**关于服务器端基础知识的简答题知识点及答案**

Author：zsdostar

（ 题目参考： ）( <http://blog.csdn.net/acingdreamer/article/details/78479476> ）

目录

[一、计算机网络 5](#_Toc503293097)

[TCP报头格式 5](#_Toc503293098)

[UDP报头格式 5](#_Toc503293099)

[TCP/UDP区别（不仅是宏观上的，最好能根据各自的机制讲解清楚） 5](#_Toc503293100)

[HTTP状态码（最好结合使用场景，比如在缓存命中时使用哪个） 6](#_Toc503293101)

[HTTP协议（一些报头字段的作用，如cace-control、keep-alive） 6](#_Toc503293102)

[常用的HTTP方法有哪些？ 7](#_Toc503293103)

[OSI协议、TCP/IP协议以及每层对应的协议。 8](#_Toc503293104)

[SESSION机制、cookie机制 8](#_Toc503293105)

[TCP三次握手、四次挥手（这个问题真的要回答吐了，不过真的是面试官最喜欢问的，建议每天手撸一遍，而且不只是每次请求的过程，各种FIN\_WAIT、TIME\_WAIT状态也要掌握）。 9](#_Toc503293106)

[打开网页到页面显示之间的过程（涵盖了各个方面，DNS解析过程，Nginx请求转发、连接建立和保持过程、浏览器内容渲染过程，考虑的越详细越好）。 9](#_Toc503293107)

[http和https区别，https在请求时额外的过程，https是如何保证数据安全的 11](#_Toc503293108)

[POST和GET区别 11](#_Toc503293109)

[DNS解析过程 12](#_Toc503293110)

[TCP如何保证数据的可靠传输的（这个问题可以引申出很多子问题，拥塞控制慢开始、拥塞避免、快重传、滑动窗口协议、停止等待协议、超时重传机制，最好都能掌握） 13](#_Toc503293111)

[二、数据库 13](#_Toc503293112)

[基础部分 13](#_Toc503293113)

[事务四大特性（ACID） 13](#_Toc503293114)

[数据库隔离级别，每个级别会引发什么问题，mysql默认是哪个级别 14](#_Toc503293115)

[MYSQL的两种存储引擎区别（事务、锁级别等等），各自的适用场景 15](#_Toc503293116)

[数据库的优化（从sql语句优化和索引两个部分回答） 16](#_Toc503293117)

[索引有B+索引和hash索引，各自的区别 16](#_Toc503293118)

[B+索引数据结构，和B树的区别 16](#_Toc503293119)

[索引的分类（主键索引、唯一索引），最左前缀原则，哪些情况索引会失效 17](#_Toc503293120)

[聚集索引和非聚集索引区别。 17](#_Toc503293121)

[有哪些锁（乐观锁悲观锁），select时怎么加排它锁 18](#_Toc503293122)

[关系型数据库和非关系型数据库区别 18](#_Toc503293123)

[数据库三范式，根据某个场景设计数据表（可以通过手绘ER图） 19](#_Toc503293124)

[数据库的主从复制 20](#_Toc503293125)

[使用explain优化sql和索引 20](#_Toc503293126)

[内连接、外连接、交叉连接、笛卡儿积等 20](#_Toc503293127)

[深入 20](#_Toc503293128)

[MVCC机制 20](#_Toc503293129)

[根据具体场景，说明版本控制机制 20](#_Toc503293130)

[死锁怎么解决 20](#_Toc503293131)

[varchar和char的使用场景。 20](#_Toc503293132)

[Redis 21](#_Toc503293133)

[redis数据结构有哪些 21](#_Toc503293134)

[redis队列应用场景 21](#_Toc503293135)

[redis和Memcached（支持数据持久化） 22](#_Toc503293136)

[分布式使用场景（储存session等） 22](#_Toc503293137)

[发布/订阅使用场景 22](#_Toc503293138)

[三、操作系统 22](#_Toc503293139)

[内存的页面置换算法 22](#_Toc503293140)

[进程调度算法 23](#_Toc503293141)

[进程间通信方式 24](#_Toc503293142)

[进程线程区别 25](#_Toc503293143)

[父子进程、孤儿进程 25](#_Toc503293144)

[fork进程时的操作， 25](#_Toc503293145)

[四、算法(Python实现) 25](#_Toc503293146)

[二叉树相关（层次遍历、求深度、求两个节点距离、翻转二叉树、前中后序遍历） 26](#_Toc503293147)

[链表相关（插入节点、链表逆置、使用链表进行大数字的加减，双向链表实现队列、寻找链表中的环） 27](#_Toc503293148)

[堆（大量数据中寻找最大N个数字几乎每次都会问，还有堆在插入时进行的调整） 27](#_Toc503293149)

[排序（八大排序，各自的时间复杂度、排序算法的稳定性。快排几乎每次都问） 29](#_Toc503293150)

[二分查找（一般会深入，如寻找数组总和为K的两个数字） 29](#_Toc503293151)

[两个栈实现队列。 29](#_Toc503293152)

[图（深度广度优先遍历、单源最短路径、最小生成树） 30](#_Toc503293153)

[动态规划问题。 30](#_Toc503293154)

[具体题目 30](#_Toc503293155)

[五、LINUX 30](#_Toc503293156)

[硬链接和软连接区别 30](#_Toc503293157)

[kill用法，某个进程杀不掉的原因（进入内核态，忽略kill信号） 30](#_Toc503293158)

[linux用过的命令 31](#_Toc503293159)

[系统管理命令（如查看内存使用、网络情况） 31](#_Toc503293160)

[管道的使用 | 31](#_Toc503293161)

[grep的使用，一定要掌握，每次都会问在文件中查找 31](#_Toc503293162)

[shell脚本 31](#_Toc503293163)

[find命令 31](#_Toc503293164)

[awk使用 31](#_Toc503293165)

[六、项目(自行总结) 31](#_Toc503293166)

[项目中遇到的困难（提前想好，并且把实现或者优化方法说清楚） 32](#_Toc503293167)

[系统的量级、pv、uv等 32](#_Toc503293168)

[应对高并发的解决办法（分布式） 32](#_Toc503293169)

[在项目中主要负责了哪些工作。 32](#_Toc503293170)

[nginx的负载均衡 32](#_Toc503293171)

[分布式缓存的一致性，服务器如何扩容（哈希环） 32](#_Toc503293172)

[项目中用到了哪些技术 32](#_Toc503293173)

[服务器性能如何测试 32](#_Toc503293174)

[项目中印象深刻的东西 32](#_Toc503293175)

[项目中遇到过哪些坑，怎么解决的 32](#_Toc503293176)

[项目中为什么用redis以及怎么实现相关功能 32](#_Toc503293177)

[用过的开源框架（路由问题），框架的原理 32](#_Toc503293178)

[消息队列的应用场景（想一个） 32](#_Toc503293179)

[有没有用到非关系型数据库 32](#_Toc503293180)

[网站负载变大时怎么办 32](#_Toc503293181)

[SOA? 32](#_Toc503293182)

[WebService? 32](#_Toc503293183)

[《大型网站技术架构：核心原理与案例分析》 32](#_Toc503293184)

[redis的消息队列 redis的数据类型 32](#_Toc503293185)

[OAuth认证过程 32](#_Toc503293186)

[单点登陆系统 分布式缓存的使用场景 32](#_Toc503293187)

[一个Controller调用两个Service，这两Service又都分别调用两个Dao，问其中用到了几个数据库连接池的连接？ 32](#_Toc503293188)

