1 C++虚函数原理和实现机制 虚继承

虚函数表的位置在每个对象内存地址的最前面，每一个含有虚函数的类至少有一个与之对应的虚函数表，其中，虚函数表中存放着此类所有虚函数对应的函数指针，派生类的指针会直接替换基类的指针。C实现使用结构和函数指针进行实现。而如何在虚函数表中进行查找，则是通过虚指针来调用，在内存结构中它一般都会放在类最开始的地方，而对于普通函数则不需要通过查表操作。这张虚函数表是什么时候被创建的呢？它是在编译的时候产生，否则这个类的结构信息中也不会插入虚指针的地址信息。

只要某一个类X包含虚函数，无论是它的父类或者它本身拥有，那么这个类的对象都会包含一个虚指针vptr，至于vptr要指向哪张表，取决于类X它本身是否含有虚函数。此处，类B中存在虚函数，那么它就会拥有自己的一张虚函数表。pa指向的是一个类B的对象，因此 p-vptr 指代的是类B中虚指针，所以它查找的是类B的虚函数表。

**虚函数表在编译器确定，属于常量数据区。**

虚继承是为了解决多重继承的问题，根据虚继承的特性，虚基类的构造函数由最终的子类负责构造。虚继承的作用是减少了对基类的重复，代价是增加了虚表指针的负担（更多的虚表指针）。并且还会添加一个指向父类的虚表指针（不是虚函数指针）

普通继承的话，有多少个虚基类，就有多少虚指针。

**类的大小关系运算：**

1，普通单继承，只需将自身成员变量的大小加上父类大小（父类中 有虚函数，子类中不管有没有）若父类没有虚函数，则子类大小需要加上指向虚表的指针大小。

2，普通多继承，若几个父类都有虚表，则子类与第一个父类公用一个虚表指针，其他有几个有虚函数的父类则就有几个虚表指针。

3，虚拟单继承，此时若子类有虚函数则加上一个自身的虚表指针的大小，（若没有则不加）再加上自身的成员变量大小，还要加上一个虚类指针ptr\_sonclass\_fatherclass，最后加上父类的大小。

4，多重虚拟继承，此时若子类有虚函数则加上一个自身的虚表指针的大小，（若没有则不叫）再加上自身的成员变量大小，还要加上 一个公用的虚类指针（不管有几个虚拟父类，只加一个），在加上所有父类的大小。

5、普通、虚拟混合多继承，此时子类的大小为自身大小（若子类或普通父类有虚函数，则为成员变量+虚表指针大小；若都没虚函数，则就为成员变量大小），加上一个虚类指针大小，在加上虚拟父类的大小，在加上普通父类的大小（除虚表指针，因为它和子类公用一个虚表指针）。

2 C++关键字define和const区别

①define是在预处理的时候进行全部的替换，const是在编译和运行时才起作用

②define只是简单的替换，并没有类型检查，而const是有数据类型的，必须要有内存检查

③define只是简单的内存展开，有多少地方使用就替换多少次，有多个内存副本，const定义的数据变量只有一件备份节省空间

④const因为在编译和运行期起作用可以调试

3 delete this 的使用及注意事项

this对象是必须是用new操作符分配的；delete this后，不能访问该对象任何的成员变量及虚函数（delete this回收的是数据，这包括对象的数据成员以及vtable，不包括函数代码）；delete this后，不能再访问this指针。换句话说，你不能去检查它、将它和其他指针比较、和 NULL比较、打印它、转换它，以及其它的任何事情；

作用：个人认为保证以上禁忌列表基本手段可以包括：

将析构函数私有化（如果有子类，则protected化，保证子类能够正确继承）--以保证对象必须使用new在堆上分配内存；

提供（可以在仅仅在基类中）Destroy(void)函数，里面仅有一句delete this--以保证第三方能够将分配的内存回收；

4 TCP和UDP区别

TCP与UDP区别总结：1、TCP面向连接(如打电话要先拨号建立连接);UDP是无连接的，即发送数据之前不需要建立连接;2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付;3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的;UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低(对实时应用很有用，如IP电话，实时视频会议等);4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信;5、TCP首部开销20字节;UDP的首部开销小，只有8个字节;6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

5 TCP怎么保证数据的可靠传输？

序号、确认、超时和重传

6 UDP实现TCP

添加seq/ack机制，确保数据发送到对端

添加发送和接收缓冲区，主要是用户超时重传；添加超时重传机制

详细说明：发送端发送数据时，生成一个随机seq=x，然后每一片按照数据大小分配seq。数据到达接收端后接收端放入缓存，并发送一个ack=x的包，表示对方已经收到了数据。发送端收到了ack包后，删除缓冲区对应的数据。时间到后，定时任务检查是否需要重传数据。

7 大端和小端的区别？网络字节序（大端）是哪一个及转换？

小端：低地址存放低字节，高地址存放高字节；

大端：高地址存放低字节，低地址存放高字节；

网络字节序是：大端。

uint32\_t htonl(uint32\_t hostlong); //将uint32\_t 型数据的主机序转为网络序

uint16\_t htons(uint16\_t hostshort); //将uint16\_t 型数据的主机序转为网络序

uint32\_t ntohl(uint32\_t netlong); //将uint32\_t 型数据的网络序转为主机序

uint16\_t ntohs(uint16\_t netshort); //将uint16\_t 型数据的网络序转为主机序

#define BigLittleSwap16(A)  ((((uint16)(A) & 0xff00) >> 8) | (((uint16)(A) & 0x00ff) << 8))

// 长整型大小端互换

#define BigLittleSwap32(A)  ((((uint32)(A) & 0xff000000) >> 24) | /

                                                 (((uint32)(A) & 0x00ff0000) >> 8) | /

                                                 (((uint32)(A) & 0x0000ff00) << 8) | /

                                                 (((uint32)(A) & 0x000000ff) << 24))

8 哪些排序算法是稳定的？哪些不稳定？

堆排序、快速排序、希尔排序、选择排序不是稳定的排序算法，而基数排序、冒泡排序、插入排序、归并排序是稳定的排序算法。

9 什么是乐观锁和悲观锁？

乐观锁：总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

悲观锁：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被阻塞直到拿到锁，共享锁、排它锁、更新锁和意向锁。

10 进程和线程与协程的区别

进程是资源分配的最小单位，线程是CPU调度的最小单位；进程可以说一个小实体，线程基本上不拥有系统资源，只是暂用一些计数器和栈；一个进程可能有多个线程，但是至少有一个线程；同一个进程中的线程共享该进程的所有资源；线程因为共享内存，在运行时需要同步，而进程必须要使用通信的方式进行同步；

一个进程挂掉不会影响别的进程，而一个线程挂掉会使所有的线程都挂掉。

协程：用户态的轻量级线程，协程拥有自己的寄存器上下文和栈，允许协程在特定的地方挂起恢复。

11 一个程序从源程序到可执行程序的全过程

1. 预编译：**主要处理源代码文件中的以“#”开头的预编译指令**

删除所有的#define，展开所有的宏定义；处理所有的条件预编译指令#if.处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件(#include< >引用的是编译器的类库路径里面的头文件标准的, #include“ ”引用的是你程序目录的[相对路径](https://www.baidu.com/s?wd=%E7%9B%B8%E5%AF%B9%E8%B7%AF%E5%BE%84&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)中的头文件自带的)；删除所有的注释，添加行号和文件标识，

1. 编译：把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件
2. **汇编：将汇编代码转变成机器可以执行的指令(机器码文件)。**产生目标文件(与可执行文件格式几乎一样)xxx.o
3. **链接：**.链接的内容：把各个模块之间相互引用的部分都处理好，使得各个模块之间能够正确地衔接.c文件经过编译器、汇编器之后得到目标文件.o，目标文件再与库进行链接得到可执行文件.out。 库其实就是一组目标文件的打包，这些目标文件中都是一些常用的代码

12 C++关键字static和extern作用

static关键字

static可以用来修饰局部变量，全局变量以及函数。在不同的情况下static的作用不尽相同，具有文件作用域，全局变量具有整个源程序的作用域，static可以用来隐藏，只会被初始化一次，默认初始化为0，

external 关键字

修饰符extern用在变量或者函数的声明前，用来说明“此变量/函数是在别处定义的，要在此处引用”。extern “c”告诉编译器按照C方式进行连接

13 TCP粘包问题

TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾，不能分开。可能发送方数据小于发送窗口，所以需要进行填充，也可能发送数据过大，需要进行分割导致粘包；接收方接收速度太慢，在接受缓存区没有一次接收完整，下次的数据已经到达。

**TCP无保护消息边界的解决**

**针对这个问题，一般有3种解决方案：(1)发送固定长度的消息(2)把消息的尺寸与消息一块发送(3)使用特殊标记来区分消息间隔**

**14** 进程间通信方式

管道、消息队列、信号量、信号、共享内存、套接字

15 select、poll和epoll

select/poll/epoll函数的区别？

int select (int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

参数介绍：

n表示待测试的最大描述符编号加1；通过设置n告知内核只需要检测编号小于n的描述符空间，提高效率。

返回值大于0表示有多少个文件描述符就绪（可以通过这个提高select返回后对文件描述符检测的效率）；返回值等于0，表示超时；返回值小于0，表示出现异常。

timeout超时时间，当timeout为0时，表示select检查完立即返回，不需要等待超时时间到达。

poll

int poll (struct pollfd \*fds, unsigned int nfds, int timeout);

基本与select相同，但是poll用链表来记录需要检测的设备文件描述符，无最大检测文件描述符限制。

int epoll\_create(int size)；//创建一个epoll的句柄，内部用哈希表实现，size表示哈希表容量；目前采用epoll\_create1（int flags），内部采用红黑树实现。

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int

timeout);

1. select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。

1. poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

**epoll 可以理解为event poll**，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是**事件驱动（每个事件关联上fd）**的，此时我们对这些流的操作都是有意义的。**（复杂度降低到了O(1)）**

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的**，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

**select的几大缺点：**

**（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大**

**（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大**

**（3）select支持的文件描述符数量太小了，默认是1024**

**Poll:**

**Poll的实现和select几乎没有任何的区别，只是poll没有最大文件描述符的限制，使用的是链表**

16 epoll和select，poll

Epoll本地  
红黑树将存储epoll所监听的套接字。上面mmap出来的内存如何保存epoll所监听的套接字，必然也得有一套数据结构，epoll在实现上采用红黑树去存储所有套接字，当添加或者删除一个套接字时（epoll\_ctl），都在红黑树上去处理，红黑树本身插入和删除性能比较好，时间复杂度O(logN)。

1，epoll最大的好处是：相比于select和poll，不会随着监听fd的数目增长而降低效率

2，内核中select和poll是通过轮寻的方式来进行处理的，而轮寻的fd越多，自然耗时越多

3，epoll的实现是基于回调的，如果fd有期望的事件发生，那么就通过回调函数将其加入epoll的就绪队列中。也就是说： epoll只关心活跃的fd，与fd的数目并没有关系。

4，如何让内核把fd消息通知给用户空间呢？select和poll采取的是：内存拷贝方法，将文件描述符和消息拷贝过去（内核）

epoll采取的是：共享内存的方法，不需要进行拷贝，效率高

5，epoll不仅会告诉应用程序有i/o事件的到来，还会告诉应用程序相关的信息，而这些信息是由应用程序来填充的，所以应用程序能直接定位到事件，而不用再次的遍历i/o

**17 Linux常用命令**

命令辅助操作、命令帮助、查看文件内容（cat、more、less）、统计命令（wc）、查找命令（grep）、截取命令（awk、cut）、文本处理命令(sed)

18 MVCC

全称Mutli Version Concurreny Control,多版本并发控制，也可称之为一致性非锁定读；它通过行的多版本控制方式来读取当前执行时间数据库中的行数据。实质上使用的是快照数据，这样就可以实现不加锁读。MVCC 主要应用于 Read Commited  和 Repeatable read  两个事务隔离级别，MVCC可以认为是行级锁的一个变种，它可以在很多情况下避免加锁操作，因此开销更低

MVCC版本控制机制：

下面,我们通过InnoDB的MVCC实现来分析MVCC使怎样进行并发控制的.

InnoDB的MVCC,是通过在每行记录后面保存两个隐藏的列来实现的,这两个列，分别保存了这个行的创建时间，一个保存的是行的删除时间。

这里存储的并不是实际的时间值,而是系统版本号(可以理解为事务的ID)，每开始一个新的事务，系统版本号就会自动递增，事务开始时刻的系统版本号会作为事务的ID.

下面看一下在REPEATABLE READ隔离级别下,MVCC具体是如何操作的.

INSERT

InnoDB为新插入的每一行保存当前系统版本号作为版本号.

SELECT

InnoDB会根据以下两个条件检查每行记录:

a.InnoDB只会查找版本早于当前事务版本的数据行

(也就是,行的系统版本号小于或等于事务的系统版本号)，这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的.

b.行的删除版本要么未定义,要么大于当前事务版本号

这可以确保事务读取到的行，在事务开始之前未被删除.

只有a,b同时满足的记录，才能返回作为查询结果.

DELETE

InnoDB会为删除的每一行保存当前系统的版本号(事务的ID)作为删除标识.

UPDATE

InnoDB执行UPDATE，实际上是新插入了一行记录，并保存其创建时间为当前事务的ID，同时保存当前事务ID到要UPDATE的行的删除时间.

19 HTTPS协议和HTTP协议的区别

HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、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。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。在三次握手之后还需要使用SSL加密技术

20 https连接建立流程

（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

（5）Web服务器利用自己的私钥解密出会话密钥。

（6）Web服务器利用会话密钥加密与客户端之间的通信。

21 HTTP协议状态码

1xx：信息，服务器收到请求，需要请求者继续执行操作

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--要完成请求必须进行更进一步的操作

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

200 OK //客户端请求成功

301、302：重定向，301永久、302临时重定向

304 Not Modified — 未按预期修改文档。客户端有缓冲的文档并发出了一个条件性的请求(一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档)。服务器告诉客户，原来缓冲的文档还可以继续使用。

