计算机网络：

网络模型与相关协议：

OSI 七层模型:物理层，数据链路层，网络层，运输层，会话层，表示层， 应用层。

TCP/IP5 层模型：物理层，数据链路层，网络层，运输层，应用层。

TCP/IP4 层模型：网络接口层，网络层(IP)，运输层(TCP/UDP)，应用层(HTTP)。

常见应用层协议：DNS，HTTP, SMTP, POP3(接受文件)，FTP,Telent 远程终端协议。

传输层：TCP，UDP; 网络层：ICMP,IGMP，IP

可以这样做，在 OSI 模型中 ARP 协议属于链路层；而在 TCP/IP 模型中，ARP 协议属于网络层。

ARP 是 IP 地址解析为 MAC 地址 RARP 将 MAC 地址解析为 IP 地址

关于 TCP 协议建立和拥塞控制

一条 TCP 连接上发送数据速度的影响因素有哪些。（服务器到客户端之间带宽瓶颈、客户端接收能力限制、服务器网卡处理能力）

TCP、UDP 协议的区别

TCP 提供可靠的、面向连接的运输服务。在传输数据之前必须三次握手建立连接，数据传输结束之后，4 次挥手释放连接，而且在数据传递时，又有**确认应答、超时重传、滑动窗口、拥塞控制**等机制保证传送数据的可靠性。TCP 经常用于对网络通信质量有很高要求的地方，如**文件传输，邮件发送，远程登录**等场景。SMTP、TELNET、HTTP、FTP

UDP 在传送数据之前不需要建立连接，目的主机收到 UDP 报文后，不需要给出确认。UDP 不提供可靠交付，一般用于即时通信，如语音、视频、直播等。**RIP(路由选择协议)，DNS**

TCP 粘包：

TCP 粘包就是指发送方发送的若干包数据到达接收方时粘成了一包，从接收缓冲区来看，后一包数据的头紧接着前一包数据的尾。原因可能是发送方也可能是接收方造成的。

发送方原因：TCP 默认使用 Nagle 算法，将多次间隔较小、数据量较小的数据，合并成一个数据量大的数据块，然后进行封包。

接收方原因：TCP 将接收到的数据包保存在接收缓存里，然后应用程序主动从缓存读取收到的分组。这样一来，如果 TCP 接收数据包到缓存的速度大于应用程序从缓存中读取数据包的速度，多个包就会被缓存，应用程序就有可能读取到多个首尾相接粘到一起的包。

如果多个分组毫不相干，甚至是并列关系，那么这个时候就一定要处理粘包现象了。处理方法：发送方关闭 Nagle 算法。

接收方：接收方没有办法来处理粘包现象，只能将问题交给应用层来处理。应用层循环读取所有的数据，根据报文的长度判断每个包开始和结束的位置。

TCP 为了保证可靠传输并减少额外的开销（每次发包都要验证），采用了基于流的传输，基于流的传输不认为消息是一条一条的，是无保护消息边界的。而 UDP 则是面向消息传输的，是有保护消息边界的，接收方一次只接受一条独立的信息，所以不存在粘包问题。

数据报文的结构：应用程序+TCP/UDP 报文头部+IP 报文头部(到这是以太网帧，46-1500)+以太网头部；

TCP 报文头部结构：（前 20 字节固定）16 为源端口号+16 位目的端口号+32 位序号+32 位确认号+4 位头部长度（单位 4 字节）+6 位保留+6 个关键字（SYN,ACK，FIN）+16 位窗口大小（指接收窗口）+16 位校验和+16 位紧急指针+最多 40 字节的选项；

UDP 报文头部结构首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。：

IP 报文头部结构：

4 位版本+4 位首部长度（单位 4 字节）+8 位服务类型+16 位总长度（字节）

16 位标识（分组）+3 位标志（是否分组）+13 为片内偏移

8 位 TTL+8 位上层协议+16 位首部检验和
32 位源 IP 地址
32 位目的 IP 地址

以太网头部：
6 字节目的地址+6 字节源地址+2 字节类型。这个地址指 MAC 地址。

点对点和端对端的区别：
点到点通信是针对数据链路层或网络层来说的，因为数据链路层只负责直接相连的两个节点之间的通信，一个节点的数据链路层接受 ip 层数据并封装之后，就把数据帧从链路上发送到与其相邻的下一个节点。 点对点是基于 MAC 地址和或者 IP 地址，是指一个设备发数据给与该这边直接连接的其他设备，这台设备又在合适的时候将数据传递给与它相连的下一个设备，通过一台一台直接相连的设备把数据传递到接收端。

端到端通信是针对传输层来说的，传输层为网络中的主机提供端到端的通信。因为无论 tcp 还是 udp 协议，都要负责把上层交付的数据从发送端传输到接收端，不论其中间跨越多少节点。只不过 tcp 比较可靠而 udp 不可靠而已。所以称之为端到端，也就是从发送端到接收端。它是一个网络连接，指的是在数据传输之前，在发送端与接收端之间（忽略中间有多少设备）为数据的传输建立一条链路，链路建立以后，发送端就可以发送数据。

IP 数据报传输的过程：(无论怎么传，IP 源和目的地址不变，但 MAC 目的和源地址会变)
IP 数据报需从主机 A 上传送到主机 B 上，主机 A 首先查找路由表;
if(目的主机是与自己在同一个网段内)
主机 A 查询自己的 ARP 表;如果存在目的 IP 地址到 MAC 的映射，将 MAC 地址作为目的 MAC 地址封装成帧，发 给主机 B.如果没有，发送 ARP 请求广播给网段内的所有主机，来查询该目的 IP 地址的 MAC 地址
else if(发现了能与目的网络号相匹配的表目)
则把报文发给该路由表目指定的下一站的路由器或直接连接的网络接口;
报文发送到下一站时，数据帧的目的 MAC 地址是下一个站路由器或者网络接口的 MAC 地址，而 IP 头部的目的 IP 地址是主机 B 的 IP 地址;
else
寻找标为“默认”的表目，把报文发送给该表目指定的下一站路由器;

滑动窗口：
TCP 通过滑动窗口的概念来进行流量控制，抑制发送端发送数据的速率，以便接收端来得及接收。
窗口：对应一段发送者可以发送的字节序列。这个序列是可以改变的。接收端发给发送端自己的接受能力。然后发送端根据已确认接受的序列号和接受能力滑动窗口，一下子全部发送，等待接收端确认。
拥塞控制：
和流量控制的区别：
拥塞控制是防止过多的数据注入到网络中，可以使网络中的路由器或链路不致过载，是一个全局性的过程。
流量控制是点对点通信量的控制,主要就是抑制发送端发送数据的速率，以便接收端来得及接收。
(设置拥塞控制窗口 cwnd, 在发送数据时，将拥塞窗口的大小与接收端 ack 的窗口大小做比较，取较小者作为发送数据量的上限。)
慢开始: 设置**拥塞控制窗口 cwnd = 1**，没收到一个 ACK,cwnd++;每过 1RTT,cwnd = cwnd * 2;呈**指数增长**。
拥塞避免：当拥塞窗口 cwnd 达到一个阈值时（cwnd >= ssthresh），窗口大小不再呈指数上升，而是以线性上升，避免增长过快导致网络拥塞。**每当过了一个 RTT，cwnd = cwnd + 1；**

无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络**出现拥塞**(没有收到确认 ACK);拥塞窗口设置为 1，阈值为**拥塞时发送窗口的一半，执行慢开始算法**。

快重传：当发送方连续收到三个重复确认时，就立即重传对方尚未收到的报文段。并执行快恢复算法

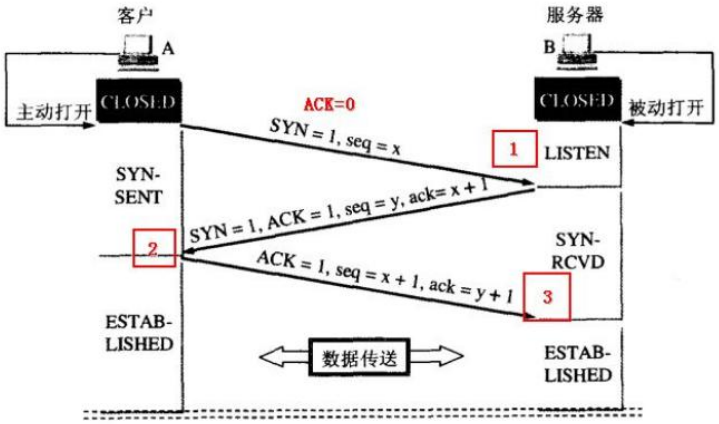
快恢复：将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。

