Py2 VS Py3

- print成为了函数，python2是关键字
- 不再有unicode对象，默认str就是unicode
- python3除号返回浮点数
- 没有了long类型
- xrange不存在，range替代了xrange
- 可以使用中文定义函数名变量名
- 高级解包 和*解包
- 限定关键字参数 *后的变量必须加入名字=值
- raise from
- iteritems移除变成items()
- yield from 链接子生成器
- asyncio,async/await原生协程支持异步编程
- 新增enum,mock,ipaddress,concurrent.futures,asyncio urllib，selector
  - 不同枚举类间不能进行比较
  - 同一枚举类间只能进行相等的比较
  - 枚举类的使用(编号默认从1开始)
  - 为了避免枚举类中相同枚举值的出现，可以使用@unique装饰枚举类

```python
#枚举的注意事项
from enum import Enum

class COLOR(Enum):
    YELLOW=1
#YELLOW=2#会报错
    GREEN=1#不会报错,GREEN可以看作是YELLOW的别名
    BLACK=3
    RED=4
print(COLOR.GREEN)#COLOR.YELLOW,还是会打印出YELLOW
for i in COLOR:#遍历一下COLOR并不会有GREEN
    print(i)
#COLOR.YELLOW\nCOLOR.BLACK\nCOLOR.RED\n怎么把别名遍历出来
for i in COLOR.__members__.items():
    print(i)
# output:('YELLOW', <COLOR.YELLOW: 1>)\n('GREEN', <COLOR.YELLOW: 1>)\n('BLACK', <
for i in COLOR.__members__:
    print(i)
# output:YELLOW\nGREEN\nBLACK\nRED

#枚举转换
#最好在数据库存取使用枚举的数值而不是使用标签名字字符串
#在代码里面使用枚举类
a=1
print(COLOR(a))# output:COLOR.YELLOW
```

py2/3转换工具

- six模块：兼容pyton2和pyton3的模块
- 2to3工具：改变代码语法版本
- __future__：使用下一版本的功能

常用的库

- 必须知道的collections
- python排序操作及heapq模块
- itertools模块超实用方法

不常用但很重要的库

- dis(代码字节码分析)
- inspect(生成器状态)
- cProfile(性能分析)

- bisect(维护有序列表)
- fnmatch
  - fnmatch根据系统决定
    - fnmatch(string,"*.txt") #win下不区分大小写
  - fnmatchcase完全区分大小写
- timeit(代码执行时间)

```
    def isLen(strString):
        #还是应该使用三元表达式，更快
        return True if len(strString)>6 else False

    def isLen1(strString):
        #这里注意false和true的位置
        return [False,True][len(strString)>6]
    import timeit
    print(timeit.timeit('isLen1("5fsdfsdfsaf")',setup="from __main__ import isLen

    print(timeit.timeit('isLen("5fsdfsdfsaf")',setup="from __main__ import isLen"
```

- contextlib
  - @contextlib.contextmanager使生成器函数变成一个上下文管理器
- types(包含了标准解释器定义的所有类型的类型对象,可以将生成器函数修饰为异步模式)

```
    import types
    types.coroutine #相当于实现了__await__
html(实现对html的转义)
    import html
    html.escape("<h1>I'm Jim</h1>") # output:'&lt;h1&gt;I&#x27;m Jim&lt;/h1&gt;'
    html.unescape('&lt;h1&gt;I&#x27;m Jim&lt;/h1&gt;') # <h1>I'm Jim</h1>
```

- mock(解决测试依赖)
- concurrent(创建进程池河线程池)

```python
from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor()
task = pool.submit(函数名, (参数)) #此方法不会阻塞，会立即返回
task.done()#查看任务执行是否完成
task.result()#阻塞的方法，查看任务返回值
task.cancel()#取消未执行的任务，返回True或False,取消成功返回True
task.add_done_callback()#回调函数
task.running()#是否正在执行      task就是一个Future对象

for data in pool.map(函数，参数列表):#返回已经完成的任务结果列表，根据参数顺序执行
    print(返回任务完成得执行结果data)

from concurrent.futures import as_completed
as_completed(任务列表)#返回已经完成的任务列表，完成一个执行一个

wait(任务列表,return_when=条件)#根据条件进行阻塞主线程，有四个条件
```