[图片是怎么存储的 32](#_Toc503293189)

[分布式一致性协议、分布式锁（应用场景） 32](#_Toc503293190)

[项目中的网络编程 33](#_Toc503293191)

[划一下项目的架构 33](#_Toc503293192)

[MVC的路由机制怎么实现的（路由很重要） 33](#_Toc503293193)

[为什么要用hadoop 33](#_Toc503293194)

[自己写的接口如何防止被别人恶意调用？印象笔记 33](#_Toc503293195)

[如何实现数据容灾， 33](#_Toc503293196)

[怎样实现负载均衡 33](#_Toc503293197)

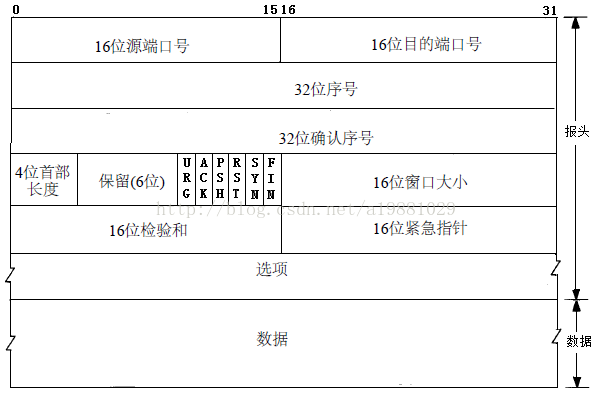
[如何解决单节点负载过 33](#_Toc503293198)

[Redis和memcache区别 33](#_Toc503293199)

[redis的持久化 33](#_Toc503293200)

# 一、计算机网络

### TCP报头格式



### UDP报头格式



### TCP/UDP区别（不仅是宏观上的，最好能根据各自的机制讲解清楚）

1.基于连接与无连接；

2.对系统资源的要求（TCP较多，UDP少）；

3.UDP程序结构较简单；

4.流模式与数据报模式 ；

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

### HTTP状态码（最好结合使用场景，比如在缓存命中时使用哪个）

1XX：消息

2XX：成功

3XX：重定向

4XX：请求错误

5XX、6XX：服务器错误

### HTTP协议（一些报头字段的作用，如cace-control、keep-alive）

请求报文包含三部分：

a、请求行：包含请求方法、URI、HTTP版本信息

b、请求首部字段

c、请求内容实体

响应报文包含三部分：

a、状态行：包含HTTP版本、状态码、状态码的原因短语

b、响应首部字段

c、响应内容实体

HTTP通信机制是在一次完整的HTTP通信过程中，Web浏览器与Web服务器之间将完成下列7个步骤：

1. 建立TCP连接

在HTTP工作开始之前，Web浏览器首先要通过网络与Web服务器建立连接，该连接是通过TCP来完成的，该协议与IP协议共同构建 Internet，即著名的TCP/IP协议族，因此Internet又被称作是TCP/IP网络。HTTP是比TCP更高层次的应用层协议，根据规则， 只有低层协议建立之后才能，才能进行更层协议的连接，因此，首先要建立TCP连接，一般TCP连接的端口号是80。

2. Web浏览器向Web服务器发送请求行

一旦建立了TCP连接，Web浏览器就会向Web服务器发送请求命令。例如：GET /sample/hello.jsp HTTP/1.1。

3. Web浏览器发送请求头

浏览器发送其请求命令之后，还要以头信息的形式向Web服务器发送一些别的信息，之后浏览器发送了一空白行来通知服务器，它已经结束了该头信息的发送。

4. Web服务器应答

客户机向服务器发出请求后，服务器会客户机回送应答， HTTP/1.1 200 OK ，应答的第一部分是协议的版本号和应答状态码。

5. Web服务器发送应答头

正如客户端会随同请求发送关于自身的信息一样，服务器也会随同应答向用户发送关于它自己的数据及被请求的文档。

6. Web服务器向浏览器发送数据

Web服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着，它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据。

7. Web服务器关闭TCP连接

一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，然后如果浏览器或者服务器在其头信息加入了这行代码：

Connection:keep-alive

TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

建立TCP连接->发送请求行->发送请求头->（到达服务器）发送状态行->发送响应头->发送响应数据->断TCP连接

### 常用的HTTP方法有哪些？

GET： 用于请求访问已经被URI（统一资源标识符）识别的资源，可以通过URL传参给服务器

POST：用于传输信息给服务器，主要功能与GET方法类似，但一般推荐使用POST方式。

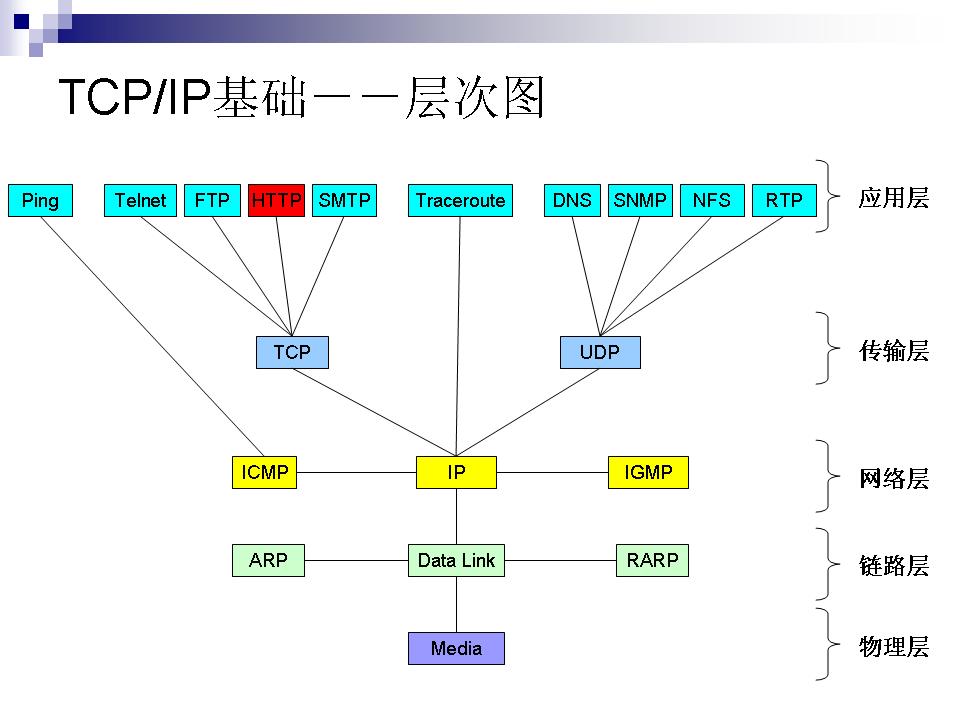
PUT： 传输文件，报文主体中包含文件内容，保存到对应URI位置。

HEAD： 获得报文首部，与GET方法类似，只是不返回报文主体，一般用于验证URI是否有效。

DELETE：删除文件，与PUT方法相反，删除对应URI位置的文件。

OPTIONS：查询相应URI支持的HTTP方法。

### OSI协议、TCP/IP协议以及每层对应的协议。



### SESSION机制、cookie机制

太多了。。有点放不下

常用的会话跟踪技术是Cookie与Session。**Cookie通过在客户端记录信息确定用户身份**，**Session通过在服务器端记录信息确定用户身份**。

Web应用程序是使用HTTP协议传输数据的。**HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接。这就意味着服务器无法从连接上跟踪会话**。

**Cookie机制：**

Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

**Session机制：**

除了使用Cookie，Web应用程序中还经常使用Session来记录客户端状态。**Session是服务器端使用的一种记录客户端状态的机制**，使用上比Cookie简单一些，相应的也**增加了服务器的存储压力**。

Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

### TCP三次握手、四次挥手（这个问题真的要回答吐了，不过真的是面试官最喜欢问的，建议每天手撸一遍，而且不只是每次请求的过程，各种FIN\_WAIT、TIME\_WAIT状态也要掌握）。

**TCP三次握手**

1 主机A通过向主机B 发送一个含有同步序列号的标志位的数据段给主机B ,向主机B 请求建立连接,通过这个数据段,

主机A告诉主机B 两件事:我想要和你通信;你可以用哪个序列号作为起始数据段来回应我.

