Table of Contents

Introduction

笔记尚待完善，陆续会有项目专题。

深度学习

我的仓库

1.python语言

1.1.python语言特点

动态语言

- 编译器还是运行期确定类型
- python是在运行期确定类型的

强类型

- 会不会发生隐式转换
- python是强类型语言

优缺点

- 胶水语言，轮子多，应用广泛
- 语言灵活，生产力高
- 性能问题，代码维护，python2/3差异

拥有自省功能

- 在运行时能够获得对象的类型
- type()，判断对象类型
- dir()， 带参数时获得该对象的所有属性和方法；不带参数时，返回当前范围内的变量、方法和定义的类型列表
- isinstance()，判断对象是否是已知类型
- hasattr()，判断对象是否包含对应属性
- getattr()，获取对象属性
- setattr()， 设置对象属性

猴子补丁

- 猴子补丁是程序在本地扩展或修改支持系统软件的方式（仅影响程序的运行实例）
- 所谓mankey patch 就是运行时替换
- 比如gevent库需要修改内置的socket
- from gevent import monkey； monkey.patch_socket() 将阻塞soket替换成非阻塞socket

鸭子类型

- 如果里看到一个鸟，走起来像鸭子，叫起来像鸭子，那么它就是鸭子
- 更关注接口

1.2.python2-3差异

不同

- print成为了函数
- 不再有Unicode，默认str就是nuicode
- 除法，除号返回浮点数
- 优化super函数

```
    retru super(C,self).func()#py2
    return super().func()#py3
```

- keyword only argument限定关键字参数

```
def add(a,b,*,c):
    pass
def add(**kwargs):
    pass
```

- 高级解包操作，a,b，*rest = range(10)
- 类型注解：type hint:def hello(name:str) ->str:
- chaied exception， py3重新抛出异常不会丢失栈信息
- 一切返回迭代器 range,zip,map,dict,values,etc,are all iterators

新增

- yeld from 连接
- asyncio，async/wait 原生协程支持异步编程
- 新增enum，mock，asyncio，ipaddress，concurrent.futures
- 生成的pyc文件统一放到**pycache**
- 内置库修改，urllib，selector

兼容2/3的代码

- six
- 2to3等工具转换代码格式
- __future__

可变/不可变对象

- 不可变对象：bool/int/float/tuple/str/frozenset
- 可变对象：list/set/dict

2.算法与数据结构

2.1.排序算法

2.1.排序算法

快速排序算法

```python {.line-numbers} def quicksort(array): if len(array) < 2: return array else: pivot_index = 0 #第一个元素作为主元 pivot = array[pivot_index] less_part = [i for i in array[pivot_index + 1:] if i <= pivot] great_part = [i for i in array[pivot_index + 1:] if i > pivot] return quicksort(less_part) + [pivot] + quicksort(great_part) def test_quicksort(): import random l1 = [range(10)] random.shuffle(l1) assert quicksort(l1) == sorted(l1)

test_quicksort()

---

**归并排序算法算法**

```python {.line-numbers}
def merge_sorted_seq(sorted_a, sorted_b):
    length_a, length_b = len(sorted_a), len(sorted_b)
    a = b = 0
    new_sorted_seq = []
    while a < length_a and b < length_b:
        if sorted_a[a] < sorted_b[b]:
            new_sorted_seq.append(sorted_a[a])
            a += 1
        else:
            new_sorted_seq.append(sorted_b[b])
            b += 1
    if a < length_a:
        new_sorted_seq.extend(sorted_a[a:])
    else:
        new_sorted_seq.extend(sorted_b[b:])
    return new_sorted_seq


def test_merge_sorted_seq():
    a = [1, 2, 5]
    b = [0, 3, 4, 8]
    print(merge_sorted_seq(a, b))


def merge_sort(array):
    if len(array) <= 1:
        return array
    else:
        mid = int(len(array) / 2)
        left_array = merge_sort(array[:mid])
        right_array = merge_sort(array[mid:])
        return merge_sorted_seq(left_array, right_array)


def test_merge_sort():
    import random
    l1 = list(range(10))
    random.shuffle(l1)
    assert merge_sort(l1) == sorted(l1)


test_merge_sort()
```
```

> 由于一次面试没答出来，回来恶补了一下 

2.2.数据结构

链表

```
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        pre =None
        cur = head
        while cur:
            nextnode = cur.next
            cur.next = pre
            pre = cur
            cur = nextnode
        return pre
```

队列

```
from collections import deque


class Queue:
    def __init__(self):
        self.item = deque()

    def append(self, val):
        return self.item.append(val)

    def pop(self):
        return self.item.popleft()

    def empty(self):
        return len(self.item) == 0
```

栈

```
class MinStack:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack=[]

    def push(self, x: int) -> None:
        self.stack.append(x)
    def pop(self) -> None:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return min(self.stack)


# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(x)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

树

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        stack = []
        if root is not None:
            stack.append((1, root))

        depth = 0
        while stack != []:
            current_depth, root = stack.pop()
            if root is not None:
                depth = max(depth, current_depth)
                stack.append((current_depth + 1, root.left))
                stack.append((current_depth + 1, root.right))

        return depth
```

3.编程范式

3.1.装饰器

函数装饰器

```
import time


def log_time(func):
    def _log(*args, **kwargs):
        beg = time.time()
        res = func(*args, **kwargs)
        print('use time: {}'.format(time.time() - beg))
        return res

    return _log


@log_time
def mysleep():
    time.sleep(1)


mysleep()
```