- selector(封装select,用户多路复用io编程)
- asyncio

```
future=asyncio.ensure_future(协程)  等于后面的方式  future=loop.create_task(协程)
future.add_done_callback()添加一个完成后的回调函数
loop.run_until_complete(future)
future.result()查看写成返回结果

asyncio.wait()接受一个可迭代的协程对象
asynicio.gather(*可迭代对象,*可迭代对象)      两者结果相同，但gather可以批量取消，gather对象

一个线程中只有一个loop

在loop.stop时一定要loop.run_forever()否则会报错
loop.run_forever()可以执行非协程
最后执行finally模块中 loop.close()

asyncio.Task.all_tasks()拿到所有任务 然后依次迭代并使用任务.cancel()取消

偏函数partial(函数，参数)把函数包装成另一个函数名   其参数必须放在定义函数的前面

loop.call_soon(函数,参数)
call_soon_threadsafe()线程安全
loop.call_later(时间,函数,参数)
在同一代码块中call_soon优先执行，然后多个later根据时间的升序进行执行

如果非要运行有阻塞的代码
使用loop.run_in_executor(executor,函数，参数)包装成一个多线程，然后放入到一个task列表中，通
```

通过asyncio实现http

```
reader,writer=await asyncio.open_connection(host,port)
writer.writer()发送请求
async for data in reader:
    data=data.decode("utf-8")
    list.append(data)
```

然后list中存储的就是html

```
as_completed(tasks)完成一个返回一个,返回的是一个可迭代对象

协程锁
```

```
async with Lock():
```

Python进阶

- 进程间通信：
  - Manager(内置了好多数据结构，可以实现多进程间内存共享)

```
from multiprocessing import Manager,Process
def add_data(p_dict, key, value):
    p_dict[key] = value

if __name__ == "__main__":
    progress_dict = Manager().dict()
    from queue import PriorityQueue

    first_progress = Process(target=add_data, args=(progress_dict, "bobby1", 22))
    second_progress = Process(target=add_data, args=(progress_dict, "bobby2", 23)

    first_progress.start()
    second_progress.start()
    first_progress.join()
    second_progress.join()

    print(progress_dict)
```

  - Pipe(适用于两个进程)

```python
from multiprocessing import Pipe,Process
#pipe的性能高于queue
def producer(pipe):
    pipe.send("bobby")

def consumer(pipe):
    print(pipe.recv())

if __name__ == "__main__":
    recevie_pipe, send_pipe = Pipe()
    #pipe只能适用于两个进程
    my_producer= Process(target=producer, args=(send_pipe, ))
    my_consumer = Process(target=consumer, args=(recevie_pipe,))

    my_producer.start()
    my_consumer.start()
    my_producer.join()
    my_consumer.join()
```

- Queue(不能用于进程池,进程池间通信需要使用Manager().Queue())

```
from multiprocessing import Queue,Process
def producer(queue):
    queue.put("a")
    time.sleep(2)

def consumer(queue):
    time.sleep(2)
    data = queue.get()
    print(data)

if __name__ == "__main__":
    queue = Queue(10)
    my_producer = Process(target=producer, args=(queue,))
    my_consumer = Process(target=consumer, args=(queue,))
    my_producer.start()
    my_consumer.start()
    my_producer.join()
    my_consumer.join()
进程池
def producer(queue):
    queue.put("a")
    time.sleep(2)

def consumer(queue):
    time.sleep(2)
    data = queue.get()
    print(data)

if __name__ == "__main__":
    queue = Manager().Queue(10)
    pool = Pool(2)

    pool.apply_async(producer, args=(queue,))
    pool.apply_async(consumer, args=(queue,))

    pool.close()
    pool.join()
```

- sys模块几个常用方法
  - argv 命令行参数list,第一个是程序本身的路径
  - path 返回模块的搜索路径
  - modules.keys() 返回已经导入的所有模块的列表
  - exit(0) 退出程序
- a in s or b in s or c in s简写
  - 采用any方式：all() 对于任何可迭代对象为空都会返回True

```
    # 方法一
    True in [i in s for i in [a,b,c]]
    # 方法二
    any(i in s for i in [a,b,c])
    # 方法三
    list(filter(lambda x:x in s,[a,b,c]))
```

- set集合运用
  - {1,2}.issubset({1,2,3})#判断是否是其子集
  - {1,2,3}.issuperset({1,2})
  - {}.isdisjoint({})#判断两个set交集是否为空,是空集则为True
- 代码中中文匹配
  - [u4E00-u9FA5]匹配中文文字区间[一到龥]
- 查看系统默认编码格式

```
    import sys
    sys.getdefaultencoding()    # setdefaultencodeing()设置系统编码方式
```