2 主机B 收到主机A的请求后,用一个带有确认应答(ACK)和同步序列号(SYN)标志位的数据段响应主机A,也告诉主机A两件事:

我已经收到你的请求了,你可以传输数据了;你要用哪佧序列号作为起始数据段来回应我

3 主机A收到这个数据段后,再发送一个确认应答,确认已收到主机B 的数据段:"我已收到回复,我现在要开始传输实际数据了

这样3次握手就完成了,主机A和主机B 就可以传输数据了.

3次握手的特点

没有应用层的数据

SYN这个标志位只有在TCP建产连接时才会被置1

握手完成后SYN标志位被置0

**TCP四次挥手**

1 当主机A完成数据传输后,将控制位FIN置1,提出停止TCP连接的请求

2 主机B收到FIN后对其作出响应,确认这一方向上的TCP连接将关闭,将ACK置1

3 由B 端再提出反方向的关闭请求,将FIN置1

4 主机A对主机B的请求进行确认,将ACK置1,双方向的关闭结束.

由TCP的三次握手和四次断开可以看出,TCP使用面向连接的通信方式,大大提高了数据通信的可靠性,使发送数据端

和接收端在数据正式传输前就有了交互,为数据正式传输打下了可靠的基础

### 打开网页到页面显示之间的过程（涵盖了各个方面，DNS解析过程，Nginx请求转发、连接建立和保持过程、浏览器内容渲染过程，考虑的越详细越好）。

1. 在浏览器地址栏输入URL
2. 浏览器查看缓存，如果请求资源在缓存中并且新鲜，跳转到转码步骤
   1. 如果资源未缓存，发起新请求
   2. 如果已缓存，检验是否足够新鲜，足够新鲜直接提供给客户端，否则与服务器进行验证。
   3. 检验新鲜通常有两个HTTP头进行控制Expires和Cache-Control：
      * HTTP1.0提供Expires，值为一个绝对时间表示缓存新鲜日期
      * HTTP1.1增加了Cache-Control: max-age=,值为以秒为单位的最大新鲜时间
3. 浏览器解析URL获取协议，主机，端口，path
4. 浏览器组装一个HTTP（GET）请求报文
5. 浏览器获取主机ip地址，过程如下：
   1. 浏览器缓存
   2. 本机缓存
   3. hosts文件
   4. 路由器缓存
   5. ISP DNS缓存
   6. DNS递归查询（可能存在负载均衡导致每次IP不一样）
6. 打开一个socket与目标IP地址，端口建立TCP链接，三次握手如下：
   1. 客户端发送一个TCP的SYN=1，Seq=X的包到服务器端口
   2. 服务器发回SYN=1， ACK=X+1， Seq=Y的响应包
   3. 客户端发送ACK=Y+1， Seq=Z
7. TCP链接建立后发送HTTP请求
8. 服务器接受请求并解析，将请求转发到服务程序，如虚拟主机使用HTTP Host头部判断请求的服务程序
9. 服务器检查HTTP请求头是否包含缓存验证信息如果验证缓存新鲜，返回304等对应状态码
10. 处理程序读取完整请求并准备HTTP响应，可能需要查询数据库等操作
11. 服务器将响应报文通过TCP连接发送回浏览器
12. 浏览器接收HTTP响应，然后根据情况选择关闭TCP连接或者保留重用，关闭TCP连接的四次握手如下：
    1. 主动方发送Fin=1， Ack=Z， Seq= X报文
    2. 被动方发送ACK=X+1， Seq=Z报文
    3. 被动方发送Fin=1， ACK=X， Seq=Y报文
    4. 主动方发送ACK=Y， Seq=X报文
13. 浏览器检查响应状态吗：是否为1XX，3XX， 4XX， 5XX，这些情况处理与2XX不同
14. 如果资源可缓存，进行缓存
15. 对响应进行解码（例如gzip压缩）
16. 根据资源类型决定如何处理（假设资源为HTML文档）
17. 解析HTML文档，构件DOM树，下载资源，构造CSSOM树，执行js脚本，这些操作没有严格的先后顺序，以下分别解释
18. 构建DOM树：
    1. Tokenizing：根据HTML规范将字符流解析为标记
    2. Lexing：词法分析将标记转换为对象并定义属性和规则
    3. DOM construction：根据HTML标记关系将对象组成DOM树
19. 解析过程中遇到图片、样式表、js文件，启动下载
20. 构建CSSOM树：
    1. Tokenizing：字符流转换为标记流
    2. Node：根据标记创建节点
    3. CSSOM：节点创建CSSOM树
21. 根据DOM树和CSSOM树构建渲染树:
    1. 从DOM树的根节点遍历所有可见节点，不可见节点包括：1）script,meta这样本身不可见的标签。2)被css隐藏的节点，如display: none
    2. 对每一个可见节点，找到恰当的CSSOM规则并应用
    3. 发布可视节点的内容和计算样式
22. js解析如下：
    1. 浏览器创建Document对象并解析HTML，将解析到的元素和文本节点添加到文档中，此时document.readystate为loading
    2. HTML解析器遇到没有async和defer的script时，将他们添加到文档中，然后执行行内或外部脚本。这些脚本会同步执行，并且在脚本下载和执行时解析器会暂停。这样就可以用document.write()把文本插入到输入流中。同步脚本经常简单定义函数和注册事件处理程序，他们可以遍历和操作script和他们之前的文档内容
    3. 当解析器遇到设置了async属性的script时，开始下载脚本并继续解析文档。脚本会在它下载完成后尽快执行，但是解析器不会停下来等它下载。异步脚本禁止使用document.write()，它们可以访问自己script和之前的文档元素
    4. 当文档完成解析，document.readState变成interactive
    5. 所有defer脚本会按照在文档出现的顺序执行，延迟脚本能访问完整文档树，禁止使用document.write()
    6. 浏览器在Document对象上触发DOMContentLoaded事件
    7. 此时文档完全解析完成，浏览器可能还在等待如图片等内容加载，等这些内容完成载入并且所有异步脚本完成载入和执行，document.readState变为complete,window触发load事件
23. 显示页面（HTML解析过程中会逐步显示页面）

### http和https区别，https在请求时额外的过程，https是如何保证数据安全的

a、通信使用明文不加密，内容可能被窃听

b、不验证通信方身份，可能遭到伪装

c、无法验证报文完整性，可能被篡改

HTTPS就是HTTP加上加密处理（一般是SSL安全通信线路）+认证+完整性保护

### POST和GET区别

1.GET提交，请求的数据会附在URL之后（就是把数据放置在HTTP协议头＜request-line＞中），以?分割URL和传输数据，多个参数用&连接;例如：login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0 %E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如： %E4%BD%A0%E5%A5%BD，其中％XX中的XX为该符号以16进制表示的ASCII。