TCP 三次握手：

1、客户端向服务器发送 TCP 连接请求数据包，其中同步标志为 SYN=1,ACK=0,初始序列号为 x;(syn_sent,等待回复)

2、服务器收到请求后，发回连接确认包。SYN=1，ACK=1,ack=x+1,初始序列号为 y;(syn_rcvd,收到请求，等待回复)

3、客户端收到确认报文后，还需发送确认数据包。ACK=1,ack=y+1,序列号为 x+1;(ESTABLISHED,完成三次握手)



用三次握手建立 TCP 连接

三次握手是为了建立可靠的通信信道，双方都确认自己和对方的发送和接受都是正常的。SYN=1 表示这是个连接请求或连接接受报文，不携带数据。不能变成两次，已失效的连接请求报文段突然又传送到了服务端，服务端误以为是正常的连接请求，直接发送连接确认报文，导致 TCP 建立，浪费资源，因而产生错误。

1、TCP 握手协议

第一次握手：建立连接时，客户端发送 syn 包(syn=j)到服务器，并进入 SYN_SEND 状态，等待服务器确认；

第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=j+1），同时自己也发送一个 SYN 包（syn=k），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK(ack=k+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

在上述过程中，还有一些重要的概念：

未连接队列：在三次握手协议中，服务器维护一个未连接队列，该队列为每个客户端的 SYN 包（syn=j）开设一个条目，该条目表明服务器已收到 SYN 包，并向客户发出确认，正在等待客户的确认包。这些条目所标识的连接在服务器处于 Syn_RECV 状态，当服务器收到客户的确认包时，删除该条目，服务器进入 ESTABLISHED 状态。

2、SYN 攻击原理

SYN 攻击属于 DOS 攻击的一种，它利用 TCP 协议缺陷，通过发送大量的半连接请求，耗费 CPU 和

内存资源。

配合 IP 欺骗，SYN 攻击能达到很好的效果，通常，客户端在短时间内伪造大量不存在的 IP 地址，向服务器不断地发送 syn 包，服务器回复确认包，并等待客户的确认，由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的 SYN 包将长时间占用未连接队列，正常的 SYN 请求被丢弃，目标系统运行缓慢，严重者引起网络堵塞甚至系统瘫痪。

第一种是缩短 SYN Timeout 时间，由于 SYN Flood 攻击的效果取决于服务器上保持的 SYN 半连接数，这个值=SYN 攻击的频度 x SYN Timeout，所以通过缩短从接收到 SYN 报文到确定这个报文无效并丢弃改连接的时间，例如设置为 20 秒以下（过低的 SYN Timeout 设置可能会影响客户的正常访问），可以成倍的降低服务器的负荷。

第二种方法是设置 SYN Cookie，就是给每一个请求连接的 IP 地址分配一个 Cookie，如果短时间内连续受到某个 IP 的重复 SYN 报文，就认定是受到了攻击，以后从这个 IP 地址来的包会被一概丢弃。

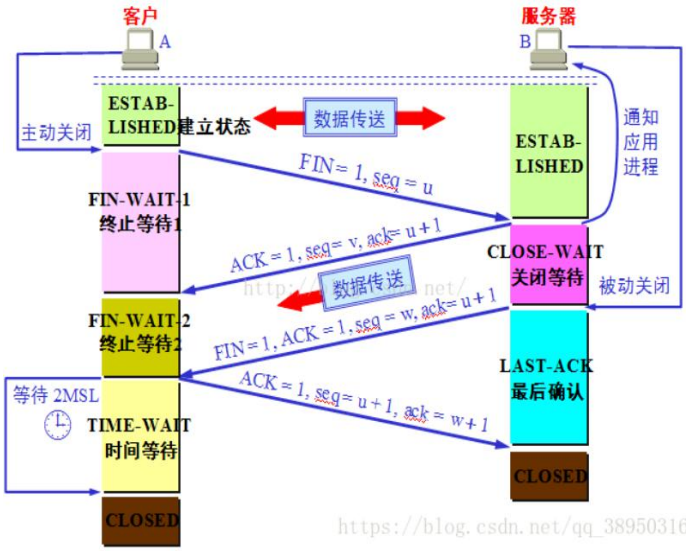
可是上述的两种方法只能对付比较原始的 SYN Flood 攻击，缩短 SYN Timeout 时间仅在对方攻击频度不高的情况下生效，SYN Cookie 更依赖于对方使用真实的 IP 地址，如果攻击者以数万/秒的速度发送 SYN 报文，同时利用 SOCK_RAW 随机改写 IP 报文中的源地址，以上的方法将毫无用武之地。

net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_max_syn_backlog = 8192
net.ipv4.tcp_synack_retries = 2
分别为启用 SYN Cookie、设置 SYN 最大队列长度以及设置 SYN+ACK 最大重试次数。

SYN Cookie 的作用是缓解服务器资源压力。启用之前，服务器在接到 SYN 数据包后，立即分配存储空间，并随机化一个数字作为 SYN 号发送 SYN+ACK 数据包。然后保存连接的状态信息等待客户端确认。启用 SYN Cookie 之后，服务器不再分配存储空间，而且通过基于时间种子的随机数算法设置一个 SYN 号，替代完全随机的 SYN 号。发送完 SYN+ACK 确认报文之后，清空资源不保存任何状态信息。直到服务器接到客户端的最终 ACK 包，通过 Cookie 检验算法鉴定是否与发出去的 SYN+ACK 报文序列号匹配，匹配则通过完成握手，失败则丢弃。
net.ipv4.tcp_synack_retries 是降低服务器 SYN+ACK 报文重试次数，尽快释放等待资源。
tcp_max_syn_backlog 则是使用服务器的内存资源，换取更大的等待队列长度，让攻击数据包不至于占满所有连接而导致正常用户无法完成握手。

SYN Flood 是当前最流行的 DoS（拒绝服务攻击）与 DDoS（分布式拒绝服务攻击）的方式之一，这是一种利用 TCP 协议缺陷，发送大量伪造的 TCP 连接请求，从而使得被攻击方资源耗尽（CPU 满负荷或内存不足）的攻击方式。

TCP 四次挥手：



由于 **TCP 连接是全双工的，因此每个方向都必须单独关闭**。当一方数据发送任务完成之后，可以发送 FIN 来终止这个方向的连接。而另一方可以继续发送数据。

1、客户端发送请求释放连接报文，FIN=1,seq=u;客户端进入 **FIN-WAIT-1** 状态；

2、服务器收到请求，发送确认报文。ACK=1,seq=v,ack=u+1;服务器进入 CLOSE-WAIT 状态。客户端收到后进入终止等待 2FIN-WAIT-2;

3、服务器发送完数据之后，向客户端发送请求释放连接报文，FIN=1,ACK=1,seq=w,ack=u+1;进入 LAST-ACK 状态

4、客户端收到请求后，发送确认报文，ACK=1,seq=u+1,ack=w+1。客户端进入 TIME-WAIT 状态，等待 2MSL 后进如 CLOSED 状态，服务器收到确认后进如 CLOSED 状态。

为什么关闭的时候是四次挥手，因为 TCP 连接时全双工的，每个方向都要发送关闭请求，而另一方向都要确认。

TIME_WAIT 状态为什么要等待 2MSL：因为第四次的确认报文可能丢失，这个状态是用来重发可能丢失的 ACK 报文。

为什么会有 CLOSE_WAIT,因为服务器可能有数据未发送完毕，这段时间是继续发送数据的。

如果建立连接之后出现故障：TCP 有个保活计时器，通常设置为 2 小时，两小时内没有收到客户端发送的数据，服务器发送探测报文，每 75s 发送一次，10 次之后探测报文没有反应，认为出现故障，关闭连接。