- **getattr** VS **getattribute**

```
class A(dict):
    def __getattr__(self,value):#当访问属性不存在的时候返回
        return 2
    def __getattribute__(self,item):#屏蔽所有的元素访问
        return item
```

- 类变量是不会存入实例__dict__中的，只会存在于类的__dict__中
- globals/locals(可以变相操作代码)
  - globals中保存了当前模块中所有的变量属性与值
  - locals中保存了当前环境中的所有变量属性与值
- python变量名的解析机制(LEGB)

    - 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)
- 实现从1-100每三个为一组分组

```
    print([[x for x in range(1,101)][i:i+3] for i in range(0,100,3)])
```

- 什么是元类?
    - 即创建类的类，创建类的时候只需要将metaclass=元类，元类需要继承type而不是object,因为type就是元类

```
type.__bases__  #(<class 'object'>,)
object.__bases__    #()
type(object)    #<class 'type'>
```

```
    class Yuan(type):
        def __new__(cls,name,base,attr,*args,**kwargs):
            return type(name,base,attr,*args,**kwargs)
    class MyClass(metaclass=Yuan):
        pass
```

- 什么是鸭子类型(即:多态)?
    - Python在使用传入参数的过程中不会默认判断参数类型，只要参数具备执行条件就可以执行
- 深拷贝和浅拷贝
    - 深拷贝拷贝内容，浅拷贝拷贝地址(增加引用计数)
    - copy模块实现神拷贝
- 单元测试
    - 一般测试类继承模块unittest下的TestCase
    - pytest模块快捷测试(方法以test_开头/测试文件以test_开头/测试类以Test开头，并且不能带有 init 方法)
    - coverage统计测试覆盖率

```
    class MyTest(unittest.TestCase):
        def tearDown(self):# 每个测试用例结束后执行
            print('本方法结束测试了')

        def setUp(self):# 每个测试用例执行之前做操作
            print('本方法测试开始了')

        @classmethod
        def tearDownClass(self):# 必须使用 @ classmethod装饰器, 所有test运行完后运行一
            print('开始测试')
        @classmethod
        def setUpClass(self):# 必须使用@classmethod 装饰器,所有test运行前运行一次
            print('结束测试')

        def test_a_run(self):
            self.assertEqual(1, 1)  # 测试用例
```

- gil会根据执行的字节码行数以及时间片释放gil，gil在遇到io的操作时候主动释放
- 什么是monkey patch?
  - 猴子补丁，在运行的时候替换掉会阻塞的语法修改为非阻塞的方法
- 什么是自省(Introspection)?
  - 运行时判断一个对象的类型的能力，id,type,isinstance
- python是值传递还是引用传递?
  - 都不是，python是共享传参，默认参数在执行时只会执行一次
- try-except-else-finally中else和finally的区别
  - else在不发生异常的时候执行，finally无论是否发生异常都会执行
  - except一次可以捕获多个异常，但一般为了对不同异常进行不同处理，我们分次捕获处理
- GIL全局解释器锁
  - 同一时间只能有一个线程执行，CPython(IPython)的特点，其他解释器不存在
  - cpu密集型：多进程+进程池
  - io密集型：多线程/协程
- 什么是Cython
  - 将python解释成C代码工具
- 生成器和迭代器
  - 实现__next__和__iter__方法的对象就是迭代器
    - 可迭代对象只需要实现__iter__方法
  - 使用生成器表达式或者yield的生成器函数(生成器是一种特殊的迭代器)

- 什么是协程
  - 比线程更轻量的多任务方式
  - 实现方式
    - yield
    - async-awiat
- dict底层结构
  - 为了支持快速查找使用了哈希表作为底层结构
  - 哈希表平均查找时间复杂度为o(1)
  - CPython解释器使用二次探查解决哈希冲突问题
- Hash扩容和Hash冲突解决方案
  - 循环复制到新空间实现扩容
  - 冲突解决：
    - 链接法
    - 二次探查(开放寻址法)：python使用

```
    for gevent import monkey
    monkey.patch_all()  #将代码中所有的阻塞方法都进行修改，可以指定具体要修改的方法
```

- 判断是否为生成器或者协程

```
    co_flags = func.__code__.co_flags

    # 检查是否是协程
    if co_flags & 0x180:
        return func

    # 检查是否是生成器
    if co_flags & 0x20:
        return func
```