类装饰器

```
import time

class LogTime:
     def __call__(self,func):
        def _log(*args, **kwargs):
            beg = time.time()
            res = func(*args, **kwargs)
            print('use time: {}'.format(time.time() - beg))
            return res

        return _log


@LogTime()
def mysleep():
    time.sleep(1)


mysleep()
```

3.2.面向对象

概念

- 把对象作为基本单元，把对象抽象成类
- 数据封装，继承，多态

举个例子：

```
class person(object): #py3可以直接class person
    def __init__(self,name,age):
        self.name = name
        self.age = age

    def print_name(self):
        print("my name is {}".format(self.name))
```

方法中我们可以看到方法的参数中都带有self这个关键字，这么的意义是指，这些方法都只能通过类创建的对象调用，self代表对象自己。我们称之为”方法“。而那些如self.attr_name的，我们成为实例属性 组合与继承

- 组合是使用其他的类实例作为自己的一个属性(Has-a关系)
- 子类继承父类的属性和方法(Is-a)
- 优先使用组合保持代码简单

类变量和实例变量

- 类变量由所有实例共享
- 实例变量单独享有，不同实例之间不影响
- 当我们需要在一个类的不同实例之间共享变量的时候使用类变量

方法装饰器

- 都可以通过Class.method()的方式使用
- classmethod第一个参数是cls，可以引用类变量
- staticmethod使用起来和普通函数，只不过放在类里去组织

元类

元类是创建类的类

- 元类允许我们控制类的生成，比如修改类的属性
- 使用type来定义元类
- 也可使用metaclass属性来替换元类
- 元类最常见的一个使用场景就是ORM框架

```
#观看文档这两种定义是等价的。
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

```
class Base:
    pass

class Child(Base):
    pass
```

```
#等价定义注意Base后要加逗号否证就不是tuple了
SameChild = type('Child',(Base,)m|{})
#此操作与上无区别

#加上方法
class ChildWithMethod(Base):
    bar = True

    def hello(self):
        print('hello')
def hello():
    print('hello')

type('ChildWithMethod',(Base,),{'bar':Ture,'hello':hello})
```

4.网络协议

4.1.基本概念

协议表

浏览器输入一个**url**中间经历的过程

TCP三次握手

TCP四次挥手

[reference：https://www.cnblogs.com/huangjianping/p/7998067.html==](

4.2.HTTP协议

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器之间的传输协议是HTTP，所以：

- 是一种用来定义网页的文本，会HTML，就可以编写网页；
- 是在网络上传输HTML的协议，用于浏览器和服务器的通信。

> 步骤*1*：浏览器首先向服务器发送HTTP请求，请求包括： 方法：GET还是POST，GET仅请求资源，POST会附带用户数据； 路径：/full/url/path； 域名：由Host头指定：Host: www.sina.com.cn 以及其他相关的Header； 如果是POST，那么请求还包括一个Body，包含用户数据。
>
> 步骤*2*：服务器向浏览器返回HTTP响应，响应包括： 响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误； 响应类型：由Content-Type指定； 以及其他相关的Header； 通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。
>
> 步骤*3*：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。 Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当我们编写一个页面时，我们只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源。

什么是长链接？

- 短链接： 连接请求——数据传输——断开连接
- 长连接： 保持一段时间不断开tcp连接
- 告诉他长度 conten—length

cookies和**sesion**区别

- Session 一般是服务器生成后给客户端
- Cookie 是实现session的一种机制，通过HTTP cookie字段实现
- Session 通过在服务器保存sessionid 识别用户，cookie 存储在客户端

HTTP1.0和**HTTP1.1**的一些区别

- TTP1.1则引入了更多的缓存控制策略
- 支持断点续传功能，解决了浪费资源的问题
- 请求消息和响应消息都应支持Host头

HTTPS与**HTTP**的一些区别

- HTTPS协议需要到CA申请证书，一般免费证书很少，需要交费。
- HTTP协议运行在TCP之上，所有传输的内容都是明文，HTTPS运行在SSL/TLS之上，SSL/TLS运行在TCP之上，所有传输的内容都经过加密的。
- HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- HTTPS可以有效的防止运营商劫持，解决了防劫持的一个大问题。

HTTP请求方法

- 根据HTTP标准，HTTP请求可以使用多种请求方法。
- HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方法。
- HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

column0	column1	column2
序号	方法	描述

1	GET	请求指定的页面信息，并返回实体主体。
2	HEAD	类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头
3	POST	向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。
4	PUT	从客户端向服务器传送的数据取代指定的文档的内容。
5	DELETE	请求服务器删除指定的页面。
6	CONNECT	HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。
7	OPTIONS	允许客户端查看服务器的性能。
8	TRACE	回显服务器收到的请求，主要用于测试或诊断。

5.Linux相关

5.1.Linux储备

常考察方向

- 在Linux服务器上操作
- 了解Linux工作原理和常用工具
- 需要了解查看文件，进程，内存相关的一些命令，用来调试

如何查询**linux**命令用法

- 使用man命令查询用法。但是man手册比较晦涩
- 工具自带dehelp
- man的替代品tldr

文件/目录操作命令

- chown/chmode/chgrp
- ls/rm/cd/mv/touch/rename/ln(软链接和硬链接)等
- locate/find/grep定位查找搜索

文件查看

- 编辑器vi/nano
- cat/head/tail查看文件
- more/less交互式查看文件

进程操作命令

- ps查看进程
- kill杀死进程
- top/htop监控进程

内存操作命令

- free查看可用内存
- 了解每一列的具体含义
- 排查内存泄漏问题

网络操作命令

- ifconfig查看网卡信息
- lsof/netstat查看端口信息
- ssh/scp远程登录复制
- tcpdump抓包

用户/组操作命令

- useradd/usermod
- groupadd/groundmod

5.2.操作系统内存管理机制

5.2.操作系统内存管理机制

什么是分页机制 操作系统为了搞笑管理内存，减少碎片，逻辑地址和物理地址分离的内存分配管理方案

- 程序的逻辑地址划分为固定大小的页(Page)
- 物理地址划分为同样大小的帧(Frame)
- 通过页表对应逻辑地址和物理地址

什么是分段机制 分段式为了满足代码的一些逻辑需求

- 数据共享，数据保护，动态链接库等
- 通过段表实现逻辑地址和物理地址的映射关系
- 每个段类不式连续内存分配，段和段直接式离散分配的(每个段是出于实现相同的一个功能来进行分配)

分页和分段的区别

- 页是出于内存利用率的角度提出离散分配机制
- 段是出于用户角度，出于用户数据保护，数据隔离等用途的管理机制
- 页的大小是固定的，操作系统决定；段的大小不确定，用户程序决定

什么是虚拟内存 通过把一部分了暂时不用的的内存信息放到硬盘上

- 局部性原理，程序运行时候只有部分必要的信息转入内存
- 内存中暂时不需要的内容放到硬盘上
- 系统似乎提供了比实际内存大得很多的内存容量，称之为虚拟内存

什么是内存抖动

- 本质是频繁的页调度行为
- 频繁的页调度，进程不断产生缺页中断
- 置换一个页，又不断再次需要这个页
- 运行程序太多；分页替换策略不好。终止进程或者增加物理内存

Python的垃圾回收机制原理

- 引用计数为主(缺点：循环引用无法解决)
- 引用标记清楚和分代回收解决引用计数的问题
- 引用计数为主+标记清除和分代回收为辅

引用计数

