# 1. select/poll/epoll有什么区别

本质上三者都是阻塞式I/O多路复用，也就是我们常说的同步I/O

（1）在select这种I/O多路复用机制下，我们需要把想监控的文件描述集合通过函数参数的形式告诉select，然后select会将这些文件描述符集合**拷贝**到内核中，我们知道数据拷贝是有性能损耗的，因此为了减少这种数据拷贝带来的性能损耗，Linux内核对集合的大小做了限制，并规定用户监控的文件描述集合不能超过1024个，同时当select返回后（将文件描述符从内核空间拷贝到用户空间）我们仅仅能知道有些文件描述符可以读写了，但是我们不知道是哪一个，因此程序员必须再遍历一边找到具体是哪个文件描述符可以读写了。

因此，总结下来select有这样几个缺点：

* 我能照看的文件描述符数量有限，不能超过1024个
* 用户给我的文件描述符需要拷贝到内核中
* 我只能告诉你有文件描述符满足要求了，但是我不知道是哪个，你自己一个一个去找吧(遍历轮询)

因此我们可以看到，select机制的特性在高性能网络服务器动辄几万几十万并发链接的场景下无疑是低效的

（2）poll和select是非常相似的，poll是已链表形式存储文件描述符，poll相对于select（数组存储fds）的优化仅仅在于解决了文件描述符不能超过1024个的限制，select和poll都会随着监控的文件描述增加而出现性能下降，因此不适合高并发场景。

（3）epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

调用epoll\_create时，做了以下事情：

1. 内核帮我们在epoll文件系统里建了个file结点；
2. 在内核cache里建了个红黑树用于存储以后epoll\_ctl传来的socket；
3. 建立一个list链表，用于存储准备就绪的事件。

调用epoll\_ctl时，做了以下事情：

1. 增加socket句柄时检查红黑树中是否存在，存在立即返回，不存在则把socket放到epoll文件系统里file对象对应的红黑树上；
2. 给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。

调用epoll\_wait时，做了以下事情：

观察list链表里有没有数据。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回。而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，epoll\_wait仅需要从内核态copy少量的句柄到用户态而已。

有两种模式，LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll\_wait时重复返回这个句柄，而ET模式仅在第一次返回。

epoll水平触发（LT）: 只要监听的文件描述符中有数据，就会触发epoll\_wait有返回值，这是默认的epoll\_wait的方式;

epoll边沿触发 （ET）: 只有监听的文件描述符的读/写事件发生，才会触发epoll\_wait有返回值;

**总结：**

**select和poll实现需要不断轮询所有的fd集合，直到设备就绪，期间可能要经过睡眠和唤醒多次交替，而且两者每次调用都需要把fd集合从用户态往内核态拷贝一次，这造成极大的开销。而epoll是当设备就绪时才调用回调函数，把就绪fd放入就绪链表中，只需要轮询就绪链表即可，判断一下就绪链表是否为空就行。**

**表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善**

# 2. time\_wait和close——wait

命令：netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

常用的三个状态是：ESTABLISHED表示正在通信 、TIME\_WAIT表示主动关闭、CLOSE\_WAIT表示被动关闭。

引入time\_wait的原因：

* 假设最后的一个ACK丢失，那么被动关闭一方收不到这最后一个ACK则会重发FIN。此时主动关闭一方必须保持一个有效的(time\_wait状态下维持)状态信息，以便可以重发ACK。如果主动关闭的socket不维持这种状态而是进入close状态，那么主动关闭的一方在收到被动关闭方重新发送的FIN时则响应给被动方一个RST。被动方收到这个RST后会认为此次回话出错了。所以如果TCP想要完成必要的操作而终止双方的数据流传输，就必须完全正确的传输四次握手的四步，不能有任何的丢失。这就是为什么在socket在关闭后，任然处于time\_wait状态的第一个原因。因为他要等待可能出现的错误(被动关闭端没有接收到最后一个ACK)，以便重发ACK。（**保证被动关闭连接的一方能被正确地关闭，即保证最后得ACK能被被动关闭方介绍，从而帮助其正常关闭**）
* 假设目前连接的通信双方都调用了close(),双方同时进入closed的终结状态，而没有走time\_wait状态。则会出现如下问题：假如现在有一个新的连接建立起来，使用的IP地址与之前的端口完全相同，现在建立的一个连接是之前连接的完全复用，我们还假定之前连接中有数据报残存在网络之中，这样的话现在的连接收到的数据有可能是之前连接的报文。为了防止这一点。TCP不允许新的连接复用time\_wait状态下的socket。处于time\_wait状态的socket在等待2MSL时间后（之所以是两倍的MSL，是由于MSL是一个数据报在网络中单向发出 到认定丢失的时间，即(Maximum Segment Lifetime)报文最长存活时间，一个数据报有可能在发送途中或是其响应过程中成为残余数据报，确认一个数据报及其响应的丢弃需要两倍的MSL），将会转为closed状态。这就意味着，一个成功建立的连接，必须使得之前网络中残余的数据报都丢失了。（**让两个方向上的旧数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的**）

time\_wait状态过多会占用端口资源，一个TCP连接至少消耗一个本地端口，会导致无法创建新连接。time\_wait问题可以通过调整内核参数和适当的设置web服务器的keep-Alive值来解决。

而出现大量close\_wait只有一种情况，那就是对方发送一个FIN后，程序自己这边没有进一步发送ACK以确认。换句话说就是在对方关闭连接后，程序里没有检测到，或者程序里本身就已经忘了这个时候需要关闭连接，于是这个资源就一直被程序占用着。

**在TIME-WAIT状态必须等待2MSL时间：**

第一，为了保证A发送的最后一个ACK报文能够到达B。这个ACK报文段有可能丢失，因而使处在LAST-ACK状态的B收不到对已发送的FIN+ACK报文段的确认。B会超时重传这个FIN+ACK报文段，而A就能在2MSL时间内收到这个重传的FIN+ACK报文段。如果A在TIME-WAIT状态不等待一段时间，而是在发送完ACK报文段后就立即释放连接，就无法收到B重传的FIN+ACK报文段，因而也不会再发送一次确认报文段。这样，B就无法按照正常的步骤进入CLOSED状态。

第二，A在发送完ACK报文段后，再经过2MSL时间，就可以使本连接持续的时间所产生的所有报文段都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求的报文段。