400 Bad Request — 服务器未能理解请求或是请求参数有误。

401 Unauthorized — 被请求的页面需要用户名和密码。

403 Forbidden — 对被请求页面的访问被禁止。

404 Not Found — 服务器无法找到被请求的页面。

500 Internal Server Error — 请求未完成。服务器遇到不可预知的情况。

501 Not Implemented — 请求未完成。服务器不支持所请求的功能。

502 Bad Gateway — 请求未完成。服务器从上游服务器收到一个无效的响应。

503 Service Unavailable — 请求未完成。服务器临时过载或当机。

504 Gateway Timeout — 网关超时。

505 HTTP Version Not Supported — 服务器不支持请求中指明的HTTP协议版本。

22 浏览器输入url的流程

a、在浏览器中输入网址，先检查本地缓存，如果有就直接弹出

b、发送至DNS服务器并获得域名对应的WEB服务器的IP地址；检查本地有没有DNS缓存，没有向本地域名服务器获取IP-顶级域名服务器-根域名服务器

c、与WEB服务器建立TCP连接；

d、浏览器向WEB服务器的发送相应的HTTP请求；

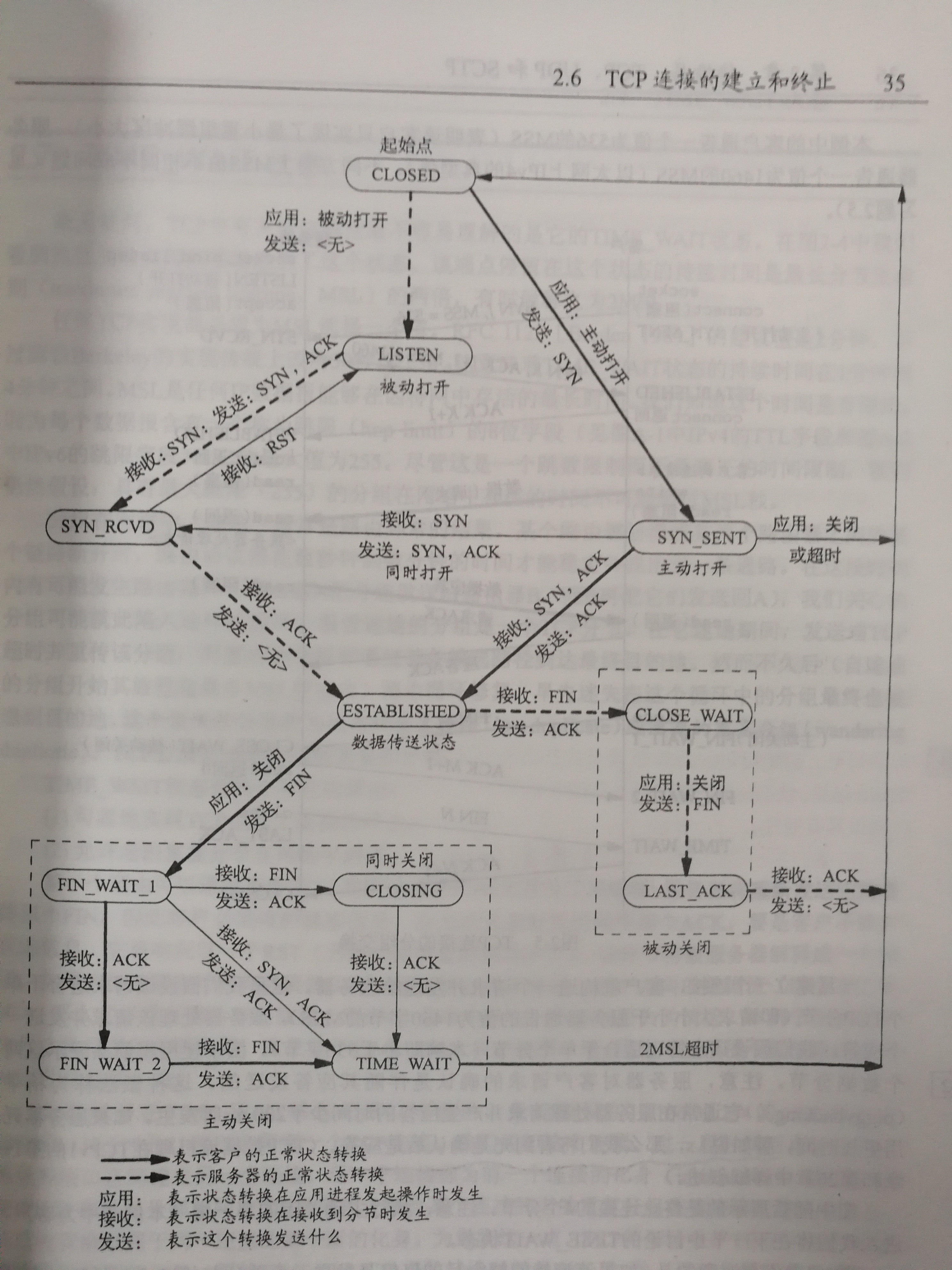
e、WEB服务器响应请求并返回指定URL的数据，或错误信息，如果设定重定向，则重定向到新的URL地址；

f、浏览器下载数据后解析HTML源文件，解析的过程中实现对页面的排版，解析完成后在浏览器中显示基础页面；

g、分析页面中的超链接并显示在当前页面，重复以上过程直至无超链接需要发送，完成全部显示；

h、如果采用的是短连接，通信完成关闭连接；如果是长连接，连接保持，等待发送新消息。

23 TCP三次握手四次挥手的11状态图



24 redis简介

a、Redis 是一个基于内存的高性能key-value数据库。Redis通过Key-Value的单值不同类型来区分, 支持以下的类型: string、List、Set、sort set、hash；

b、Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。

c、redis支持主从的模式。采用读写分离方式，利用master来插入数据，slave提供检索服务。这样可以有效减少单个机器的并发访问数量 主从模式是一个redis实例通过一定的配置告诉其相互关系

**Redis是基于内存的，持久化方式**：aof（记录服务器的写操作），rdb（通过保存内存快照）

25 使用Redis有哪些好处？

a、速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

b、 支持丰富数据类型，支持string，list，set，zset，hash

c、支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

d、丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

26 Redis学习：过期策略

redis的过期策略有定期删除（主动）和惰性删除（被动）。

**（主动）定期删除**，就是redis每隔段时间就会随机抽取一些设置了过期时间的key，检查是否过期，若过期则删除，默认是每隔100毫秒。

这里redis具体的操作是：1.从设置了过期时间的key中随机抽取20个key做测试，2.从这20个key中找到已过期的并删除，3.如果这20个key中已过期的超过了25%则从第一步再重新开始。以上操作redis每秒执行10次。

**（被动）惰性删除**，就是在获取某个key时，redis检查一下是否设置了过期时间以及是否过期，如果过期了就会删除，并且不会返回任何东西。

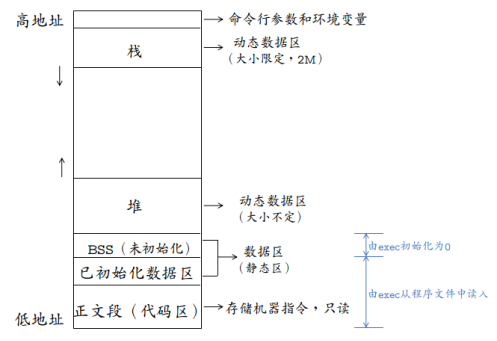
但如果定时删除漏掉了一些过期key，并且后续也没查这些key，那么也就不会走惰性删除，那么这些key也就堆积在内存中了，会消耗内存，如果已使用的内存大于maxmemory，这时redis还有一个内存淘汰机制。

内存淘汰的过程是，客户端要执行新命令，例如新增，redis先检查内存使用情况，如果大于maxmemory，根据已配置的策略清除key，然后再执行新命令。

27 Linux环境变量位置

shell环境依赖于多个文件的设置。当shell被调用时，它从两个初始文件读取命令。/etc/profile包含了系统变量，它由[系统管理员](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=795592&ss_c=ssc.citiao.link)维护，由系统管理员设置本地系统变量和特殊命令。普通用户的启动信息文件($HOME/.bash\_project)由各用户自己维护，该文件可以被修改以实现任何特定的[系统初始化](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=862917&ss_c=ssc.citiao.link)。

应该记住profile文件和.bashrc文件的区别：两个profile都只在系统启动时被读取一次，而.bashrc在系统启动和每次调用shell的时候都要被读取。



28 数据库的隔离等级

未提交读：事务中的修改即使没有提交，对其他事务也是可见的。事务可以读取其他事务修改完但未提交的数据，这种问题称为脏读。这个级别还存在不可重复读和幻读，很少使用。

提交读：多数数据库的默认隔离级别，事务只能看见已提交事务的修改。存在不可重复读，两次执行同样的查询可能会得到不同结果。

可重复读（MySQL默认的隔离级别）：解决了不可重复读，保证同一个事务中多次读取同样的记录结果一致，InnoDB 通过 MVCC 解决。但无法解决幻读，幻读指当某个事务在读取某个范围内的记录时，会产生幻行。

可串行化：最高隔离级别，通过强制事务串行执行避免幻读。在读取的每一行数据上都加锁，可能导致大量的超时和锁争用的问题。实际很少使用，只有非常需要确保数据一致性时考虑

29 数据库索引，索引的数据结构B/B+树，二者的区别？

**为什么要使用索引：**

a、表中的数据以块的形式存储，逻辑上相邻的数据块在物理上可以相隔很远。遍历查询时，需要大量的磁盘IO。

b、在对未经排序的字段中查找时，需要线性查找。复杂度为O（n）；

**使用索引的优点：**

a、大大加快查询速度

b、索引记录只包含一个地址指针，索引实体比真实数据行要小很多，索引页较数据页要密集很多。所以一个索引页可以储存更多的索引记录，这样可以减少将页面载入内存的次数。

c、建立唯一索引可以保证行的唯一性；

d、可以给非主键的列添加索引，提高根据非主键列进行查询的速度。

**使用索引的缺点：**

a、增加了数据库的存储空间；

b、在增加、删除和修改数据时，需要维护索引，增加开销；

30 常用的索引数据结构

我们常见的数据库系统，其索引使用的数据结构多是B-Tree或者B+Tree

为什么选择使用B树和B+树，为什么不用哈希表？

a、hash只能用=查询，不能用于大于、小于的范围查询；

b、hash表是无序的，无法利用hash表对键值进行排序；

c、hash无法用于组合查询；

d、hash需要处理hash冲突、扩容的问题，且占用的内存空间比B/B+树高。

B/B+树的比较：

B+树是B树的变种，相对于B树，B+树有以下不同：

a、将所有关键字（data）存储在叶子节点中，非叶子节点不存储真正的data；

b、为所有叶子节点增加了一个指向下一个链指针。

因为这些不同，在数据库索引的应用上，B+树性能要优于B树：

a、IO次数更少。由于B+树内节点无 data信息，所以一个页面可以存储更多的内结点信息，每次能索引的范围更大更精确，磁盘IO的次数更少，I/O效率更高。

b、每个关键字的查询次数一样长，查询更稳定。

c、B+树范围查询更快。

31 聚集索引和非聚集索引？

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。

a、一个表只能有一个聚集索引，但可以有多个非聚集索引

b、聚集索引查询数据速度快，聚集索引的叶子节点中存储的就是数据记录，非聚焦索引叶子节点存的是记录的指针。

c、聚焦索引查找范围内的数据很快

d、增删数据时，聚集索引维护慢。

32 怎样合理使用索引

a、记录很少时，不要使用索引；

b、对于主键和外键，创建索引；

c、在需要经常搜索的列创建索引，对需要经常被排序的列创建聚集索引；

d、对取值很少的列（比如性别）不使用索引；

e、需要频繁增加、删除、修改的列，使用非聚集索引；

使用索引的优点：

a、大大加快查询速度

b、索引记录只包含一个地址指针，索引实体比真实数据行要小很多，索引页较数据页要密集很多。所以一个索引页可以储存更多的索引记录，这样可以减少将页面载入内存的次数。

c、建立唯一索引可以保证行的唯一性；

d、可以给非主键的列添加索引，提高根据非主键列进行查询的速度。

使用索引的缺点：

a、增加了数据库的存储空间；

b、在增加、删除和修改数据时，需要维护索引，增加开销；

33 主键和外键？

若某一个属性组（注意是组）能唯一标识一条记录，该属性组就是一个主键。主键不能重复，且只能有一个，也不允许为空。定义主键主要是为了维护关系数据库的完整性。

外键用于与另一张表的关联，是能确定另一张表记录的字段。外键是另一个表的主键，可以重复，可以有多个，也可以是空值。定义外键主要是为了保持数据的一致性。

34 数据库索引，最左前缀原则，哪些情况索引会失效

索引的作用：**（1）提高查询速度；（2）确保数据的唯一性；（3）可以加速表与表之间的连接，实现表和表之间的参照完整性；**（4）使用分组和排序子句进行数据检索时，可以减少分组和排序的时间；（5）全文检索字段进行搜索优化。

索引的分类：（1）主键索引（Primay key）;(2)唯一索引；（3）常规索引；（4）全文索引；

主键索引：某一个属性组能唯一标识一条记录；

特点：最常见的索引类型；确保数据记录的唯一性；确定特定数据记录在数据库中的位置；

唯一索引：避免同一个表中某数据列中的值重复；

特点：主键索引索引只能有1个；唯一索引可以有多个；

常规索引：快速定位特定数据

特点：index 和 key关键字都可以设置常规索引，

全文索引：快速定位特定数据

数据库在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构实现高级查找算法。这种数据结构就是索引。

**最左前缀原则**：联合索引（a,b,c）必须要先比较a才可以

**这是非常重要、非常重要、非常重要（重要的事情说三遍）的原则，MySQL会一直向右匹配直到遇到范围查询 （>,<,BETWEEN,LIKE）就停止匹配。**

索引是因为B+树结构 所以查找快 如果单看第三列 是非排序的。   
多列索引是先按照第一列进行排序，**然后在第一列排好序的基础上再对第二列排序**，**如果没有第一列的话，直接访问第二列，那第二列肯定是无序的**，直接访问后面的列就用不到索引了。

.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

索引失效：

**a隐式转换导致索引失效.这一点应当引起重视.也是开发中经常会犯的错误.**

**b**对索引列进行运算导致索引失效,我所指的对索引列进行运算包括(+，-，\*，/，! 等)（where id-1=9）

c统计信息失效      需要重新搜集统计信息

d在查询条件上没有使用引导列   
e 查询的数量是大表的大部分，应该是30％以上。   
f 索引本身失效   
g 查询条件使用函数在索引列上（见12）   
h 对小表查询   
j 提示不使用索引

35 关系型库和非关系型数据库

**非关系型数据库的优势：（MongoDB、Redis）**

特点：

* 1. 非结构化存储；
  2. **部署容易，开源免费，成本低；**

优点：

1. 处理高并发、大批量数据的能力强；
2. **支持分布式集群、负载均衡，性能高**；
3. **内存级数据库，查询速度快**；
4. 没有多表查询机制的限制，扩展性高；

缺点：

1. 没有完整性约束，对于复杂业务场景支持较差；
2. 不支持sql查询；

性能：NOSQL是基于键值对的，不需要经过SQL的解析，所以性能搞

可扩展性：数据之间没有耦合关系，非常容易水平扩展

**关系型数据库的优势：**

采用了关系模型来组织数据的数据库。关系模型就是指二维表格模型，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。

关系：表名

元组：一行，记录

属性：字段，一列

域：取值范围

关键字：主键

**复杂查询**：可以用SQL语句方便的在一个表或多个表中做非常复杂的数据查询；易于维护：

难点：

* 1. 数据读写必须经过sql解析，大量数据、高并发下读写性能不足；
  2. 为保证数据一致性，需要加锁，影响并发性能；
  3. 扩展性难
  4. 数据库庞大，价格昂贵

36 主从复制

MySQL 主从复制是指数据可以从一个MySQL数据库服务器主节点复制到一个或多个从节点。MySQL 默认采用异步复制方式，这样从节点不用一直访问主服务器来更新自己的数据，数据的更新可以在远程连接上进行，从节点可以复制主数据库中的所有数据库或者特定的数据库，或者特定的表。

MySQL 主从复制主要用途：

读写分离  
在开发工作中，有时候会遇见某个sql 语句需要锁表，导致暂时不能使用读的服务，这样就会影响现有业务，使用主从复制，让主库负责写，从库负责读，这样，即使主库出现了锁表的情景，通过读从库也可以保证业务的正常运作。  
l 数据实时备份，当系统中某个节点发生故障时，可以方便的故障切换  
l高可用HA  
l架构扩展  
随着系统中业务访问量的增大，如果是单机部署数据库，就会导致I/O访问频率过高。有了主从复制，增加多个数据存储节点，将负载分布在多个从节点上，降低单机磁盘I/O访问的频率，提高单个机器的I/O性能。