TIME_WAIT 存在的两个理由：

1 可靠的实现 TCP 全双工连接的终止

2 允许老的重复的分节在网络上的消逝（TCP 不允许处于 TIME_WAIT 状态的连接启动一个新的化身，因为 TIME_WAIT 状态持续 2MSL，就可以保证当成功建立一个 TCP 连接的时候，来自连接先前化身的重复分组已经在网络中消逝。）

netstat -an |grep "TIME_WAIT" 查看处于 Time_wait 状态的连接详细情况

netstat -ae|grep "TIME_WAIT" |wc -l 查看处于 Time_wait 状态的连接个数

在高并发短连接的 TCP 服务器上，当服务器处理完请求后立刻主动正常关闭连接。这个场景下会出现大量 socket 处于 TIME_WAIT 状态。如果客户端的并发量持续很高，此时部分客户端就会显示连接不上。

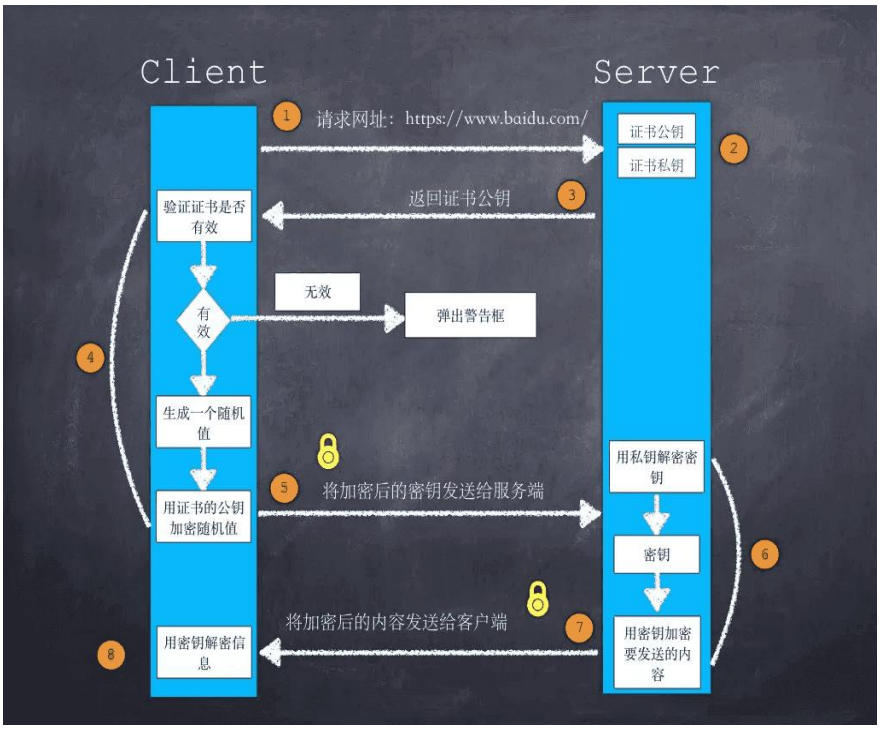
正常的 TCP 客户端连接在关闭后，会进入一个 TIME_WAIT 的状态，持续的时间一般在 1~4 分钟，短时间内（例如 1s 内）进行大量的短连接，则可能出现这样一种情况：客户端所在的操作系统的 socket 端口和句柄被用尽，系统无法再发起新的连接！

解决方法：
net.ipv4.tcp_syncookies = 1 表示开启 SYN Cookies。当出现 SYN 等待队列溢出时，启用 cookies 来处理，可防范少量 SYN 攻击，默认为 0，表示关闭；
net.ipv4.tcp_tw_reuse = 1 表示开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接，默认为 0，表示关闭；
net.ipv4.tcp_tw_recycle = 1 表示开启 TCP 连接中 TIME-WAIT sockets 的快速回收，默认为 0，表示关闭。
net.ipv4.tcp_fin_timeout 修改系默认的 TIMEOUT 时间
简单来说，就是打开系统的 TIMEWAIT 重用和快速回收。

浏览器输入 url 并回车的过程以及相关协议
1、根据域名查询域名的 IP 地址，DNS 解析。2、TCP 连接 3、发送 HTTP 请求 4、服务器处理请求并返回 HTTP 报文 5、浏览器解析渲染页面 6、连接结束。
使用的协议：DNS(获取域名的 IP 的地址);TCP(与服务器建立 TCP 连接)；IP(建立 TCP 协议时，需发送数据，在网络层用到 IP 协议)；OPSF(IP 数据包在路由之间传送，路由选择使用 OPSF 协议)；ARP(路由器与服务器通信时，将 IP 地址转化为 MAC 地址，使用 ARP 协议)HTTP(TCP 建立之后，使用 HTTP 协议访问网页)；
DNS 寻址：先查找浏览器缓存，如果没命中，查询系统缓存，即 hosts 文件。如果没命中，查询路由器缓存。如果没命中，请求本地域名服务器解析域名，没有命中就进入根服务器进行查询。没有命中就返回顶级域名服务器 IP 给本地 DNS 服务器。本地 DNS 服务器请求顶级域名服务器解析，没有命中就返回主域名服务器给本地 DNS 服务器。本地 DNS 服务器请求主域名服务器解析域名，将结果返回给本地域名服务器。本地域名服务器缓存结果并反馈给客户端。
HTTP1.0、1.1、2.0 之间的区别
HTTP1.1 与 1.0 之间的区别:1、**HTTP1.1 默认开启长连接，在一个 TCP 连接上可以传送多个 HTTP 请求和响应**。而 1.0 不支持长连接。客户端和服务器每进行一次 HTTP 操作，就建立一次连接。2、**缓存处理**：在 HTTP1.0 中主要使用 header 里的 If-Modified-Since,Expires 来做为缓存判断的标准，HTTP1.1 则引入了更多的缓存控制策略例如 **Entity tag，If-Unmodified-Since, If-Match, If-None-Match** 等更多可供选择的缓存头来控制缓存策略。3、Host 头处理：1.0 请求的 url 并没有传递主机名(服务器与 IP 地址绑定)，1.1 请求和响应都支持 Host 头域(虚拟主机共享 IP 地址)4、1.1 新增 24 个错误状态响应码。409：请求的资源和资源的当前状态冲突，410：服务器资源永久性删除。5、带宽优化以及网络连接的使用：**1.1 允许只请求资源的某个部分.**
HTTP2.0 和 HTTP1.X 相比的新特性
1、**新的二进制格式：1.x 的解析是基于文本的，而 2.0 的协议解析是采用二进制格式。2、多路复用**，即连接共享，即每一个 request 都是是用作连接共享机制的；。一个 request 对应一个 id，这样一个连接上可以有多个 request。**3、header 压缩。4、服务端推送。**
HTTP 与 HTTPS
1、HTTPS 协议需要到 CA 申请证书，一般免费证书很少，需要交费。
2、HTTP 协议运行在 TCP 之上，传输的内容都是明文。HTTPS 运行在 SSL/TLS(运行在 TCP 之上)之上，内容加密。
3、连接端口不一样，http 是 80，https 是 443.
4、http 连接简单，没有状态，https 是 ssl 加密的传输，身份认证的网络协议，更安全。
HTTPS 在传统的 HTTP 和 TCP 之间加了一层用于**加密解密的 SSL/TLS 层；采用 对称加密和非对称加密结合的方式来保护浏览器和服务端之间的通信安全。对称加密：加密和解密都是同一个密钥。非对称加密：密钥成对出现，分为公钥和私钥，公钥和私钥之间不能互相推导，公钥加密需要私钥解密，私钥加密需要公钥解密。**

浏览器使用 Https 的 URL 访问服务器，建立 SSL 链接；
1、发送非对称加密的公钥 A 给浏览器

2、客户端(SSL/TLS)解析证书（无效会弹出警告）
3、生成随机值(这个相当于传送数据的密钥)，作为对称加密的密钥 B。
4、浏览器使用服务器返回的公钥 A，对自己生成的对称加密密钥 B 进行加密，得到密钥 C。
5、浏览器将密钥 C 发送给服务器
6、务器使用自己的私钥 D 对接受的密钥 C 进行解密，得到对称加密密钥 B。
7、将信息和密钥 B 混合在一起进行对称加密
8、将加密的内容发送给客户端
9、客户端用密钥 B 解密信息