```
a = [1]     #ref 1
b = a       #ref 2
b=None      #ref 1
del a       #ref 0  回收
```

标记清除

```
a = [1]     #a ref 1
b = [2]     #b ref 1
```

```
a.append(b) #b ref 2
b.append(a) #a ref 2
del a       #a ref 1
del b       #b ref 1    无法归零回收
```


通过root节点搜索可以达到的节点，不可达到的点标为灰色，回收

分代回收

- 给对象记录下一个age，随着每一次垃圾回收，这个age会增加；
- 给不同age的对象分配不同的堆内内存空间，称为某一代；
- 对某一代的空间，有适合其的垃圾回收算法；
- 对每代进行不同垃圾回收，一般会需要一个额外的信息：即每代中对象被其他代中对象引用的信息。这个引用信息对于当前代来说，扮演与"root"一样的角色，也是本代垃圾回收的起点。

5.3.操作系统线程和进程常考题

进程和线程对比

- 进程是对运行时程序的封装，时系统资源调度的基本单位
- 线程是进程的子任务，cpu调度和分配的基本单位，实现进程内并发，并行
  - 并行是真正的多核运行
  - python由于GIL不能真正的并行，看似并发
- 一个进程都可以包含多个线程，线程依赖进程存在，并共享进程内存

什么是线程安全

- 一个操作可以再多线程环境中安全使用，获取正确的结果
- 线程安全的操作好比线程是顺序执行而不是并发执行的(i+=1)
- 一般如果涉及到写操作需要考虑如何让多个线程安全访问数据

线程同步方式

- 互踩量：通过互斥机制防止多个线程同时访问公共资源
- 信号量(Semphare):threading.Semphare(valuse = 1)控制同一时刻多个线程访问同一个资源的线程数
- 事件(信号)：通过通知的方式保持多个线程同步

进程间通信的方式**(IPC)**

- 管道/匿名管道/有名管道(pipe)
- 信号(Signal):比如用户使用Ctrl+c产生SIGINT程序终止信号
- 消息队列(Message)
- 共享内存(share memory)
- 信号量(Semaphore)
- 套接字(socket):最常用的方式，我们的web应用都是这种方式

Python使用多线程(python开发适用于I/O密集型的应用)

- threading.Thread类来创建线程
- start()方法启动进程
- 可以用join()等待线程结束

Python使用多进程程(python开发使用于计算密集型的应用)

- mltiprocessing多进程模块
- Multiprocessing.Process类实现对进程
- 避免GIL的影响

6.数据库

6.1.数据库种类

如图:

关系型数据库介绍

1. 关系型数据库的由来　　虽然网状数据库和层次数据库已经很好的解决了数据的集中和共享问题，但是在数据库独立性和抽象级别上扔有很大欠缺。用户在对这两种数据库进行存取时，仍然需要明确数据的存储结构，指出存取路径。而关系型数据库就可以较好的解决这些问题。
2. 关系型数据库介绍　　关系型数据库模型是把复杂的数据结构归结为简单的二元关系（即二维表格形式）。在关系型数据库中，对数据的操作几乎全部建立在一个或多个关系表格上，通过对这些关联的表格分类、合并、连接或选取等运算来实现数据库的管理。

   关系型数据库诞生40多年了，从理论产生发展到现实产品，例如：Oracle和MySQL，Oracle在数据库领域上升到霸主地位，形成每年高达数百亿美元的庞大产业市场。
3. 关系型数据库表格之间的关系举例 

非关系型数据库介绍

1. 非关系型数据库诞生背景　　NoSQL，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSql数据库在特定的场景下可以发挥出难以想象的高效率和高性能，它是作为对传统关系型数据库的一个有效的补充。

   NoSQL(NoSQL = Not Only SQL)，意即“不仅仅是SQL”，是一项全新的数据库革命性运动，早期就有人提出，发展至2009年趋势越发高涨。NoSQL的拥护者们提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新的思维的注入。
2. 非关系型数据库种类

   （1）键值存储数据库（key-value）　　键值数据库就类似传统语言中使用的哈希表。可以通过key来添加、查询或者删除数据库，因为使用key主键访问，所以会获得很高的性能及扩展性。 键值数据库主要使用一个哈希表，这个表中有一个特定的键和一个指针指向特定的数据。Key/value模型对于IT系统来说的优势在于简单、易部署、高并发。 典型产品：Memcached、Redis、MemcacheDB

   （2）列存储（Column-oriented）数据库　　列存储数据库将数据存储在列族中，一个列族存储经常被一起查询的相关数据，比如人类，我们经常会查询某个人的姓名和年龄，而不是薪资。这种情况下姓名和年龄会被放到一个列族中，薪资会被放到另一个列族中。

   这种数据库通常用来应对分布式存储海量数据。

   典型产品：Cassandra、HBase

   （3）面向文档（Document-Oriented）数据库

   文档型数据库的灵感是来自于Lotus Notes办公软件，而且它同第一种键值数据库类似。该类型的数据模型是版本化的文档，半结构化的文档以特定的格式存储，比如JSON。文档型数据库可以看作是键值数据库的升级版，允许之间嵌套键值。而且文档型数据库比键值数据库的查询效率更高。

   面向文档数据库会将数据以文档形式存储。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名词与对应值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用XML、JSON或JSONB等多种形式存储。

典型产品：MongoDB、CouchDB

（4）图形数据库

图形数据库允许我们将数据以图的方式存储。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple和Next，则会有两个“Founded by”的边将Apple和Next连接到Steve Jobs。

典型产品：Neo4J、InforGrid

> [reference：https://blog.csdn.net/qq_27565769/article/details/80731213](

6.2.MYSQL

6.2.1.MYSQL概念

常考察点

- 事务原理，特性，事务的并发控制
- 常用字段，含义，区别
- 常用数据库引擎的区别

事务 Transaction

- 事务是数据库并发控制的基本单位
- 事务可以看作是一系列SQL语句的集合
- 事务必须要么全部执行成功，要么全部执行失败

特性ACID

- 原子性(Atomicity)：一个事务所有操作全部完成或失败
- 一致性(Consistency):事务开始和结束后数据的完整性没有被破坏
- 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，修改时永久不会丢失的