# 3. LRU算法机制

type LRUCache struct {

    Size       int

    Cap        int

    Cache      map[int]\*DLinkedList

    head, tail \*DLinkedList

}

type DLinkedList struct {

    key, val   int

    prev, next \*DLinkedList

}

func initDLinkedList(key, val int) \*DLinkedList {

    return &DLinkedList{key: key, val: val}

}

func Constructor(capacity int) LRUCache {

    lru := LRUCache{

        Cap:   capacity,

        Cache: map[int]\*DLinkedList{},

        head:  initDLinkedList(0, 0),

        tail:  initDLinkedList(0, 0),

    }

    lru.head.next = lru.tail

    lru.tail.prev = lru.head

    return lru

}

func (this \*LRUCache) Get(key int) int {

    if \_, ok := this.Cache[key]; !ok {

        return -1

    }

    node := this.Cache[key]

    this.moveToHead(node)

    return node.val

}

func (this \*LRUCache) Put(key int, value int) {

    if \_, ok := this.Cache[key]; !ok {

        node := initDLinkedList(key, value)

        this.Cache[key] = node

        this.addToHead(node)

        this.Size++

        if this.Size > this.Cap {

            temp := this.removeTail()

            delete(this.Cache, temp.key)

            this.Size--

        }

    } else {

        this.Cache[key].val = value

        this.moveToHead(this.Cache[key])

    }

}

func (this \*LRUCache) removeNode(node \*DLinkedList) {

    node.prev.next = node.next

    node.next.prev = node.prev

}

func (this \*LRUCache) addToHead(node \*DLinkedList) {

    node.prev = this.head

    node.next = this.head.next

    this.head.next.prev = node

    this.head.next = node

}

func (this \*LRUCache) moveToHead(node \*DLinkedList) {

    this.removeNode(node)

    this.addToHead(node)

}

func (this \*LRUCache) removeTail() \*DLinkedList {

    node := this.tail.prev

    this.removeNode(node)

    return node

}

# 4. 进程切换和线程切换

线程是调度的基本单位，而进程则是资源拥有的基本单位。说白了，所谓内核中的任务调度，实际上的调度对象是线程；而进程只是给线程提供了虚拟内存、全局变量等资源。

操作系统保持跟踪进程运行所需的所有状态信息，这种状态，就是上下文，上下文就是内核重新启动一个被抢占的进程所需的状态。上下文包括：通用目的寄存器、浮点寄存器、状态寄存器、程序计数器、用户栈、内核栈、描绘地址空间的页表、包含当前进程信息的进程表以及包含进程已打开文件的信息的文件表。

虚拟内存是操作系统为每个进程提供的一种抽象，每个进程都会有属于自己的、私有的、地址连续的虚拟内存，页表（地址空间映射）负责把虚拟地址空间映射到物理内存地址。每个进程都有自己的虚拟地址空间，进程内的所有线程共享进程的虚拟地址空间。

进程切换和线程切换的区别在于进程切换涉及虚拟地址空间的切换而线程不会。因为每个进程都有自己的虚拟地址空间，而线程是共享所在进程的虚拟地址空间的，因此同一个进程中的线程切换时不涉及虚拟地址空间的转换。

为什么虚拟地址空间切换会比较耗时呢？

现在我们已经知道了进程都有自己的虚拟地址空间，把虚拟地址转换为物理地址需要查找页表，页表查找是一个很慢的过程，因此通常使用Cache来缓存常用的地址映射，这样可以加速页表查找，这个cache就是TLB（translation Lookaside Buffer，我们不需要关心这个名字只需要知道TLB本质上就是一个cache，是用来加速页表查找的）。由于每个进程都有自己的虚拟地址空间，那么显然每个进程都有自己的页表，那么**当进程切换后页表也要进行切换，页表切换后TLB就失效了**，cache失效导致命中率降低，那么虚拟地址转换为物理地址就会变慢，表现出来的就是程序运行会变慢，而线程切换则不会导致TLB失效，因为线程线程无需切换地址空间，因此我们通常说线程切换要比较进程切换块，原因就在这里。

协程是有程序自身控制的，执行效率高，按照一定的顺序交替执行

**4.1 进程和线程的区别**

     a）进程是资源分配的最小单位，线程是任务执行的最小单位。

     b）进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此 CPU 切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。

     c）线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。不过如何处理好同步与互斥是编写多线程程序的难点。

     d）但是多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

# 5. 僵尸进程和孤儿进程

* 僵尸进程：一个子进程在父进程还没有调用wait()方法或者waitpid()方法的情况下退出，或子进程结束后向父进程发出SIGCHLD信号，父进程默认忽略了它，这个子进程就是僵尸进程；
* 孤儿进程：一个父进程退出，它的一个或多个子进程还在运行，子进程将成为孤儿进程，孤儿进程将被init进程所收养；

僵尸进程将会导致资源浪费，因为僵尸进程会一直占用进程号，而操作系统的进程号是有限的，如果僵尸进程过多，会导致系统无法分配新的进程号，而孤儿进程则不会。进程设置僵尸状态的目的是维护子进程的信息，以便父进程在以后某个时间获取。

解决方案

1)kill杀死元凶父进程(一般不用)

严格的说，僵尸进程并不是问题的根源，罪魁祸首是产生大量僵死进程的父进程。因此，我们可以直接除掉元凶，通过kill发送SIGTERM或者SIGKILL信号。元凶死后，僵尸进程进程变成孤儿进程，由init充当父进程，并回收资源。

或者运行：kill -9 父进程的pid值、

2)父进程用wait或waitpid去回收资源(方案不好)

父进程通过wait或waitpid等函数去等待子进程结束，但是不好，会导致父进程一直等待被挂起，相当于一个进程在干活，没有起到多进程的作用。

3)通过信号机制，在处理函数中调用wait，回收资源

通过信号机制，子进程退出时向父进程发送SIGCHLD信号，父进程调用signal(SIGCHLD,sig\_child)去处理SIGCHLD信号，在信号处理函数sig\_child()中调用wait进行处理僵尸进程。什么时候得到子进程信号，什么时候进行信号处理，父进程可以继续干其他活，不用去阻塞等待。

# 6. TCP如何保证传输可靠性（稳定传输）

TCP协议保证数据传输可靠性的方式主要有：

（校序重流拥）

**校验和**

       发送的数据包的二进制相加然后取反，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。

**确认应答+序列号**

       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。

**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。

**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。

        接收方有即时窗口（滑动窗口），随ACK报文发送

**拥塞控制**

       当网络拥塞时，减少数据的发送。

       发送方有拥塞窗口，发送数据前比对接收方发过来的即使窗口，取小

       慢启动、拥塞避免、拥塞发送、快速恢复

       （1）慢开始与拥塞避免

       发送的最初执行慢开始，令拥塞窗口大小为1，发送方只能发送1个报文段；当收到确认后，将拥塞窗口大小加倍。设置一个慢开始门限，当 拥塞窗口的大小大于慢开始门限 时，进入拥塞避免，每个轮次只将拥塞窗口加1。如果出现了超时，则令慢开始门限 = 拥塞窗口大小 / 2，然后重新执行慢开始。

      （2）快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。在 发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传， 立即重传下一个报文段。在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此 执行快恢复，令慢开始门限 = 拥塞窗口大小 / 2 ，拥塞窗口大小 = 慢开始门限 ，注意到此时直接进入拥塞避免。

应用数据被分割成TCP认为最适合发送的数据块。 TCP的接收端会丢弃重复的数据。

[TCP的流量控制和拥塞控制](https://zhuanlan.zhihu.com/p/37379780)