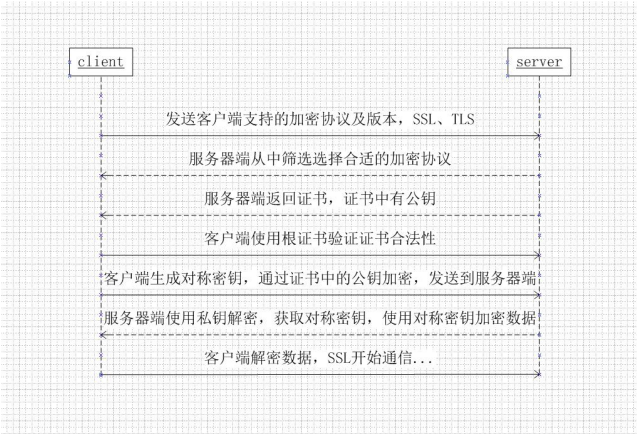


加密过程使用了对称加密和非对称加密。

对称加密：客户端和服务端采用相同的密钥经行加密
encrypt(明文，秘钥)＝ 密文
decrypt(密文，秘钥)＝ 明文
非对称加密：客户端通过公钥加密。服务端通过私钥解密
encrypt(明文，公钥)＝ 密文
decrypt(密文，私钥)＝ 明文

验证证书：
客户端获取到了站点证书，拿到了站点的公钥
客户端找到其站点证书颁发者的信息
站点证书的颁发者验证服务端站点是否可信

非对称加密算法：RSA，DSA/DSS
对称加密算法：AES，RC4，3DES
HASH 算法：MD5，SHA1，SHA256



HTTP 请求有哪些，get 和 post 的差别：

get:请求特定资源 2、post:向指定资源提交数据进行处理请求。Put:向指定资源上传最新内容。Delete:请求删除资源。

Get 是从指定资源请求数据，而 Post 是向指定资源提交要被处理的数据。Get 请求刷新无害，而 post 刷新，数据会被重复提交。Get 请求的数据会附加到 URL 中，多个参数用&连接，URL 编码采用 ASCII 编码。而 POST 请求会把请求的数据放到 body 中。因此 get 请求的数据会暴露在地址栏中，而 post 不会。又浏览器和服务器对 url 的长度有限制，所以 get 传输数据的大小受到 url 的限制。Get 请求的资源会被浏览器缓存。post 比 get 慢，因为 post 在发送数据之前会先将请求头发送给服务器进行确认，然后才真正发送数据。而 get 请求直接发送请求头和数据。

状态码：

304：客户端请求一个有缓存的资源，服务器返回 304 告诉客户端，自上次请求后，资源并没有更新，原本的缓存可以继续使用。

1xx：指示信息--表示请求已接收，继续处理

100 --客户端必须继续发出请求 101-客户端要求服务器转换 HTTP 协议版本。

2xx：成功--表示请求已被成功接收、理解、接受

200—OK 204--请求收到，但返回信息为空 206--服务器已经完成了部分用户的 GET 请求

3xx：重定向--信息不完整需要进一步补充

300 --- 请求资源在多处可得到。301—永久重定向，隐式重定向。302 临时重定向，显示重定向。

304—请求的资源没有改变，可以使用缓存。

4xx：客户端错误--请求有语法错误或请求无法实现

401---未授权 403—禁止访问。 404 –找不到。409：对当前资源状态，请求不能完成

5xx：服务器端错误--服务器未能实现合法的请求

500 内部服务器错误，501 未实现， 502 网关错误，503 服务不可用，504 网关超时。

HTTP 请求报文主要由请求行、请求头部、请求正文 3 部分组成

HTTP 响应报文主要由状态行、响应头部、响应正文 3 部分组成

转发和重定向的区别:

转发（forward）是服务器行为，重定向是客户端行为。

转发是服务器直接向目标地址访问 URL,将相应内容读取之后发给浏览器，地址栏 URL 不变，转发页面和转发到的页面可以共享 request 里面的数据。效率高，可用于用户登录之后将角色转发到相应的模块。

重定向是利用服务器返回的状态码来实现的，**如果服务器返回 301 或者 302，浏览器到新的网址重新请求资源**。地址栏 url 会发生改变，而且不能共享数据。效率低，可用于用户注销之后，跳转到其他网站。

Session 和 Cookie 的区别：

Cookie 技术是客户端的解决方案，Cookie 就是由服务器发给客户端的特殊信息，而这些信息以文本文件的方式存放在客户端， 然后客户端每次向服务器发送请求的时候都会带上这些特殊的信息。

客户端发送一个 http 请求到服务器端

服务器端发送一个 http 响应到客户端，其中包含 Set-Cookie 头部

客户端发送一个 http 请求到服务器端，其中包含 Cookie 头部

服务器端发送一个 http 响应到客户端

Cookie 的不可跨域名性

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。服务器一般把 Session 放在内存中。 每个用户都会有一个独立的 Session。Session 在用户第一次访问服务器的时候自动创建，创建 Session 的同时，服务器会为该 Session 生成唯一的 session id,session id 会以 cookie 的方式发送个客户端。客户端下次访问时，带上这个 session id，就可以跟踪会话了。如果浏览器不支持 cookie，可以用 url 重写的方式，将 sessionId 写入 url 传给服务器。

操作系统：

进程：是并发执行的**程序在执行过程中分配和管理资源的基本单位，是一个动态概念，竞争计算机系统资源的基本单位。**

线程：是进程的一个执行单元，是进程内部的调度实体。比进程更小的独立运行的基本单位。线程也被称为轻量级进程。

一个程序至少一个进程，一个进程至少一个线程。

进程是程序运行的实例。运行一个 Java 程序的实质就是启动一个 java 虚拟机进程。进程是程序向操作系统申请资源的基本单位。线程是进程中可独立执行的最小单位。一个进程可以包括多个线程，同一个进程中的所有线程共享该进程的资源。

竞态：多线程编程中一个问题，对于同样的输入，程序的输出有时是正确的，有时是错误的，这种计算结果的正确性与时间有关的现象被成为竞态。

线程安全和非线程安全： 一个类在单线程环境下能够正常运行，并且在多线程环境下，使用方不做特别处理也能运行正常，我们就称其实线程安全的。反之，一个类在单线程环境下运行正常，而在多线程环境下无法正常运行，这个类就是非线程安全的。

多线程编程的实质就是将任务的处理方式由串行改为并发。

进程是资源分配最小单位，线程是程序执行的最小单位；进程有自己独立的地址空间，每启动一个进程，系统都会为其分配地址空间。线程是进程的一个实体，一个进程至少有一个线程,同一个进程的所有线程，共享所属进程的资源。

线程占用的资源比线程少很多，所以创建线程和切换线程的开销相对来说很小。但多进程程序更安全，生命力更强。一个进程的死亡不会对其他进程造成影响。而一个线程死掉，（可能会锁住资源）造成整个进程都死掉了。

多进程和多线程的对比：

多进程，**每个进程拥有自己的数据，互不干涉**，共享复杂需要 IPC，同步简单。占用内存多，切换复杂，CPU 利用率低，创建销毁、切换复杂，速度慢， 进程间不会相互影响。

多线程，**共享所属进程的数据**，共享简单，**同步比较难。占用内存少，切换简单，CPU 利用率高，创建销毁、切换简单**，速度快，一个线程挂掉将导致整个进程挂掉。

协程：

协程是一种用户态的轻量级线程，**协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。**

协程调度切换时，**将寄存器上下文和栈保存到其他地方。在切回来的时候，恢复先前保存的寄存器上下文和栈，**直接操作栈则**基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。进程线程都是同步机制，而协程则是异步。协程不需要多线程的锁机制。**

进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。
进程间通信方式：管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket、Streams等
管道:无名管道，半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。只能用于父子进程和兄弟进程。
FIFO，也称为命名管道，是一种文件类型。FIFO 的通信方式类似于在进程中使用文件来传输数据，只不过 **FIFO 类型文件同时具有管道的特性。在数据读出时，FIFO 管道中同时清除数据，并且"先进先出"。**可用于任何进程。
消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列 ID）来标识。消息队列独立于发送与接收进程。消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。
信号量（semaphore）与已经介绍过的 IPC 结构不同，**它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。**信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。
共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。因为多个进程可以同时操作，所以需要进行同步。信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