37 数据库的范式

**第一范式：**

数据表中的每一列(字段)，必须是不可拆分的最小单元，也就是确保每一列的原子性。（比如字符串，int，bool之类的，不能是数组，容器等）

**第二范式**：

在满足1NF的基础上，增加两个内容：表中要有主键；非主键列不能部分依赖于主键。

例如：一个表四列：A、B、C、D，其中主键为A和B，D完全依赖与A-B主键，C只依赖于B，此时，根据第二范式，需要拆分为A、B、D一张表，B、C一张表。

**第三范式：**

满足2NF后，要求：表中的每一列都要与主键直接相关，而不是间接相关（表中的每一列只能依赖于主键）。

例如：一张表三列A、B、C，其中A为主键，B依赖于A，C依赖于B。此时需要拆分为两张表：A、B一张表，B、C一张表。

**BC范式(BCNF)：**

多个关键字，不能耦合到一个表中。此时需要拆分表。

即：A依赖B，C也依赖B，但是A和C之间没有关系。此时将导致以下问题：当B和A的对应关系改动时，整张表都需要改动；而且，当没有C时，不能单独存储A、B的关系。

38 数据库的优化

a、合理创建索引，并且只通过索引访问数据。通过索引访问数据，可以大幅度减少数据库系统访问磁盘的次数。

b、优化查询语句，每次查询只返回需要的字段，减小数据处理和传输开销。尽量不要使用模糊查询。

c、减少交互次数，客户端查询后将结果存储起来，防止对数据库做重复查询操作。

d、将大量复杂运算在客户端进行，减轻服务器CPU压力；

e、优化服务器硬件、优化网络质量；

f、使用数据库连接池

39 long query

慢查询是指超过指定时间的[SQL语句](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=12370852&ss_c=ssc.citiao.link)查询，分析MySQL语句查询性能的方法除了使用EXPLAIN输出执行计划，还可以让MySQL记录下查询超过指定时间的语句

40 数据库死锁

在数据库中有两种基本的锁类型：排它锁（Exclusive Locks，即X锁）和共享锁（Share Locks，即S锁）。当数据对象被加上排它锁时，其他的事务不能对它读取和修改。加了共享锁的数据对象可以被其他事务读取，但不能修改。数据库利用这两 种基本的锁类型来对数据库的事务进行并发控制。

解决方法：MySQL死锁检测会检测到，回滚事务。避免死锁（理论上应该叫死锁预防 （Deadlock Prevention），死锁避免（Deadlock Avoidance）通过一些算法，如银行家算法，动态的去检测加锁请求是否会产生死锁危险，很难应用在数据库用户层上），只要打破死锁发生的条件(死锁的四个条件)即可。数据库用户层面可做的，一般是破坏环路条件，按顺序加锁就不会产生环。拿这个例子来说，不管是从A转到B，还是从B转到A，我们都先写A，后写B就可以避免死锁。

41 linux常用命令，查看网络状态用什么命令？

Cd—切换当前目录；ls：查看文件与目录；grep：分析一行的信息，提取需要的信息，常和管道命令一起使用；find：查找指定目录下满足某些特点的文件；mv-移动文件目录或者更换名称；rm-删除文件或目录；cp-复制文件；ps-查看进程并输出；

kill-向pid发送信号； kill-9 3366

1 SIGHUP，启动被终止的进程 2 SIGINT，相当于输入ctrl+c，中断一个程序的进行 9 SIGKILL，强制中断一个进程的进行 15 SIGTERM，以正常的结束进程方式来终止进程 17 SIGSTOP，相当于输入ctrl+z，暂停一个进程的进行

tar：对文件进行压缩和解压

eg：压缩：tar -jcv -f filename.tar.bz2 要被处理的文件或目录名称

查询：tar -jtv -f filename.tar.bz2

解压：tar -jxv -f filename.tar.bz2 -C 欲解压缩的目录

cat：查看文本文件内容，通常与more和less一起使用，从而可以一页页查看数据

chgrp命令改变文件所属的用户组; gcc：将C语言的源程序文件，编译成可执行程序; time：用于测算一个命令（即程序）的执行时间

网络状态：netstat：显示路由表、实际的网络连接以及每一个网络接口设备的状态信息

Netstat-a显示所有的socket端口；netstat-at显示所有的TCP端口；

Netstat-au显示所有的udp端口

ln ：创建链接；

eg：ln -s source target //软链接，将文件名为target的文件链接到文件source上

ln source target //硬链接，将文件名为target的文件链接到文件source

echo：字符串输出；

sed：sed命令是一个面向字符流的非交互式编辑器，也就是说sed不允许用户与它进行交互操作。sed是按行来处理文本内容的。在shell中，使用sed来批量修改文本内容是非常方便的。sed命令的选项sed [选项] [动作]

42 统计一个文件中某个字符串出现的个数grep

a、grep

eg：grep -o ‘keyword’ my\_file.txt

grep-i -n‘keywprd’—color my\_file.txt –color高亮显示 –B1显示之前的一行-A1显示之后的一行 –C1显示之前之后的一行-n显示结果所在的行号

-i 检索关键字的时候不区分大小写

**-o:只显示符合条件的字符串，但不是整行显示，每个符合条件的字符串单独显示一行**

**-c 输入符合的总行号**

-v:输出不带关键字的行（反向查询，反向匹配）

-w:匹配整个单词，如果是字符串中包含这个单词，则不匹配

-E使用正则表达式查找

-n显示匹配字符串的行数；

-e:实现多个选项的匹配，逻辑or关系

-q：静默模式，不输出任何信息，当我们只关心有没有匹配到，缺不关心匹配到什么内容时，我们可以使用次命令，然后，使用“echo$?”查看是否匹配到，0表示匹配到，1表示没有匹配到。

b、awk

eg：awk -v RS=’keyword’ ‘END {print –NR}’ my\_file.txt

43 linux中查看系统资源的命令

top用来显示当前系统中各项资源情况，包括系统启动时间、登陆用户树、系统中的进程、CPU使用情况、物理内存使用情况。

ps用于查看当前系统中的所有进程状态。

ps -L可以查看进程的所有线程信息，线程ID表示未LWP

free用于查看系统内存使用情况；

netstat查看各网络端口情况。（netstat -a显示所有连线中的socket）

vmstat查看CPU使用情况。

uptime查看系统负载，包括平均运行的进程数量。

iostat查看系统设备的IO负载情况

lsof查看当前打开的文件，可用于查看当前端口占用情况

df查看磁盘占用情况

44 Linux文件读写使用的系统调用？

**打开、创建文件open()**

int open(const char \*pathname, int flags);

int open(const char\*pathname, int flags, mode\_t mode);

pathname：文件路径；flags：读/写/创建；mode：文件权限；

关闭文件close()

int close(int fd);

读取文件read()

ssize\_t read(int fd, void \*buf, size\_t count);

读取成功，返回读取的字节数；读取函数出错，返回-1；如果达到文件末尾，返回0；

写文件write()

ssize\_t write(int fd, const void \*buf, size\_t count);

文件偏移lseek()

off\_t lseek( int fildes, off\_t offset, int whence)；

建立内存映射函数mmap()

void \*mmap(void \*addr, size\_t length, int prot, int flags,int fd, off\_t offset);

mmap()函数将普通文件映射到进程虚拟地址空间，进程可以像访问普通内存一样对文件进行访问。 mmap()还使得进程之间可以通过映射同一个文件实现共享内存。

解除内存映射munmap()

int munmap(void \*start, size\_t length);

45 mmap函数（内存映射）

a、文件被打开后（open的过程是不能省的，缓存到内核的地址空间），使用mmap，将物理内存上的文件的一部分直接添加到进程的共享内存区，进程可以直接访问该部分内存。（通过read调用，需要陷入内核，由内核将对应部分的文件内容拷贝给进程）

b、多个线程可以通过映射物理内存中文件的同一部分，实现通信；

c、父进程调用mmap后fork子进程，子进程和父进程的共享内存区指向同一文件的同一部分，可以直接通信。

mmap函数第一种用法是映射磁盘文件到内存中；而malloc使用的mmap函数的第二种用法，即匿名映射，匿名映射不映射磁盘文件，而是向映射区申请一块内存。munmap函数是用于释放内存，第一个参数为内存首地址，第二个参数为内存的长度。接下来看下mmap函数的参数。

46 硬链接和软链接

硬链接：不存在实体文件和链接对应，通过在目录中增加一个链接到某inode号码的记录实现硬链接；

软链接：即符号链接，建立一个独立文件，文件中记录了被链接文件的路径名。

**区别：**

a、软链接对应有实体文件，硬链接没有；

b、软链接可以对文件和目录创建，硬链接不可以（如果子目录硬链接了父目录，则出现目录环）；

c、创建软链接时，文件的链接计数不会增加；硬链接会增加计数。

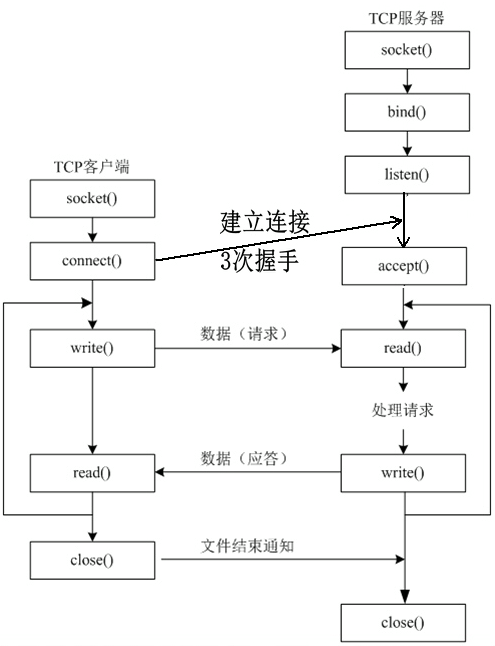
d、当链接原文件移动路径时，硬链接不受影响；软链接失效。

e、软链接可以跨文件系统创建、硬链接不可以；

47 同步和异步、阻塞和非阻塞

同步和异步其实指的是，请求发起方对消息结果的获取是主动发起的，还是等被动通知的。如果是请求方主动发起的，一直在等待应答结果（同步阻塞），或者可以先去处理其他的事情，但要不断轮询查看发起的请求是否有应答结果（同步非阻塞 ）因为不管如何都要发起方主动获取消息结果，所以形式上还是同步操作。如果是由服务方通知的，也就是请求方发出请求后，要么在一直等待通知（异步阻塞），要么就先去干自己的事了（异步非阻塞），当事情处理完成之后，服务方会主动通知请求方，它的请求已经完成，这就是异步。异步通知的方式一般是通过状态改变，消息通知，或者回调函数来完成，大多数时候采用的都是回调函数。

48 TCP服务器端



//服务器端代码

int main()

{

int listenfd = Socket(AF\_INET, SOCK\_STREAM,0);

//设置端口号和地址

Bind(listenfd, &servAddr, sizeof(servAddr));//servAddr设置协议族、IP地址、端口号

Listen(listenfd, LISTENQ); //Listen第二个参数是套接字排队的最大连接个数

Signal(SIGCHLD, sig\_chld);

for()

{

connfd = Accept(listenfd, cliAddr, cliLen); //Accept从已连接队列中取出第一个连

接，返回连接套接字

if(fock() == 0)

{

Close(listenfd); //在子进程中，不再对listenfd进行读写，但并没有关闭套接字；

while((n = readn(connfd, buf, MAXLINE)) > 0)

{

writen(connfd, buf, n)

}

exit(0);

}

Close(connfd);//父子进程共享listenfd、connfd两个连接描述符，通过引用计数的

//方式保证只有最后一个进程退出时，该描述符才被真正销毁，即连接才被断开

}

}

//信号处理函数

int sig\_chld()

{

while((pid = waitpid(-1, stat, WNOHANG)) > 0)

//处理pid子进程

}

//客户端代码

int main()

{

int connfd = Socket(AF\_INET, SOCK\_STREAM, 0);

Connect(connfd, servAddr, sizeof(servAddr));

while(Fgets(sendLine, MAXLINE, fp) != NULL)

{

Writen(connfd, sendLine, strLen(sendLine) );

Readn(socket, recvline, MAXLINE);

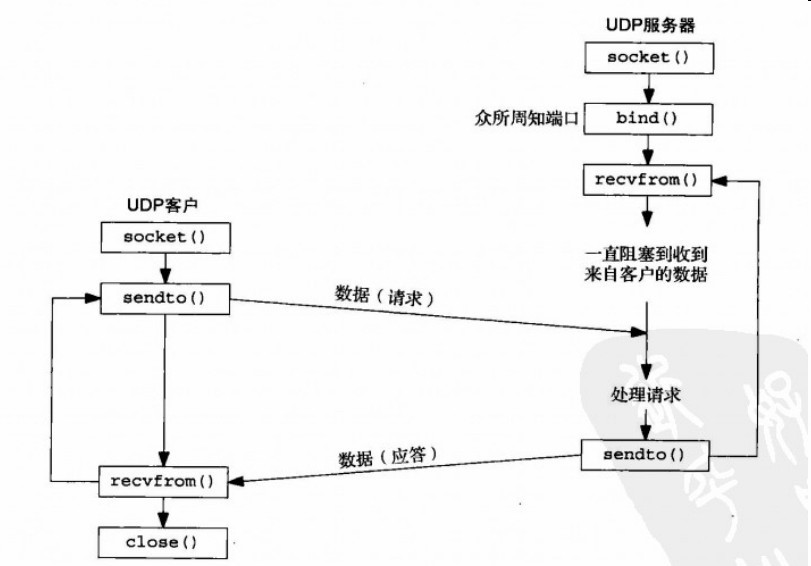
Fputs(recvline, stdout);

}

exit();

}

49 UDP服务器端



int main()

{

sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

Bind(sockfd, &servAddr, sizeof(servAddr));

for()

{

n = Recvfrom(sockfd, mesg, MAXLINE, 0, cliAddr, len);

Sendto(sockfd, mesg, n, 0, cliAddr, len);

}

}

//客户端程序

int main()

{

sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

while(Fgets(sendline, MAXLINE, fd))

{

Sendto(socket, sendline, strlen(sendline), 0, servAddr, sizeof(servAddr));

n = Recvfrom(socket, mes, MAXLINE, 0, NULL, NULL);

Fputs(mes, stdout);

}

}

50 TCP和UDP网络编程比较

TCP/UDP网络编程对比：

1、大多数TCP服务器是并发的，大多数UDP服务器是迭代的；

2、UDP客户端程序不需要调用Connect函数， 当调用Connect函数后，该套接字只能与Connect关联的客户端套接字通信；

51 UDPConnect函数

由于UDP无连接，所以UDP传输过程中产生的异步错误（例如对端关闭错误）不会返回到UDP套接口，调用connect就是为了解决这个问题。

a、UDP调用connect不会引起三次握手，仅仅通知内核记录对端的IP和端口；

b、UDP可以重复调用connect，指定新的对端IP和端口；

52 僵尸进程、孤儿进程，编程时如何避免产生僵尸进程，如何kill僵尸进程？

描述init进程，它是内核启动的第一个用户级进程。

僵尸进程：子进程先退出，父进程没有调用wait/waitpid处理它，子进程称为僵尸进程。僵尸进程进程ID仍然保存在系统中，但是不占用任何内存空间。僵尸进程父进程结束后，如果还没回收僵尸进程，僵尸进程变成孤儿进程，被init进程接管，最后由init进程回收其资源。

孤儿进程：父进程先于子进程退出，子进程称为孤儿进程。孤儿进程被init进程收养。孤儿进程退出时，init进程将回收其资源。

所以说，孤儿进程死后不占用系统进程ID，僵尸进程仍然需要占用。

**僵尸进程的避免：**

a、signal（SIGCHLD，SIG\_IGN），设置忽略子进程终止信号。这样子进程终止时，自动释放所有资源。

b、fork两次，子进程fork孙进程后直接退出，孙进程成为孤儿进程，退出时由init进程负责释放其资源。

c、通过调用wait/waitpid函数。

**僵尸进程的处理：**

a、通过ps命令找到所有进程状态为Z的进程的PID及其父进程的PPID

ps -A -ostat,ppid,pid,cmd | grep -e '^[Zz]'

b、kill子进程；

kill -HUP PID

c、如果子进程无法kill，kill父进程

53 wait/waitpid函数

pid\_t wait(int \* statloc);

pid\_t waitpid(pid\_t pid,int \*statloc,int options);

wait/waitpid用于处理已终止子进程。两个函数均返回两个值，已终止子进程的ID、子进程终止状态statloc。通过statloc可以判断子进程是正常终止、被信号杀死还是作业控制停止。