# 7. cookie和session

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和Session就是这样的一种机制。它可以弥补HTTP协议无状态的不足。

cookie是客户端会话技术，session是服务端会话技术。

session与Cookie的区别：

1. session存储数据在服务器端，Cookie在客户端
2. session没有数据大小限制，Cookie有
3. session数据安全，Cookie相对于不安全

# 8. 进程间通信

IPC方式包括：管道、系统IPC（消息队列、共享内存、信号量、信号 ）和套接字（socket）

（1）匿名管道的通信范围是存在父子关系的进程，命名管道可以在不相关的进程间互相通信。通信数据遵循先进先出原则，效率低。

（2）A进程要给B进程发送消息，A进程把数据放在对应的消息队列就可以正常返回了，B进程需要的时候再去读取就可以了，这种通信方式是消息队列。消息队列是保存在内核中的消息链表，用户需自定义消息体的数据类型，按照这个消息体去发送和接收数据。消息队列的生命周期由内核决定，如果没有释放消息队列或者没有关闭操作系统，消息队列就会一直存在。但消息队列通信不及时，而且不适合大数据的传输（因为内核限制了消息的大小以及队列的最大长度），消息队列通信过程中，存在用户态和内核态之间的数据拷贝开销。

（3）共享内存解决了消息队列在用户态和内核态之间的数据拷贝问题，共享内存的机制就是拿出进程拿出一块虚拟地址空间，映射到相同的物理内存中，数据通信都发生在这个共享的内存里，大大提升了进程间通信的速度，但是如果多个进程同时修改同一个共享内存，那就发生了冲突。

（4）信号量解决共享内存中同时修改数据的冲突问题，信号量保证了共享的资源在任意时刻只能被一个进程访问。信号量其实就是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据。进程的互斥（mutual exclusion ）是解决进程间竞争关系( 间接制约关系) 的手段。 进程互斥指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到占有资源的进程释放该资源。进程同步指两个以上进程基于某个条件来协调它们的活动。一个进程的执行依赖于另一个协作进程的消息或信号，当一个进程没有得到来自于另一个进程的消息或信号时则需等待，直到消息或信号到达才被唤醒。

（5）前面所说的进程间通信都是常规状态下的工作模式，对于异常情况下的工作模式，就需要用到信号的方式来通知进程。在Linux操作系统汇总，为了响应各种事件，提供了几十种信号，分别代表不同的意义，可以通过kill -l命令查看所有的信号。信号事件的来源主要由硬件来源（如键盘cltr+c）和软件来源（如kill命令）。信号是进程间通信机制中的唯一的异步通信机制，因为可以在任何时候发送信号给某一进程，一旦有信号发生，进程就对信号进程处理，处理方式有：执行默认操作、捕捉信号然后执行相应的信号处理函数、忽略信号不做任何处理。

（6）前面说的通信机制，都是工作于同一台主机，如果要与不同主机的进程间通信，那么就需要socket通信了。socket实际上不仅用于不同的主机进程间通信，还可以用于本地主机进程间通信，根据创建socket的类型不同，可分为三种常见的通信方式，一个是基于TCP协议的通信方式，一个是基于UDP协议的通信方式，一个是本地进程间的通信方式。

同个进程下的线程之间都是共享进程的资源，只要是共享变量都可以做到线程间通信，比如全局变量，所以对于线程间关注的不是通信方式，而是关注多线程竞争共享资源的问题，信号量也同样可以在线程间实现互斥与同步

线程间的通信方式：

a) 互斥量(Mutex)：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。比如 Java 中的 synchronized 关键词和各种 Lock 都是这种机制。

b) 信号量(Semphares)：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量。

c) 事件(Event):Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作。

# 9. iNode和block

表面上，用户通过文件名打开文件，实际上，系统内部将这个过程分为三步：

     1.系统找到这个文件名对应的inode号码；

     2.通过inode号码，获取inode信息；

     3.根据inode信息，找到文件数据所在的block，并读出数据。

其实系统还要根据inode信息，看用户是否具有访问的权限，有就指向对应的数据block，没有就返回权限拒绝。

文件名并不存在iNode元信息中，而是放在目录中

# 10. 硬链接和软链接

**硬链接(hard link)：**

　　　　A是B的硬链接（A和B都是文件名），则A的目录项中的inode节点号与B的目录项中的inode节点号相同，即一个inode节点对应两个不同的文件名，两个文件名指向同一个文件，A和B对文件系统来说是完全平等的。如果删除了其中一个，对另外一个没有影响。每增加一个文件名，inode节点上的链接数增加一，每删除一个对应的文件名，inode节点上的链接数减一，直到为0，inode节点和对应的数据块被回收。注：文件和文件名是不同的东西，rm A删除的只是A这个文件名，而A对应的数据块（文件）只有在inode节点链接数减少为0的时候才会被系统回收。

**软链接(soft link)：**

　　　　A是B的软链接（A和B都是文件名），A的目录项中的inode节点号与B的目录项中的inode节点号不相同，A和B指向的是两个不同的inode，继而指向两块不同的数据块。但是A的数据块中存放的只是B的路径名（可以根据这个找到B的目录项）。A和B之间是“主从”关系，如果B被删除了，A仍然存在（因为两个是不同的文件），但指向的是一个无效的链接。

**主要区别、限制：**

硬链接：

　　a.不能对目录创建硬链接，原因有几种，最重要的是：文件系统不能存在链接环（目录创建时的".."除外，这个系统可以识别出来）,存在环的后果会导致例如文件遍历等操作的混乱(du，pwd等命令的运作原理就是基于文件硬链接，顺便一提，ls -l结果的第二列也是文件的硬链接数，即inode节点的链接数)

　　b：不能对不同的文件系统创建硬链接,即两个文件名要在相同的文件系统下。

　　c：不能对不存在的文件创建硬链接，由原理即可知原因。

软链接：

　　a.可以对目录创建软链接，遍历操作会忽略目录的软链接。

　　b:可以跨文件系统

　　c:可以对不存在的文件创建软链接，因为放的只是一个字符串，至于这个字符串是不是对于一个实际的文件，就是另外一回事了

# 11. 五种IO模型的特点

IO模型：阻塞式IO、非阻塞式IO、IO多路复用、信号驱动式IO、异步IO

[五种IO模型的特点和区别](https://juejin.cn/post/6844903728718462990)

# 12. HTTP状态码

|  |  |
| --- | --- |
| 分类 | 分类描述 |
| 1\*\* | 信息，服务器收到请求，需要请求者继续执行操作 |
| 2\*\* | 成功，操作被成功接收并处理 |
| 3\*\* | 重定向，需要进一步的操作以完成请求 |
| 4\*\* | 客户端错误，请求包含语法错误或无法完成请求 |
| 5\*\* | 服务器错误，服务器在处理请求的过程中发生了错误 |