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**，而且只能在具有血缘关系的进程间使用。
进程的血缘关系通常指父子进程关系。**管道分为 pipe（无名管道）和fifo（命名管道）两种**，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通信。
管道是一个**固定大小的缓冲区。在 Linux 中，该缓冲区的大小为 1 页**，即 4K 字节，使得它的大小不象**文件那样不加检验地增长。从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃.**
2. 信号量（semophore）：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。它通常作为一种锁
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。
3. **消息队列（message queue）：消息队列是由消息组成的链表，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。消息队列与管道通信相比，其优势是对每个消息指定特定的消息类型，接收的时候不需要按照队列次序，而是可以根据自定义条件
接收特定类型的消息。
4 信号（signal）：信号是一种比较复杂的通信方式，用于通知接收进程某一事件已经发生。
5. 共享内存（shared memory）：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的通信方式运

行效率低而专
门设计的。它往往与其他通信机制，如信号量配合使用，来实现进程间的同步和通信。
6. **套接字（socke. t）：socket，即套接字是一种通信机制，凭借这种机制，客户/服务器（即要进行通信的进程）**
系统的开发工作既可以在本地单机上进行，也可以跨网络进行。也就是说它可以让不在同一台计算机但通过网
络连接计算机上的进程进行通信。也因为这样，套接字明确地将客户端和服务器区分开来。

用户态与核心态：

防止用户进程访问对操作系统的稳定运行造成破坏。对一些资源的访问进行了等级划分。**内核态和用户态是操作系统的两种运行级别，内核态权限高，用户态权限低。**

操作系统的很多操作会消耗系统的物理资源，例如创建一个新进程时，要做很多底层的细致工作，如分配物理内存，从父进程拷贝相关信息，拷贝设置页目录、页表等，这些操作显然不能随便让任何程序都可以做，**于是就产生了特权级别的概念，与系统相关的一些特别关键性的操作必须由高级别的程序来完成，这样可以做到集中管理，减少有限资源的访问和使用冲突**

当一个进程在执行用户自己的代码时处于用户运行态（用户态），此时特权级最低，为 3 级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态。Ring3 状态不能访问 Ring0 的内核**地址空间，**包括代码和数据；**当一个进程因为系统调用陷入内核代码中执行时处于内核运行态（内核态），此时特权级最高，为 0 级。执行的内核代码会使用当前进程的内核栈，每个进程都有自己的内核栈。**

用户态和内核态的切换：

进程大部分时间是运行在用户态下的，在其需要操作系统帮助完成一些用户态自己没有特权和能力完成的操作时就会切换到内核态。切换到内核的方式有：系统调用、发生异常、外围设备的中断。

进程空间：

内核态内存空间、用户态的堆栈（一般 8M，从高地址向低地址增长）、**数据段、进程代码段、**
线程共享的有：进程代码段、进程共有数据、文件描述符、信号处理器、进程当前目录、进程用户 ID、进程组 ID.
线程私有的：线程 ID、寄存器的值、线程的栈、线程优先级、错误返回码、线程信号屏蔽码。

线程上下文切换开销：上下文切换的开销包括**直接开销和间接开销**。直接开销有如下几点：

操作系统**保存恢复上下文(CPU 寄存器值，程序计数器值等)所需的开销**

线程调度器**调度线程的开销**

间接开销有如下几点：

处理器**高速缓存重新加载**的开销

上下文切换可能导致整个一级高速缓存中的内容被冲刷，即被写入到下一级高速缓存或主存

线程间的几种通信方式知道不？

1、**锁机制**

互斥锁：提供了以排它方式阻止数据结构被并发修改的方法。

读写锁：允许多个线程同时读共享数据，而对写操作互斥。

条件变量：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件测试是在互斥锁的保护下进行

的。条件变量始终与互斥锁一起使用。

2、**信号量机制：**包括无名线程信号量与有名线程信号量

3、**信号机制：**类似于进程间的信号处理。

线程间通信的主要目的是用于线程同步，所以线程没有象进程通信中用于数据交换的通信机制。

死锁条件：互斥、不可剥夺、循环等待、请求与保持。

分页分段：

传统存储管理方式：作业必须一次全部加载到内存中，方可运行。当作业很大，就无法运行。而且多道作业运行时，内存不足容纳所有作业，导致多道程序性能下降。操作系统引入了虚拟内存的概念，利用计算机的空间局部性和时间局部性原理，将程序分的一部分装入内存运行，其余部分留在外存，等需要的时候再讲外村的程序装入内存继续运行。虚拟内存好像给用户提供了一个比实际内存大得多的存储器。叫，虚拟存储器，大小由计算机地址结构决定。

虚拟内存实现方式：请求分页，请求分段，请求段页式存储管理。请求分页存储管理中，将虚拟地址内存空间划分位大小相等的页块，同时内存地址空间，也划分位等大小的页块。系统维持一个页表，存储这虚拟页号到物理快块号的映射。程序中的逻辑地址由两部分组成：页号 P 和页内位移量 W。相邻的页面在内存中不一定相邻，即分配给程序的内存块之间不一定连续。逻辑地址转化为物理地址时，根据页表将页号转化为块号，块号*块大小加上页内偏移得到物理地址。
如果程序执行时，调用到不再内存中的虚拟页面时，发生缺页中断，将页由外村调入内存。如果内存已满，采用页面置换算法将老的淘汰，载入新的。页面置换算法常见的有 FIFO,LRU。

优点：没有外碎片，每个内碎片不超过页的大小。
缺点：程序全部装入内存，要求有相应的硬件支持，如地址变换机构缺页中断的产生和选择淘汰页面等都要求有相应的硬件支持。增加了机器成本和系统开销。

请求分段存储管理：将用户程序地址空间分成若干个大小不等的段，每段能够定义一组相对完整的逻辑信息。存储分配时，以段为单位，段内地址连续，段间不连续。虚拟地址由段号和段内地址组成，虚拟地址到实存地址的变换通过段表来实现。 分页对程序猿而言是不可见的。而分段通常对程序猿而言是可见的，因而分段为组织程序和数据提供了方便。段页式存储组织是分段式和分页式结合的存储组织方法。这样可充分利用分段管理和分页管理的长处。

优点：可以分别编写和编译，可以针对不同类型的段采取不同的保护，可以按段为单位来进行共享，包括通过动态链接进行代码共享。
缺点：会产生碎片。

用分段方法来分配和管理虚拟存储器。程序的地址空间按逻辑单位分成基本独立的段，而每一段有自己的段名，再把每段分成固定大小的若干页。用分页方法来分配和管理实存。
优点：段页式管理是段式管理和页式管理相结合而成，具有两者的优点。
缺点：由于管理软件的增加，复杂性和开销也增加。另外需要的硬件以及占用的内存也有所增加，使得执行速度下降。

操作系统虚拟内存换页的过程

在进程开始运行之前，不是全部装入页面，而是装入一个或者零个页面，之后根据进程运行的需要，动态装入其他页面；当内存已满，而又需要装入新的页面时，则根据某种算法淘汰某个页面，以便装入新的页面。

在使用虚拟页式存储管理时需要在页表中增加一些内容：
页号、驻留位（中断位）、内存块号、外存地址、访问号、修改位
驻留位：表示该页在外存还是内存；
访问位：表示该页在内存期间是否被访问过，又称 R 位；
修改位：表示该页在内存中是否被修改过，又称 M 位；

缺页本身是一种中断，与一般的中断一样，需要经过 4 个处理步骤：
1、保护 CPU 现场
2、分析中断原因
3、转入缺页中断处理程序进行处理
4、恢复 CPU 现场，继续执行

虚拟内存换页的算法有哪些？
最优页面置换算法、先进先出页面置换算法（FIFO）及其改进、最近最少使用页面置换算法（LRU）、时钟页面置换算法（clock）、最近未使用页面置换算法（NRU）；

换页算法里面，FIFO 有什么缺点？怎么改进？
先进先出实现简单，但并没有考虑局部性原理，最近访问过的数据不久之后很可能会再次被访问。性能可能会很差。还会发生 Belady 异常现象， 使用 FIFO 算法时，四个页框时缺页次数比三个页框时多。这种奇怪的情况称为 Belady 异常现象。
称为第二次机会算法。即给每个页面增加一个 R 位，表示最近访问过，每次先从链表头开始查找，如果 R 置 1 位，清除 R 位并且把该页面节点放到链表结尾；如果 R 是 0，那么就是又老又没用到，替换掉。