**wait/waitpid比较：**

a、waitpid可以处理多个子进程同时终止的情况。

waitpid提供了一个无阻塞的wait，通过设置options为WNOHANG，waitpid在没有子进程终止时会立即返回。那么在子进程终止信号被触发时，通过循环调用waitpid可以处理所有已终止进程。由于wait在子进程尚未终止时会阻塞，所以不能在处理函数中循环调用wait，每次只能处理第一个终止的子进程。

b、设置pid大于零，使waitpid等待指定的进程

c、waitpid支持工作控制，对终止状态（正常终止、异常终止、暂停终止）进行检查。设置options为WUNTRACED。

**waitpid中pid的含义：**

a、pid==-1 等待任何一个子进程；

b、pid >0等待进程ID与pid值相同的子进程

c、pid==0等待与调用者进程组ID相同的任意子进程

d、pid<-1等待进程组ID与pid绝对值相等的任意子进程

54 epoll介绍，即其两种触发模式，哪种效率高？

a、**LT水平触发模式**：缺省的工作方式，当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，直到程序处理完所有数据，否则下次调用epoll\_wait时，会再次响应应用程序并通知此事件。 （poll默认的方式）

b、**ET边缘触发模式**：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，epoll\_wait只被触发一次，如果数据没有处理完，下次调用epoll\_wait不会再响应。一定使用非阻塞IO，根据上面的讨论，若ET模式下使用阻塞IO，则程序一定会阻塞在最后一次write或read操作。

**效率：**

LT模式触发的次数更多；ET模式使用起来更复杂，在数据读写不完全时调用epoll\_wait会永久阻塞，所以需要应用程序自身维护就绪队列，保证数据的读写完全。

accept函数不论什么时候，都要设置为**非阻塞模式**，给出理由：TCP连接被客户端夭折，即在服务器调用accept之前，客户端主动发送RST终止连接，导致刚刚建立的连接从就绪队列中移出，如果套接口被设置成阻塞模式，服务器就会一直阻塞在accept调用上，直到其他某个客户建立一个新的连接为止。但是在此期间，服务器单纯地阻塞在accept调用上，就绪队列中的其他描述符都得不到处理

55 Shutdown和Close

a、close关闭描述符时，采用引用计数的方式，只有在计数值变为0时才关闭套接字。使用shutdown可以不管计数就激发TCP的正常终止过程。

b、close终止读和写两个方向的数据传输。shutdown可以只关闭其中一半。例如，shutdown关闭写端后，仍然可以接收从对方过来的数据，直到对方关闭连接。howto可以设置为SHUT\_RD，SHUT\_WR，SHUT\_RDWR.

int shutdown（int sockfd， int howto）；

56 recv函数的返回值及其意义？

1、接收成功，返回其实际copy的字节数（读到指定字节数量时，返回len；当读到文件结尾时，

返回值小于len）。

2、如果recv在copy时出错，那么它返回-1，并设置相应错误号（SOCKET\_ERROR）；

3、返回0表示对方关闭了套接口。

57 有哪些网络I/O？阻塞IO和多路复用IO的区别

**阻塞IO**：在IO执行的两个阶段，进程一直被阻塞。一个进程同时只能处理一个IO。

**非阻塞IO**：等待数据到达期间，如果没有数据到达，内核立即返回一个错误，进程不阻塞，继续发送请求。数据到达，从内核拷贝数据期间，进程阻塞。（ 每个进程可以同时处理多个IO请求，但是循环调用请求占用大量CPU）

**信号驱动IO：**进程提前建立SIGIO的信号处理程序，当数据到达时通知线程。所以等待数据到达期间，进程不阻塞。数据到达，内核通过SIGIO信号通知进程，进程调用recv拷贝数据，期间阻塞。（单进程可以处理多个IO，但是信号处理开销较大）

**多路复用IO**：用select、poll、epoll同时对多个IO进行监测。等待数据到达期间，进程阻塞于select、poll、epoll上，期间进程也不能处理其它任务。有数据到达时，系统调用返回，对应的IO再去拷贝数据。（每个进程可以处理多个IO）

**异步IO**：调用aio\_read函数，把IO的两个过程都交给内核，期间进程不阻塞，继续处理其它任务。由内核等待数据到达，然后将数据拷贝到用户内存空间，拷贝完成后再通过信号通知进程。（单个进程处理多个IO）

阻塞IO和多路复用IO比较：

a、都属于同步IO。在等待数据到达过程中，对于阻塞IO，进程阻塞于IO调用；对于非阻塞IO，进程阻塞于select、epoll调用；拷贝数据期间，进程阻塞。

b、阻塞IO单进程只能同时处理一个IO；

非阻塞IO和异步IO比较：

a、都支持单线程对多个IO的监控；

b、非阻塞IO是同步IO，从内核拷贝数据的过程需要进程自己完成；

c、非阻塞IO通过轮询的方式确认数据到达，CPU开销非常大。

同步和异步、阻塞和非阻塞：

a、同步和异步指的是，处理IO时，需不需要一个严格的顺序，在一个调用没有返回结果之前，该调用就不算完成；对应到IO，就是一次IO完成是不是以进程把数据拷贝到进程空间为标志，除异步IO以外的所有IO，最后都需要自己去拷贝数据，所以都是同步IO。

b、阻塞和非阻塞是等待消息时，进程有没有被挂起

58 kill 杀不死进程的原因判断进程是否异常退出

使用kill(B\_pid,0). POSIX.1将信号编号0定义为空信号。如果sign参数是0，则kill仍执行正常的错误检查，但不发送信号。这常被用来确定一个特定进程是否仍旧存在。如果向一个并不存在的进程发送空信号，则kill返回-1，errno则被设置为ESRCH。

**a**、该进程处于”Zombie”状态（使用ps命令返回defunct的进程）。 此时进程已经释放所有资源，但还未得到其父进程的确认。”zombie”进程要等到下次重启时才会消失，但它的存在不会影响系统性能。

**b**、该进程处于”kernel mode”（核心态）且在等待不可获得的资源。处于核心态的进程忽略所有信号处理，因此对于这些一直处于核心态的进程只能通过重启系统实现。进程在AIX 中会处于两种状态，即用户态和核心态。只有处于用户态的进程才可以用“kill”命令将其终止。

59 http版本间的区别

a、长连接和短连接：1.0短连接、1.1之后长连接

b、节约带宽：1.1之后支持传输资源的一部分。

c、host域：1.1在request消息头中增加了host域

d、多路复用：2.0中增加了多路复用技术

e、数据压缩：2.0增加了对header数据压缩

f、服务器推送：2.0增加了服务器给客户端推送消息的机制。

60 Get和Post的区别？

a、提交方式：Get方式通过头部的URL提交数据，数据在URL中可以看到；POST方式，数据放置在HTTP请求消息中提交。

b、安全性：使用 Get 的时候，参数会显示在地址栏上，他人可以通过URL获取参数，而 Post 不会。

c、get只允许发ASCII字符，post没有限制。

d、数据长度：GET方式提交的数据最多只能有1024字节（因为URL有长度限制），而POST则没有此限制。

e、get可以被缓存并存入书签，参数会被保存在浏览历史中；post不可以

61 守护进程

a、孤儿进程；

b、以root权限运行，使用保留端口号（1-1024）

c、非交互式程序，没有控制终端，没有任何输出。

创建守护进程的步骤：

a、在父进程中执行fork并exit推出；（保证进程不是进程组组长，否则不能调用setsid）setsid让子进程脱离

b、在子进程中调用setsid函数创建新的会话，子进程成为会话组长，并与控制终端脱离；

c、重新fork，使进程不再成为会话组长，从而禁止进程重新打开控制终端；

d、在子进程中调用chdir函数，让根目录 ”/” 成为子进程的工作目录；

e、在子进程中关闭任何不需要的文件描述符

f、重设文件创建掩模，设置进程的umask为0；

常用的守护进程：

inetd超级服务器、FTP服务器、Web服务器httpd、邮件服务器sendmail、系统日志服务syslogd。

**设想有一个守护进程管理的服务挂掉了怎么拉起来？**

重启对应的socket

62 read、readn、recv

a、read函数可以从任何文件io中读取数据，recv只能用于从套接口读取数据；

b、read函数读取数据时，可能会由于缓冲区已满，导致读不到指定大小的数据，所以通常需要在while循环中读数据；readn可以避免这个问题。

c、recv相对于read，多了第四个参数flags，当其为0时，功能和read相同。当设置为MSG\_WAITALL时，功能同readn。另外，还可以设置为MSG\_OOB接收紧急指针中的数据。设置为MSG\_PEEK可以读取数据，但是不从缓冲区中移走接收到的数据。

63 客户端bind的意义

无连接的socket的客户端和服务端以及面向连接socket的服务端通过调用bind函数来配置本地信息；需要在建连前就知道端口的话，需要bind；需要通过指定的端口来通讯的话，需要 bind。

客户端进程bind端口：由进程选择一个端口去连服务器，（如果默认情况下，调用bind函数时，内核指定的端口是同一个，那么调用多个调用了bind（）的client程序，会出现端口被占用的错误）注意这里的端口是客户端的端口。如果不分配就表示交给内核去选择一个可用端口。

客户端进程bind IP地址：相当于为发送出去的IP数据报分配了源IP地址，但交给进程分配IP地址的时候（就是这样写明了bind IP地址的时候）这个IP地址必须是主机的一个接口，不能分配一个不存在的IP。如果不分配就表示由内核根据所用的输出接口来选择源IP地址。

64 浅拷贝和深拷贝的区别

简单的来说就是，在有指针的情况下，浅拷贝只是增加了一个指针指向已经存在的内存，而深拷贝就是增加一个指针并且申请一个新的内存，使这个增加的指针指向这个新的内存，采用深拷贝的情况下，释放内存的时候就不会出现在浅拷贝时重复释放同一内存的错误。

65 线程共享进程的哪些内容，协程与线程对比？

a、线程共享进程的堆区，包括进程代码段、进程打开文件描述符、全局变量、进程目录、进程用户ID和进程组ID；

b、线程有各自独立的栈，包括线程ID、线程寄存器、线程栈；

**协程与线程对比：**

a、一个线程可以多个协程，一个进程也可以单独拥有多个协程，一个CPU同时只能有一个协程运行。

b、线程进程都是同步机制，而协程则是异步

c、协程能保留上一次调用时的状态，每次过程重入时，就相当于进入上一次调用的状态

66 线程池对线程的管理方式，包括初始化线程的方法、线程创建后的管理、指派任务的方式

线程池的组成：

a、线程池管理器，用户和线程池交互的接口类；

b、线程池，维护两个链表，空闲链表、忙绿链表。负责线程的调度和释放。

b、工作线程，处理实际的工作。

线程池的作用

a、重用线程池中的线程，减少进程的创建、销毁过程；

b、能有效的控制线程池的最大并发数，避免以为其共同抢占资源而造成的阻塞；

c、能够对线程进行简单的管理，提供制定间隔和定时执行。

线程池的适用场景

a、单位时间内处理任务频繁，且任务处理时间短；

b、对实时性要求高；

c、需要处理突然的高并发事件。

67 linux系统日志函数syslog,openlog,closelog函数

三个函数openlog， syslog， closelog是一套系统日志写入接口。另外那个vsyslog和syslog功能一样，只是参数格式不同。

原理：通常，syslog守护进程读取三种格式的记录消息。此守护进程在启动时读一个配置文件。一般来说，其文件名为/etc/syslog.conf（注释:if you want to redirect log to other place,you need to change this），该文件决定了不同种类的消息应送向何处。例如，紧急消息可被送向系统管理员（若已登录），并在控制台上显示，而警告消息则可记录到一个文件中。

68 线程是不是越多越好，一般设置几个？

**不是。**

a、使用多线程是为了进程内并发，这种并发只有对IO密集型程序才是有意义的，因为在IO阻塞阶段CPU可以切换其它进程处理其它事情。对于CPU密集型程序，进程运行期间不会阻塞，此时使用多线程还会增加线程切换开销，这样反而会使性能降低。

b、线程是操作系统的资源，操作系统对线程数有限制。一个进程中创建了太多无意义的进程，可能会使其它程序无法创建线程。

**数量：**

N核服务器，通过执行业务的单线程分析出本地计算时间为x，等待时间为y，则工作线程数（线程池线程数）设置为 N\*(x+y)/x，能让CPU的利用率最大化。所以，对于CPU密集型进程来说，设置线程数和CPU数量一致就行。

69 单核CPU设置多线程有意义吗

a、是用多线程可以让代码逻辑结构更清晰。

b、对于CPU密集型程序来说，单核CPU上使用多线程并无意义，还会增加CPU切换开销。

c、对于IO密集型程序来说，在进程调用IO阻塞时，使用多线程的话，CPU可以切换其它线程处理其它任务，可以很好的提高性能。

# 70 Linux之进程调度算法 Fork

**子进程继承父进程**

用户号UIDs和用户组号GIDs

环境Environment

堆栈

共享内存

打开文件的描述符

执行时关闭（Close-on-exec）标志

信号（Signal）控制设定

进程组号

当前工作目录

根目录

文件方式创建屏蔽字

资源限制

控制终端

**子进程独有**

进程号PID

不同的父进程号

自己的文件描述符和目录流的拷贝

子进程不继承父进程的进程正文（text），数据和其他锁定内存（memory locks）

不继承异步输入和输出

**我们说子进程和父进程共享代码，但是数据并不是共享的，也就是说，他们的虚拟地址是一样的，但是物理地址就肯定不一样了**。

**1. 先来先服务（FCFS)调度算法**

FCFS调度算法是一种最简单的调度算法，在进程调度中，FCFS调度算法每次从就绪队列中选择最先进入该队列的进程，将处理机分配给它，使之投入运行，直到完成或因某种原因而阻塞时才释放处理机。

**2.短作业优先（SJF）调度算法**

SJF（进程）优先调度算法是指对短作业（进程）优先调度的算法。短作业优先(SJF)调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法，则是从就绪队列中选择一个估计运行时间最短的进程，将处理机分配给它，使之立即执行，直到完成或发生某事件而阻塞时，才释放处理机。

**3.优先调度算法**

优先级调度算法又称优先权调度算法，该算法中的优先级用于描述作业运行的紧迫程度。在进程调度中，优先级调度算法每次从就绪队列中选择优先级最高的进程，将处理机分配给它，使之投入运行。

**4.高响应比优先调度算法**

高响应比优先调度算法主要用于作业调度，该算法是对FCFS调度算法和SJF调度算法的一种综合平衡，同时考虑每个作业的等待时间和估计的运行时间。在每次进行作业调度时，先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。

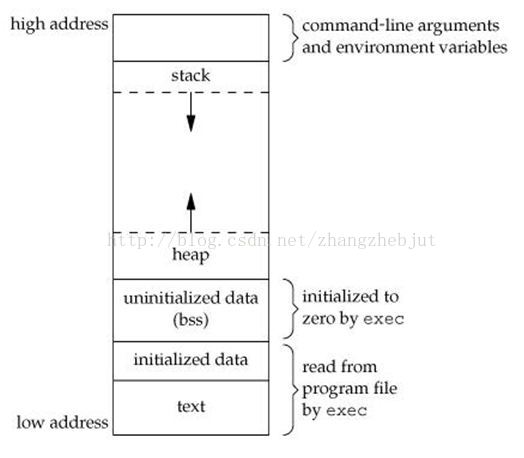
**5.时间片轮转调度算法**

时间片轮转调度算法主要适用于分时系统。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片，如100ms。在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

**6.多级反馈队列调度算法（集合了前几种算法的优点）**

多级反馈队列调度算法是时间片轮转调度算法和优先级调度算法的综合和发展，动态调整进程优先级和时间片大小，多级反馈队列调度算法可以兼顾多方面的系统目标。按优先度分级，每一级时间片比上一级高2倍，每一级里面采用先来先服务。没有执行完就插入到下一级