- 斐波那契解决的问题及变形

```python
#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。
#请问用n个2*1的小矩形无重叠地覆盖一个2*n的大矩形，总共有多少种方法?
#方式一:
fib = lambda n: n if n <= 2 else fib(n - 1) + fib(n - 2)
#方式二:
def fib(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return b

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

- 获取电脑设置的环境变量

```python
    import os
    os.getenv(env_name,None)#获取环境变量如果不存在为None
```

- 垃圾回收机制
  - 引用计数
  - 标记清除
  - 分代回收

```python
    #查看分代回收触发
    import gc
    gc.get_threshold()  #output:(700, 10, 10)
```

- True和False在代码中完全等价于1和0,可以直接和数字进行计算，inf表示无穷大
- C10M/C10K
  - C10M:8核心cpu，64G内存，在10gbps的网络上保持1000万并发连接
  - C10K：1GHz CPU,2G内存，1gbps网络环境下保持1万个客户端提供FTP服务
- yield from与yield的区别：
  - yield from跟的是一个可迭代对象，而yield后面没有限制
  - GeneratorExit生成器停止时触发
- 单下划线的几种使用
  - 在定义变量时，表示为私有变量
  - 在解包时，表示舍弃无用的数据

    - 在交互模式中表示上一次代码执行结果
    - 可以做数字的拼接(111_222_333)
- 使用break就不会执行else
- 10进制转2进制

```
    def conver_bin(num):
        if num == 0:
            return num
        re = []
        while num:
            num, rem = divmod(num,2)
            re.append(str(rem))
        return "".join(reversed(re))
    conver_bin(10)
```

- list1 = ['A', 'B', 'C', 'D'] 如何才能得到以list中元素命名的新列表 A=[],B=[],C=[],D=[]呢

```
    list1 = ['A', 'B', 'C', 'D']

    # 方法一
    for i in list1:
        globals()[i] = []   # 可以用于实现python版反射

    # 方法二
    for i in list1:
        exec(f'{i} = []')   # exec执行字符串语句
```

- memoryview与bytearray$\color{#000}(不常用，只是看到了记载一下)$

```
    # bytearray是可变的，bytes是不可变的,memoryview不会产生新切片和对象
    a = 'aaaaaa'
    ma = memoryview(a)
    ma.readonly  # 只读的memoryview
    mb = ma[:2]  # 不会产生新的字符串

    a = bytearray('aaaaaa')
    ma = memoryview(a)
    ma.readonly  # 可写的memoryview
    mb = ma[:2]      # 不会会产生新的bytearray
    mb[:2] = 'bb'    # 对mb的改动就是对ma的改动
```

- Ellipsis类型

```
# 代码中出现...省略号的现象就是一个Ellipsis对象
L = [1,2,3]
L.append(L)
print(L)    # output:[1,2,3,[…]]
```

- lazy惰性计算

```
    class lazy(object):
        def __init__(self, func):
            self.func = func

        def __get__(self, instance, cls):
            val = self.func(instance)    #其相当于执行的area(c),c为下面的Circle对象
            setattr(instance, self.func.__name__, val)
            return val`

    class Circle(object):
        def __init__(self, radius):
            self.radius = radius

        @lazy
        def area(self):
            print('evalute')
            return 3.14 * self.radius ** 2
```

- 遍历文件，传入一个文件夹，将里面所有文件的路径打印出来(递归)

```
all_files = []
def getAllFiles(directory_path):
    import os
    for sChild in os.listdir(directory_path):
        sChildPath = os.path.join(directory_path,sChild)
        if os.path.isdir(sChildPath):
            getAllFiles(sChildPath)
        else:
            all_files.append(sChildPath)
    return all_files
```

- 文件存储时，文件名的处理

```
#secure_filename将字符串转化为安全的文件名
from werkzeug import secure_filename
secure_filename("My cool movie.mov") # output:My_cool_movie.mov
secure_filename("../../../etc/passwd") # output:etc_passwd
secure_filename(u'i contain cool \xfcml\xe4uts.txt') # output:i_contain_cool_umla
```

- 日期格式化