  POST提交：把提交的数据放置在是HTTP包的包体＜request-body＞中。上文示例中红色字体标明的就是实际的传输数据

  因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变

2.传输数据的大小：

   首先声明,HTTP协议没有对传输的数据大小进行限制，HTTP协议规范也没有对URL长度进行限制。 而在实际开发中存在的限制主要有：

   GET:特定浏览器和服务器对URL长度有限制，例如IE对URL长度的限制是2083字节(2K+35)。对于其他浏览器，如Netscape、FireFox等，理论上没有长度限制，其限制取决于操作系统的支持。

   因此对于GET提交时，传输数据就会受到URL长度的限制。

   POST:由于不是通过URL传值，理论上数据不受限。但实际各个WEB服务器会规定对post提交数据大小进行限制，Apache、IIS6都有各自的配置。

3.安全性：

    POST的安全性要比GET的安全性高。注意：这里所说的安全性和上面GET提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的Security的含义，比如：通过GET提交数据，用户名和密码将明文出现在URL上，因为(1)登录页面有可能被浏览器缓存， (2)其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了。

**区别一：**

get重点在从服务器上获取资源，post重点在向服务器发送数据；

**区别二：**

get传输数据是通过URL请求，以field（字段）= value的形式，置于URL后，并用"?"连接，多个请求数据间用"&"连接，如http://127.0.0.1/Test/login.action?name=admin&password=admin，这个过程用户是可见的；

post传输数据通过Http的post机制，将字段与对应值封存在请求实体中发送给服务器，这个过程对用户是不可见的；

**区别三：**

Get传输的数据量小，因为受URL长度限制，但效率较高；

Post可以传输大量数据，所以上传文件时只能用Post方式；

**区别四：**

get是不安全的，因为URL是可见的，可能会泄露私密信息，如密码等；

post较get安全性较高；

**区别五：**

get方式只能支持ASCII字符，向服务器传的中文字符可能会乱码。

post支持标准字符集，可以正确传递中文字符。

### DNS解析过程

DNS是应用层协议，事实上他是为其他应用层协议工作的，包括不限于HTTP和SMTP以及FTP，用于将用户提供的主机名解析为ip地址。  
具体过程如下：  
①用户主机上运行着DNS的客户端，就是我们的PC机或者手机客户端运行着DNS客户端了  
②浏览器将接收到的url中抽取出域名字段，就是访问的主机名，比如

http://www.baidu.com/

, 并将这个主机名传送给DNS应用的客户端  
③DNS客户机端向DNS服务器端发送一份查询报文，报文中包含着要访问的主机名字段（中间包括一些列缓存查询以及分布式DNS集群的工作）  
④该DNS客户机最终会收到一份回答报文，其中包含有该主机名对应的IP地址  
⑤一旦该浏览器收到来自DNS的IP地址，就可以向该IP地址定位的HTTP服务器发起TCP连接

### TCP如何保证数据的可靠传输的（这个问题可以引申出很多子问题，拥塞控制慢开始、拥塞避免、快重传、滑动窗口协议、停止等待协议、超时重传机制，最好都能掌握）

1、应用数据被分割成TCP认为最适合发送的数据块。

2、超时重传：当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

3、TCP给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。

4、校验和：TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。

5、TCP的接收端会丢弃重复的数据。

6、流量控制：TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的我数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP使用的流量控制协议是可变大小的滑动窗口协议。

7、拥塞控制：当网络拥塞时，减少数据的发送。

# 二、数据库

## 基础部分

### 事务四大特性（ACID）

数据库如果支持事务的操作，那么就具备以下四个特性：

　　1、原子性(Atomicity)

　　事务是数据库的逻辑工作单位，事务中包括的诸操作要么全做，要么全不做。

　　2、一致性(Consistency)

　　事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

　　3、隔离性(Isolation)

　　一个事务的执行不能被其他事务干扰。

　　4、持续性/永久性(Durability)

　　一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。

### 数据库隔离级别，每个级别会引发什么问题，mysql默认是哪个级别

　MySQL数据库针对这四种特性，为我们提供的四种隔离级别，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

　　1、Read uncommitted 读未提交

　　公司发工资了，把50000元打到我的账号上，但是该事务并未提交，而我正好去查看账户，发现工资已经到账，是50000元整，非常高兴。可是不幸的是，领导发现发给的工资金额不对，是2000元，于是迅速回滚了事务，修改金额后，将事务提交，最后我实际的工资只有2000元，空欢喜一场。

　　脏读是两个并发的事务，“事务A：领导发工资”、“事务B：我查询工资账户”，事务B读取了事务A尚未提交的数据。

　　当隔离级别设置为Read uncommitted时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

　　2、Read committed 读提交

　　我拿着工资卡去消费，系统读取到卡里确实有2000元，而此时老婆也正好在网上转账，把工资卡的2000元转到她账户，并在我之前提交了事务，当我扣款时，系统检查到工资卡已经没有钱，扣款失败，十分纳闷，明明卡里有钱，为何......

　　不可重复读是两个并发的事务，“事务A：消费”、“事务B：老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

　　当隔离级别设置为Read committed时，避免了脏读，但是可能会造成不可重复读。

　　3、Repeatable read 重复读

　　当隔离级别设置为Repeatable read时，可以避免不可重复读。当我拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），我老婆就不可能对该记录进行修改，也就是不能在此时转账。

　　虽然Repeatable read避免了不可重复读，但还有可能出现幻读。例如：老婆工作在银行部门，她时常通过银行内部系统查看我的信用卡消费记录。有一天，她正查询到我当月信用卡的总消费金额（select sum(amount) from transaction where month = 本月）为80元，而我此时正好在外面吃完大餐后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction ... ），并提交了事务，随后老婆将我的当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，老婆很诧异，以为出现了幻觉，幻读就这样产生了。

注：MySQL的默认隔离级别就是Repeatable read。



总结：

　　·Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

　　·Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

　　·Read committed (读已提交)：可避免脏读的发生。

　　·Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

　　以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。

　　在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

　　在MySQL数据库中查看当前事务的隔离级别：

　　1select @@tx\_isolation;

　　在MySQL数据库中设置事务的隔离 级别：

　　1set [glogal session] transactionisolationlevel 隔离级别名称;

### MYSQL的两种存储引擎区别（事务、锁级别等等），各自的适用场景

MyISAM引擎设计简单，数据以紧密格式存储，所以某些读取场景下性能很好。

但是MyISAM最典型的还是表锁问题，这样会导致长期处于"Locked"状态。而且数据恢复时间长，无事务等问题或短板。

虽然5.1之前MyISAM的读比Innodb快很多，但是在5.1之后，默认引擎已经变为Innodb。

Innodb读写有很大的提高，采用MVCC来支持高并发，针对行加锁，是使用最广泛的存储引擎。

官方建议尽量将MyISAM都换为Innodb。

### 数据库的优化（从sql语句优化和索引两个部分回答）

太多了。。

### 索引有B+索引和hash索引，各自的区别

1、B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接。在B+树上的常规检索，从根节点到叶子节点的搜索效率基本相当，不会出现大幅波动，而且基于索引的顺序扫描时，也可以利用双向指针快速左右移动，效率非常高。

2、哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快。

3、B+树索引和哈希索引的明显区别是：

* **如果是等值查询，那么哈希索引明显有绝对优势**，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；
* 从示意图中也能看到，**如果是范围查询检索，这时候哈希索引就毫无用武之地了**，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；
* 同理，**哈希索引也没办法利用索引完成排序**，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；
* **哈希索引也不支持多列联合索引的最左匹配规则**；
* B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，**在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题**。