71 linux进程地址空间



程序段(Text):程序代码在内存中的映射，存放函数体的二进制代码。

初始化过的数据(Data):在程序运行初已经对变量进行初始化的数据。

未初始化过的数据(BSS):在程序运行初未对变量进行初始化的数据。

堆 (Heap):存储动态内存分配,需要程序员手工分配,手工释放.注意它与数据结构中的堆是两回事，分配方式类似于链表。

栈 (Stack):存储局部、临时变量，函数调用时，存储函数的返回指针，用于控制函数的调用和返回。在程序块开始时自动分配内存,结束时自动释放内存，其操作方式类似于数据结构中的栈。

71 Linux内存管理机制

**伙伴系统：**

为了避免出现这种情况，Linux内核中引入了伙伴系统算法(buddy system)。把所有的空闲页框分组为11个块链表，每个块链表分别包含大小为1，2，4，8，16，32，64，128，256，512和1024个连续页框的页框块。最大可以申请1024个连续页框，对应4MB大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。一个页面大小是4kb

 假设要申请一个256个页框的块，先从256个页框的链表中查找空闲块，如果没有，就去512个页框的链表中找，找到了则将页框块分为2个256个页框的块，一个分配给应用，另外一个移到256个页框的链表中。如果512个页框的链表中仍没有空闲块，继续向1024个页框的链表查找，如果仍然没有，则返回错误。页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。

**Slab机制：**

slab是Linux操作系统的一种内存分配机制。其工作是针对一些经常分配并释放的对象，如进程描述符等，这些对象的大小一般比较小，如果直接采用伙伴系统来进行分配和释放，不仅会造成大量的内碎片，而且处理速度也太慢。而slab分配器是基于对象进行管理的，相同类型的对象归为一类(如进程描述符就是一类)，每当要申请这样一个对象，slab分配器就从一个slab列表中分配一个这样大小的单元出去，而当要释放时，将其重新保存在该列表中，而不是直接返回给伙伴系统，从而避免这些内碎片。slab分配器并不丢弃已分配的对象，而是释放并把它们保存在内存中。当以后又要请求新的对象时，就可以从内存直接获取而不用重复初始化。

**页面置换算法：**

最佳置换算法（OPT）（理想置换算法）；先进现出置换算法（FIFO）；最近最久未使用（LRU）算法；最少使用（LFU）置换算法:

72 惊群效应

惊群问题又名惊群效应。简单来说就是多个进程或者线程在等待同一个事件，当事件发生时，所有线程和进程都会被内核唤醒。唤醒后通常只有一个进程获得了该事件并进行处理，其他进程发现获取事件失败后又继续进入了等待状态，在一定程度上降低了系统性能。多出现在select epoll poll获取事件的时候

可能很多人想问，惊群效应为什么会占用系统资源？降低系统性能？

多进程/线程的唤醒，涉及到的一个问题是上下文切换问题。频繁的上下文切换带来的一个问题是数据将频繁的在寄存器与运行队列中流转。极端情况下，时间更多的消耗在进程/线程的调度上，而不是执行

73 TCP判断对方是否依旧存活

**1.心跳包**

由应用程序自己发送心跳包来检测连接是否正常，大致的方法是：服务器在一个 Timer事件中定时向客户端发送一个短小精悍的数据包，然后启动一个低级别的线程，在该线程中不断检测客户端的回应， 如果在一定时间内没有收到客户端的回应，即认为客户端已经掉线；同样，如果客户端在一定时间内没有收到服务器的心跳包，则认为连接不可用。

注意：心跳包一般都是很小的包，或者只包含包头的一个空包

**2.设置TCP属性: SO\_KEEPALIVE**

1.因为要考虑到一个服务器通常会连接多个客户端，因此由用户在应用层自己实现心跳包，代码较多 且稍显复杂，而利用TCP／IP协议层为内置的KeepAlive功能来实现心跳功能则简单得多

74 C和C++的区别

C语言文件后组为.C；C++为.cpp；3.返回值不同C语言中，如果一个函数没有指定返回值类型，默认为int类型，并返回一个随机数，一般为0XCCCCCCCC

在C++中，如果函数没有返回值则必须指定为void型，否则编译不能通过；

参数列表：

在C语言中，函数没有指定的参数列表时，默认可接收任意多个参数

在C++中，有严格的参数类型检测，没有参数列表的函数，默认为void，不接收任何参数。

**C++支持函数重载，C语言不支持；C++有命名空间;C语言可以传值和传地址，C++还可以传引用**

74 SGI STL allocator（空间配置器）

SGI空间配置器分两级，第一级配置器直接采用malloc和free管理内存；第二级采用内存池管理内存。申请空间时，如果区块够大，超过128B，移交第一级配置器处理。当区块小于128B时，以内存池管理。

**内存池管理流程：**

客户端请求分配大小为n bytes的内存（n < 128，n已经圆整为8的倍数）

1、检查n对应的free\_list是否为空，不为空，直接分配一个内存块给客户，如果为空，到第2步；

2、向内存池申请20个n bytes的内存，如果申请成功，一个返回给客户，另外19个挂到对应free\_list节点下；如果有不到20个n bytes的空间，返回一个给客户，另外的全部交给对应free\_list节点；如果连一个n bytes的空间都没法分配，进入第3步；

3、向堆申请40个n bytes的空间，如果申请成功，20个n bytes交给内存池，1个返回给客户，19个留给free\_list下的节点；如果堆内存不够，进入第四步；

4、从比当前n还大的free\_list节点中查找是否有内存可用，如果有，分n bytes给客户，另外部分交给内存池；如果n之后的所有free\_list上都没有内存可用，进入第5步；

5、调用第一级配置器的out\_of\_memory处理机制，尝试释放其它内存。如果可以，返回给客户。

75 数字证书

在采用加密和对称加密过程中，数字证书用于验证发送消息的是本人。

a、A和B通信时，A为了证明自己的身份，需要向证书机构CA申请证书，证书机构采用非对称加密的方式，通过用自己的私钥对A的公钥等信息进行加密，并返回给A；

b、A向B发消息时，附上CA加密过的信息；B接收到消息后，尝试用CA的公钥解密，如果解密出的A的公钥和A发送给B的一致，说明对方是A；

76 加密算法有哪些？对称加密和非对称加密的区别？

对称加密和非对称加密，常用的对称加密算法有DES和AES，非对称加密算法有RSA、ECC等。

区别：

a、加密效率上来看，对称加密简单，加密解密效率快；

b、安全性上看，非对称加密消除了最终用户交换密钥的需要，保密性比较好；

由于非对称加密效率较低，非对称加密常用于少量数据传输。通信前，先用非对称加密传输对称加密的密钥，再用对称加密传输数据。

77 数据库的join含义及其分类：内联结（INNER JOIN）、外联结（OUTER JOIN，包括左联结（LEFT OUTER JOIN）、右联结（RIGHT OUTER JOIN））的区别？

含义：数据库中join语句是指将多个表中的数据并联在一起，然后根据要求找出需要的数据。

内联结是保证两个表中所有的行都要满足联结条件。在外联结中，某些不满条件的列也会显示出来。左联结保证左表全部显示，右表中不匹配的部分显示为NULL；全联结全部显示，不匹配位置显示为NULL。

**1.INNER JOIN** 产生的结果是AB的交集（内连接）

SELECT \* FROM TableA INNER JOIN TableB ON TableA.name = TableB.name

**2.LEFT [OUTER] JOIN** 产生表A的完全集，而B表中匹配的则有值，没有匹配的则以null值取代。（左连接）  
SELECT \* FROM TableA LEFT OUTER JOIN TableB ON TableA.name = TableB.name

**3.RIGHT [OUTER] JOIN**产生表B的完全集，而A表中匹配的则有值，没有匹配的则以null值取代。（右连接）  
SELECT \* FROM TableA RIGHT OUTER JOIN TableB ON TableA.name = TableB.name

**4.FULL [OUTER] JOIN**产生A和B的并集。对于没有匹配的记录，则会以null做为值。  
SELECT \* FROM TableA FULL OUTER JOIN TableB ON TableA.name = TableB.name

**5. CROSS JOIN** 把表A和表B的数据进行一个N\*M的组合，即笛卡尔积。如本例会产生4\*4=16条记录，在开发过程中我们肯定是要过滤数据，所以这种很少用。  
SELECT \* FROM TableA CROSS JOIN TableB

78 Redis和memcache的区别 Redis分布式锁原理

setNX啥的先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

Memaache的所有值都是简单的字符串，redis拥有丰富的数据结构

Redis快

Memachae可以持久化数据

1)、存储方式

Memecache把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。

Redis有部份存在硬盘上，这样能保证数据的持久性。

2)、数据支持类型

Memcache对数据类型支持相对简单。

Redis有复杂的数据类型。

3)、使用底层模型不同

它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。

Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

4），value大小

redis最大可以达到1GB，而memcache只有1MB

79 怎么实现redis持久化？

rdb持久化方式能够在指定的时间间隔内对数据进行快照存储；Rdb的功能就是将redis在内存中的数据库状态保存到磁盘里面，避免数据意外丢失；

aof持久化方式记录每次对服务器的写操作，当服务器重启的时候会重新执行这些命令来恢复原始的数据，aof命令以redis协议追加保存每次写的操作的文件末尾。

可以同时开启两种持久化方式，在这种情况下当redis重启的时候会有限载入aof文件来恢复原始的数据，因为在通常情况下aof文件保存的数据集要比rdb文件保存的数据集更加完整，

**为什么会高效？**

Redis文件事件处理器以单线程方式运行，但是通过使用I/O多路复用程序来监听多个套接字，文件事件处理器即实现了高性能的网络通信模型，又可以很好地与Redis服务器中其他同样以单线程运行的模块进行对接，这保持了Redis内部单线程设计的简单性。

80 redis五种数据类型及实现方式

**字符串：**

字符串是redis中最基本的数据。Redis使用key作为存取value的唯一标识符，而key的通俗理解就是字符串。Redis中的字符串分为两类：二进制安全（binary safe）和非二进制安全。**二进制安全**的字符串，是指字符串中所有字符均可以用**256个编码**。Redis中的value都是通过二进制安全的字符串存储的，而key使用的是非二进制的。

Redis内部实现了字符串类型，由sds.h和sds.c定义。Sds 本质是char\*.

Typedef char \* sds;

SDS与C字符串的区别：

常数复杂度获取字符串长度，Sds记录字符串的长度，c字符串不记录；

杜绝缓冲区溢出，C字符串不记录自身长度带来的另一个问题是容易造成缓冲区溢出。Strcat。sds中记录了它的容量，如果不够则会进行扩容；

减少修改字符串带来的内存重新分配次数

二进制安全，所有ads api都会以处理二进制的方式来处理sds存放在buf数组里的数据，程序不会对其中的数据做任何限制、过滤、或者假设，数据在写入时是什么样的，它被读取时就是什么样；sds使用len属性的值来判断字符串是否结束

**链表：**

链表提供了高效的节点重排能力，以及顺序性的节点访问方式，并且可以通过增删节点来灵活调整链表的长度。

Redis的链表实现的特性：

双端：链表节点带有prev和next指针，获取某个节点的前置节点和后置节点的复杂度是O(1);

无环：表头节点的prev指针和表尾节点的next指针都指向NULL，对链表的方位以NULL为终点；

带表头指针和表尾指针：通过list结构的head指针和tail指针，程序获得链表的表头节点和表尾节点的复杂度为O（1）；

带链表长度计数器：程序使用list结构的len属性来对list持有的链表节点进行计数，程序获取链表中节点数量的复杂度的时间复杂度为O（1）；

**多态**：链表节点使用void\* 指针来保存节点值，并且可以通过list结构的dup、free、match三个属性为节点值设置类型特定函数，所以链表可以用于保存各种不同类型的值；

**哈希表：**

字典，又称为符号表、关联数组或映射，是一种用于保存键值对的抽象数据结构。在字典中，一个键可以和一个值进行关联，这些关联的键和值就称为键值对。字典中的每个键都是独一无二的，程序可以在字典中根据键查找与之关联的值，或者通过键来更新值，又或者根据键来删除整个键值对。

Redis的数据库就是使用字典来作为底层实现的，对数据库的增、删、查、该操作也是建立在字典的操作之上的。

Redis的哈希表最大的特色就是**自动扩容**。当它的哈希表容量不够时，可以0/1切换，然后自动扩容。

整个哈希系统由结构体dict定义，其中type包含一系列哈希表需要用到的函数，dictht类型的数组ht[2]表示两个哈希表实例，由rehashidx指明下一个需要扩容的哈希实例的编号，iterators记录外部使用哈希表的迭代器的数目。

Redis中的字典使用哈希表作为底层实现，每个字典带有两个哈希表，一个平时使用，另一个仅在进行rehash时使用。

分摊转移：在redis数据库中，使用两个哈希表进行扩容和转移工作：当从第一个hash表的装载因子为1时，假设要往字典里插入一个元素。首先为第二个hash表开辟2倍第一个hash表的容量。同一时候将第一个hash表的一个非空桶中元素所有转移到第二个hash表中。然后把待插入元素存储到第二个hash表里。继续往字典里插入第二个元素，又会将第一个hash表的一个非空桶中元素所有转移到第二个hash表中，然后把元素存储到第二个hash表里……直到第一个hash表为空。这样的策略就把第一个hash表全部元素的转移分摊为多次转移，并且每次转移的期望时间复杂度为O(1)。

**跳跃表（有序集合的低层实现）**

跳跃表是一种有序数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。

跳跃表支持平均O(logN)、最坏O(N)复杂度的查找，还可以通过顺序性操作来批量处理节点。在大部分情况下，调表的效率可以和平衡树相媲美，并且因为跳跃表的实现比平衡树要来的更为简单，所以不少程序都是用跳跃表来替代平衡树。

Redis中使用跳跃表作为有序集合键的底层实现之一。Redis只有在两个地方用到了跳跃表，一个是实现有序集合键，另一个是集群节点中用作内部数据结构。

跳跃表是有序集合的底层实现之一；

Redis的跳跃表实现由zskiplist和zskiplistNode两个结构组成，其中zskiplist用于保存跳跃表信息（比如表头节点、表尾节点、长度），而zskiplistNode则用于表示跳跃表节点。

每个跳跃表节点的层高都是1至32之间的随机数；

在同一个跳跃表中，多个节点可以包含相同的分值，但每个节点的成员对象必须是唯一的；

跳跃表中的节点按照分值大小进行排序，当分值相同时，节点按照成员对象的大小进行排序；

**整数集合（集合）**

整数集合（intset）是集合键的底层实现之一，当一个集合只包含整数值元素，并且这个集合的元素数量不多时，redis就会使用**整数集合作为集合键**的底层实现。

数组实现的，各个项在数组中按值的大小从小到大有序地排列，并且数组中不包含任何重复项。

**整数集合的升级**

当我们要将一个新元素添加到整数集合里面，并且新元素的类型比整数集合现有所有元素的类型都要长时，整数集合需要先进行升级，然后才能将新元素添加到整数集合里面。

升级整数集合并添加新元素共分为三步进行：

根据新元素的类型，扩展整数集合底层数组的空间大小，并为新元素分配空间；

将底层数组现有的所有元素都转换成与新元素相同的类型，并将类型转换后的元素放置到正确的位置上，

将新元素添加到底层数组里面。

**81 redis集群**

Redis集群是redis体用的分布式数据库方案，集群通过分片来进行数据共享，并提供复制和故障转移功能。（通过分布式锁setnx）expire

一个redis集群通常由多个节点（node）组成，在刚开始的时候，每个节点都是相互独立的，它们都处于一个只包含自己的集群当中，要组建一个真正可工作的集群，我们必须将各个独立的节点连接起来，构成一个包含多个节点的集群。