```
from datetime import datetime

datetime.now().strftime("%Y-%m-%d")

import time
#这里只有localtime可以被格式化，time是不能格式化的
time.strftime("%Y-%m-%d",time.localtime())
```

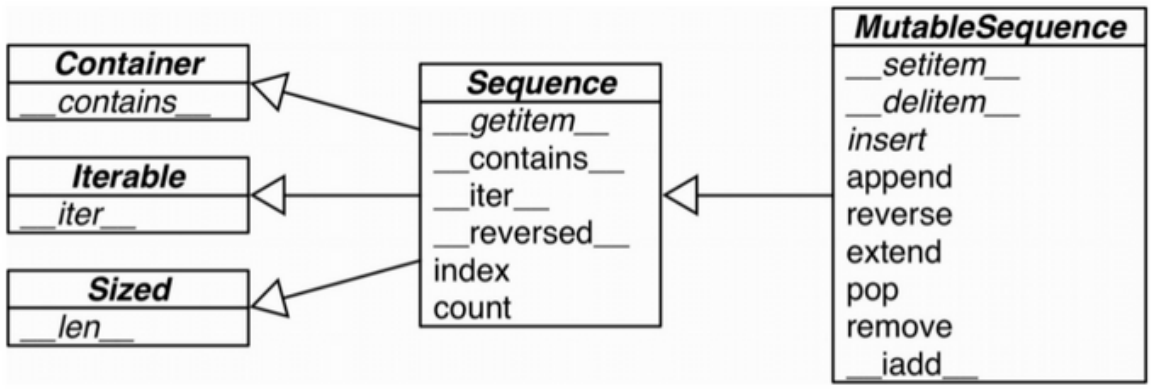
- tuple使用+=奇怪的问题
  - # 会报错，但是tuple的值会改变，因为t[1]id没有发生变化 t=(1,[2,3]) t[1]+=[4,5] # t[1]使用append\extend方法并不会报错，并可以成功执行
- __missing__你应该知道

```
class Mydict(dict):
    def __missing__(self,key): # 当Mydict使用切片访问属性不存在的时候返回的值
        return key
```

- +与+=
  - # +不能用来连接列表和元祖，而+=可以（通过iadd实现，内部实现方式为extends(),所以可以增加元组），+会创建新对象 不可变对象没有__iadd__方法，所以直接使用的是__add__方法，因此元祖可以使用+=进行元祖之间的相加
- 如何将一个可迭代对象的每个元素变成一个字典的所有键?

```
dict.fromkeys(['jim','han'],21) # output:{'jim': 21, 'han': 21}
```

- wireshark抓包软件
- 可变/不可变关系图



网络知识

- 什么是HTTPS?
  - 安全的HTTP协议，https需要cs证书，数据加密，端口为443，安全，同一网站https seo排名会更高
- 常见响应状态码

204 No Content //请求成功处理，没有实体的主体返回，一般用来表示删除成功 206 Partial Content //Get范围请求已成功处理 303 See Other //临时重定向，期望使用get定向获取 304 Not Modified //求情缓存资源 307 Temporary Redirect //临时重定向，Post不会变成Get 401 Unauthorized //认证失败 403 Forbidden //资源请求被拒绝 400 //请求参数错误 201 //添加或更改成功 503 //服务器维护或者超负载

- http请求方法的幂等性及安全性
- WSGI

```
    # environ: 一个包含所有HTTP请求信息的dict对象
    # start_response: 一个发送HTTP响应的函数
    def application(environ, start_response):
        start_response('200 OK', [('Content-Type', 'text/html')])
        return '<h1>Hello, web!</h1>'
```

- RPC
- CDN

- SSL(Secure Sockets Layer 安全套接层),及其继任者传输层安全（Transport Layer Security，TLS）是为网络通信提供安全及数据完整性的一种安全协议。
- SSH（安全外壳协议） 为 Secure Shell 的缩写，由 IETF 的网络小组（Network Working Group）所制定；SSH 为建立在应用层基础上的安全协议。SSH 是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。利用 SSH 协议可以有效防止远程管理过程中的信息泄露问题。SSH最初是UNIX系统上的一个程序，后来又迅速扩展到其他操作平台。SSH在正确使用时可弥补网络中的漏洞。SSH客户端适用于多种平台。几乎所有UNIX平台—包括HP-UX、Linux、AIX、Solaris、Digital UNIX、Irix，以及其他平台，都可运行SSH。
- TCP/IP
  - TCP:面向连接/可靠/基于字节流
  - UDP:无连接/不可靠/面向报文
  - 三次握手四次挥手
    - 三次握手(SYN/SYN+ACK/ACK)
    - 四次挥手(FIN/ACK/FIN/ACK)
  - 为什么连接的时候是三次握手，关闭的时候却是四次握手?
    - 因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。
  - 为什么TIME_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态?
    - 虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。
- XSS/CSRF
  - HttpOnly禁止js脚本访问和操作Cookie,可以有效防止XSS

Mysql

- 索引改进过程
  - 线性结构->二分查找->hash->二叉查找树->平衡二叉树->多路查找树->多路平衡查找树(B-Tree)
- Mysql面试总结基础篇
- Mysql面试总结进阶篇
- 深入浅出Mysql
- 清空整个表时，InnoDB是一行一行的删除，而MyISAM则会从新删除建表
- text/blob数据类型不能有默认值，查询时不存在大小写转换
- 什么时候索引失效
  - 以%开头的like模糊查询
  - 出现隐士类型转换

- 没有满足最左前缀原则
  - 对于多列索引，不是使用的第一部分，则不会使用索引
- 失效场景：
  - 应尽量避免在 where 子句中使用 != 或 <> 操作符，否则引擎将放弃使用索引而进行全表扫描
  - 尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，即使其中有条件带索引也不会使用，这也是为什么尽量少用 or 的原因
  - 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来，否则不会使用索引
  - 应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描

