如何排查和消除慢查询

- SQL语句编写常考题 常用连接为重点 掌握程度 需要了解语句含义 同时可以手写

    - 内连接：两个表都存在匹配时，才会返回匹配行
    - inner join
    - select A.id as a_id,B.id as b_id,A.val as a_val,B.val as b_val from A inner join B on A.id=B.id;
    - 外连接：返回一个表的行，即使另一个没有匹配
    - 左连接
        - 返回左表中的所有记录，即使右表中没有任何满足匹配条件的记录
        - select A.id as a_id,B.id as b_id,A.val as a_val,B.val as b_val from A left join B on A.id=B.id;
    - 右连接
        - 返回右表中的所有记录，即使左表中没有任何满足匹配条件的记录
        - select A.id as a_id,B.id as b_id,A.val as a_val,B.val as b_val from A right join B on A.id=B.id;
    - 全连接：只要某一个表存在匹配就返回
    - 只要某一个表存在匹配，就返回，MySQL不支持，可以使用leftjoin和union 和right join联合使用来模拟
    - select A.id as a_id,B.id as b_id,A.val as a_val,B.val as b_val from A left join B on A.id=B.id union select A.id as a_id,B.id as b_id,A.val as a_val,B.val as b_val from A right join B on A.id=B.id;

Redis

- 为什么使用缓存？使用场景？

    i. 缓解关系数据库并发访问的压力（热点数据）
    ii. 减少响应时间：内存的IO速度比磁盘快
    iii. 提升吞吐量：Redis等内存数据库单机片就可以支撑很大并发

- redis与memcached的区别

    - 数据存储类型
    - 网络IO模型
    - 持久化：RDB AOF

- redis的常用数据类型，使用场景

    - string 字符串 ：用来实现简单的KV键值对存储，比如计数器
    - list链表：实现双向链表，比如用户的关注 粉丝列表
    - hash哈希表：用来存储彼此相关信息的键值对
    - set集合：存储不重复元素，比如用户的关注
    - sorted set 有序集合：实时信息排行榜

- redis 内置实现 针对中高级工程师 C语言实现

    - string：整数或者sds（simply dynamic string）
    - list：ziplist 或者double linked list
    - hash：ziplist或者hashtable
    - set：intset或者hashtable
    - sortedset:skiplist 跳跃表

- redis两种持久化

    - 快照方式RDB：把数据快照放在磁盘二进制文件中
    - AOF(append only file):每一个写命令追加到appendonly.aof中
    - 可以通过修改redis配置实现

- redis事务

    - 概念：将多个请求打包，一次性、按序执行多个命令的机制
    - redis 通过MULTI EXEC WATCH 等命令实现事务

- redis如何实现分布式锁？

    - 使用setnx实现加锁，可以通过expire添加超时时间
    - 锁的value值可以使用一个随机的UUID或者特定的命令
    - 释放锁的时候，通过UUID判断时候是该锁，是则执行delete释放锁

- 使用缓存的模式？

    i. cache aside：同时更新缓存和数据库，先从缓存获取数据，不存在则去数据库取，然后再设置缓存
    - 针对并发写操作 一般先更新数据库然后再删除缓存，在下一次读取时缓存没有则去数据读取，防止出现脏数据
    i. read/w rite through(了解)：
    ii. w rite behind caching(了解)：

- 缓存使用问题：数据一致性问题；缓存穿透 击穿 雪崩问题
    - 缓存穿透问题
    - 由于大量缓存查不到就去数据库取值，数据库也没有要查的数据
    - 解决方法：对于没查到的返回None的数据也缓存
    - 插入数据的时候删除响应的缓存或者设置较短的超时时间
    - 缓存击穿问题：非常热点的数据key过期，大量请求达到后端数据库
    - 热点数据key失效导致大量请求打到数据库增加数据库压力
    - 分布式锁：获取锁的线程从数据库拉数据更新缓存，其他线程等待
    - 异步后台更新：后台任务针对过期的key自动刷新
    - 缓存雪崩问题：缓存不可用或者大量缓存key同时失效，大量请求直接打到数据库
    - 多级缓存：不同级别的key设置不同的超时时间
    - 随机超时：key的超时时间随机设置，防止同时超时
    - 架构层：提升系统可用性，监控 报警完善
- MySQL 练习题
    - 为什么MySQL数据库的主键使用自增的整数比较好？
    - 使用UUID可以吗？为什么？
    - 如果是分布式系统下我们怎么生成数据库的自增id呢？
- Redis 练习题
    - 基于redis编写代码实现一个简单的分布式锁
    - 要求：支持超时参数
    - 如果redis单节点宕机了，如何处理？

web安全

- HTTPS 与HTTP的区别 wireshark抓包
- HTTPS
    - 对称加密
    - 非对称加密
    - HTTPS通信过程
- SQL注入
- XSS
- CSRF

系统设计

- 考点
    - 什么是系统设计？
        i. 系统设计师一个定义系统架构、模块、接口和数据满足特定需求的过程
        ii. 微服务
    - 系统设计的难点？
        i. 需要具备相关领域 算法的经验，有一定的架构设计能力
        ii. 熟悉后端技术组件，比如消息队列 缓存 数据库 框架
        iii. 具备文档撰写 流程图 架构设计 编码能力
    - 系统设计如何回答？
    - 场景和限制条件 使用场景，用户估计多少，峰值qps 平均qps
    - 数据存储系统 按需求设计数据表 需要哪些字段 数据库选型 索引设计
    - 设计算法相关模块 需要哪些接口 使用什么算法或模型 不同实现之间的优劣对比
    - 深入问题
    - qps高了如何处理
    - 数据扩容
    - 故障如何处理
    - 如何设计和和实现一个短网址系统？
    - 将长网址转换为短网址
    - 系统设计需要掌握的哪些知识？
    - 如何设计和实现一个后端系统服务的设计？
    - 如何设计一个秒杀系统

面试

- 网上搜索意向公司的面试题
- 多刷题
- 了解技术重点