81 进程与子进程

创建进程之后，子进程与父进程之间有什么区别：

相同处：全局变量、.data、.date、栈、对、环境变量、用户id、进程工作目录、信号处理方式等；

不同之处：1、进程id号；2、fork返回值；3、父进程id；4、未决信号集

子进程复制了父进程0-3G用户空间内容，以及父进程的pcb，但是pid不同，写时复制。

82 linux的四种锁

**互斥锁、读写锁、自旋锁、RCU**

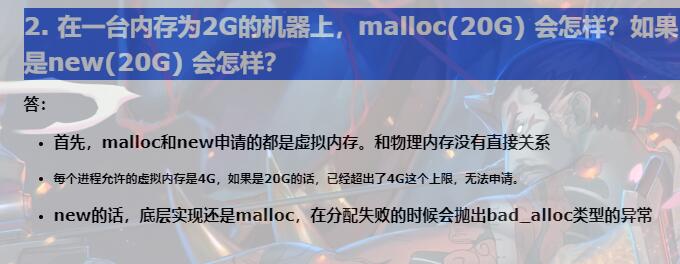
互斥锁：用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程则进入睡眠状态，等待锁释放时被唤醒；

读写锁：分为读锁和写锁。处于读操作时，可以允许多个线程同时获得该读操作。但是同一时刻只能允许一个线程获得写锁。其他获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。写锁会阻塞其他的读写锁，当有一个线程获得写锁写时，读锁不能被其他线程获取到。

自旋锁：spinlock，在任何时刻同样只能有一个线程访问对象，但是当获取锁操作失败时，不会进入睡眠，只会在原地自旋，直到锁被释放。这样节省了线程从睡眠状态到被唤醒期间的事件消耗，在加锁事件短暂的环境下会有极大地提高效率。“忙等”

RCU：read-copy-update，在修改数据时，需要读取数据，生成一个副本，然后对副本进行修改，然后更新。使用RCU时，读者几乎不需要同步开销，既不需要获得锁，也不需要原子操作，没有锁竞争。对于写者的同步开销很大，他需要复制被修改的数据，还需要使用锁机制并行其他写者的修改操作，在有大量读操作，少量写的操作时效率很高。

83 在一台内存为2G的机器上，malloc（20G会怎么样）？如果是new（20G）会怎么样



84 cookie和session的区别是什么

在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。cookie的出现就是为了解决这个问题，第一次登录后服务器返回一些数据（cookie）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会自动的把上次请求存储的cookie数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前用户是哪个了。cookie存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4KB。因此使用cookie只能存储一些小量的数据。

session和cookie的作用有点类似，都是为了存储用户相关的信息。不同的是，cookie是存储在本地浏览器，而session存储在服务器。存储在服务器的数据会更加的安全，不容易被窃取。但存储在服务器也有一定的弊端，就是会占用服务器的资源，但现在服务器已经发展至今，一些session信息还是绰绰有余的。

85 mysql查询缓慢原因和解决方案

查询速度慢的原因很多，常见如下几种情况：

1. **所有索引或者没有用到索引**（这是查询最常见的问题，是程序设计的缺陷）
2. I/O吞吐量小，形成了瓶颈效应；
3. 没有创建计算列导致查询不优化；
4. 内存不足；
5. 网络速度慢；
6. 查询的数据量过大（可以采用多次查询，其他的方法降低数据量）；
7. **锁或者死锁**（这也是查询慢最常见的问题，是程序设计的缺陷）
8. 查询语句不好；

优化查询：

1. 减少表的尺寸（sp\_spaceuse）
2. 升级硬件
3. 根据查询条件、建立索引、优化访问方式，限制结果集的数据量。注意填充因子要适当。索引应该尽量小，使用字节数小的列建索引号，不要对有限的几个值的字段建单一索引如性别字段；
4. 提高网速；
5. 扩大服务器的内存；
6. 增加服务器CPU个数，但是必须明白并行处理串行处理更需要资源例如内存；
7. 重建索引。

86 多线程和多进程对比

1、调度方面：在引入线程的OS中，线程是独立的调度和分派单位，而进程作为资源的拥有单位。 2、安全性方面，多进程比多线程更安全。进程中可以创建多个线程，进程结束，所有线程都会被销毁，任意一个线程崩溃也会导致进程崩溃。

3、进程拥有PCB、页表、内核堆栈等资源，进程的创建、撤销和切换需要保存PCB和内核堆栈等信息，开销比线程大。

4、进程间互相独立，所以协作时主要考虑通信问题；同一进程的线程共享进程空间，通信容易，主要考虑同步问题

87 Redis缓存雪崩、缓存击穿和缓存穿透

缓存雪崩是因为大面积的缓存失效，打崩了DB，大面积的存储节点。

处理缓存雪崩简单，在批量往Redis存数据的时候，把每个Key的失效时间都加个随机值就好了，这样可以保证数据不会在同一时间大面积失效，我相信，Redis这点流量还是顶得住的。

Redis的缓存击穿，如何处理这个问题；

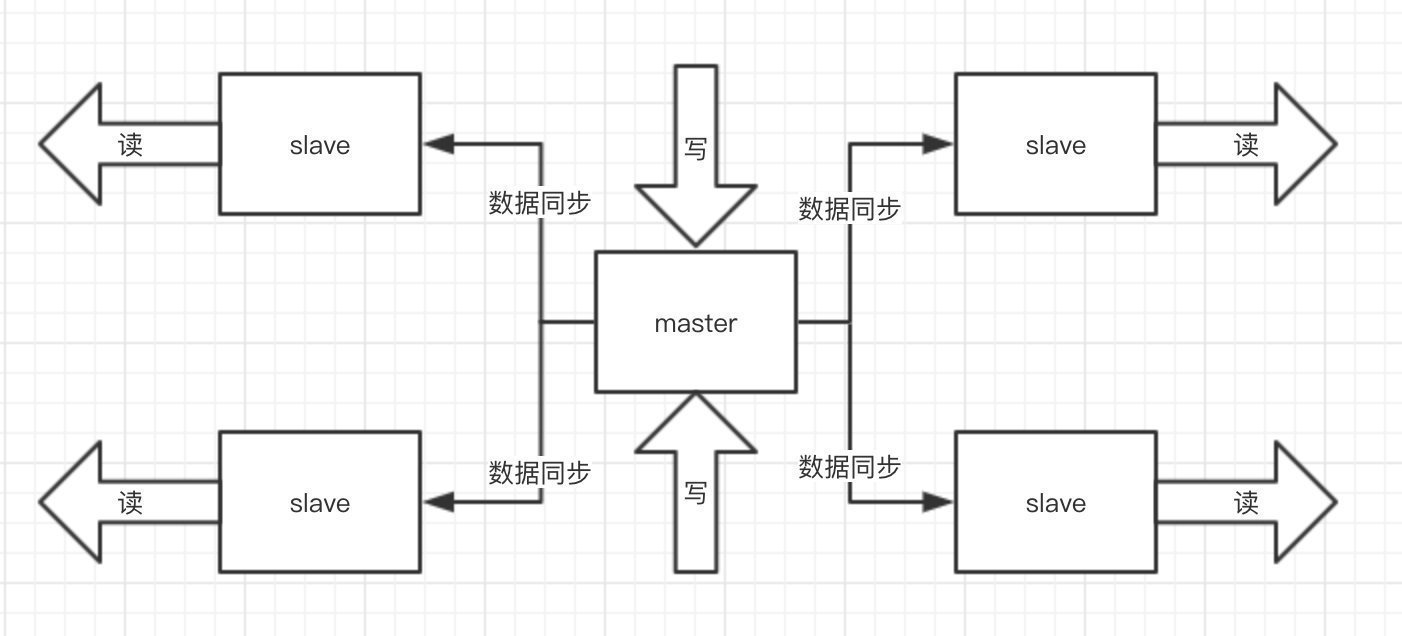
缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。缓存击穿的话，设置热点数据永远不过期。或者加上互斥锁就能搞定了。或者加上互斥锁

缓存穿透：

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，我们数据库的 id 都是1开始自增上去的，如发起为id值为 -1 的数据或 id 为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大，严重会击垮数据库。

使用布隆过滤器（Bloom Filter）这个也能很好的防止缓存穿透的发生，他的原理也很简单就是利用高效的数据结构和算法快速判断出你这个Key是否在数据库中存在，不存在你return就好了，存在你就去查了DB刷新KV再return。

88 Redis的主从复制读写分离



Master写slave读（查询）减少服务器压力

你启动一台slave 的时候，他会发送一个psync命令给master ，如果是这个slave第一次连接到master，他会触发一个全量复制。master就会启动一个线程，生成RDB快照，还会把新的写请求都缓存在内存中，RDB文件生成后，master会将这个RDB发送给slave的，slave拿到之后做的第一件事情就是写进本地的磁盘，然后加载进内存，然后master会把内存里面缓存的那些新命名都发给slave。

89 幂等模型

幂等（idempotent、idempotence）是一个数学与计算机学概念，常见于抽象代数中。

在编程中一个幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。

幂等函数，或幂等方法，是指可以使用相同参数重复执行，并能获得相同结果的函数。这些函数不会影响系统状态，也不用担心重复执行会对系统造成改变。

例如，“setTrue()”函数就是一个幂等函数,无论多次执行，其结果都是一样的.更复杂的操作幂等保证是利用唯一交易号(流水号)实现.

90 构造函数中能不能抛出异常？

**可以抛出异常**

a、因为构造函数没有返回值，所以在构造函数中抛出异常是C++中通知对象构造失败的唯一方法。

b、构造函数中抛出异常将导致对象的析构函数不被执行；因为构造不完全，C++拒绝为没有完成构造操作的对象调用析构函数。

c、当对象发生部分构造时，已经构造完毕的子对象（非动态分配）将会逆序地被析构。

91 构造函数能否写成虚函数

不可以，虚函数的调用依赖于指向类的虚表指针，而虚表指针是在构造函数中初始化。如果构造函数是虚函数，则调用构造函数时，虚表指针并未初始化，此时会引起错误。

92 构造函数和析构函数中调用虚函数为什么不行？

**在**[**构造函数**](https://wenwen.sogou.com/s/?w=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&ch=ww.xqy.chain)**不要调用虚函数**。在基类构造的时候，虚函数是非虚，不会走到[派生类](https://wenwen.sogou.com/s/?w=%E6%B4%BE%E7%94%9F%E7%B1%BB&ch=ww.xqy.chain)中，既是采用的[静态绑定](https://wenwen.sogou.com/s/?w=%E9%9D%99%E6%80%81%E7%BB%91%E5%AE%9A&ch=ww.xqy.chain)。显然的是：当我们构造一个子类的对象时，先调用基类的构造函数，构造子类中基类部分，子类还没有构造，还没有初始化，如果在基类的构造中调用虚函数，如果可以的话就是调用一个还没有被初始化的对象，那是很危险的，所以C++中是不可以在构造父类对象部分的时候调用子类的虚函数实现。

**在析构函数中也不要调用虚函数**。在析构的时候会首先调用子类的析构函数，析构掉对象中的子类部分，然后在调用基类的析构函数析构基类部分，如果在基类的析构函数里面调用虚函数，会导致其调用已经析构了的子类对象里面的函数，这是非常危险的，记得在写派生类的拷贝函数时，调用基类的拷贝函数拷贝基类的部分，不能忘记了

93 什么情况下会调用拷贝构造函数

1。当使用类的一个对象去初始化该类的另一个新对象时。

2。如果函数的形参是类的对象，那么当调用该函数时拷贝构造函数也会被调用。

3。如果函数的返回值是类的对象，那么函数执行完成返回调用者时。

94 使用TCP和UDP的应用及端口号

**OSPF使用IP数据报**

**TCP应用：**

SMTP（简单邮件传输协议）——25 TELENT（远程控制）——23

HTTP（超文本传输协议）——80 HTTPS（加密的超文本传输协议）——443

FTP（文件传输协议）——21 BGP(外部网关协议)

**UDP应用：**

DNS（域名解析）——53 RIP（路由信息协议520）DHCP（动态主机配置67Sev68Client）SNMP(简单网络管理协议)——161

95 Host文件，同一个IP多个域名

hosts文件是一个没有扩展名的系统文件，其基本作用就是将IP和域名建立联系，之前在DNS的文章中提到过互联网中的互访，是通过IP地址进行的，Hosts文件就是本地的域名解析。域名解析到同一个IP，确实不是问题，服务器上配置的有host文件啊，Host文件指定了对应的域名可以访问相应目录的文件，从而保证打开相应的域名访问的是你想要打开的内容；

96 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存储引擎：表数据和索引自动分开存储，各种是一个独立的存储文件，表的文件有：.MYD数据文件 .MYI所有文件.frm结构文件**非聚集索引**

INNODB存储引擎：表的数据和索引都是存储在同一个共享表空间里面，但是有多个文件组成，表的文件有：IBDATA1.IBD 数据库文。**聚集索引**

MYISAM 不支持事务，不支持外键，表锁，插入数据时，锁定整个表，查表总行数时，不需要全表扫描

INNODB 支持事务，支持外键，行锁，查表总行数时，全表扫描

数据库隔离级别---MySQL的默认隔离级别就是Repeatable,Oracle默认Read committed，最高级别Serializable

**共享表空间方式（一个文件索引和数据在一起） 独立表空间（很多个文件索引和数据分离）**

97 为什么使用数据库连接池？

创建数据库连接是一个很耗时的操作，频繁的打开和关闭连接数据库操作会影响服务器性能。数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。

**使用数据库连接池的优点：**

a、资源重用。连接池里的连接可以重复使用，避免了频繁创建、释放连接引起的大量性能开销。

b、更快的系统响应速度 。连接池中的连接已经做了初始化工作，相对于重新创建可以提高效率。

98 什么是视图，视图可以更改吗？

a、视图是是从一个或多个表导出的虚拟的表，其内容由查询定义。它具有普通表的结构，但是不实现数据存储。

b、可以通过使用表的方式使用视图，简化复杂的sql操作，隐藏具体的细节，保护数据；

c、视图不能被索引，不能有关联的触发器或默认值。

**对视图的修改：**

单表视图对视图的修改会改变基本表的数据，多表视图不会改变基本表的数据。

99 文件的三种时间（mtime, atime，ctime），分别在什么时候会改变

1、访问时间（atime->access time）：读一次这个文件的内容，这个时间就会更新。比如对这个文件运用 more、cat等命令。ls、stat命令都不会修改文件的访问时间。

2、修改时间（mtime->modifiy time）：修改时间是文件内容最后一次被修改时间。比如：vi后保存文件。ls -l列出的时间就是这个时间。  
3、状态改动时间（ctime->change time）：ctime是在写入文件、更改所有者、权限或链接设置时随i节点的内容更改而更改的，是该文件的i节点最后一次被修改的时间，通过chmod、chown命令修改一次文件属性，这个时间就会更新。

4、ls命令簇   
1) ls -lc filename 列出文件的 ctime （最后状态更改时间）  
2) ls -lu filename 列出文件的 atime（最后访问时间）  
3) ls -l filename 列出文件的 mtime （最后修改时间）

5、stat命令   
stat命令可以一次性更精确的看到文件的三种时间属性。

6、操作对三种时间属性的影响：   
1、cat、less等只访问文件，不修改文件的操作，只会修改atime的值。  
2、chmod、chown等修改文件权限、所有者等的操作，会修改atime和ctime的值。  
3、vim、emacs等修改文件内容的操作，会修改atime、ctime、mtime的值。