```
例如：
select id from t where substring(name,1,3) = 'abc' - name;
以abc开头的，应改成：
select id from t where name like ‘abc%’
例如：
select id from t where datediff(day, createdate, '2005-11-30') = 0 - '2005-11-30'
应改为：
```

  - 不要在 where 子句中的 “=” 左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引
  - 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描

```
如：
select id from t where num/2 = 100
应改为：
select id from t where num = 100*2;
```

  - 不适合键值较少的列（重复数据较多的列）比如：set enum列就不适合(枚举类型(enum)可以添加null,并且默认的值会自动过滤空格集合(set)和枚举类似，但只可以添加64个值)
  - 如果MySQL估计使用全表扫描要比使用索引快，则不使用索引
- 什么是聚集索引
  - B+Tree叶子节点保存的是数据还是指针
  - MyISAM索引和数据分离，使用非聚集
  - InnoDB数据文件就是索引文件，主键索引就是聚集索引

Redis命令总结

- 为什么这么快?

    - 基于内存，由C语言编写
    - 使用多路I/O复用模型，非阻塞IO
    - 使用单线程减少线程间切换
        - 因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。
    - 数据结构简单
    - 自己构建了VM机制，减少调用系统函数的时间
- 优势
    - 性能高 – Redis能读的速度是110000次/s,写的速度是81000次/s
    - 丰富的数据类型
    - 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行
    - 丰富的特性 – Redis还支持 publish/subscribe（发布/订阅）, 通知, key 过期等等特性
- 什么是redis事务?
    - 将多个请求打包，一次性、按序执行多个命令的机制
    - 通过multi,exec,watch等命令实现事务功能
    - Python redis-py pipeline=conn.pipeline(transaction=True)
- 持久化方式
    - RDB(快照)
        - save(同步，可以保证数据一致性)
        - bgsave(异步，shutdown时，无AOF则默认使用)
    - AOF(追加日志)
- 怎么实现队列
    - push
    - rpop
- 常用的数据类型(Bitmaps,Hyperloglogs,范围查询等不常用)
    - String(字符串):计数器
        - 整数或sds(Simple Dynamic String)
    - List(列表)：用户的关注，粉丝列表
        - ziplist(连续内存块，每个entry节点头部保存前后节点长度信息实现双向链表功能)或double linked list
    - Hash(哈希)：
    - Set(集合)：用户的关注者
        - intset或hashtable
    - Zset(有序集合)：实时信息排行榜
        - skiplist(跳跃表)
- 与Memcached区别

    - Memcached只能存储字符串键
    - Memcached用户只能通过APPEND的方式将数据添加到已有的字符串的末尾，并将这个字符串当做列表来使用。但是在删除这些元素的时候，Memcached采用的是通过黑名单的方式来隐藏列表里的元素，从而避免了对元素的读取、更新、删除等操作
    - Redis和Memcached都是将数据存放在内存中，都是内存数据库。不过Memcached还可用于缓存其他东西，例如图片、视频等等
    - 虚拟内存–Redis当物理内存用完时，可以将一些很久没用到的Value 交换到磁盘
    - 存储数据安全–Memcached挂掉后，数据没了；Redis可以定期保存到磁盘（持久化）
    - 应用场景不一样：Redis出来作为NoSQL数据库使用外，还能用做消息队列、数据堆栈和数据缓存等；Memcached适合于缓存SQL语句、数据集、用户临时性数据、延迟查询数据和Session等
- Redis实现分布式锁
    - 使用setnx实现加锁，可以同时通过expire添加超时时间
    - 锁的value值可以是一个随机的uuid或者特定的命名
    - 释放锁的时候，通过uuid判断是否是该锁，是则执行delete释放锁
- 常见问题
    - 缓存雪崩
        - 短时间内缓存数据过期，大量请求访问数据库
    - 缓存穿透
        - 请求访问数据时，查询缓存中不存在，数据库中也不存在
    - 缓存预热
        - 初始化项目，将部分常用数据加入缓存
    - 缓存更新
        - 数据过期，进行更新缓存数据
    - 缓存降级
        - 当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级
- 一致性Hash算法
    - 使用集群的时候保证数据的一致性
- 基于redis实现一个分布式锁，要求一个超时的参数
    - setnx
- 虚拟内存
- 内存抖动

Linux

- Unix五种i/o模型
    - 阻塞io
    - 非阻塞io

    - 多路复用io(Python下使用selectot实现io多路复用)
        - select
            - 并发不高，连接数很活跃的情况下
        - poll
            - 比select提高的并不多
        - epoll
            - 适用于连接数量较多，但活动链接数少的情况
    - 信号驱动io
    - 异步io(Gevent/Asyncio实现异步)
- 比man更好使用的命令手册
    - tldr:一个有命令示例的手册
- kill -9和-15的区别
    - -15：程序立刻停止/当程序释放相应资源后再停止/程序可能仍然继续运行
    - -9：由于-15的不确定性，所以直接使用-9立即杀死进程
- 分页机制（逻辑地址和物理地址分离的内存分配管理方案）：
    - 操作系统为了高效管理内存，减少碎片
    - 程序的逻辑地址划分为固定大小的页
    - 物理地址划分为同样大小的帧
    - 通过页表对应逻辑地址和物理地址
- 分段机制
    - 为了满足代码的一些逻辑需求
    - 数据共享/数据保护/动态链接
    - 每个段内部连续内存分配，段和段之间是离散分配的
- 查看cpu内存使用情况?
    - top
    - free 查看可用内存，排查内存泄漏问题

设计模式

单例模式