### B+索引数据结构，和B树的区别

相对于B树，

（1）B+树空间利用率更高，可减少I/O次数，

     一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。而因为B+树的内部节点只是作为索引使用，而不像B-树那样每个节点都需要存储硬盘指针。

     也就是说：B+树中每个非叶节点没有指向某个关键字具体信息的指针，所以每一个节点可以存放更多的关键字数量，即一次性读入内存所需要查找的关键字也就越多，减少了I/O操作。

 e.g.假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内   部结点需要2个盘快。而**B+**树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比**B+**树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

（2）增删文件（节点）时，效率更高，

**因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。**

（3）B+树的查询效率更加稳定，

因为B+树的每次查询过程中，都需要遍历从根节点到叶子节点的某条路径。所有关键字的查询路径长度相同，导致每一次查询的效率相当。

### 索引的分类（主键索引、唯一索引），最左前缀原则，哪些情况索引会失效

**Mysql索引概念：**  
说说Mysql索引，看到一个很少比如：索引就好比一本书的目录，它会让你更快的找到内容，显然目录（索引）并不是越多越好，假如这本书1000页，有500也是目录，它当然效率低，目录是要占纸张的,而索引是要占磁盘空间的。  
**Mysql索引方式主要有两种结构：**B+树和hash.

hash:hsah索引在mysql比较少用,他以把数据的索引以hash形式组织起来,因此当查找某一条记录的时候,速度非常快.当时因为是hash结构,每个键只对应一个值,而且是散列的方式分布.所以他并不支持范围查找和排序等功能.  
B+树:b+tree是mysql使用最频繁的一个索引数据结构,数据结构以平衡树的形式来组织,因为是树型结构,所以更适合用来处理排序,范围查找等功能.相对hash索引,B+树在查找单条记录的速度虽然比不上hash索引,但是因为更适合排序等操作,所以他更受用户的欢迎.毕竟不可能只对数据库进行单条记录的操作.   
**Mysql常见索引类别有：**主键索引、唯一索引、普通索引、全文索引、组合索引

PRIMARY KEY（主键索引）  ALTER TABLE `table\_name` ADD PRIMARY KEY ( `column` ) UNIQUE(唯一索引)     ALTER TABLE `table\_name` ADD UNIQUE (`column`)  
INDEX(普通索引)     ALTER TABLE `table\_name` ADD INDEX index\_name ( `column` ) FULLTEXT(全文索引)      ALTER TABLE `table\_name` ADD FULLTEXT ( `column` )  
组合索引   ALTER TABLE `table\_name` ADD INDEX index\_name ( `column1`, `column2`, `column3` )

**Mysql各种索引区别：**  
普通索引：最基本的索引，没有任何限制  
唯一索引：与"普通索引"类似，不同的就是：索引列的值必须唯一，但允许有空值。  
主键索引：它 是一种特殊的唯一索引，不允许有空值。   
全文索引：仅可用于 MyISAM 表，针对较大的数据，生成全文索引很耗时好空间。  
组合索引：为了更多的提高mysql效率可建立组合索引，遵循”最左前缀“原则。

### 聚集索引和非聚集索引区别。

**根本区别**

    聚集索引和非聚集索引的根本区别是**表记录的排列顺序和与索引的排列顺序是否一致。**

**聚集索引**

    聚集索引表记录的排列顺序和索引的排列顺序一致，所以查询效率快，只要找到第一个索引值记录，其余就连续性的记录在物理也一样连续存放。聚集索引对应的缺点就是修改慢，因为为了保证表中记录的物理和索引顺序一致，在记录插入的时候，会对数据页重新排序。

**非聚集索引**

    非聚集索引制定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致，两种索引都采用B+树结构，非聚集索引的叶子层并不和实际数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针方式。非聚集索引层次多，不会造成数据重排。

**例子对比两种索引**

    聚集索引就类似新华字典中的拼音排序索引，都是按顺序进行，例如找到字典中的“爱”，就里面顺序执行找到“癌”。而非聚集索引则类似于笔画排序，索引顺序和物理顺序并不是按顺序存放的。

### 有哪些锁（乐观锁悲观锁），select时怎么加排它锁

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

排他锁的申请前提：没有线程对该结果集中的任何行数据使用排他锁或共享锁，否则申请会阻塞。用法： select … for update;

for update仅适用于InnoDB，且必须在事务块(BEGIN/COMMIT)中才能生效。在进行事务操作时，通过“for update”语句，MySQL会对查询结果集中每行数据都添加排他锁，其他线程对该记录的更新与删除操作都会阻塞。排他锁包含行锁、表锁。

### 关系型数据库和非关系型数据库区别

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库 类型 | 特性 | 优点 | 缺点 |
| 关系型数据库 SQLite、Oracle、mysql | 1、关系型数据库，是指采用了关系模型来组织 数据的数据库； 2、关系型数据库的最大特点就是事务的一致性； 3、简单来说，关系模型指的就是二维表格模型， 而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。 | 1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解； 2、使用方便：通用的SQL语言使得操作关系型数据库非常方便； 3、易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率； 4、支持SQL，可用于复杂的查询。 | 1、为了维护一致性所付出的巨大代价就是其读写性能比较差； 2、固定的表结构； 3、高并发读写需求； 4、海量数据的高效率读写； |
| 非关系型数据库 MongoDb、redis、HBase | 1、使用键值对存储数据； 2、分布式； 3、一般不支持ACID特性； 4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。 | 1、无需经过sql层的解析，读写性能很高； 2、基于键值对，数据没有耦合性，容易扩展； 3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。 | 1、不提供sql支持，学习和使用成本较高； 2、无事务处理，附加功能bi和报表等支持也不好； |

### 数据库三范式，根据某个场景设计数据表（可以通过手绘ER图）

**第一范式**

* 第一范式（1NF）要求数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值。
* 若某一列有多个值，可以将该列单独拆分成一个实体，新实体和原实体间是一对多的关系。
* 在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

**第二范式**

* 满足第二范式（2NF）必须先满足第一范式（1NF）。
* 第二范式要求实体中没一行的所有非主属性都必须完全依赖于主键；即：非主属性必须完全依赖于主键。
* 完全依赖：主键可能由多个属性构成，完全依赖要求不允许存在非主属性依赖于主键中的某一部分属性。
* 若存在哪个非主属性依赖于主键中的一部分属性，那么要将发生部分依赖的这一组属性单独新建一个实体，并且在旧实体中用外键与新实体关联，并且新实体与旧实体间是一对多的关系。

**第三范式**

* 满足第三范式必须先满足第二范式。
* 第三范式要求：实体中的属性不能是其他实体中的非主属性。因为这样会出现冗余。即：属性不依赖于其他非主属性。
* 如果一个实体中出现其他实体的非主属性，可以将这两个实体用外键关联，而不是将另一张表的非主属性直接写在当前表中。

### 数据库的主从复制

主从属于高可用的范畴: 主机挂了 备用马上接替. 平时只使用主机(高可用HA)

### 使用explain优化sql和索引

Explain命令在解决数据库性能上是第一推荐使用命令，大部分的性能问题可以通过此命令来简单的解决，Explain可以用来查看SQL语句的执行效 果，可以帮助选择更好的索引和优化查询语句，写出更好的优化语句。

Explain语法：explain select … from … [where …]

例如：explain select \* from news;