2xx 类状态码表示服务器**成功**处理了客户端的请求，也是我们最愿意看到的状态。

「**200 OK**」是最常见的成功状态码，表示一切正常。如果是非 HEAD 请求，服务器返回的响应头都会有 body 数据。

「**204 No Content**」也是常见的成功状态码，与 200 OK 基本相同，但响应头没有 body 数据。

「**206 Partial Content**」是应用于 HTTP 分块下载或断点续传，表示响应返回的 body 数据并不是资源的全部，而是其中的一部分，也是服务器处理成功的状态。

*3xx*

3xx 类状态码表示客户端请求的资源发送了变动，需要客户端用新的 URL 重新发送请求获取资源，也就是**重定向**。

「**301 Moved Permanently**」表示永久重定向，说明请求的资源已经不存在了，需改用新的 URL 再次访问。

「**302 Found**」表示临时重定向，说明请求的资源还在，但暂时需要用另一个 URL 来访问。

301 和 302 都会在响应头里使用字段 Location，指明后续要跳转的 URL，浏览器会自动重定向新的 URL。

「**304 Not Modified**」不具有跳转的含义，表示资源未修改，重定向已存在的缓冲文件，也称缓存重定向，用于缓存控制。

*4xx*

4xx 类状态码表示客户端发送的**报文有误**，服务器无法处理，也就是错误码的含义。

「**400 Bad Request**」表示客户端请求的报文有错误，但只是个笼统的错误。

「**403 Forbidden**」表示服务器禁止访问资源，并不是客户端的请求出错。

「**404 Not Found**」表示请求的资源在服务器上不存在或未找到，所以无法提供给客户端。

*5xx*

5xx 类状态码表示客户端请求报文正确，但是**服务器处理时内部发生了错误**，属于服务器端的错误码。

「**500 Internal Server Error**」与 400 类型，是个笼统通用的错误码，服务器发生了什么错误，我们并不知道。

「**501 Not Implemented**」表示客户端请求的功能还不支持，类似“即将开业，敬请期待”的意思。

「**502 Bad Gateway**」通常是服务器作为网关或代理时返回的错误码，表示服务器自身工作正常，访问后端服务器发生了错误。

「**503 Service Unavailable**」表示服务器当前很忙，暂时无法响应服务器，类似“网络服务正忙，请稍后重试”的意思。

# 13. ARP协议的工作过程

* 首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，已表示IP地址和MAC地址之间的对应关系。
* 当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP数据包，该数据包包括的内容有：源主机IP地址，源主机MAC地址，目的主机IP地址。
* 当本网络的所有主机收到该ARP数据包时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在则覆盖，然后将自己的MAC地址写入ARP响应包中，告诉源主机自己是它想要找的MAC地址。
* 源主机收到ARP响应包后，将目的主机的IP和MAC地址写入ARP列表，并利用该信息发送数据。如果源主机一直没有收到ARP响应包，表示ARP查询失败。

# 14. 长连接和短连接

[**长短连接的区别**](https://www.jianshu.com/p/a697e504b1ab)

短连接的操作步骤是：

建立连接——数据传输——关闭连接...建立连接——数据传输——关闭连接

长连接的操作步骤是：

建立连接——数据传输...（保持连接）...数据传输——关闭连接

# 15. 红黑树定义和性质

红黑树是一种含有红黑结点并能自平衡的二叉查找树。它必须满足下面性质：

* 性质1：每个节点要么是黑色，要么是红色。
* 性质2：根节点是黑色。
* 性质3：每个叶子节点（NIL）是黑色。
* 性质4：每个红色结点的两个子结点一定都是黑色。
* **性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。**

从性质5又可以推出：

* 性质5.1：如果一个结点存在黑子结点，那么该结点肯定有两个子结点

# 16. 分段机制和分页机制

分段机制就是把虚拟地址空间中的虚拟内存组织成一些长度可变的称为段的内存块单元，每个段由三个参数定义：段基地址、段限长和段属性，段可以用来存放程序的代码、数据、堆栈，或者用来存放系统数据结构。

分页机制在段机制后进行的，它进一步将线性地址转换成物理地址，分页机制支持虚拟存储技术，在使用虚拟存储的环境中，大容量的线性地址空间需要使用小块的物理内存来模拟。线性地址是连续的，如果直接使用线性地址作为物理地址，那么为每个段分配的物理内存就必须是连续的物理内存。这不利用碎片化内存的利用，为内存管理增大了难度。所以引入了分页机制，将地址分为大小固定的页（一般为4096字节），按页为单位进行映射。连续的线性地址可以映射到不连续的的物理内存上。

分页的另一个优点是，当物理内存不足时，可以按页为单位将内存的内容转换到磁盘上保存起来。

如果不使用分页，则只能整个段整个段的进行转换

分段机制和分页机制的区别

　　1、分页机制会使用大小固定的内存块，而分段管理则使用了大小可变的块来管理内存。

　　2、分页使用固定大小的块更为适合管理物理内存，分段机制使用大小可变的块更适合处理复杂系统的逻辑分区。

　　3、段表存储在线性地址空间，而页表则保存在物理地址空间

# 17. 堆和栈的区别

栈由操作系统**自动分配释放** ，用于存放函数的参数值、局部变量等，其操作方式类似于数据结构中的栈

1. 函数中定义的局部变量按照先后定义的顺序依次压入栈中,也就是说相邻变量的地址之间不会存在其它变量。

2. 栈的内存地址生长方向与堆相反，由高到底，所以后定义的变量地址低于先定义的变量

3. 栈中存储的数据的生命周期随着函数的执行完成而结束

堆由**程序员分配释放**， 若程序员不释放，程序结束时由OS回收，分配方式倒是类似于链表

1. 堆的内存地址生长方向与栈相反，由低到高，但需要注意的是，后申请的内存空间并不一定在先申请的内存空间的后面

原因是先申请的内存空间一旦被释放，后申请的内存空间则会利用先前被释放的内存，从而导致先后分配的内存空间在地址上不存在先后关系

关于堆上内存空间的分配过程:

1. 操作系统有一个记录空闲内存地址的链表

2. 当系统收到程序的申请时,会遍历该链表,寻找第一个空间大于所申请空间的堆节点

3. 将该节点从空闲节点链表中删除，并将该节点的空间分配给程序

对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确地释放本内存空间。

由于找到的堆节点的大小不一定正好等于申请的大小，**系统会自动的将多余的那部分重新放入空闲链表**。

堆与栈区别:

堆与栈实际上是操作系统对进程占用的内存空间的两种管理方式，主要有如下几种区别：

（1）**管理方式不同**。栈由操作系统自动分配释放，无需我们手动控制；堆的申请和释放工作由程序员控制，容易产生内存泄漏；

（2）**空间大小不同**。每个进程拥有的栈的大小要远远小于堆的大小。理论上，程序员可申请的堆大小为虚拟内存的大小，进程栈的大小64bits的Windows默认1MB，64bits的Linux默认10MB；