事务的并发可能会产生的四种异常情况

- 幻读(phanton 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结果
- 脏读(dirty read):一个事务读取到另外一个事务没有提交的修改
- 丢失修改(lost update):并发写入造成其中一些修改丢失

四种事务隔离级别

- 读取提交(read uncommitted):别的事务可以读取到未提交改变
- 读已提交(read committed):只能读取已提交的数据
- 可重复读(repeatable read):同一个事务先后查询结果一样(Mysql InoDB默认实现可重复读级别)
- 串行化(Serialzavle)：事务串行化的执行，隔离级别最高，执行效率最低

如何解决并发场景下的插入重复

- 使用数据库的唯一索引(一般情况用不了，一般项目会建库建表)
- 使用队列异步写入
- 使用redis等实现分布式锁

乐观锁和悲观锁

- 悲观锁是先获取在进行操作。一锁二查三更新select for update
- 乐观锁先修改，更新的时候发现数据已经变了就回滚(测check and set)
- 根据响应速度，冲突频率，重试代价来判断选择哪种

MYSQL数据类型

1. 字符串 CHAR:存储定长字符串 VARCHAR：存储不定长字符串 TEXT:存储较长的文章
2. 数值 TINTINT,INT,BIGINT,DOUBLE等
3. 日期和时间 DATE：YYYY-MM-DD DATETIME:YYYY-MM-DD HH:MM:SS

Mysql常用引擎

- MyISAM不支持事务，InnoDB支持事务
- MyISAM不支持外键，InnoDB支持外键
- MyISAM只支持表锁，InnoDB支持表锁和行锁

6.2.2.索引原理以及优化

- 索引的原理，类型，结构
- 创建索引的注意事项，使用原则
- 创建排查和消除慢查询

什么是索引?

- 索引是数据表中一个或多个列进行排序的数据结构
- 索引能够大幅度提升检索速度(回顾下查找结构：二叉搜索树，平衡数，多路平衡数)
- 创建，更新索引本身也会消耗空间和时间

查找结构进化史

- 线性查找：一个一个找，实现简单，速度慢
- 二分查找：简单，查找快，但要求是有序的，插入特别慢
- HASH：查询快，占用空间，不太适合存储大规模的数据
- 二叉查找树：插入和查询很快(log(n))，无法存大规模数据，复杂退化问题
- 平衡数：解决bst退化的问题，树是平衡的；节点非常多的时候，树依然很高
- 多路查找树：一个父亲多个孩子节点，书不会特别深
- 多路平衡查找树：B-Tree

> 数据结构可视化网站

什么是**B-Tree?**

- 多路平衡查找树(每个节点最多m(m>=2)个孩子，称为m阶或者度)
- 叶节点具有相同深度
- 节点中的数据key从做到右四递增的

什么是**B+Tree**

- Mysql实际使用的是B+Tree作为索引的数据结构
- 只在叶子节点带有指向的指针，可以增加书的度
- 叶子节点通过指针相连，实现范围查询

Mysql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

- 经常用作查询条件的字段(WHERE条件)
- 经常用锁表连接的字段
- 经常出现order by，ground by之后的字段

创建索引右那些需要注意的

- 非空字典NOT NULL，Mysql很难多空值查询优化
- 区分度高，离散度大，作为索引的字段值尽量不要右大量相同值

- 索引长度不要太长(比较耗费时间)

索引什么时候失效

- 模糊匹配：以%开头的LIKE语句，模糊搜索
- 类型隐转：出现隐式类型转换(在python这种动态语言中查询需要特别注意)
- 没有满足最左前缀原则

什么式聚集索引和非聚集索引

- 聚集还是非聚集指的式B+Tree叶节点的是指针还是数据记录
- MyISAM索引和数据分离，使用的是非聚集索引
- InnoDB数据文件就是索引文件，主键索引就是聚集索引

如何排查慢查询

- 慢查询通常是缺少索引，索引不合理或业务逻辑代码实现导致
- slow_query_log_file开启并且查询了慢查询日志
- 通过explain排查索引问题
- 调整数据修改索引；业务代码层限制不合理访问

6.2.3.Mysql语句常考题

SQL语句已考察各种各种连接为重点

- 内链接(INNER JOIN)：两个表存在匹配时，才会返回匹配行
  - 将左表和右表能关联起来的数据连接后返回
  - 类似于求两个表的“交集”
  - select * from A innner join B on a .id =v .id
- 外连接(LEFT/RIGHT JOIN)：返回一个表的行，即使另外一个没有匹配
  - 左连接返回坐标中所有记录，几时右表中没有匹配的记录
  - 左连接返回右表中所有记录，几时坐标中没有匹配的记录
  - 没有匹配的字段会设置成NULL
  - Mysql中使用left join和right jion实现
- 全链接(FULL JOIN):只要某一个表存在匹配就返回
  - 只要某一个表存在匹配，就返回行
  - 类似求两个表的“并集”
  - 但是Mysql不支持，可以用left jion，union，right join联合使用模拟

6.2.4.Mysql思考题

- 为什么Mysql数据库的主键使用自增的增数比较好？　　对于这个问题需要从MySQL的索引以及存储引擎谈起：　　InnoDB的primary key为cluster index,除此之外，不能通过其他方式指定cluster index,如果InnoDB不指定primary key,InnoDB会找一个unique not null的field做cluster index,如果还没有这样的字段，则InnoDB会建一个非可见的系统默认的主键---row_id(6个字节长)作为cluster_index。　　建议使用数字型auto_increment的字段作为cluster index。不推荐用字符串字段做cluster index (primary key) ,因为字符串往往都较长， 会导致secondary index过大(secondary index的叶子节点存储了primary key的值),而且字符串往往是乱序。cluster index乱序插入容易造成插入和查询的效率低下。
- 使用uuid可以？为什么？
  - 自增ID节省一半磁盘空间
  - 单个数据走索引查询，自增id和uuid相差不大
  - 范围like查询，自增ID性能优于UUID
  - 写入测试，自增ID是UUID的4倍

    - 备份和恢复，自增ID性能优于UUID
- 如果是分布式系统下怎么生成数据库的自增? 分布式架构，意味着需要多个实例中保持一个表的主键的唯一性。这个时候普通的单表自增ID主键就不太合适，因为多个mysql实例上会遇到主键全局唯一性问题。
    - 自增ID主键+步长，适合中等规模的分布式景　　在每个集群节点组的master上面，设置（auto_increment_increment），让目前每个集群的起始点错开 1，步长选择大于将来基本不可能达到的切分集群数，达到将 ID 相对分段的效果来满足全局唯一的效果。　　优点是：实现简单，后期维护简单，对应用透明。　　缺点是：第一次设置相对较为复杂，因为要针对未来业务的发展而计算好足够的步长;