### 内连接、外连接、交叉连接、笛卡儿积等

不粘概念了，没用，拗口。靠理解。

## 深入

### MVCC机制

　MySQL InnoDB存储引擎，实现的是基于多版本的并发控制协议——MVCC (Multi-Version Concurrency Control) (注：与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。MVCC最大的好处，相信也是耳熟能详：读不加锁，读写不冲突。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的RDBMS，都支持了MVCC。

对于delete操作，innodb是**通过先将要删除的那一行标记为删除**，而不是马上清除这一行，因为innodb实现了MVCC，这些undo段用来实现MVCC多版本机制。锁不阻塞读，读也不阻塞写，这样大大提高了并发性。

### 根据具体场景，说明版本控制机制

# 搜不到太合适的答案 数据库的直接备份？或者把数据备份传到SVN？或者用别的工具管理控制？

### 死锁怎么解决

# 没搜到太靠谱的答案

### varchar和char的使用场景。

1、char。char存储定长数据很方便，char字段上的索引效率级高，比如定义char(10)，那么不论你存储的数据是否达到了10个字节，都要占去10个字节的空间。

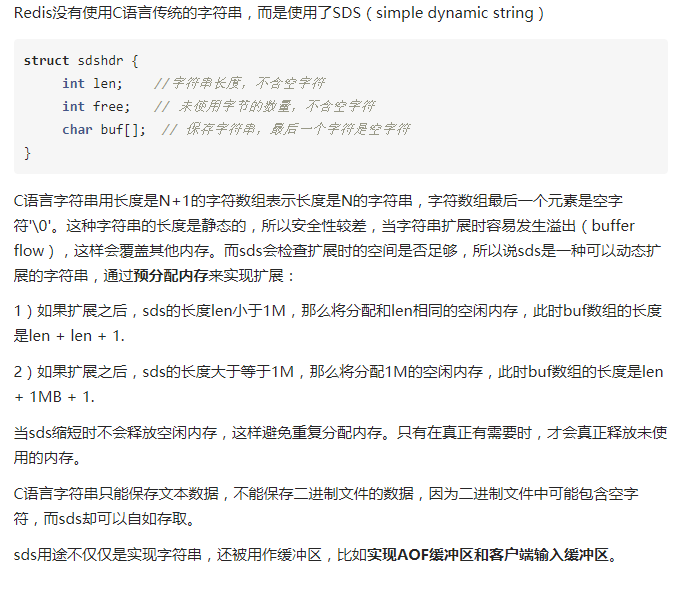
2、varchar。存储变长数据，但存储效率没有char高。如果一个字段可能的值是不固定长度的，我们只知道它不可能超过10个字符，把它定义为 varchar(10)是最合算的。varchar类型的实际长度是它的值的实际长度+1。为什么“+1”呢？这一个字节用于保存实际使用了多大的长度。   
从空间上考虑，用varchar合适；从效率上考虑，用char合适，关键是根据实际情况找到权衡点。

## Redis

### redis数据结构有哪些

这里的数据结构指的是key-value中value的数据结构，分别为string、hash、list、set、sorted set。

煊哥可以在算法与数据结构方面做如下解释：



### redis队列应用场景

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。

使用Redis作为消息队列服务场景应用案例

高并发的用户请求是网站成长过程中必不可少的过程，也是一个必须要解决的难题。在众多的实践当中，除了增加服务器数量配置服务器集群实现伸缩性架构设计之外，异步操作也被广泛采用。而异步操作中最核心的就是使用消息队列。通过消息队列将高并发用户请求进行异步操作，然后一一对消息队列进行出队的同步操作，也避免了并发控制的难题。大家可能会想到这尼玛不就是生产者消费者模式么？对的，消息队列就是生产者消费者模式的典型场景。

### redis和Memcached（支持数据持久化）

Redis虽然是基于内存的存储系统，但是它本身是支持内存数据的持久化的，而且提供两种主要的持久化策略：RDB快照和AOF日志。而memcached是不支持数据持久化操作的。

但是一个持续写入的数据库如何生成快照呢？Redis借助了fork命令的copy on write机制。在生成快照时，将当前进程fork出一个子进程，然后在子进程中循环所有的数据，将数据写成为RDB文件。

AOF文件是可识别的纯文本，它的内容就是一个个的Redis标准命令。只有那些会导致数据发生修改的命令才会追加到AOF文件。每一条修改数据的命令都生成一条日志，AOF文件会越来越大，所以Redis又提供了一个功能，叫做AOF rewrite。其功能就是重新生成一份AOF文件，新的AOF文件中一条记录的操作只会有一次，而不像一份老文件那样，可能记录了对同一个值的多次操作。其生成过程和RDB类似，也是fork一个进程，直接遍历数据，写入新的AOF临时文件。

### 分布式使用场景（储存session等）

redis实现分布式session共享

### 发布/订阅使用场景

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在Redis中，你可以设定对某一个key值进行消息发布及消息订阅，当一个key值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是构建实时消息系统，比如普通的即时聊天，群聊等功能。

# 三、操作系统

### 内存的页面置换算法

具体内容太多了。。。简写一下，还得去详细搜一下复习

1．最佳置换算法（OPT）（理想置换算法）：从主存中移出永远不再需要的页面；如无这样的页面存在，则选择最长时间不需要访问的页面。于所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。

2．先进先出置换算法（FIFO）：是最简单的页面置换算法。这种算法的基本思想是：当需要淘汰一个页面时，总是选择驻留主存时间最长的页面进行淘汰，即先进入主存的页面先淘汰。其理由是：最早调入主存的页面不再被使用的可能性最大。

3．最近最久未使用（LRU）算法：这种算法的基本思想是：利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可能也不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择在最近一段时间内最久不用的页面予以淘汰。

### 进程调度算法

1. ***先来先服务 （FCFS，first come first served）***   
   在所有调度算法中，最简单的是非抢占式的FCFS算法。   
   **算法原理**：进程按照它们请求CPU的顺序使用CPU.就像你买东西去排队，**谁第一个排，谁就先被执行，在它执行的过程中，不会中断它**。当其他人也想进入内存被执行，就要排队等着，如果在执行过程中出现一些事，他现在不想排队了，下一个排队的就补上。此时如果他又想排队了，只能站到队尾去。   
   **算法优点**：易于理解且实现简单，只需要一个队列(FIFO)，且相当公平   
   **算法缺点**：比较有利于长进程，而不利于短进程，有利于CPU 繁忙的进程，而不利于I/O 繁忙的进程
2. ***最短作业优先（SJF, Shortest Job First）***   
   短作业优先（SJF, Shortest Job First）又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，**其目标是减少平均周转时间。**   
   **算法原理**：对预计执行时间短的进程优先分派处理机。通常后来的短进程不抢先正在执行的进程。   
   **算法优点**：相比FCFS 算法，该算法可改善平均周转时间和平均带权周转时间，缩短进程的等待时间，提高系统的吞吐量。   
   **算法缺点**：对长进程非常不利，可能长时间得不到执行，且未能依据进程的紧迫程度来划分执行的优先级，以及难以准确估计进程的执行时间，从而影响调度性能。
3. ***最高响应比优先法(HRRN，Highest Response Ratio Next)***   
   最高响应比优先法(HRRN，Highest Response Ratio Next)是对FCFS方式和SJF方式的一种综合平衡。**FCFS方式只考虑每个作业的等待时间而未考虑执行时间的长短，而SJF方式只考虑执行时间而未考虑等待时间的长短**。因此，这两种调度算法在某些极端情况下会带来某些不便。HRN调度策略同时考虑每个作业的等待时间长短和估计需要的执行时间长短，从中选出响应比最高的作业投入执行。这样，即使是长作业，随着它等待时间的增加，W / T也就随着增加，也就有机会获得调度执行。这种算法是介于FCFS和SJF之间的一种折中算法。   
   **算法原理**：**响应比R定义如下： R =(W+T)/T = 1+W/T**   
   其中T为该作业估计需要的执行时间，W为作业在后备状态队列中的等待时间。每当要进行作业调度时，系统计算每个作业的响应比，选择其中R最大者投入执行。   
   **算法优点**：由于长作业也有机会投入运行，在同一时间内处理的作业数显然要少于SJF法，从而采用HRRN方式时其吞吐量将小于采用SJF 法时的吞吐量。   
   **算法缺点**：由于每次调度前要计算响应比，系统开销也要相应增加。
4. ***时间片轮转算法（RR，Round-Robin）***   
   该算法采用剥夺策略。时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法，**又称RR调度**。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。   
   **算法原理**：让就绪进程以FCFS 的方式按时间片轮流使用CPU 的调度方式，即将系统中所有的就绪进程按照FCFS 原则，排成一个队列，每次调度时将CPU 分派给队首进程，让其执行一个时间片，时间片的长度从几个ms 到几百ms。在一个时间片结束时，发生时钟中断，调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程，进程可以未使用完一个时间片，就出让CPU（如阻塞）。   
   **算法优点**：时间片轮转调度算法的特点是简单易行、平均响应时间短。   
   **算法缺点**：不利于处理紧急作业。在时间片轮转算法中，时间片的大小对系统性能的影响很大，因此时间片的大小应选择恰当   
   怎样确定时间片的大小：