100 进程的内存映像包含部分？

进程的内存映像可以很好地说明进程的组成。简单的说，一个进程映像（Process Image）包括：  
**进程程序块**，即被执行的程序，规定了进程一次运行应完成的功能。通常它是纯代码，作为一种系统资源可被多个进程共享。  
**进程数据块**，即程序运行时加工处理对象，包括全局变量、局部变量和常量等的存放区以及开辟的工作区，常常为一个进程专用。  
**系统/用户堆栈**，每一个进程都将捆绑一个系统/用户堆栈。用来解决过程调用或系统调用时的信息存储和参数传递。  
**进程控制块**，每一个进程都将捆绑一个进程控制块，用来存储进程的标志信息、现场信息和控制信息。进程创建时，建立进程控制块；进程撤销时，回收进程控制块，它与进程一一对应。

可见每个进程有四个要素组成：控制块、程序块、数据块和堆栈。

101 抓包

Sniffer程序是一种利用以太网的特性把[网络适配卡](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=7922621&ss_c=ssc.citiao.link)（NIC，一般为以太网卡）置为杂乱（promiscuous）模式状态的工具，一旦网卡设置为这种模式，它就能接收传输在网络上的每一个[信息包](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=8645613&ss_c=ssc.citiao.link)。

102 帧同步和状态同步

**状态同步：**

所以大概总结一下就是，状态同步是通过每个客户端发送自己的操作给服务器，这时客户端不进行任何动作，服务器统一计算后并把结果同步个每一个客户端。

**帧同步：**

帧同步是在关键帧的时候同步操作给服务器，服务器转发操作给每个客户端。客户端之间要接受到关键帧才可以进行操作。

帧同步里面有一个关键技术就锁帧，也就是如果没有收到广播的关键帧不能进行下一步操作。大家静止不动。

而现在的游戏会有一个聪明一点的方法叫做乐观帧锁定

乐观帧锁定通过定时发送关键帧的方法，不锁定任何客户端，服务器一定时间间隔就发包给每个客户端，包里可能是空包也有可能是你自己活着别人的操作。而客户端就通过定时或者每当操作就发包的形式告诉服务器自己的操作。

103 GCC和G++编译区别

gcc和g++的区别主要是在对cpp文件的编译和链接过程中，因为cpp和c文件中库文件的命名方式不同，那为什么g++既可以编译C又可以编译C++呢，这时因为g++在内部做了处理，默认编译C++程序，但如果遇到C程序，它会直接调用gcc去编译。

104 强一致性、弱一致性和最终一致性

**强一致性：**

当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据  
CAP 理论，这种实现需要牺牲可用性。

**CAP原则又称CAP定理**：指的是在一个分布式系统中，Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性）这三个基本需求，最多只能同时满足其中的2个。

**弱一致性：**

系统并不保证续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。

**最终一致性：**

弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。DNS  
是一个典型的最终一致性系统。

105 MYSQL常用语句

**SELECT 列名 FROM 表名 WHERE 列名 LIKE 模式或取值;**

SELECT \* FROM Products; 检索所有的列

Select prod\_name FROM Products LIMIT 5 OFFSET 5;

从product表中返获取prod\_name的数据，从第5行开始到第十行。

Select prod\_name From Products ORDERED BY prod\_name，prod\_price;

按prod\_name检索排序语句当name相同时，按照prices排序

Ordered by a desc,b; 排序按照a降序,b升序排列

create database scort //建立数据库

use scort //使用数据库

create table tb (姓名varchar(10),课程varchar(10),分数int)//创建表格

SELECT vect\_id,prod\_name FROM Products WHERE vect\_id<>’DLL01’ AND（prod\_name IS NULL）

检索数据，当id值不等于DLL01且prod\_name 是空值

SELECT vect\_id,prod\_name FROM Products WHERE vect\_id BETWEEN 10 AND ；20 检索数据，价格在10-20之间

SELECT \* FROM Products WHERE vect\_id IN(‘DLL01’,’BRS101’); ID在这个2个里面

SELECT vect\_id,prod\_name FROM Products WHERE vect\_id LIKE ‘Fish%’；检索所有以Fish开头的产品 LIKE ‘F%y’匹配以F开头y结尾的字符

LIKE ‘\_A’\_只匹配一个字符 LIKE ‘[IM]%’匹配以I和M开头的字符 ‘[^IM]%’不以IM开头

SELECT Concat(RTRIM(vend\_name),’(’,vend\_country,’)’) AS vec\_title

FROM Products WHERE vect\_id LIKE ‘Fish%’; 为名字添加括号和国家

RTRIM——删除字符右边的空格 LTRIM——删除字符左边的空格

AS 添加别名

SELECT AVG(DISTINST prod\_prices) AS Avg\_prices FROM Products;

AVG返回某列的平均值 COUNT 统计行数 MAX返回某列的最大值 MIN返回某列的最小值，SUM返回某列值之和 DISTINST不同

SELECT vend\_id,COUNT(\*) AS num\_prods FROM Products GROUP BY vend\_id

HAVING COUNT(\*)>=2;

GROUP BY 按vend\_id分组并排序 按分组排序后统计行数至少在2以上

SELECT Customers.cust\_id,Ordereds.order\_num FORM Customers INNER JOIN

Orders ON Customers. cust\_id= Ordereds. order\_num; 内连接

组合查询UNION

SELECT vect\_id,prod\_name FROM Products WHERE vect\_id LIKE ‘Fish%’ UNION

SELECT vect\_id,prod\_name FROM Products WHERE vect\_id LIKE ‘aa%’；

INSERT INTO Custom VALUE(‘100’,’ADAD’,’13131’);插入一行新的数据

SELECT \* INTO CustomCopy FROM Custom;//复制表格数据从一个到另外一个。

UPDATE Customers SET cust\_email=NULL WHERE cust\_id=’1000004’;更新语句

DELETE FROM Customers WHERE cust\_id=’11111’;删除某一行

DROP TABLE Custome；删除这个表

**mysql强制指定索引查询**

语法：  
select \* from table\_name force index (index\_name) where conditions;

**null与not null和null与空值''的区别:**

空值是不占用空间的 my NOT NULL 的字段是不能插入“NULL”的，只能插入“空值”

sql中的NULL其实是占用空间的

106 批处理系统、分时系统和实时系统

**批处理系统：**又名[批处理操作系统](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=72371026)。批处理是指用户将一批作业提交给操作系统后就不再干预，由操作系统控制它们自动运行。

适用的调度策略：

先来先服务：[FCFS](https://www.baidu.com/s?wd=FCFS&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)最短作业优先；最短剩余时间优先；响应比最高者优先

**分时系统：**是一种联机的多用户交互式的操作系统。一般采用时间片轮转的方式使一台计算机为多个终端服务。对每个用户能保证足够快的响应时间，并提供交互会话能力。

适用的调度策略：

时间片轮转调度；优先级调度；多级队列调度；彩票调度

**实时系统：**所谓“实时”，是表示“及时”，而实时系统是指系统能及时响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有[实时任务](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=67907684&ss_c=ssc.citiao.link)协调一致的运行。

适应的调度策略：

单比率调度；限期调度；最少裕度法

107 ping命令

Ping发送一个ICMP(Internet Control Messages Protocol）即因特网信报控制协议；回声请求消息给目的地并报告是否收到所希望的ICMP echo （ICMP回声应答）。

主机B收到这个数据帧后，先检查它的目的地址，并和本机的物理地址对比，如符合，则接收；否则丢弃。接收后检查该数据帧，将IP数据包从帧中提取出来，交给本机的IP层协议。同样，IP层检查后，将有用的信息提取后交给ICMP协议，后者处理后，马上构建一个ICMP应答包，发送给主机A，其过程和主机A发送ICMP请求包到主机B一模一样。示成功，返回-1表示失败，一般指的是没有内存了。

108 Linux调用malloc()发生了什么？

Int brk(void \*addr); 参数意义是brk的入参就是指定堆的结束地，brk()返回 0表

Void \*sbrk(intptr\_t increment); sbrk的入参与返回值都和brk不一样，brk的入参是一个绝对地址，表示自己想要设置的brk\_end 值，而sbrk的入参是相对地址，sbrk的返回是新设置的brk\_end。

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk和mmap（不考虑共享内存）。

1、brk是将数据段(.data)的最高地址指针\_edata往高地址推；

2、mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。

malloc小于128k的内存，使用brk分配内存，将\_edata往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)；

**malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)。**

****109 Inline关键字****

a、显式设置：在头文件中，函数声明的前面加上inline；

b、隐式设置：类的成员函数默认是inline。

**inline的优点：**

a、**编译期**以函数本体替换函数调用，免除了执行时的函数调用开销；

b、编译器的最优化机制可以对inline函数执行相关最大优化；

**inline的缺点：**

a、过度声明inline函数会导致代码膨胀；

b、inline函数的存在会导致版本升级时，需要重新编译代码，无法用于动态链接。

c、inline只是一个申请，编译器可以进行忽略；太过复杂的函数（例如带有递归、循环），编译器拒绝inline；

d、通过函数指针调用的函数拒绝inline；

e、拒绝对虚函数inline；因为虚函数是运行时确定，inline是编译期替换；

110从汇编的角度看函数调用的过程

a、当前栈基地址压栈；（保存当前基地址，重新开始一个新的栈）

b、建立新的栈帧，参数从右往左入栈（从右向左压入是因为支持动态参数变化，从左向右必须要提前知道参数个数）；

c、下一条指令地址进栈；（函数调用返回后从这里继续）

d、修改指令寄存器rip的值，使其指向被调函数执行位置；

d、栈顶指针esp归位，回到本函数ebp；（清空当前函数所使用的栈）

e、基地址回退到上一个函数的基地址；

f、eip退回到上一个函数即将要执行的语句上

111 多线程死锁检测和解决方案 区分死锁和死循环

1、按顺序加锁

如果每个线程都按同一个的加锁顺序这样就不会出现死锁。

2、获取锁时限

每个获取锁的时候加上个时限，如果超过某个时间就放弃获取锁之类的。

3、死锁检测

按线程间获取锁的关系检测线程间是否发生死锁，如果发生死锁就执行一定的策略，如终断线程或回滚操作等。

**死锁和死循环区别：**

PS 查询 相关进程的PID

top -P   PID   查询该进程占用资源情况    主要查看的是  cpu  和 内存  的占用率 （如果CPU占用率高就是死循环， 如果CPU占用率正常，而且内存占用正常，那就是死锁）

如果有问题，就去详分析该进程底下 各个是哪个线程出现的问题

top -Hp  PID     该进程底下各线程的CPU、内存占用情况

112 redies为什么使用跳表而不使用红黑树？

在做范围查找的时候，平衡树比skiplist操作要复杂。在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在skiplist上进行范围查找就非常简单，只需要在找到小值之后，对第1层链表进行若干步的遍历就可以实现。

平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。

从内存占用上来说，skiplist比平衡树更灵活一些。一般来说，平衡树每个节点包含2个指针（分别指向左右子树），而skiplist每个节点包含的指针数目平均为1/(1-p)，具体取决于参数p的大小。如果像Redis里的实现一样，取p=1/4，那么平均每个节点包含1.33个指针，比平衡树更有优势。

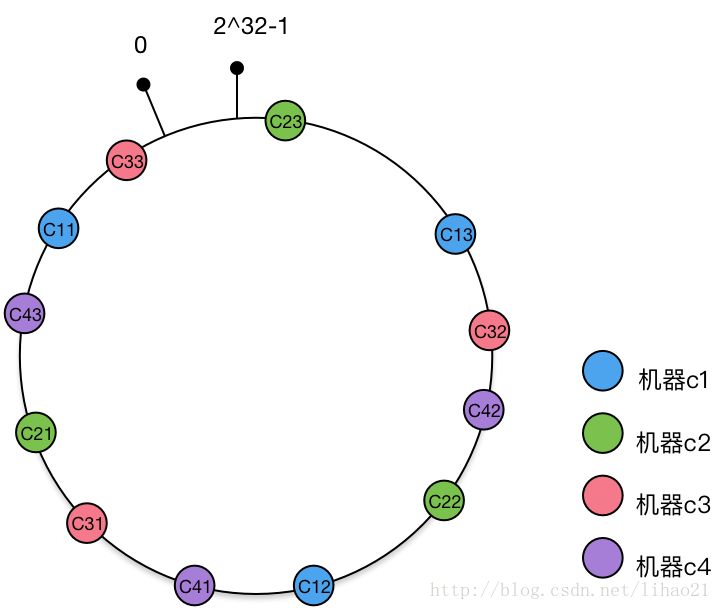
查找单个key，skiplist和平衡树的时间复杂度都为O(log n)，大体相当；而哈希表在保持较低的哈希值冲突概率的前提下，查找时间复杂度接近O(1)，性能更高一些。所以我们平常使用的各种Map或dictionary结构，大都是基于哈希表实现的。

从算法实现难度上来比较，skiplist比平衡树要简单得多。

113 一致性哈希

在分布式缓存系统中，由多台服务器提供服务，来自每个客户端的请求被hash到不同服务器上，计算方式是通过对服务器数量取模的方式。这样当服务器数量增加或减少时，将会出现大量的缓存访问不命中的情况。一致性哈希算法就是为了解决这个问题，一致性哈希采用一致性哈希环结构，环的起点为0，终点为2^32 - 1，且起点和终点相连。将机器和资源对象hash到对应环上的节点，哈希方法为求哈希值后对2^32取模，然后顺时针分配服务器。

对每个服务器和资源对象用不同的hash函数计算出多个值，对每个位置都放置一个虚拟节点，节点多了--，根据哈希函数的性质，平衡性自然就好了。



114 什么时候会生成默认构造函数？

含有类对象数据成员，该类对象类型有默认构造函数；基类带有默认构造函数的派生类；带有虚函数的类；带有虚基类的类；

115 消息对列

**解耦、异步、削峰**

A系统调用B系统、C系统，传统的调用是直接调用，但是当B系统说我不需要你提供数据了，这时候A需要改代码，Ｃ系统说我不需要某个字段了，这时候Ａ也要改代码，如果又多了一个D系统，A又要写代码。为了实现解耦，引入消息队列，Ａ将产生的数据丢到消息队列中，哪个系统需要 哪个系统就去取；

A系统调用B系统，B系统由于某个需要调用第三方接口超时，导致A系统响应速度慢，而B系统的好坏又不会影响业务逻辑，所以可以改为A异步调用B，A将消息丢到消息队列中，B系统订阅消息，实现A的快速响应；

当大量流量请求到系统A时，由于数据库的处理能力有限，造成数据库连接异常。使用消息队列，大量请求先丢到消息队列中，系统A使用按批拉数据的方式，批量处理数据，生产中，高峰期短暂的消息积压是允许的。

消息队列的缺点：

系统复杂性增加：加了消息队列，需要保证消息不会重复消费，需要保证消息的可靠性，需要保证消息队列的高可用

系统的可用性降低：如果消息队列挂了，那么系统也会受到影响；

MQ能否保证消息必达，即消息的可靠性？

为了降低消息丢失的概率，MQ需要进行超时和重传

(1) MQ-client-sender 发送消息给MQ-server

(2) MQ-server接收到消息后，发送 ACK消息给发送方

(3) MQ-client-sender 接收到 ACK消息后，则 消息已经投递成功

如果上述 2 消息丢失或者超时，MQ-client-sender 内的 timer 会重发消息，直到收到 ACK消息，如果重试N次后还未收到，则回调发送失败。需要注意的是，这个过程中 MQ-server 可能会收到同一条消息的多次重发。

对每条消息，MQ系统内部必须生成一个inner-msg-id，作为去重和幂等的依据，这个内部消息ID的特性是：

* 全局唯一
* MQ生成，具备业务无关性，对消息发送方和消息接收方屏蔽

(4) MQ-server 将消息发送给 MQ-client-receiver

(5) MQ-client-receiver 得到消息处理业务逻辑

(6) MQ-client-receiver 回复 ACK消息给 MQ-server

(7) MQ-server收到 ACK消息，将已消费的消息删除

如果上述 6 消息丢失或者超时，MQ-server 内的 timer 会重发消息，直到 MQ-server 收到ACK消息 并且 将已消费的消息删除，这个过程也可能会重发多次，MQ-client-receiver 也可能会收到同一条消息的多次重发。