```python
    # 方式一
    def Single(cls,*args,**kwargs):
        instances = {}
        def get_instance (*args, **kwargs):
            if cls not in instances:
                instances[cls] = cls(*args, **kwargs)
            return instances[cls]
        return get_instance
    @Single
    class B:
        pass
    # 方式二
    class Single:
        def __init__(self):
            print("单例模式实现方式二。。。")

    single = Single()
    del Single  # 每次调用single就可以了
    # 方式三(最常用的方式)
    class Single:
        def __new__(cls,*args,**kwargs):
            if not hasattr(cls,'_instance'):
                cls._instance = super().__new__(cls,*args,**kwargs)
            return cls._instance
```

工厂模式

```
    class Dog:
        def __init__(self):
            print("Wang Wang Wang")
    class Cat:
        def __init__(self):
            print("Miao Miao Miao")


    def fac(animal):
        if animal.lower() == "dog":
            return Dog()
        if animal.lower() == "cat":
            return Cat()
        print("对不起，必须是：dog,cat")
```

构造模式

```
    class Computer:
        def __init__(self,serial_number):
            self.serial_number = serial_number
            self.memory = None
            self.hadd = None
            self.gpu = None
        def __str__(self):
            info = (f'Memory:{self.memoryGB}',
            'Hard Disk:{self.hadd}GB',
            'Graphics Card:{self.gpu}')
            return ''.join(info)
    class ComputerBuilder:
        def __init__(self):
            self.computer = Computer('Jim1996')
        def configure_memory(self,amount):
            self.computer.memory = amount
            return self #为了方便链式调用
        def configure_hdd(self,amount):
            pass
        def configure_gpu(self,gpu_model):
            pass
    class HardwareEngineer:
        def __init__(self):
            self.builder = None
        def construct_computer(self,memory,hdd,gpu)
            self.builder = ComputerBuilder()
            self.builder.configure_memory(memory).configure_hdd(hdd).configure_gp
        @property
        def computer(self):
            return self.builder.computer
```

数据结构和算法内置数据结构和算法

python实现各种数据结构

快速排序