进程调度算法：
先来先服务调度算法、短作业(进程)优先调度算法、优先权调度算法的类型、高响应比优先调度算法、时间片轮转法、多级反馈队列调度算法、

线程与协程的区别：
进程的内存空间是怎么样的？
堆快还是栈快？
JVM 对堆栈只进行两种操作:以帧为单位的压栈和出栈操作。堆是应用程序在运行的时候请求操作系统分配给自己内存，由于从操作系统管理的内存分配,所以在分配和销毁时都要占用时间，因此用堆的效率非常低。
Java 的同步机制有哪些：几种锁把。
Java 中如果我有两个 long 数组，想要取他们的交集
Java 有没有遇到过 fullGC，触发 fullGC 的条件以及如何排查
网页跳转怎么实现，结合具体的场景 前端
301 和 302 什么区别？结合具体场景
301 和 302 都是重定向状态码。说浏览器在拿到服务器返回的这个状态码后会自动跳转到一个新的 URL 地址，这个地址可以从响应的 Location 首部中获取。301 永久重定向，表示旧地址的资源永久性移除了，搜索引擎在抓取新内容的同时也将旧的网址交换为重定向之后的网址。302 临时重定向。表示旧地址 A 的资源还在（仍然可以访问），这个重定向只是临时地从旧地址 A 跳转到地址 B，搜索引擎会抓取新的内容而保存旧的网址。
网页目录结构调整，网页扩展名调整，网页被移动到一个新的地址，这些都要永久重定向 301.如果网页调整是临时的，之后会调整过来，就用临时重定向 302；
504 什么意思？
网关超时。
怎么在前端实现网页跳转，用什么函数？
onclick 跳转
设置 window 的 location.href 属性
onclick="window.location.href='URL'"
onclick="location='URL'"
调用 window 的 open 方法

onclick="window.open('URL','_blank');" // 在新窗口打开
onclick="window.open('URL','_self');" // 覆盖当前页

a 标签跳转

Preface // 在新窗口打开
Preface // 覆盖当前页，target 属性默认为_self，此处可省略

```
req.getRequestDispatcher("ok.jsp").forward(req,resp)
```

```
resp.sendRedirect("ok.jsp");//URL 发生变化
```

任务处理时间 100ms，服务器 4 核 8G 如何设计线程池达到 1000qps？任务是 90ms 在 IO，10ms 在计算的情况下怎么弄？全在计算呢？
10G 数据，1G 内存，如何快速找到重复出现的数据？
10G 数据，1G 内存，如何快速找到重复出现次数最多的数据？
如何检测跳转页面的登录状态 session,cookie
5.java 设计模式， jdk 里用到了哪些设计模式。
6.NIO 讲一讲。
不用 final 还可以用什么办法使得这个类不被继承、
java 初始化的顺序
自旋锁 是公平吗？
自旋锁 怎么才能公平。
客户抱怨你们网站太慢，怎么排查问题？
SDS 优点，链表、跳表的实现与复杂度
内核态和用户态的切换
select poll epoll 三连
虚拟内存的作用
CPU 二级缓存
如何快速复用处于 TIME_WAIT 的连接
三种 I/O 多路复用
一个 TCP 报文从本机到对方主机的过程
路由表的结构是什么样的
海量数据(100 亿)找最大的 100w 个数，时间复杂度 nlogm，n 时 100 亿，m 是 100w
衍生出堆排序过程讲一下
redis 里几个常用命令？
linux 常用命令？
解释 XSS 和 CSRF，它们之间有什么区别，以及如何防范。
交换机和路由器区别
17、linux 读文件的过程
18、ping 127.0.0.1 经过哪些层
19、设置 mtu 有什么用
Linux 中能否删除一个正在运行的文件
Linux 常用的命令，如何排查 bug

集合框架

ArrayList

底层实现是 Object 数组（transient Object[] elementData）；
默认长度是 10，扩容是变成 1.5 倍：int newCapacity = oldCapacity + (oldCapacity >> 1);
扩容是创建新的数组，elementData = Arrays.copyOf(elementData, newCapacity)

增减删除，会造成数组元素的移动，使用 System.arraycopy();
支持随机访问;

LinkedList

LinkedList 实际上是通过双向链表去实现的,内部类 Node(val,next,pre)是链表的节点;
它也可以被当作堆栈、队列或双端队列进行操作。
不支持随机访问，但删除、插入元素时间不受元素位置影响，近似 O(1),而 ArryList 是 O(n);

Vector

底层实现也是 Object 数组，线程安全,支持随机访问
默认长度是 10， capacityIncrement 增长系数，如果设置，每次扩展都是增加这个数，否则扩容为 2 倍。

(01) 对于需要快速插入，删除元素，应该使用 LinkedList。
(02) 对于需要快速随机访问元素，应该使用 ArrayList。
(03) 对于“单线程环境” 或者 “多线程环境，但 List 仅仅只会被单个线程操作”，此时应该使用非同步的类(如 ArrayList)。
对于“多线程环境，且 List 可能同时被多个线程操作”，此时，应该使用同步的类(如 Vector)。
Collections.synchronizedList(new ArrayList<Map<String,Object>>());

快速失败（fail—fast）

在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出 Concurrent Modification Exception;
迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变 modCount 的值。**每当迭代器使用 hashNext()/next()遍历下一个元素之前，都会检测 modCount 变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常，终止遍历。**

当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。这个机制是防止多线程并发访问 list 可能带来错误，所以抛出异常提醒一下。
Fail-Fast : List 维护一个 modCount 变量，add、remove、clear 等涉及了改变 list 元素的个数的方法都会导致 modCount 的改变。如果判断出 modCount != expectedModCount，抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。
并发安全的解决方案：
1、Collections.synchronizedList。或者每个访问操作 sychronized 加锁。
2 CopyOnWriteArrayList 来替换 ArrayList。推荐使用该方案。
CopyOnWriterArrayList 所代表的核心概念就是：**任何对 array 在结构上有所改变的操作**（add、remove、clear 等），CopyOnWriterArrayList 都会 copy 现有的数据，再在 copy 的数据上修改，这样就不会影响 COWIterator 中的数据了，修改完成之后改变原有数据的引用即可。同时这样造成的代价就是产生大量的对象，同时数组的 copy 也是相当有损耗的。

安全失败（fail—safe）

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。
java.util.concurrent 包下的容器都是安全失败，可以在多线程下并发使用，并发修改。
fail-safe 机制有两个问题
（1）需要复制集合，产生大量的无效对象，开销大
（2）无法保证读取的数据是目前原始数据结构中的数据。

HashMap

HashMap 底层是数组和链表结合在一起，也就是链表散列； 怎么更通顺的说出来？？？

hashMap 底层是数组+链表+红黑树。首先是 table 数组，每个数组存放的是链表，链表的每一个节点都是<Key,Value>型的 Node 节点。

JDK1.8 之后，当链表长度超过阈值，默认为 8 时，为加快检索速度，将链表转化成红黑树。

默认加载因子是 0.75, 默认的初始容量是 16，容量就是数组长度，如果 size 大于容量*加载因子，就进行 rehash;

容量扩容成两倍;

如果初始化时指定初始容量，要向上取到 2 的 N 次方。因为对 key 的 hashCode 进行扰动函数处理之后，是根据（n-1）& hash 判断元素在数组中的位置的。

使用(n-1) & hash 是为了效率考虑，而只有数组长度为 2 的 N 次方，才能使 n-1 & hash 起到取余操作的作用。

hash(key) ----> return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);-----key 的扰动函数;减少碰撞;

(n-1) & hash---判断元素在数组中的位置。数组中存放的是 Node<k,v>节点。

查询 get(key)时，先判断在数组中的位置，key 的 hashCode 无符号右移 16，再异或，取余;然后在链表中搜索;

put,也是先查询在数组中的问题，然后判断链表中，key 是否存在，存在就更新， 不存在头插法，插入链表;

红黑树的平均查找长度是 log(n)，长度为 8，查找长度为 log(8)=3，链表的平均查找长度为 n/2，当长度为 8 时，平均查找长度为 8/2=4，这才有转换成树的必要；链表长度如果是小于等于 6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。中间有个差值 7 可以防止链表和树之间频繁的转换。