    - UUID，适合小规模的分布式环境　　对于InnoDB这种聚集主键类型的引擎来说，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，而且由于索引和数据存储在一起，字符串做主键会造成存储空间增大一倍。　　在存储和检索的时候，innodb会对主键进行物理排序，这对auto_increment_int是个好消息，因为后一次插入的主键位置总是在最后。但是对uuid来说，这却是个坏消息，因为uuid是杂乱无章的，每次插入的主键位置是不确定的，可能在开头，也可能在中间，在进行主键物理排序的时候，势必会造成大量的IO操作影响效率，在数据量不停增长的时候，特别是数据量上了千万记录的时候，读写性能下降的非常厉害。优点：搭建比较简单，不需要为主键唯一性的处理。 缺点：占用两倍的存储空间（在云上光存储一块就要多花2倍的钱），后期读写性能下降厉害。
    - 雪花算法自造全局自增ID，适合大数据环境的分布式场景。由twitter公布的开源的分布式id算法snowflake

6.3.Redis

6.3.1.Redis概念

- 为什么使用缓存？使用场景？
  - 常用的内存缓存有Redis和Memcached
  - 缓存关系数据库并访问的压力：热点数据
  - 减少响应时间：内存IO速度必磁盘快
  - 提升吞吐量：Redis等内存数据库单机可以支撑很大并发

- Redis的常用数据类型，使用方式
  - Sring:用来实现简单的KV键值对，比如计数器
  - List：实现双向链表，比如用户的关注，粉丝列表
  - Hash：用来存储彼此相关的键值对，HSET key filed value
  - Set：存储不重复元素，比如用户的关注者
  - Sorted Set：实时信息排行榜
- Redis内置实现
  - C语言底层实现
  - String：整数或者sds(Simple 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有哪些持久化方式？
  - 快照方式：把数据快照放在磁盘二进制文件中，dump.rdb，指定时间间隔把Redis数据库状态保存到一个压缩的二进制文件中，缺点：若宕机，间隔内的数据全部丢失
  - AOF(Append Only File)：每一个写命令保存到appendonly.aof中。缺点，虽然不会丢失大量数据，但文件比较大，恢复速度比较慢
- Redis事务
  - 将多个请求打包，一次性，按序执行多个命令的机制
  - Rdis通过MULTI,EXEC,WATCH等命令实现事务功能
  - Python redis-py pipeline = conn.pipeline(transaction =True)
- Redis如何实现分布式锁
  - 使用setnx实现加锁，可以同时通过expire添加超时时间
  - 锁的valuse值可以使用一个随机的uuid或者待定的命名
  - 释放锁的时候，通过uuid判断是否是该锁，是则执行delete释放锁

  Redis分布式锁的实现原理看这篇就够了
- 使用缓存的模式？
  - Cache Aside：同时更新缓存和数据库(先更新数据库后更新缓存，并发写操作可能导致缓存读取的是脏数据，一般先更新数据库然后删除缓存，下次读取时再重建缓存)
  - Read/Write Throught：先更新缓存，缓存负责同步更新数据库
  - Write Behind Caching：先更新缓存，缓存顶起异步更新数据库
- 缓存使用问题：数据一致性问题；缓存穿透，击穿，雪崩
  - 缓存穿透：大量查询不到的数据请求落到后端数据库，数据库压力增大(很多无脑爬虫通过自增id的方式爬取网

站，网站查不到相关id的数据)
    - 原因：由于大量缓存查不到就去数据库取，数据库也没有要差的数据
    - 解决：对于没查到返回为None的数据也缓存
    - 插入数据的时候删除相应缓存，或者设置较短的超时时间
- 缓存击穿：某些非常热点的数据key过期，大量请求打到后端数据库
    - 原因：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
    - 解决：分布式锁：获取锁的线程从数据库拿去数据更新缓存，其他线程等待。异步后台更新：后代任务针对过期的key自动刷新
- 缓存雪崩：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
    - 解决：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

6.3.2.Redis分布式锁应用

- 请里基于redis编写实现一个简单的分布式锁(要求支持超时参数)
- 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么？

7.爬虫

7.1.技术储备

7.1.1.开发环境

- pycharm
- mysql+redis+etri

技术选型

- scrapy vs requests + beatifulsoup
- request 和beatifulsoup都是库，scrapy是框架
- scrapy框架加入
- scrapy基于twsted，性能最大的优势
- scrapy方便拓展，提供了很对内置的功能
- scrapy内置的css和xpath selector非常方便，beautifulsoup最大的缺点就是慢

网页分类

- 静态网页
- 动态网页
- webservice(restapi)

7.1.2.正则表达式

字符	描述
\cx	匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。
\S	匹配任何非空白字符。等价于 [^\f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 $。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*	匹配前面的子表达式零次或多次。要匹配 字符，请使用 \。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 . 请使用

.	匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。
[	标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 { ，请使用 \{。

\		指明两项之间的一个选择。要匹配 \	，请使用 \\	。

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo 能匹配 *"z"* 以及 *"zoo"*。 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。 + 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。 ? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