```
    def quick_sort(_list):
            if len(_list) < 2:
                return _list
            pivot_index = 0
            pivot = _list(pivot_index)
            left_list = [i for i in _list[:pivot_index] if i < pivot]
            right_list = [i for i in _list[pivot_index:] if i > pivot]
        return quick_sort(left) + [pivot] + quick_sort(right)
```

选择排序

```
    def select_sort(seq):
        n = len(seq)
        for i in range(n-1)
        min_idx = i
            for j in range(i+1,n):
                if seq[j] < seq[min_inx]:
                    min_idx = j
            if min_idx != i:
                seq[i], seq[min_idx] = seq[min_idx],seq[i]
```

插入排序

```
    def insertion_sort(_list):
        n = len(_list)
        for i in range(1,n):
            value = _list[i]
            pos = i
            while pos > 0 and value < _list[pos - 1]
                _list[pos] = _list[pos - 1]
                pos -= 1
            _list[pos] = value
            print(sql)
```

归并排序

```
    def merge_sorted_list(_list1,_list2):   #合并有序列表
        len_a, len_b = len(_list1),len(_list2)
        a = b = 0
        sort = []
        while len_a > a and len_b > b:
            if _list1[a] > _list2[b]:
                sort.append(_list2[b])
                b += 1
            else:
                sort.append(_list1[a])
                a += 1
        if len_a > a:
            sort.append(_list1[a:])
        if len_b > b:
            sort.append(_list2[b:])
        return sort

    def merge_sort(_list):
        if len(list1)<2:
            return list1
        else:
            mid = int(len(list1)/2)
            left = mergesort(list1[:mid])
            right = mergesort(list1[mid:])
            return merge_sorted_list(left,right)
```

希尔排序
堆排序heapq模块

```
    from heapq import nsmallest
    def heap_sort(_list):
        return nsmallest(len(_list),_list)
```

栈

```python
from collections import deque
class Stack:
    def __init__(self):
        self.s = deque()
    def peek(self):
        p = self.pop()
        self.push(p)
        return p
    def push(self, el):
        self.s.append(el)
    def pop(self):
        return self.pop()
```

队列

```python
from collections import deque
class Queue:
    def __init__(self):
        self.s = deque()
    def push(self, el):
        self.s.append(el)
    def pop(self):
        return self.popleft()
```

二分查找

```python
def binary_search(_list,num):
    mid = len(_list)//2
    if len(_list) < 1:
        return Flase
    if num > _list[mid]:
        BinarySearch(_list[mid:],num)
    elif num < _list[mid]:
        BinarySearch(_list[:mid],num)
    else:
        return _list.index(num)
```

面试加强题：

关于数据库优化及设计

- 如何使用两个栈实现一个队列

- 反转链表
- 合并两个有序链表
- 删除链表节点
- 反转二叉树
- 设计短网址服务？62进制实现
- 设计一个秒杀系统(feed流)?
- 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InnoDB表的数据写入顺序能和B+树索引的叶子节点顺序一致的话，这时候存取效率是最高的。为了存储和查询性能应该使用自增长id做主键。
    - 对于InnoDB的主索引，数据会按照主键进行排序，由于UUID的无序性，InnoDB会产生巨大的IO压力，此时不适合使用UUID做物理主键，可以把它作为逻辑主键，物理主键依然使用自增ID。为了全局的唯一性，应该用uuid做索引关联其他表或做外键
- 如果是分布式系统下我们怎么生成数据库的自增id呢?
    - 使用redis
- 基于redis实现一个分布式锁，要求一个超时的参数
    - setnx
    - setnx + expire
- 如果redis单个节点宕机了，如何处理？还有其他业界的方案实现分布式锁码?
    - 使用hash一致算法

缓存算法

- LRU(least-recently-used):替换最近最少使用的对象
- LFU(Least frequently used):最不经常使用，如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小

服务端性能优化方向

- 使用数据结构和算法
- 数据库
    - 索引优化
    - 慢查询消除
        - slow_query_log_file开启并且查询慢查询日志
        - 通过explain排查索引问题
        - 调整数据修改索引
    - 批量操作，从而减少io操作
    - 使用NoSQL:比如Redis
- 网络io
    - 批量操作
    - pipeline
- 缓存

    - Redis
- 异步
    - Asyncio实现异步操作
    - 使用Celery减少io阻塞
- 并发
    - 多线程
    - Gevent