而且理想情况下，**在随机哈希代码下，桶中的节点频率遵循泊松分布**，文中给出了桶长度 k 的频率表。

由频率表可以看出，**桶的长度超过 8 的概率非常非常小（千万分之一）。所以作者应该是根据概率统计而选择了 8 作为阀值。**

TreeMap 是基于红黑树实现的一个保证有序性的 Map 基于红黑树，所以 TreeMap 的时间复杂度是 O(log n)，

HashMap 冲突解决方法：开发地址法，当冲突发生时，使用某种探查(亦称探测)技术（线性探测，随机探测，不同的关键字使用不同的探测距离）在散列表中形成一个探查(测)序列。沿此序列逐个单元地查找，直到找到给定的关键字，或者找到空。

2、拉链法

多线程 PUT 可能发生的问题：

1、当多个线程同时执行 addEntry(hash,key ,value,i)时，如果产生哈希碰撞，导致两个线程得到同样的 bucketIndex 去存储，就可能会发生元素覆盖丢失的情况

2、JDK1.7 以前。在向 HashMap put 元素时，会检查 HashMap 的容量是否足够，如果不足，则会新建一个比原来容量大两倍的 Hash 表，然后把数组从老的 Hash 表中迁移到新的 Hash 表中，迁移的过程就是一个 rehash()的过程，多个线程同时操作就有可能会形成循环链表。

扩容方式：遍历数组，然后遍历数组链表，从链表头到尾，保留 next = e.next;先计算出节点在新 hashmp 的数组位置 i，然后用**头插法将节点插入到新数组的头结点**（e.next = new B）。e = next; do while(e != null)之后的一样。

单线程是没问题的，如果是多线程的话，那么可能 next = e.next；被干扰的话 next = B;造成死循环。影响扩容。

HashMap 扩容导致死循环的主要原因在于扩容后链表中的节点在新的 hash 桶使用头插法插入。

JDK1.8 扩容时，将原节点用尾插法将节点插入到新的 hashmap 中。

当然，他们都不是线程安全的，线程安全还是得用 ConcurrentHashMap;

红黑树的特性

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL 或 NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。（确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。）

它的时间复杂度是 O(lgn)，一棵含有 n 个节点的红黑树的高度至多为 2log(n+1).

HashTable 与 hashMap 的不同

HashTable 线程安全，方法都是 sychronized 修饰;hashMap 线程不安全；

HashTable,Key 和 Value 都不能为 null;而 hashMap 可以有一个 null 可以，value 是否为 null 无所谓;

HashTable 初始容量 11，扩展 2*n+1,HashMap,初始 16，扩容加倍；初始容量向上取到 2 的 N 次方;

HashMap 当链表长度大于阈值时，转化成红黑树，而 HashTable 没有。

ConcurrentHashMap 的底层实现!!!!!!!!!!!!!!!

HashMap 多线程情况下，可能造成死循环，当扩容时，头插法+没有同步控制，很可能造成循环链表，造成死循环

JDK1.8 声明两对指针，维护两个链表，依次在末端添加新的元素,解决了死循环，但依然存在很多问题，例如数据丢失：

HashTable 是线程安全的，使用 synchronized 来保证线程安全，锁住整个 hash 表，并发效率低下;

JUC 中的 ConcurrentHashMap,1.5-1.7 使用分段锁机制，将整个数组分割分段，每一把锁只锁定 segment,

将整个 Hash 表切分多个 Segment；而每个 Segment 元素，类似于一个 Hashtable;

Segment 的数量由所谓的 concurrentcyLevel 决定， 默认是 16； 和 HashMap 的初始容量一致，也可以在相应构造函数直接指定。 同样是 2 的幂数值;

当一个线程占用锁访问其中一个 segment 的时候，其他 segment 也能被其他线程访问

ConcurrentHashMap 定位一个元素的过程需要进行两次 Hash 操作，第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表的头部

JDK1.8 之后，抛弃了 segment 的概念，直接用 Node 数组+链表+红黑树的数据结构实现，并发控制采用 CAS+synchronize;

put 的时候，判断是否 kv 为 null, hash 定位桶位置之后，如果为空，cas 写入，否则 synchronize 加锁写入(更新或插入)如果链表长度达到 8，将链表转化成红黑树；

get 的时候，没有并发控制；在链表查询的时候，会判断是红黑树还是链表，两种查询方式；

synchronize 只锁定当前链表或红黑二叉树的首节点。

① 判断存储的 key、value 是否为空，若为空，则抛出异常，否则，进入步骤②

② 计算 key 的 hash 值，随后进入无限循环，该无限循环可以确保成功插入数据，若 table 表为空或者长度为 0，则初始化 table 表，否则，进入步骤③

③ 根据 key 的 hash 值取出 table 表中的结点元素，若取出的结点为空（该桶为空），则使用 CAS 将 key、value、hash 值生成的结点放入桶中。否则，进入步骤④

④ 若该结点的的 hash 值为 MOVED，则对该桶中的结点进行转移，否则，进入步骤⑤

⑤ 对桶中的第一个结点（即 table 表中的结点）进行加锁，对该桶进行遍历，桶中的结点的 hash 值与 key 值与给定的 hash 值和 key 值相等，则根据标识选择是否进行更新操作（用给定的 value 值替换该结点的 value 值），若遍历完桶仍没有找到 hash 值与 key 值和指定的 hash 值与 key 值相等的结点，则直接新生一个结点并赋值为之前最后一个结点的下一个结点。进入步骤⑥

⑥ 若 binCount 值达到红黑树转化的阈值，则将桶中的结构转化为红黑树存储，最后，增加 binCount 的值。

JVM 内存

线程私有：PC 程序计数器，虚拟机栈，本地方法栈　　线程共享：堆，方法区，直接内存

PC：下一条执行字节码指令的地址，唯一不会发生 OutOfMemory 的地方；

虚拟机栈：线程私有，**生命周期和线程一致**；存在**局部变量表（基本数据类型和引用类型变量），操作数栈，动态链接，方法出口信息。**

两种异常：当栈的深度超过最大申请申请深度时，StackOverFlowError;内存不够 Java 虚拟机栈动态扩展时，OutOfMemory;

栈中存放的是栈帧，一个函+数一个栈帧;

本地方法栈，存放本地方法的栈，其他和虚拟机栈一样。

堆：唯一作用：存储对象内存；几乎所有的对象和数组都在这分配内存；是垃圾回收的主要地方；为更好的回收内存和分配内存，将对堆进行分代，分为新生代和老年代，不同的代采用不同的垃圾回收算法；老年代基本采用标记整理，只有 cms 采用标记清除。

新生采用复制算法，为提高内存利用率，将新生代分为伊甸区，s0,[səˈvaɪvə(r)] s1，比例为 8:1:1。每次留 1 块 s 区作为复制的备份内存，同时将老年代作为分配担保区。

方法区：存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。线程共享。JDK1.8 被元空间取代，元空间位于直接内存；

替换的一个原因是，方法区有 JVM 设置固定大小上限，而元空间直接使用内容，只受限于直接内存，不会发生 OutOfMemory;

常量池，1.8 之前，放在方法区，大小受限于方法区。**而 1.8 存放堆中**。主要有**字面量(字符串基本数据类型，final 常量值)和符号引用.**

直接内存，不是运行时数据区的一部分，也不是虚拟机规范定义的内存，也可能 OutOfMemory;

堆内存分配策略：1、对象优先在新生代伊甸区分配内存。2、大对象直接进去老年代(长字符串和数组)3、长期存活的对象进入老年代。一次 Minor[ˈmaɪnə(r)]GC，年龄加一，默认年龄 15 进入老年代。4、动态年龄判定。相同年龄所有对象大小的总和大于 Survivor 空间的一半，大于等于该年龄的对象进入老年代。

新生代 GC（Minor GC）,发生在新生代的垃圾回收动作，频繁且快。

老年代 GC（Major GC/Full GC，发生在老年代的垃圾回收，通常伴随着至少一次的 minor gc。速度慢。

Full GC 触发：1、老年代空间不足 2、方法区空间不足。3、调用 System.gc(),建议 JVM 进行 full gc. 4、没有足够的连续空间分配给大对象。5、长期存活的对象转入老年代，空间不足。6、新生代垃圾回收存活的对象太多，S1 放不下，老年代担保空间不足。