**时间片大小的确定**   
1.系统对响应时间的要求   
2.就绪队列中进程的数目   
3.系统的处理能力

1. ***多级反馈队列(Multilevel Feedback Queue)***   
   多级反馈队列调度算法是一种CPU处理机调度算法，UNIX操作系统采取的便是这种调度算法。   
   **多级反馈队列调度算法描述：**   
   　　1、进程在进入待调度的队列等待时，首先进入优先级最高的Q1等待。   
   　　2、首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次优先级队列中的进程。例如：Q1,Q2,Q3三个队列，只有在Q1中没有进程等待时才去调度Q2，同理，只有Q1,Q2都为空时才会去调度Q3。   
   　　3、对于同一个队列中的各个进程，按照时间片轮转法调度。比如Q1队列的时间片为N，那么Q1中的作业在经历了N个时间片后若还没有完成，则进入Q2队列等待，若Q2的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。   
   　　4、在低优先级的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU马上分配给新到达的作业（抢占式）。   
   　　**在多级反馈队列调度算法中，如果规定第一个队列的时间片略大于多数人机交互所需之处理时间时，便能够较好的满足各种类型用户的需要。**

### 进程间通信方式

1. 管道pipe：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间使用。进程的亲缘关系通常是指父子进程关系。

2. 命名管道FIFO：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

4. 消息队列MessageQueue：消息队列是由消息的链表，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. 共享存储SharedMemory：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，如信号两，配合使用，来实现进程间的同步和通信。

6. 信号量Semaphore：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。

7. 套接字Socket：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

8. 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。

### 进程线程区别

进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

**1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.**

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。**但是线程不能够独立执行，**必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。**这就是进程和线程的重要区别。**

### 父子进程、孤儿进程

　　孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。**孤儿进程将被init进程(进程号为1)所收养**，并由init进程对它们完成状态收集工作。

　　僵尸进程：一个进程使用fork创建子进程，如果**子进程退出**，而父进程并**没有**调用wait或waitpid**获取**子进程的**状态信息**，那么子进程的**进程描述符仍然保存在系统中**。这种进程称之为僵死进程。

### fork进程时的操作，

# 四、算法(Python实现)

剑指OFFER的各个题目是最常见的，即使不是原题也是题目的变体，因为面试不像笔试，一般不会出特别困难的题目，所以剑指OFFER上小而精的题目就非常适合。建议手刷一遍。

### 二叉树相关（层次遍历、求深度、求两个节点距离、翻转二叉树、前中后序遍历）

**class node**(object)**:  
 def \_\_init\_\_**(self, *data***=**None, *left***=**None, *right***=**None)**:** self.data **=** *data* self.left **=** *left* self.right **=** *right*

*# 深度***def depth**(*tree*)**:  
 if** *tree* **==** None**:  
 return** 0  
 left, right **= depth**(*tree*.left), **depth**(*tree*.right)  
 **return** max(left, right) **+** 1

*# 前序遍历***def pre\_order**(*tree*)**:  
 if** *tree* **==** None**:  
 return  
 print** *tree*.data  
 **pre\_order**(*tree*.left)  
 **pre\_order**(*tree*.right)

*# 中序遍历***def mid\_order**(*tree*)**:  
 if** *tree* **==** None**:  
 return  
 mid\_order**(*tree*.left)  
 **print** *tree*.data  
 **mid\_order**(*tree*.right)

*# 后序遍历***def post\_order**(*tree*)**:  
 if** *tree* **==** None**:  
 return  
 post\_order**(*tree*.left)  
 **post\_order**(*tree*.right)  
 **print** *tree*.data

**if** \_\_name\_\_ **==** '\_\_main\_\_'**:** tree **= node**('D', **node**('B', **node**('A'), **node**('C')), **node**('E', right**=node**('G', **node**('F'))))

**def inverttree**(*self*, *treenode*)**:** *# 真正的翻转只有这8行代码* **if** *treenode* **==** None**:  
 return** None  
 temp **=** *treenode*.left  
 *treenode*.left **=** *treenode*.right  
 *treenode*.right **=** temp  
 *self*.**inverttree**(*treenode*.left)  
 *self*.**inverttree**(*treenode*.right)

树的翻转其实就是递归遍历每一个结点，把所有结点的左孩子和右孩子互换

### 链表相关（插入节点、链表逆置、使用链表进行大数字的加减，双向链表实现队列、寻找链表中的环）

*# 单向链表***class Node**(object)**:  
 def \_\_init\_\_**(self, *value*, *nextnode***=**None)**:** self.value **=** *value* self.\_nextnode **=** *nextnode* **def append**(self, *n*)**:  
 if not** isinstance(*n*, Node)**:** n **= Node**(*n*)  
 self.\_nextnode, n **=** *n*, self.\_nextnode  
 self.\_nextnode.\_nextnode **=** *n*

### 堆（大量数据中寻找最大N个数字几乎每次都会问，还有堆在插入时进行的调整）

*# 通过保留K个数的槽，然后不断去N个数中取数，如果取出的数比K个槽中的数的最小值要大，那么就替换这个最小值。所以时间复杂度为N\*logK。***import** time  
**import** random  
  
  
**def left**(*i*)**:  
 return** 2 **\*** *i* **+** 1  
  
  
**def right**(*i*)**:  
 return** 2 **\*** *i* **+** 2  
  
  
**def min\_heapify**(*a*, *heap\_size*, *i*)**:** l **= left**(*i*)  
 r **= right**(*i*)  
 **if** l **<** *heap\_size* **and** *a*[l] **<** *a*[*i*]**:** least **=** l  
 **else:** least **=** *i* **if** r **<** *heap\_size* **and** *a*[r] **<** *a*[least]**:** least **=** r  
 **if** least **!=** *i***:** *a*[*i*], *a*[least] **=** *a*[least], *a*[*i*]  
 **min\_heapify**(*a*, *heap\_size*, least)  
  