（3）**生长方向不同**。堆的生长方向向上，内存地址由低到高；栈的生长方向向下，内存地址由高到低。

（4）**分配方式不同**。堆都是动态分配的，没有静态分配的堆。

        栈有2种分配方式：

              静态分配和动态分配。静态分配是由操作系统完成的，比如局部变量的分配。

              动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由操作系统进行释放，无需我们手工实现

（5）分配效率不同。

         栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。

         堆则是由C/C++提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，堆的效率比栈要低得多。

（6）存放内容不同。栈存放的内容，函数返回地址、相关参数、局部变量和寄存器内容等。当主函数调用另外一个函数的时候，要对当前函数执行断点进行保存，需要使用栈来实现，

        首先入栈的是主函数下一条语句的地址，即扩展指针寄存器的内容（EIP），然后是当前栈帧的底部地址，

        即扩展基址指针寄存器内容（EBP），再然后是被调函数的实参等，一般情况下是按照从右向左的顺序入栈，

        之后是被调函数的局部变量，注意静态变量是存放在数据段或者BSS段，是不入栈的。出栈的顺序正好相反，

        最终栈顶指向主函数下一条语句的地址，主程序又从该地址开始执行。堆，一般情况堆顶使用一个字节的空间来存放堆的大小，而堆中具体存放内容是由程序员来填充的。

# 18. 用户态和内核态的切换和区别

当一个任务(进程)执行系统调用而陷入内核代码中执行时，我们就称进程处于内核状态。此时处理器处于特权级最高的(0级)内核代码。当进程处于内核态时，执行的内核代码会使用当前的内核栈。每个进程都有自己的内核栈。当进程在执行用户自己的代码时，则称其处于用户态。即此时处理器在特权级最低的用户代码中运行。当正在执行用户程序而突然中断时，此时用户程序也可以象征性地处于进程的内核态。因为中断处理程序将使用当前进程的内核态。

用户态切换到内核态的3种方式

a.系统调用

这是用户进程主动要求切换到内核态的一种方式，用户进程通过系统调用申请操作系统提供的服务程序完成工作。

而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如Linux的ine 80h中断。

b.异常

当CPU在执行运行在用户态的程序时，发现了某些事件不可知的异常，这是会触发由当前运行进程切换到处理此

异常的内核相关程序中，也就到了内核态，比如缺页异常。

c.外围设备的中断

当外围设备完成用户请求的操作之后，会向CPU发出相应的中断信号，这时CPU会暂停执行下一条将要执行的指令

转而去执行中断信号的处理程序，如果先执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了有

用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

# 19. http1.0/htpp1.1/http2.0的区别

（1）http1.0是短连接，每发起一个请求，都要新建一次TCP连接（三次握手），而且是串行请求，也就是每个请求都得得等响应回来了才能发起另一个请求。

    （2）http1.1采用的是长连接，也叫持久连接，多个请求可以复用同一个TCP连接，减少了TCP连接的重复建立和断开的开销，持久连接的特点是，只要任意一端没有明确提出断开连接，则保持TCP连接状态。采用了管道网络传输，也就是在同一个TCP连接中，客户端可以发起多个请求，只要第一个请求发出去了，不必等其回来，就可以发送第二个请求出去，可以减少整体的响应时间。http1.1的缺点：

* 请求/响应头部未经压缩就发送，首部信息越多延迟越大。只能压缩body数据。
* 发送冗长的首部，每次互相发送相同的首部造成的浪费比较多。
* 服务器还是按照请求的顺序来响应请求，要是前面的响应特别慢，后面的就会有需要请求排队等着，这称为队头阻塞。
* 没有请求优先级控制
* 请求只能从客户端开始，服务器只能被动响应。

    （3）http2.0是基于HTTPS的，安全有保障，相对于http1.1的改进：

* 头部压缩：http2.0会压缩头部信息，如果同时发出多个请求，头部信息是一样的或者相似的，那么协议会消除掉重复的部分：在客户端和服务器同时维护一张头信息表，所有字段都存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就提高了速度。
* 二进制格式：不再采用纯文本报文格式，而是全面采用了二进制格式，头信息和数据体都是二进制，并且统称为帧：头信息帧和数据帧。
* 数据流：每个请求或者响应的所有数据包，称为一个数据流，都有编号，客户端的数据流编号为奇数，服务端发出的数据流编号为偶数。客户端还可以指定数据流的优先级，优先级高的请求，服务器就先响应该请求。
* 多路复用：http2.0可以在一个连接中并发多个请求或者响应，而不用按照顺序一一对应。
* 服务器推送：http2.0中服务器可以主动向客户端发送消息，比如在浏览器刚请求HTML的时候，就提前把可能用到的JS/CSS等静态资源主动发给客户端，减少延时的等待。

     在http2.0中，一旦发生丢包，就有阻塞住同一个TCP连接中的所有HTTP请求，等待重传，并没有彻底解决队头阻塞。HTTP3.0（QUIC）是基于UDP的协议，就解决了这个问题。

# 20. mysql是如何解决幻读的

    （1）多版本并发控制（MVCC）（快照读、一致性读情况下），将历史数据保存一份快照，其他事务增加和删除数据，对当前事务来说是不可见的

    （2）next-key锁（当前读情况下），next-key锁包含两部分：记录锁（行锁）、间隙锁，记录锁是加在索引上的锁，间隙锁是加在索引之间的，将当前数据行与上一条数据和下一条数据之间的间隙锁定，保证此范围内读取的数据时一致的。

# 21. 打开一个URL的过程

    （1）浏览器解析URL（比如URL的协议、web服务器名称、数据路径名），生成HTTP请求信息（HTTP报文：请求行、消息头、消息体），

    （2）DNS查询，根据域名查询到对应的IP地址（根DNS服务器、顶级域DNS服务器、权威DNS服务器）

    （3）到达传输层，利用TCP三次握手建立连接，如果HTTP请求消息比较长，超过了MSS（除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度）的长度，这时就需要把HTTP的的数据以MSS的长度为单位进行拆分；然后生成TCP报文。TCP报文头部包括：源端口号、目标端口号、包的序号、确认号、状态位、窗口大小等等。如果是HTTPS，还需要TLS四次握手建立加密通道。

    （4）到达网络层，根据源IP地址和目的IP地址等信息生成IP报文头部，然后获取目的IP的MAC地址（ARP协议），生成MAC报文头部，以上这些信息组合成一个网络包。

    （5）生成的网络包经过网卡转换为电信号，通过网线发送过去，这一历程会经过交换机、路由器等各层网络设备到达目的主机。

    （6）数据包抵达服务器后，服务器会按照前面的逆操作来对数据进行拆包，解析客户端的请求，然后把相应的内容封装成HTTP响应报文，同样需要加上TCP/IP/MAC等头部，然后发送给客户端

    （7）客户端接收到响应数据包后，解析响应报文，然后交给浏览器去渲染页面

    （8）最后，如果数据交换完毕了，那就进行TCP四次挥手断开连接

# 22. 进程调度算法

    先来先服务调度算法、最短作业优先算法、高响应比优先调度算法、时间片轮转调度算法、最高优先级调度算法、多级反馈队列调度算法

# 23. 内存页面置换算法（虚拟内存映射到物理内存）

    最佳页面置换算法、先进先出置换算法、最近最久未使用置换算法、时钟页面置换算法、最不常用置换算法

# 24. 磁盘调度算法

    先来先服务算法、最短寻道时间优先算法、扫描算法、循环扫描算法、LOOK与C-LOOK算法

# 25. HTTP 与 HTTPS 有哪些区别？

* HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
* HTTP 连接建立相对简单， TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
* HTTP 的端口号是 80，HTTPS 的端口号是 443。
* HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的