```
import re
line = 'marshenmmm'                     #.为任意字符 * 与 +为修饰次数的限定词
regek_str1 = '.*(m.*m).*'               #贪婪匹配
regek_str2 = '.*?(m.*?m).*'             #非贪婪匹配
regek_str2 = '.*?(m.+?m).*'             #至少中间有一个字符
res1 = re.match(regek_str1,line)
res2 = re.match(regek_str2,line)
print(res1.group(1))
print(res2.group(1))
"""
执行结果：
mm
marshenm
mmm
"""
```

7.1.3.深度优先与广度邮箱

> 我们可以观看网站的结构非常地与数据结构的树相似

- 深度优先

```
# ABDEICFGH(递归实现)
```

```
def depth_tree(tree_node):
    if tree_node is not None:
        print(tree_node.val)
    if tree_node._left is not None:
        return depth_tree(tree_node._left)
    if tree_node._right is not None:
        return depth_tree(tree_node._right)
```

- 广度优先

```
# ABCDEFGHI(队列实现)
def level_queue(root):
    if root is None:
        return
    my_queue = []
    my_queue.append(root)
    while my_queue:
        node = my_queue.pop()
        print(node.elem)
        if node.lchild is not None:
            my_queue.append(node.lchild)
        if node.rchild is not None:
            my_queue.appnd(node.rchild)
```

爬虫去重策略 由于网站中链接会有相互跳转的情况，如我们不处理，那么就有可能会进入无限的循环。我们就需要进行网站的除重。常用的除虫有以下策略：

- 将访问的url保存到数据库中
- 将访问的url报错到set中，只需要o(1)代价就可以查询url
- url经过md5等方法哈希保存到set
- 用bitmap方法，将访问的url通过hash函数映射到某一位
- bloomfilter方法对bitmap进行改进，对重hash函数降低冲突

7.2.scrapy安装

7.2.1启动项目

虚拟环境安装好后，在windows系统中还需安装pypiwin32库，安装好后，在cmd中输入