  
**def build\_min\_heap**(*a*, *heap\_size*)**:  
 for** i **in** range((*heap\_size* **-** 1) **/** 2, **-**1, **-**1)**:  
 min\_heapify**(*a*, *heap\_size*, i)  
  
  
**def heap\_sort**(*a*, *heap\_size*)**:  
 build\_min\_heap**(*a*, *heap\_size*)  
 **print** *a* **for** i **in** range(*heap\_size* **-** 1, 0, **-**1)**:** *a*[i], *a*[0] **=** *a*[0], *a*[i]  
 *# print a  
 heap\_size* **-=** 1  
 **min\_heapify**(*a*, *heap\_size*, 0)  
  
  
**def findMaxKInN**(*N*, *K*)**:** All\_Data **=** []  
 **for** i **in** range(*N*)**:** ele **=** random.**randint**(1, *N*)  
 All\_Data.**append**(ele)  
 a **=** []  
 **for** i **in** range(*K*)**:** a.**append**(All\_Data[i])  
 start **=** time.**clock**()  
 **build\_min\_heap**(a, len(a))  
 **for** i **in** range(*K*, *N*, 1)**:  
 if** a[0] **<** All\_Data[i]**:** a[0] **=** All\_Data[i]  
 **min\_heapify**(a, *K*, 0)  
 end **=** time.**clock**()  
 **print** "N=", *N*, "K=", *K* **print** "N\*logK" **+** "复杂度所用的时间", end **-** start  
 start **=** time.**clock**()  
 All\_Data.**sort**(cmp**=**None, key**=**None, reverse**=**True)  
 end **=** time.**clock**()  
 **print** "使用排序N\*logN", end **-** start  
  
  
**if** \_\_name\_\_ **==** "\_\_main\_\_"**:** N **=** 10000  
 K **=** 100  
 **while** (N **<=** 10000000)**:  
 findMaxKInN**(N, K)  
 N **\*=** 10;

### 排序（八大排序，各自的时间复杂度、排序算法的稳定性。快排几乎每次都问）

*# 快排***def qsort**(*L*)**:  
 if** len(*L*) **<=** 1**: return** *L* **return qsort**([lt **for** lt **in** *L*[1**:**] **if** lt **<** *L*[0]])

**+** [*L*[0]]

**+ qsort**([ge **for** ge **in** *L*[1**:**] **if** ge **>=** *L*[0]])

### 二分查找（一般会深入，如寻找数组总和为K的两个数字）

### 两个栈实现队列。

*# -\*- coding:utf-8 -\*-***class Solution:   
 def \_\_init\_\_**(self)**:** self.stack1 **=** []   
 self.stack2 **=** []   
 **def push**(self, *node*)**:** *# write code here* self.stack1.**append**(*node*)   
 **def pop**(self)**:** *# return xx* **if** self.stack2**:   
 return** self.stack2.**pop**()   
 **elif not** self.stack1**:   
 return** None   
 **else:   
 while** self.stack1**:** self.stack2.**append**(self.stack1.**pop**())   
 **return** self.stack2.**pop**()

### 图（深度广度优先遍历、单源最短路径、最小生成树）

### 动态规划问题。

### 具体题目

1. 一个文本文件中每一行中有一个URL，最多一万行，统计每一个URL的次数，输出到另外一个文件中，每一行前面是URL，后面是个数。

2. 单链表的逆序

3. 一个函数实现给定字符串，去除前面和后面的空格，比如“ ab cd ”，最后得到的结果是”ab cd”，不能改变字符串的地址。

4. 对比cookie和session,有一个值错误则不正确

5. 查找10的阶乘后面有几个0

6. 字符串匹配

7. 字符串移位，给出字符串abc##dfg##gh，实现将所有#移至字符串串头。输出####abcdfggh（个人认为可以用后向移位，减少移位次数）

8. 给出一颗二叉树，两个叶节点，找到这两个叶节点互连通的一条最短路径。

9. 两个日期计算天数差

10. 100个有序数组合并

11. 矩阵的最大子矩阵和

12. 给定一棵多叉树，每个节点有一个编号，现在要对节点排序，要求对于每个节点，它的父节点排在它后面,如果一棵树有N个节点，那么肯定有N-1条边。输入的数据形式为：Map.Entry

# 五、LINUX

### 硬链接和软连接区别

硬连接的作用是允许一个文件拥有多个有效路径名，这样用户就可以建立硬连接到重要文件，以防止“误删”的功能。只删除一个连接并不影响节点本身和其它的连接，只有当最后一个连接被删除后，文件的数据块及目录的连接才会被释放。也就是说，文件真正删除的条件是与之相关的所有硬连接文件均被删除。

软链接又称之为符号连接（Symbolic Link）。软链接文件类似于Windows的快捷方式。它实际上是一个特殊的文件。在符号连接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。

### kill用法，某个进程杀不掉的原因（进入内核态，忽略kill信号）

kill [ －s signal | －p ] [ －a ] pid ...

kill －l [ signal ]

3.参数

－s：指定发送的信号。

－p：模拟发送信号。

－l：指定信号的名称列表。

pid：要中止进程的ID号。

Signal：表示信号。

通常有两种情况是不能kill掉的：

一是进程已经成为僵死进程，当它的父进程将它回收或将它的父进程kill掉即可在ps输出看不到了；

二是进程正处在内核状态中，Linux进程运行时分内核和用户两种状态，当进程进入内核状态后，会屏蔽所有信号，包括SIGKIL，所以这个时候kill -9也变得无效了。

### linux用过的命令

…

### 系统管理命令（如查看内存使用、网络情况）

free命令是一个快速查看内存使用情况的方法

top综合显示系统资源

netstat命令的功能是显示网络连接、路由表和网络接口信息

### 管道的使用 |

…

### grep的使用，一定要掌握，每次都会问在文件中查找

…

### shell脚本

…

### find命令

…

### awk使用

…

# 六、项目(自行总结)

### 项目中遇到的困难（提前想好，并且把实现或者优化方法说清楚）

### 系统的量级、pv、uv等

### 应对高并发的解决办法（分布式）

### 在项目中主要负责了哪些工作。

### nginx的负载均衡

### 分布式缓存的一致性，服务器如何扩容（哈希环）

### 项目中用到了哪些技术

### 服务器性能如何测试

### 项目中印象深刻的东西

### 项目中遇到过哪些坑，怎么解决的

### 项目中为什么用redis以及怎么实现相关功能

### 用过的开源框架（路由问题），框架的原理

### 消息队列的应用场景（想一个）

### 有没有用到非关系型数据库

### 网站负载变大时怎么办

### SOA?

### WebService?

### 《大型网站技术架构：核心原理与案例分析》

### redis的消息队列 redis的数据类型

### OAuth认证过程

### 单点登陆系统 分布式缓存的使用场景

### 一个Controller调用两个Service，这两Service又都分别调用两个Dao，问其中用到了几个数据库连接池的连接？

### 图片是怎么存储的

### 分布式一致性协议、分布式锁（应用场景）

### 项目中的网络编程

### 划一下项目的架构

### MVC的路由机制怎么实现的（路由很重要）

### 为什么要用hadoop

### 自己写的接口如何防止被别人恶意调用？印象笔记

### 如何实现数据容灾，

### 怎样实现负载均衡

### 如何解决单节点负载过

### Redis和memcache区别

### redis的持久化

总之要把写在简历上的项目部分熟悉一遍，技术栈、项目功能、难点都要考虑好。