# 26. TCP 和 UDP 区别：

*（1）. 连接*

* TCP 是面向连接的传输层协议，传输数据前先要建立连接。
* UDP 是不需要连接，即刻传输数据。

*（2）. 服务对象*

* TCP 是一对一的两点服务，即一条连接只有两个端点。
* UDP 支持一对一、一对多、多对多的交互通信

*（3）. 可靠性*

* TCP 是可靠交付数据的，数据可以无差错、不丢失、不重复、按需到达。
* UDP 是尽最大努力交付，不保证可靠交付数据。

*（4）. 拥塞控制、流量控制*

* TCP 有拥塞控制和流量控制机制，保证数据传输的安全性。
* UDP 则没有，即使网络非常拥堵了，也不会影响 UDP 的发送速率。

*（5）. 首部开销*

* TCP 首部长度较长，会有一定的开销，首部在没有使用「选项」字段时是 20 个字节，如果使用了「选项」字段则会变长的。
* UDP 首部只有 8 个字节，并且是固定不变的，开销较小。

*（6）. 传输方式*

* TCP 是流式传输，没有边界，但保证顺序和可靠。
* UDP 是一个包一个包的发送，是有边界的，但可能会丢包和乱序。

*（7）. 分片不同*

* TCP 的数据大小如果大于 MSS 大小，则会在传输层进行分片，目标主机收到后，也同样在传输层组装 TCP 数据包，如果中途丢失了一个分片，只需要传输丢失的这个分片。
* UDP 的数据大小如果大于 MTU 大小，则会在 IP 层进行分片，目标主机收到后，在 IP 层组装完数据，接着再传给传输层，但是如果中途丢了一个分片，则就需要重传所有的数据包，这样传输效率非常差，所以通常 UDP 的报文应该小于 MTU。

# 27. TCP 和 UDP 应用场景：

由于 TCP 是面向连接，能保证数据的可靠性交付，因此经常用于：

* FTP 文件传输
* HTTP / HTTPS

由于 UDP 面向无连接，它可以随时发送数据，再加上UDP本身的处理既简单又高效，因此经常用于：

* 包总量较少的通信，如 DNS 、SNMP 等
* 视频、音频等多媒体通信
* 广播通信

# 28. TCP为什么不使用两次握手或者四次握手

     因为三次握手才能保证双方具有接收和发送的能力（这是片面的）

     三次握手的**首要原因是为了防止旧的重复连接初始化造成混乱。**

     TCP 建立连接时，通过三次握手能防止历史连接的建立，能减少双方不必要的资源开销，能帮助双方同步初始化序列号。序列号能够保证数据包不重复、不丢弃和按序传输。

     不使用「两次握手」和「四次握手」的原因：

* 「两次握手」：无法防止历史连接的建立，会造成双方资源的浪费（如果客户端的 SYN 阻塞了（变成了历史连接），重复发送多次 SYN 报文，那么服务器在收到请求后就会建立多个冗余的无效链接，造成不必要的资源浪费），也无法可靠的同步双方序列号（TCP 协议的通信双方， 都必须维护一个「序列号」， 序列号是可靠传输的一个关键因素）；
* 「四次握手」：三次握手就已经理论上最少可靠连接建立，所以不需要使用更多的通信次数。

# 29. 自旋锁和互斥锁的区别

加锁的目的就是保证共享资源在任意时间里，只有一个线程访问，这样就可以避免多线程导致共享数据错乱的问题。

当已经有一个线程加锁后，其他线程加锁则就会失败，互斥锁和自旋锁对于加锁失败后的处理方式是不一样的：

* **互斥锁**加锁失败后，线程会**释放 CPU** ，给其他线程；
* **自旋锁**加锁失败后，线程会**忙等待**，直到它拿到锁；

如果你能确定被锁住的代码执行时间很短，就不应该用互斥锁，而应该选用自旋锁，否则使用互斥锁。

# 30. B 树 和 B+ 树的区别，为什么 mysql 要用 B+ 树，mongodb 要用 B 树？

    B树的明显特点：树内的每个节点都存储数据、叶子节点之间无指针相邻

    B+树的明显特点：数据只出现在叶子节点、所有叶子节点增加了一个链指针

    （1）MongoDB是非关系性数据库，使用json格式保存数据，单一查询比较常见，使用B树作为索引结构，所有节点都有data域，只要找到了指定索引就可以进行访问，单次查询效率较好，最好的情况是O(1)。

    （2）MySQL是关系型数据库，数据的关联性是非常强的，区间访问是很常见的操作，B+树由于数据全部存储在叶子节点，并且通过指针连在一块，这样就很容易进行区间遍历访问。还有一点就是，关系型数据库数据量大，为了减少内存的占用，索引也会被存储在磁盘上，而B+树除了叶子节点其他节点并不存储数据，节点小，磁盘IO次数就少，效率就高了。

# 31. redis中为什么不用红黑树而使用跳表（sorted-set使用跳表和哈希表）

    （1）跳表的时间复杂度和红黑树一样，但实现起来更简单，而且代码可读性更好

    （2）红黑树的插入和删除有可能引发整个子树的调整，逻辑复杂，而跳表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。

    （3）跳表区间查找效率更高，跳表可以做到O(logn)的时间复杂度定位区间的起点，然后在原始链表中顺序往后遍历查找就行。

# 31. ping命令有什么用

    ping命令主要是查看与某台机器的连接情况，是基于ICMP协议的，是网络层上的协议，

    TTL：指IP包被路由器丢弃之前允许通过的最大网段数量

# 32. HTTP请求报文和响应报文

    一个HTTP请求报文由请求行、请求头部、空行和请求数据四个部分组成

    （1）请求行由请求方法字段、URL字段和HTTP协议版本字段3个字段组成，他们用空格分隔

    （2）请求头部由关键字/值对组成，每行一对，关键字和值用英文冒号“:”分隔。请求头部通知服务器有关于客户端请求的信息，比如User-Agent、Accept、Host等。

    （3）最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

    （4）请求数据不在GET方法中使用，而是在POST方法中使用。POST方法适用于需要用户填写表单的场合。与请求数据相关的最常使用的请求头是Content-Type和Content-Length。