```
scrapy startproject ArticleSpider               #创建项目
scrapy genspider jobbloe web.jobble.com         #在项目于目录下，创建爬虫
```

由于pycharm中没有scrapy的模板，需要创建主脚本启动调试爬虫，在主目录下创建

```
from scrapy.cmdline import execute
import sys
import os

sys.path.append(os.path.dirname(os.path.abspath(__file__)))
execute(['scrapy','crawl','jobbole'])    #相当于在cmd窗口执行   scrapy crawl jobble
```

7.2.2xpath

简介 使用路径表达式在xml和html中进行导航 xpath包含标准函数库 xpath时一个w3c标准

节点关系 父节点 子节点 同胞节点 先辈节点 后代节点

xpath语法

在伯乐在线选取一篇网站，使用浏览器开发者工具复制xpth路径填入 火狐浏览器和chrome浏览器的可能不一样，是因为火狐的复制xpath是浏览器运行网页代码之后生成的，其中有js生成的元素。 0

```
#jobbole.py
start_urls = ['http://blog.jobbole.com/114666/']
...
strs = '//*[@id="post-114666"]/div[1]/h1/text()'
re_selector = response.xpath(strs)
xpath("//span[contains(@class,'xxxx')]")
```

8.框架语言

8.1.什么是WSGI

8.1.什么是WSGI

- python web server gateway interface
- 解决了python webserver乱象 mode——python，CGI。fastCGI
- 描述了web server 如何与web框架交互，web框架如何请求处理

```
#一个简单的wsgi应用
def myapp(environ, start_resopnce):
    status = '200 OK'
    header = [('Conten-Typr', 'text/html;charset=utf-8')]
    start_resopnce(status, header)
    return [b'<h1>Hello world</h1>']