需要强调的是，MQ-client-receiver 回ACK给 MQ-server，是消息消费业务方的主动调用行为，不能由 MQ-client-sender 自动发起，因为MQ系统不知道消费方什么时候真正消费成功。

为了保证业务幂等性，业务消息体中，必须有一个biz-id，作为去重和幂等的依据，这个业务ID的特性是：

* 对于同一个业务场景，全局唯一
* 由业务消息发送方生成，业务相关，对MQ透明
* 由业务消息消费方负责判重，以保证幂等

116 死锁调试

调试core dump文件，一般程序异常退出时，操作系统会把当前的内存状况存储在

core 文件中。

1.先用kill命令 kill -11 pid 杀掉进程，进程会产生一个segmentation fault，从而导致一个core dump，随后得到一个core文件，里面包含了死锁的时候，进程的内存镜像，也就包括了导致死锁的那几个线程的栈。

2.用gdb打开core文件，然后输入thread apply all bt ，gdb会打出所有线程的栈，如果发现有那么几个栈停在pthread\_wait或者类似调用上，就可以大致判断就是这几个线程导致的死锁，然后就能定位到产生死锁的位置

117 fork父子进程共享什么内容？

子进程从父进程继承了用户号和用户组号，用户信息，目录信息，环境（表），打开的文件描述符，堆栈，（共享）内存等。

经过fork()以后，父进程和子进程拥有相同内容的代码段、数据段和用户堆栈，就像父进程把自己克隆了一遍。事实上，父进程只复制了自己的PCB块。而代码段，数据段和用户堆栈内存空间并没有复制一份，而是与子进程共享。只有当子进程在运行中出现写操作时，才会产生中断，并为子进程分配内存空间。

118 共享内存

进程要想操作一块共享内存，首先要先分配一块共享内存，然后把内存共享段映射到自己的虚拟地址空间中，对内存段进行操作，读、写、获取内存段的信息等，然后脱离共享内存，释放共享内存。

注意：这里需要注意的是进程虽然脱离共享内存，但是此共享内存仍然存在，用ipcs仍可以查看到，释放该内存段后（可以理解为删除），共享内存段不存在了，用ipcs查看不到。

119 信号量的类型和linux信号

信号量是一种数据结构（可以是整型数、整型数组、链表、记录型变量（结构体））。信号量的值与相应资源的使用情况有关。信号量的值仅由P、V操作改变。

SIGINT：在键盘按下<Ctrl+C>组合键后产生，默认动作为终止进程；

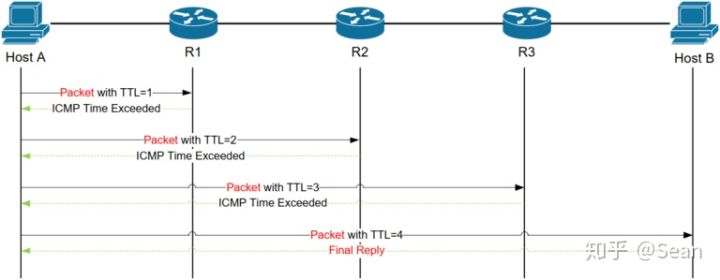
SIGQUIT：在键盘按下<Ctrl+\>组合键后产生，默认动作为终止进程；

SIGTSTP：ctrl+z产生信号

120 TraceRoute

**TTL**是 Time To Live的缩写，该字指定IP包被[路由器](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=15367&ss_c=ssc.citiao.link)丢弃之前允许通过的最大网段数量。TTL是IPv4包头的一个8 bit字段。

Traceroute 的基本原理是向外发送带有逐次递增 TTL 的数据包从而获取的路径中每一跳的信息。



Host A 向 Host B 做 traceroute，Host A 第一次会发出一个 TTL=1 的数据包，当该数据包到达 R1 时，TTL 会变为 0 （网络上每经过一跳 TTL 会减去 1），R1 会将 TTL=0 的数据包丢弃并返回一个 ICMP Time Exceeded 给 Host A。Host A 发出第二个数据包并将 TTL 增加1 （TTL=2)，该数据包到达 R2 后 TTL=0，R2 向 Host A 返回 ICMP Time Exceeded。依此类推，直到 TTL 增加到一个合适的值让数据包顺利到达 Host B，Host B 会返回一个 Final Replay 给 Host A。

UDP 和 ICMP traceroute 的区别就在于向外发送的数据包（上图中红色标明的 packet）和最后的 final reply。

UDP 向外发送的是一个 UDP 数据包，final reply 是 ICMP Destination Unreachable

ICMP 向外发送的是一个 ICMP Echo Request，final reply 是 ICMP Echo Reply

121 进程切换的步骤

1，保存程序计数其以及其他寄存器。

2， 更新当前处于“运行态”的进程的进程控制块，把进程状态改为相应状态，更新其他相关域

3， 把被切换进程的进程控制块移到相关状态的队列

4， 选择另外一个进程开始执行，把该进程进程控制块的状态改为“运行态”

5， 恢复被选择进程的处理器在最近一次被切换出运行态时的上下文，比如载入程序计数器以及其他处理器的值

进程间切换伴随着两次模式切换（用户--内核，内核--用户）。

**进程上下文的概念:**

操作系统管理很多进程的执行. 有些进程是来自各种程序、系统和应用程序的单独进程，而某些进程来自被分解为很多进程的应用或程序。当一个进程从内核中移出，另一个进程成为活动的, 这些进程之间便发生了上下文切换. 操作系统必须记录重启进程和启动新进程使之活动所需要的所有信息. 这些信息被称作**上下文, 它描述了进程的现有状态, 进程上下文是可执行程序代码是进程的重要组成部分, 实际上是进程执行活动全过程的静态描述, 可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等**

进程的上下文信息包括， 指向可执行文件的指针, 栈, 内存(数据段和堆), 进程状态, 优先级, 程序I/O的状态, 授予权限, 调度信息, 审计信息, 有关资源的信息(文件描述符和读/写指针), 关事件和信号的信息, 寄存器组(栈指针, 指令计数器)等等, 诸如此类.

**上下文切换**

进程被抢占CPU时候, 操作系统保存其上下文信息, 同时将新的活动进程的上下文信息加载进来, 这个过程其实就是**上下文切换**, 而当一个被抢占的进程再次成为活动的, 它可以恢复自己的上下文继续从被抢占的位置开始执行.

**上下文切换**(有时也称做**进程切换**或**任务切换**)是指CPU从一个进程或线程切换到另一个进程或线程

稍微详细描述一下，上下文切换可以认为是内核（操作系统的核心）在 CPU 上对于进程（包括线程）进行以下的活动：

挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处，

在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复

跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程

因此上下文是指某一时间点CPU寄存器和程序计数器的内容, 广义上还包括内存中进程的虚拟地址映射信息.

上下文切换只能发生在内核态中, 上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。  
Linux相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少.。

（同一进程内）线程间切换的步骤：

线程分两种，用户级线程和内核级线程

在用户级线程中，有关线程管理的所有工作都由应用程序完成，内核没有意识到线程的存在。

（同一进程内）用户级线程间切换时，只需要保存用户寄存器的内容，程序计数器，栈指针，不需要模式切换。

缺点：

1， 在进程的某个线程执行系统调用时，不仅该线程被阻塞，该线程所在进程的所有线程都被阻塞

2， 无法利用多处理器

在内核级线程中，有关线程的管理工作都是由内核完成的，应用程序部分没有线程管理的权限，只有一个接口（api)

（同一进程内）内核级线程间切换时，除了保存上下文，还要进行模式切换。

优点：

1， 可以利用多处理器

2， 线程阻塞不会导致进程阻塞

122 无锁线程同步机制

多线程访问共享内存，为了实现同步，常采用加锁的方式。

那么，如何采用不加锁的方式实现线程同步呢？

思路：

1、保存两块共享内存，一块用于读操作，一块用于写操作。

2、初始时，两块共享内存内容一致。读操作均是读取第一块共享内存的数据；写操作均是写第二块共享内存。

3、在多个读操作一个写操作的情况下，读操作均从第一块共享内存读取，写操作修改第二块共享内存的数据。直至写操作完成，交换两块共享内存的下标，即第二块共享内存用于接下来的读操作，第一块共享内存根据第二块共享内存数据更新，用于接来下的写操作。

123 SSL协议通信过程（对称+非对称）【双向认证】

(1) 浏览器发送一个连接请求给服务器;服务器将自己的证书(包含服务器公钥S\_PuKey)、对称加密算法种类及其他相关信息返回客户端;

(2) 客户端浏览器检查服务器传送到CA证书是否由自己信赖的CA中心签发。若是，执行4步;否则，给客户一个警告信息：询问是否继续访问。

(3) 客户端浏览器比较证书里的信息，如证书有效期、服务器域名和公钥S\_PK，与服务器传回的信息是否一致，如果一致，则浏览器完成对服务器的身份认证。

(4) 服务器要求客户端发送客户端证书(包含客户端公钥C\_PuKey)、支持的对称加密方案及其他相关信息。收到后，服务器进行相同的身份认证，若没有通过验证，则拒绝连接;

(5) 服务器根据客户端浏览器发送到密码种类，选择一种加密程度最高的方案，用客户端公钥C\_PuKey加密后通知到浏览器;

(6) 客户端通过私钥C\_PrKey解密后，得知服务器选择的加密方案，并选择一个通话密钥key，接着用服务器公钥S\_PuKey加密后发送给服务器;

(7) 服务器接收到的浏览器传送到消息，用私钥S\_PrKey解密，获得通话密钥key。

(8) 接下来的数据传输都使用该对称密钥key进行加密。

124 ARP

ARP（Address Resolution Protocol）是地址解析协议，是一种将IP地址转化成物理地址的协议。从IP地址到物理地址的映射有两种方式：表格方式和非表格方式。ARP具体说来就是将网络层（也就是相当于OSI的第三层）地址解析为数据链路层（也就是相当于OSI的第二层）的物理地址(注:此处物理地址并不一定指MAC地址)。

ARP原理：

某机器A要向主机B发送报文，会查询本地的ARP缓存表，找到B的IP地址对应的MAC地址后，就会进行数据传输。如果未找到，则A广播一个ARP请求报文（携带主机A的IP地址Ia——物理地址Pa），请求IP地址为Ib的主机B回答物理地址Pb。网上所有主机包括B都收到ARP请求，但只有主机B识别自己的IP地址，于是向A主机发回一个ARP响应报文。其中就包含有B的MAC地址，A接收到B的应答后，就会更新本地的ARP缓存。接着使用这个MAC地址发送数据（由网卡附加MAC地址）。因此，本地高速缓存的这个ARP表是本地网络流通的基础，而且这个缓存是动态的。

ARP欺骗：

ARP协议并不只在发送了ARP请求才接收ARP应答。当计算机接收到ARP应答数据包的时候，就会对本地的ARP缓存进行更新，将应答中的IP和MAC地址存储在ARP缓存中。因此，当局域网中的某台机器B向A发送一个自己伪造的ARP应答，而如果这个应答是B冒充C伪造来的，即IP地址为C的IP，而MAC地址是伪造的，则当A接收到B伪造的ARP应答后，就会更新本地的ARP缓存，这样在A看来C的IP地址没有变，而它的MAC地址已经不是原来那个了。

125 进程间加锁

<https://blog.csdn.net/qq_35396127/article/details/78942245>

默认 pthread\_mutex\_t 用于线程间同步，因为线程共享互斥量对象的内容。通过共享内存映射机制，可以使不同进程也能共享互斥量对象，因此也能通过互斥量类型实现进程间的同步

126 进程栈线程栈

<https://www.cnblogs.com/dongzhiquan/p/11415722.html>

一、进程栈

进程栈是属于用户态栈，和进程 虚拟地址空间 (Virtual Address Space) 密切相关。那我们先了解下什么是虚拟地址空间：在 32 位机器下，虚拟地址空间大小为 4G。这些虚拟地址通过页表 (Page Table) 映射到物理内存，页表由操作系统维护，并被处理器的内存管理单元 (MMU) 硬件引用。每个进程都拥有一套属于它自己的页表，因此对于每个进程而言都好像独享了整个虚拟地址空间。

Linux 内核将这 4G 字节的空间分为两部分，将最高的 1G 字节（0xC0000000-0xFFFFFFFF）供内核使用，称为内核空间。而将较低的3G字节（0x00000000-0xBFFFFFFF）供各个进程使用，称为 用户空间。每个进程可以通过系统调用陷入内核态，因此内核空间是由所有进程共享的。虽然说内核和用户态进程占用了这么大地址空间，但是并不意味它们使用了这么多物理内存，仅表示它可以支配这么大的地址空间。它们是根据需要，将物理内存映射到虚拟地址空间中使用。

二、线程栈

从 Linux 内核的角度来说，其实它并没有线程的概念。Linux 把所有线程都当做进程来实现，它将线程和进程不加区分的统一到了 task\_struct 中。线程仅仅被视为一个与其他进程共享某些资源的进程，而是否共享地址空间几乎是进程和 Linux 中所谓线程的唯一区别。线程创建的时候，加上了 CLONE\_VM 标记，这样 线程的内存描述符 将直接指向 父进程的内存描述符。

虽然线程的地址空间和进程一样，但是对待其地址空间的 stack 还是有些区别的。对于 Linux 进程或者说主线程，其 stack 是在 fork 的时候生成的，实际上就是复制了父亲的 stack 空间地址，然后写时拷贝 (cow) 以及动态增长。然而对于主线程生成的子线程而言，其 stack 将不再是这样的了，而是事先固定下来的，使用 mmap 系统调用，它不带有 VM\_STACK\_FLAGS 标记

三、进程内核栈

在每一个进程的生命周期中，必然会通过到系统调用陷入内核。在执行系统调用陷入内核之后，这些内核代码所使用的栈并不是原先进程用户空间中的栈，而是一个单独内核空间的栈，这个称作进程内核栈。进程内核栈在进程创建的时候，通过 slab 分配器从thread\_info\_cache缓存池中分配出来，其大小为THREAD\_SIZE，一般来说是一个页大小 4K；

thread\_union进程内核栈 和task\_struct进程描述符有着紧密的联系。由于内核经常要访问task\_struct，高效获取当前进程的描述符是一件非常重要的事情。因此内核将进程内核栈的头部一段空间，用于存放thread\_info结构体，而此结构体中则记录了对应进程的描述符，两者关系如下图（对应内核函数为dup\_task\_struct()）：

四、中断栈

进程陷入内核态的时候，需要内核栈来支持内核函数调用。中断也是如此，当系统收到中断事件后，进行中断处理的时候，也需要中断栈来支持函数调用。由于系统中断的时候，系统当然是处于内核态的，所以中断栈是可以和内核栈共享的。但是具体是否共享，这和具体处理架构密切相关。

X86 上中断栈就是独立于内核栈的；独立的中断栈所在内存空间的分配发生在arch/x86/kernel/irq\_32.c的irq\_ctx\_init()函数中(如果是多处理器系统，那么每个处理器都会有一个独立的中断栈)，函数使用\_\_alloc\_pages在低端内存区分配2个物理页面，也就是8KB大小的空间。有趣的是，这个函数还会为softirq分配一个同样大小的独立堆栈。如此说来，softirq将不会在hardirq的中断栈上执行，而是在自己的上下文中执行。

**127 多线程下fork**

<https://blog.csdn.net/q2519008/article/details/83824729>

我们知道通过fork创建的一个子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份拷贝，包括文本、数据和bss段、堆以及用户栈等。子进程还获得与父进程任何打开文件描述符相同的拷贝，这就意味着子进程可以读写父进程中任何打开的文件，父进程和子进程之间最大的区别在于它们有着不同的PID。

但是有一点需要注意的是，在Linux中，fork的时候只复制当前线程到子进程