    一个HTTP响应也you四部分组成：状态行、消息报头、空行和响应正文。

# 33. UDP为什么快

    不需要建立连接、对于收到的数据，不用给出确认、没有超时重发机制、没有流量控制和拥塞控制

# 34. 电子邮件的发送过程?

一个电子邮件系统由三部分组成：用户代理、邮件服务器以及邮件协议。

邮件协议包含发送协议和读取协议，发送协议常用 SMTP，读取协议常用 POP3 和 IMAP。

（1）用户A的邮箱是QQ邮箱，他要发往的邮箱是163邮箱，用户A写好一封邮件点击发送， 即提交到了QQ邮箱服务器，使用的是SMTP协议。

（2）QQ邮箱会对A发送邮件的收件地址进行解析，判断是否为内部邮箱的账号，如果也是QQ邮箱，会直接存储到自己的存储空间，如果不是则会发送到指定邮箱服务器，使用 的也是SMTP协议。

（3）163的邮箱服务器收到邮件后会再次判断该邮件是否为自己的邮件，如果是则存到自己的存储空间，等待POP3服务去读取邮件。

（4）用户B收到消息后，打开客户端访问163服务器，调用POP3服务。

Pop3服务接到指令后，读取存储空间中发送给B的未读邮件服务。

（5）将读取到的邮件返回给客户端软件。

# 35. 介绍一下数据库的事务

    事务是一个操作序列，这些操作要么全部执行，要么都不执行。事务具有四大特性：原子性、一致性、隔离性、持久性。

# 36. MySQL有哪些隔离级别

   （1） 读未提交（ru，脏读、不可重复读、幻读）

            读未提交，就是可以读取未提交的内容，在这种隔离级别下，查询是不会加锁的。

   （2）读已提交（rc，不可重复读、幻读）

            一个事务的更新操作结果只有在该事务提交之后，另一个事务才可以读取到数据更新后的结果。读已提交也没有加锁，但却能够避免脏读，这是因为“快照读”机制。

   （3）可重复读（rr，幻读）

            可重复读使用的也是“快照读”，但不同的是，但事务启动时，就不允许“修改操作了”，而不可重复读恰恰是因为两次读取之间进行了数据的修改。

   （4）串行化

            这是数据库最高的隔离级别，这种级别下，事务串行化顺序执行，也就是一个个排队执行。

# 37. MySQL什么情况下会造成脏读、不可重复读、幻读？如何解决

    （1）脏读：有两个事务A和B，A读取已经被B修改但未提交的字段，此时B回滚，那么A读取的字段就是临时且无效的。可以提高隔离级别，改成读已提交。

    （2）不可重复度：有两个事务A和B，A读取了一个字段值，然后B更新并且提交了事务，A再重新读取这个字段，结果就和之前的不一样了。可以提高隔离级别，改成可重复读。

    （3）幻读：有两个事务A和B，A读取某个范围内的记录时，B又在该范围内插入了新的记录并提交，当A再次读取该范围的记录时，会产生幻行。可以升级隔离级别到串行化，或者使用MVCC+next-key锁机制实现。

# 38. MySQL在可重复读的隔离级别下会不会有幻读的情况，为什么？

    不会，InnoDB存储引擎默认隔离级别为RR，通过MVCC+next-key锁机制解决了幻读的问题。

     PS：其实严格来说，是存在幻读的。。。可以尝试一下这个操作，A开启事务，执行查询，此时B开启事务新增一条数据并提交，此时A再查询，发现没有幻读，但是如果A执行一个update操作，再查询，会发现出现了幻读。我认为应该是A在执行update操作的时候，新建了一条创建版本号为A事务版本号的记录，然后标记B事务创建的记录为待删除的，查询的版本号依据是删除版本号为空或大于当前版本号，并且创建版本号小于等于当前事务版本号，那么这里刚刚A更新的这条数据，显然也符合查询的条件，所以也会被查出来。

# 39. MySQL事务是如何实现的

     原子性：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。通过undo log实现的。每条数据变更都伴随一条undo log日志的生成，当系统发生错误或执行回滚根据undo log做逆向操作

     持久性：持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。通过redo log实现的。redo log记录了数据的修改日志。数据持久化到磁盘，先是储存到缓冲池里，然后缓冲池中的数据定期同步到磁盘中，如果系统宕机，可能会丢失数据，系统重启后会读取redo log恢复数据

     隔离性：mysql数据库通过MVCC + next-key机制实现了隔离性

     一致性：一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。以上3大特性，保障了事务的一致性

# 40. binlog和redo log的区别是什么，分别有什么用

* binlog是二进制文件，记录了对数据库执行更改的所有操作，不包括 select、show，因为这两个操作没有对数据本身做修改。但是若操作了数据，但是数据没有发生变化，也会记录到binlog。常用来数据恢复，数据备份。
* redo log又叫做重做日志文件，记录了事务的修改，不管事务是否提交都记录下来。在实例和介质失败时，InnoDB存储引擎会使用redo log恢复到之前的状态，保证数据的完整性

# 41. Innodb 的默认加锁方式是什么，是怎么实现的

Innodb默认加锁方式是行级锁

通过给索引上的索引项加锁来实现的

# 42.  冒泡、归并、插入是稳定排序

# 43. redis有哪些优缺点

优点

* 读写性能优异，Redis能读的速度是110000次/s，写的速度是81000次/s
* 支持数据持久化，支持AOF和RDB两种持久化方式
* 支持事务，Redis的所有操作都是原子性的，同时Redis还支持对几个操作合并后的原子性执行
* 数据结构丰富，除了支持string类型的value外还支持hash、set、zset、list等数据结构（键key只能是string类型）
* 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离

缺点

* 数据库容量受到物理内存的限制，不能用作海量数据的高性能读写，因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。
* Redis 不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。
* 主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。
* Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

# 44.  为什么要使用Redis/为什么要使用缓存

     主要从“高性能”和“高并发”这两点来看待这个问题。

     高性能：

     假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！

     高并发：

     直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

# 45. 为什么要用Redis而不用map/guava做缓存

     缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

     使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached服务的高可用，整个程序架构上较为复杂。

# 46. Redis为什么这么快

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。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)；

2、数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路 I/O 复用模型，非阻塞 IO；

5、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；

# 47.是什么是Redis持久化

     持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

# 48. Redis的持久化机制是什么，各自的优缺点

     Redis 提供两种持久化机制 RDB（默认） 和 AOF 机制

     RDB：是Redis DataBase缩写快照

     RDB是Redis默认的持久化方式。按照一定的时间将内存的数据以快照的形式保存到硬盘中，对应产生的数据文件为dump.rdb。通过配置文件中的save参数来定义快照的周期。

优点：

* 只有一个文件 dump.rdb，方便持久化。
* 容灾性好，一个文件可以保存到安全的磁盘。
* 性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能
* 相对于数据集大时，比 AOF 的启动效率更高。

缺点：

* 数据安全性低。RDB 是间隔一段时间进行持久化，如果持久化之间 redis 发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候)

AOF持久化(即Append Only File持久化)，则是将Redis执行的每次写命令记录到单独的日志文件中，当重启Redis会重新将持久化的日志中文件恢复数据。

当两种方式同时开启时，数据恢复Redis会优先选择AOF恢复。

优点：

* 数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次 命令操作就记录到 aof 文件中一次。
* 通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。
* AOF 机制的 rewrite 模式。AOF 文件没被 rewrite 之前（文件过大时会对命令 进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）)