if __name__ == "__main__":
    from wsgiref.simple_server import make_server
    httpd = make_server('127.0.0.1', 8888, myapp)
    httpd.serve_forever()
```

8.2.常用pythonweb框架

8.3.RESTful

- 前后端分离的意义和方式
- 什么是RESTful
- 如何设计RESTful API

前后端解耦，接口复用，减少开发量 各司其职，前后端同步开发，提升工作效率，定义好接口规范 更有利于调试(mock)测试和运维部署

representtational state transfer 表现层状态转移，由HTTP协议的主要设计者RoyFielding提出 资源(resources)，表现层(representation)，状态转化(statr transfer) 是一种以资源为为中心的web软件架构风格，可以用ajax和resful web服务构建应用

- resources：使用url指向一个实体
- representation：资源的表现形式，比如图片，HTML文本等
- statr transfer状态转化：get，post，putdelete http动词来操作资源，实现资源的改变

resful的准则 所有思维u抽象围殴至于那，资源对应唯一的标识 资源通过接口进行操作实现状态转移，操作本身无状态 对之u按的操作不会改变资源的标识

restful api 通过get，post，put delete http 获取/新疆/更新/删除 一般使用json格式返回数据 一般web框架都有相应的插件支持resfulapi

什么是**https**

- https和http的区别是什么
- 什么是对称加密和非对称加密

9.web高并发、分布式技术

9.1.什么是微服务？

单体式开发的缺点

- 设计、开发、部署为一个单独的单元。会变得越来越复杂，最后导致维护、升级、新增功能变得异常困难。很难以敏捷研发模式进行开发和发布部分更新，都需要重新部署整个应用
- 水平扩展：必须以应用为单位进行扩展，在资源需求有冲突时扩展变得比较困难（部分服务需要更多的计算资源，部分需要更多内存资源）
- 可用性：一个服务的不稳定会导致整个应用出问题
- 创新困难：很难引入新的技术和框架，所有的功能都构建在同质的框架之上
- 运维困难：变更或升级的影响分析困难，任何一个小修改都可能导致单体应用整体运行出现故障。

微服务的出现 随着用户群体的增多，导致了web服务器的压力增大，还有一些高并发的时间端的冲击。我们并不能保证服务器的安全平稳运行。一旦宕机，若不尽快的修复就会带来难以弥补的损失。

微服务的设计原则 职责单一原则（Single Responsibility Principle）：把某一个微服务的功能聚焦在特定业务或者有限的范围内会有助于敏捷开发和服务的发布。 设计阶段就需要把业务范围进行界定。

9.2.网络IO

==本节图片摘自《UNIX网络编程》== **poll,select,epoll**

- poll是Linux中的字符设备驱动中的一个函数。Linux 2.5.44版本后，poll被epoll取代。和select实现的功能差不多，poll的作用是把当前的文件指针挂到等待队列。
- Select在Socket编程中还是比较重要的，可是对于初学Socket的人来说都不太爱用Select写程序，他们只是习惯写诸如connect、accept、recv或recvfrom这样的阻塞程序（所谓阻塞方式block，顾名思义，就是进程或是线程执行到这些函数时必须等待某个事件的发生，如果事件没有发生，进程或线程就被阻塞，函数不能立即返回）。 可是使用Select就可以完成非阻塞（所谓非阻塞方式non-block，就是进程或线程执行此函数时不必非要等待事件的发生，一旦执行肯定返回，以返回值的不同来反映函数的执行情况，如果事件发生则与阻塞方式相同，若事件没有发生则返回一个代码来告知事件未发生，而进程或线程继续执行，所以效率较高）方式工作的程序，它能够监视我们需要监视的文件描述符的变化情况——读写或是异常。
- epoll是Linux内核为处理大批量文件描述符而作了改进的poll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。==另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了==。epoll除了提供select/poll那种IO事件的水平触发（Level Triggered）外，还提供了边缘触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll_wait/epoll_pwait的调用，提高应用程序效率。

BIO(同步阻塞) 阻塞式I/O模型是最基本的I/O模型。在默认情况下，所有套接字都是阻塞的，以数据报(在python中常用monkey pack替换成非阻塞socket)

1. 收到一个IO请求，首先调用recvfrom系统调用
2. 不能立即获得数据，从磁盘读取数据到内核内存(wait for data)
3. 数据准备完毕，从内核内存复制到用户程序内存
4. 返回OK

当发生错误时recvfrom会返回出错。recvfrom只有接收或者出错时才会返回，其余时间都是阻塞的。

> 在用户量有一定规模的情况下，可以使用：
>
> - 在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。
> - 但是开启多进程或都线程的方式，在遇到要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资源，==降低系统对外界响应效率，而且线程与进程本身也更容易进入假死状态。(内存抖动)==
> - ==线程池”旨在减少创建和销毁线程的频率==，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。“连接池”维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率。这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统。
> - “线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用IO接口带来的资源占用。而且，==所谓“池”始终有其上限，当请求大大超过上限时==，“池”构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。

NIO(同步非阻塞)

1. 收到一个IO请求，首先调用recvfrom系统调用
2. 系统立即返回一个ERROR说明数据没准备好
3. 在这一阶段可以继续执行其他操作
4. 继续回到步骤2继续执行，直到数据表准备完成

5. 操作系统将数据表复制到用户程序内存(这一阶段也是阻塞状态的)
6. 返回ok

也就是说非阻塞的recvform系统调用调用之后，进程并没有被阻塞，内核马上返回给进程，如果数据还没准备好，此时会返回一个error。进程在返回之后，可以干点别的事情，然后再发起recvform系统调用。重复上面的过程，循环往复的进行recvform系统调用。这个过程通常被称之为轮询。轮询检查内核数据，直到数据准备好，再拷贝数据到进程，进行数据处理。需要注意，拷贝数据整个过程，进程仍然是属于阻塞的状态。

> 优点：能够在等待任务完成的时间里干其他活了（包括提交其他任务，也就是 “后台” 可以有多个任务在“”同时“”执行）。 缺点：
>
> - 循环调用recv()将大幅度推高CPU占用率；这也是我们在代码中留一句time.sleep(2)的原因,否则在低配主机下极容易出现卡机情况
> - 任务完成的响应延迟增大了，因为每过一段时间才去轮询一次read操作，而任务可能在两次轮询之间的任意时间完成。这会导致整体数据吞吐量的降低。

==此外，在这个方案中recv()更多的是起到检测“操作是否完成”的作用，实际操作系统提供了更为高效的检测“操作是否完成“作用的接口，例如select()多路复用模式，可以一次检测多个连接是否活跃。==

多路复用 IO multiplexing这个词可能有点陌生，但是如果我说select/epoll，大概就都能明白了。有些地方也称这种IO方式为事件驱动IO(event driven IO)。我们都知道，select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select/epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。它的流程如图：

> 

当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。 这个图和blocking IO的图其实并没有太大的不同，事实上还更差一些。因为这里需要使用两个系统调用(select和recvfrom)，而blocking IO只调用了一个系统调用(recvfrom)。但是，用select的优势在于它可以同时处理多个connection。

强调：

1. 如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。
2. 在多路复用模型中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

==结论: select的优势在于可以处理多个连接，不适用于单个连接==

> 优点： 相比其他模型，使用select() 的事件驱动模型只用单线程（进程）执行，占用资源少，不消耗太多 CPU，同时能够为多客户端提供服务。如果试图建立一个简单的事件驱动的服务器程序，这个模型有一定的参考价值。 缺点： 首先select()接口并不是实现“事件驱动”的最好选择。因为当需要探测的句柄值较大时，select()接口本身需要消耗大量时间去轮询各个句柄。很多操作系统提供了更为高效的接口，如linux提供了epoll，BSD提供了kqueue，Solaris提供了/dev/poll，…。如果需要实现更高效的服务器程序，类似epoll这样的接口更被推荐。遗憾的是不同的操作系统特供的epoll接口有很大差异，所以使用类似于epoll的接口实现具有较好跨平台能力的服务器会比较困难。 其次，该模型将事件探测和事件响应夹杂在一起，一旦事件响应的执行体庞大，则对整个模型是灾难性的。 **AIO(异步非阻塞)**
>
>  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。

通俗的例子 老张爱喝茶，废话不说，煮开水。 出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

1. 老张把水壶放到火上，立等水开。（同步阻塞）老张觉得自己有点傻
2. 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀~~~~的噪音。
3. 老张把响水壶放到火上，立等水开。（异步阻塞）老张觉得这样傻等意义不大
4. 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）老张觉得自己聪明了。

5. 所谓同步异步，只是对于水壶而言。 普通水壶，同步；响水壶，异步。 虽然都能干活，但响水壶可以在自己完工之后，提示老张水开了。这是普通水壶所不能及的。 同步只能让调用者去轮询自己（情况2中），造成老张效率的低下。

6. 所谓阻塞非阻塞，仅仅对于老张而言。 立等的老张，阻塞；看电视的老张，非阻塞。 情况1和情况3中老张就是阻塞的，媳妇喊他都不知道。虽然3中响水壶是异步的，可对于立等的老张没有太大的意义。所以一般异步是配合非阻塞使用的，这样才能发挥异步的效用。