缺点：

* AOF 文件比 RDB 文件大，且恢复速度慢。
* 数据集大的时候，比 rdb 启动效率低

两者优缺点是什么？

* AOF文件比RDB更新频率高，优先使用AOF还原数据。
* AOF比RDB更安全也更大
* RDB性能比AOF好
* 如果两个都配了优先加载AOF

# 49. 如何选择合适的持久化方式

* 一般来说， 如果想达到足以媲美PostgreSQL的数据安全性，你应该同时使用两种持久化功能。在这种情况下，当 Redis 重启的时候会优先载入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文件保存的数据集要完整。
* 如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。
* 有很多用户都只使用AOF持久化，但并不推荐这种方式，因为定时生成RDB快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比AOF恢复的速度要快，除此之外，使用RDB还可以避免AOF程序的bug。
* 如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。

# 50. Redis持久化数据和缓存怎么做扩容

* 如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容（通过给节点增加虚拟节点来解决数据倾斜问题）。
* 如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

# 51. Redis的过期键删除策略

     Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

过期策略通常有以下三种：

* 定时过期：每个设置过期时间的key都需要创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源去处理过期的数据，从而影响缓存的响应时间和吞吐量。
* 惰性过期：只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
* 定期过期：每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)
* Redis中同时使用了惰性过期和定期过期两种过期策略。

# 52. Redis key的过期时间和永久有效分别怎么设置？

     EXPIRE和PERSIST命令。

# 53. 我们知道通过expire来设置key 的过期时间，那么对过期的数据怎么处理呢?

     除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择[Redis缓存失效策略](http://www.originalxzx.top/categories/redisCache.html" \t "_blank)），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

* 定时去清理过期的缓存；
* 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，可以根据自己的应用场景来权衡。

# 54. MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

     redis内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。

# 55. Redis的内存淘汰策略有哪些

Redis的内存淘汰策略是指在Redis的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

全局的键空间选择性移除

* noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。
* allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。（这个是最常用的）
* allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。

设置过期时间的键空间选择性移除

* volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。
* volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。
* volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

总结

Redis的内存淘汰策略的选取并不会影响过期的key的处理。内存淘汰策略用于处理内存不足时的需要申请额外空间的数据；过期策略用于处理过期的缓存数据。

# 56. Redis的内存用完了会发生什么？

     如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以配置内存淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

# 57. Redis如何做内存优化？

     可以好好利用Hash,list,sorted set,set等集合类型数据，因为通常情况下很多小的Key-Value可以用更紧凑的方式存放到一起。尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key，而是应该把这个用户的所有信息存储到一张散列表里面

# 58. 什么是事务？

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

# 59. Redis事务的概念

Redis 事务的本质是通过MULTI、EXEC、WATCH等一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。

总结说：redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。Redis的事务总是具有ACID中的一致性和隔离性，其他特性是不支持的。当服务器运行在*AOF*持久化模式下，并且appendfsync选项的值为always时，事务也具有持久性。

Redis事务的三个阶段

事务开始 MULTI

命令入队

事务执行 EXEC

事务执行过程中，如果服务端收到有EXEC、DISCARD、WATCH、MULTI之外的请求，将会把请求放入队列中排队

Redis事务相关命令

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

redis 不支持回滚，“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。

如果在一个事务中的命令出现错误，那么所有的命令都不会执行；

如果在一个事务中出现运行错误，那么正确的命令会被执行。

* WATCH 命令是一个乐观锁，可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。
* MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。
* EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。 当操作被打断时，会返回空值 nil 。
* 通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。

UNWATCH命令可以取消watch对所有key的监控。

# 60. Redis事务支持隔离性吗

Redis 是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis 的事务是总是带有隔离性的。

# 61. Redis事务保证原子性吗，支持回滚吗

Redis中，单条命令是原子性执行的，但事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。

# 62. 哨兵模式

     哨兵的介绍

sentinel，哨兵，是Redis集群机构中非常重要的一个组件，主要由以下功能：

* 集群监控：负责监控 redis master 和 slave 进程是否正常工作。
* 消息通知：如果某个 redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员。
* 故障转移：如果 master node 挂掉了，会自动转移到 slave node 上。
* 配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址。

     哨兵用于实现 redis 集群的高可用，本身也是分布式的，作为一个哨兵集群去运行，互相协同工作。

故障转移时，判断一个 master node 是否宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题。

即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

哨兵的核心知识

* 哨兵至少需要 3 个实例，来保证自己的健壮性。
* 哨兵 + redis 主从的部署架构，是不保证数据零丢失的，只能保证 redis 集群的高可用性。
* 对于哨兵 + redis 主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

# 63. Redis 相比Memcached 有哪些优势？

* Memcached 所有的值均是简单的字符串， redis 作为其替代者， 支持更为丰富的数据类
* Redis 的速度比 Memcached 快很
* Redis 可以持久化其数据

# 64. Memcache 与Redis 的区别都有哪些？

* 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
* 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。
* 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。

# 65. redis是单进程单线程的？

    Redis是单进程单线程的，redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销。

# 66. Redis 常见性能问题和解决方案：

* Master 最好不要写内存快照，如果 Master 写内存快照，save 命令调度 rdbSave函数， 会阻塞主线程的工作， 当快照比较大时对性能影响是非常大的， 会间断性暂停服务
* 如果数据比较重要， 某个 Slave 开启 AOF 备份数据， 策略设置为每秒同步一
* 为了主从复制的速度和连接的稳定性， Master 和 Slave 最好在同一个局域网
* 尽量避免在压力很大的主库上增加从
* 主从复制不要用图状结构， 用单向链表结构更为稳定， 即：Master <- Slave1<- Slave2 <- Slave3… 这样的结构方便解决单点故障问题，实现 Slave 对 Master 的替换。如果 Master 挂了， 可以立刻启用 Slave1 做 Master， 其他不变。

# 67. 是否使用过 Redis 集群，集群的原理是什么？

* Redis Sentinal 着眼于高可用， 在 master 宕机时会自动将 slave 提升为master， 继续提供服务。
* Redis Cluster 着眼于扩展性， 在单个 redis 内存不足时， 使用 Cluster 进行分片存储。

# 68. 说说 Redis 哈希槽的概念？

     Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念， Redis 集群有16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽， 集群的每个节点负责一部分 hash 槽。

# 69. 如果有大量的 key 需要设置同一时间过期，一般需要注意什么？

     如果大量的 key 过期时间设置的过于集中，到过期的那个时间点，redis 可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值， 使得过期时间分散一些。

# 70. 使用过 Redis 分布式锁么，它是什么回事？

     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。

     如果在 setnx 之后执行 expire 之前进程意外 crash 或者要重启维护了， 那会怎么样？

     set 指令有非常复杂的参数， 这个可以同时把 setnx 和expire 合成一条指令来用的。

# 71. Redis 集群如何选择数据库？

     Redis 集群目前无法做数据库选择， 默认在 0 数据库。