判断对象是否存活？

1 引用计算法，给对象添加一个引用计数器，每次被引用，计算器加一，引用失效，计算减一。当引用数为 0 时，表示对象不存活。但无法解决循环引用问题。

2、可达性分析法。以 **GC Roots** 对象为起点，向下搜索，节点所走的路径成为引用链。当一个对象和引用链没有相连时，表示这个对象不可达。主要方法。GC Roots: 虚拟机栈引用的对象，本地方法栈引用的对象，方法区静态属性引用的对象，方法区常量引用的对象。

对象的存活都和引用有关，引用类型分为强引用，软引用，弱引用，虚引用。

强引用，new 的对象。垃圾回收器绝不会回收它。软引用，空间不足，回收这些对象的内存。弱引用，只要发现，马上回收。虚引用，形同虚设，任何时候都可能被回收

很少用虚和弱，软引用很多，可以加快 jvm 对垃圾内存的回收速度。

回收对象的两次标记过程：可达性分析的不可达对象，并不马上回收。真正死亡至少经过两次标记：对可达性分析不可达的对象进行第一次标记，并进行筛选。筛选条件是重写 finalize 方法且没有调用的对象，将其放入队列中，进行第二次标记。在执行 finalize 方法时，该对象依然没有被引用，才会被真正回收掉。finalize 方法只能被调用一次。

判断常量是否是**废弃常量**？ 运行时常量池是主要的回收对象。常量不被引用，就可以被回收。

判断一个类是否是**无用的类**？ 1，类的实例对象全被回收。2.加载类的 ClassLoader 被回收。3，堆中的 Class 对象(在运行时期提供或者获得某个对象的类型信息)没有被引用。

垃圾回收算法：标记-清除：先标记需清除的对象，统一回收----效率不高，会产生大量不连续的碎片。

复制算法：将内存分块，每次只使用一块，使用完后，将存活的对象复制到另一块上。

标记整理：先标记存活对象，然后让所有存活对象向一端移动，直接清理端边界以外的内存。

分代算法，堆分队新生代和老年代，新生代每次收集都会有大量的对象死去，选择复制算法。将新生代分为伊甸区，和 s0,s1.大小比例为 8:1:1

老年代存活率比较高，且没有额外空间进行分配担保，选择标记清除或者标记整理算法。

垃圾回收器：

serial[ˈsɪrɪəl],串行收集器。单线程，垃圾回收的时候，必须暂停其他工作。新生复制，老年标记整理。简单高效。

ParNew 收集器。serial 的多线程版本。

Parallel Scavenge [ˈpærəlel] [ˈskævɪndʒ] 收集器,复制算法的多线程收集器。注重吞吐量，cpu 运行代码时间/cpu 耗时总时间。新生复制，老年标记整理

Serial Old 收集器，老年代版本。

Parallel Old 收集器，Parallel 老年代版本。

CMS 收集器，注重最短时间停顿。并发收集器，垃圾收集线程与用户线程（基本上）同时工作。标记清除算法。

初始标记：暂停其他线程，标记与 GC roots 直接关联的对象。并发标记,可达性分析过程。重新标记，对并发标记过程中，用户线程修改的对象再次标记一下。并发清除：标记完成之后，和用户线程并发清除死亡对象。

主要优点：**并发收集、低停顿。**

缺点，**无法处理浮动垃圾，使用标记清除算法，导致大量不连续碎片产生。对 CPU 资源敏感，可能导致用户线程变慢。**

G1 收集器，低停顿， 高吞吐。

可预测停顿（让使用者指定垃圾回收时间），空间整合（标记整理算法），分代收集：分代概念保留，但不需要和其他收集器配合就能独立管理整个堆。并行和并发：并行：利用多 CPU、多核环境下的硬件优势。并发：两个阶段的垃圾回收能和用户线程并发。

垃圾回收时，对新生代和老年代一视同仁。将整个堆划分成多个大小相等的 Region。G1 能跟踪每个 Region 的回收价值和停顿时间成本。步骤，初始标记，标记与 GC roots 直接关联的对象，并发标记，可达性分析过程。最终标记，对并发标记过程中，用户线程修改的对象再次标记一下。筛选回收：对各个 Region 的回收价值和成本进行排序，然后根据用户所期望的 GC 停顿时间制定回收计划并回收。

创建一个对象的步骤

Step1,类加载检查，先检查对象所属类是否已被加载、解析、初始化过，如果没有，先进行类加载过程；

Step2,分配内存，为对象分配内存(两种分配方式指针碰撞(标记整理)和空闲列表(标记清除))。

Step3,初始化为 0 值，将内存空间除了对象头都初始化为 0 值。

Step4,设置对象头，类的元数据信息，对象哈希码，对象年龄等

Step5,执行 init 方法，对对象真正初始化。

对象的内存布局:在内存中主要分为三部分：对象头，实例数据和对象对齐。实例数据时对象真正的有效信息，对其填充部分起占位作用。

对象头一是**运行时数据（哈希码，GC 年龄，锁状态等）二是类型指针，指向类元数据。**

对象访问形式 1、使用句柄、2 直接使用指针。使用句柄：在内存中开辟句柄池，栈中的引用变量，指向句柄池中的句柄地址，指向堆中的实例对象数据，和方法区的对象类型数据。

直接指针，栈中的引用对象变量，直接指向堆中的对象，其中堆中的对象头又指向方法区中的对象类型数据。

类加载过程，虚拟机将 Class 文件加载到虚拟机中，并初始化。主要三步，加载，链接，初始化。链接又分验证，准备，解析。

加载：主要完成三件事：通过全类名获取类的二进制字节流。2，将类的静态存储结构转化为方法区的运行时数据结构。3,在内存中生成类的 Class 对象，作为方法区数据的入口。

验证，对文件格式，元数据，字节码，符号引用等验证正确性。

准备，在方法区内为类变量分配内存并设置为 0 值。

解析，将符号引用转化为直接引用。

初始化，执行类构造器 clinit 方法，真正初始化。

类加载过程的加载的第一步，将.class 文件加载到内存是由类加载器完成的。

BootstrapClassLoader 启动类加载器,除了他其他类加载器都继承 Java.lang.ClassLoader。 /lib 下的 jar 包和类。

ExtensionClassLoader 扩展类加载器， /lib/ext 目录下的 jar 包和类。

AppClassLoader 应用类加载器，当前 classPath 下的 jar 包和类。

双亲委派机制

一个类加载器收到类加载请求之后，首先判断当前类是否被加载过。已经被加载的类会直接返回，如果没有被加载，首先将类加载请求转发给父类加载器，一直转发到启动类加载器，只有当父类加载器无法完成时才尝试自己加载。

好处 1、可以避免类的重复加载。(相同的类被不同的类加载器加载会产生不同的类)，双亲委派保证了 java 程序的稳定运行。2、保证核心 API 不被修改。

自定义类加载器:（1）继承 ClassLoader（2）重写 findClass（）方法 （3）调用 defineClass（）方法

破坏双亲委派机制，重载 loadClass() 方法。

创建对象的几种方式：

用 new 关键字创建

调用对象的 clone 方法

利用反射，调用 Class 类的或者是 Constructor 类的 newInstance（）方法

用反序列化，调用 ObjectInputStream 类的 readObject（）方法

多线程并发

多线程相关问题

进程是程序运行的实例。运行一个 Java 程序的实质就是启动一个 java 虚拟机进程。进程是程序向操作系统申请资源的基本单位。线程是进程中可独立执行的最小单位。一个进程可以包括多个线程，同一个进程中的所有线程共享该进程的资源。

竞态：多线程编程中一个问题，对于同样的输入，程序的输出有时是正确的，有时是错误的，这种计算结果的正确性与时间有关的现象被成为竞态。

线程安全和非线程安全： 一个类在单线程环境下能够正常运行，并且在多线程环境下，使用方不做特别处理也能运行正常，我们就称其实线程安全的。反之，一个类在单线程环境下运行正常，而在多线程环境下无法正常运行，这个类就是非线程安全的。

线程状态：新建（new 之后），就绪（start 之后），运行（run 之后），阻塞（等待，wait/sleep,阻塞 IO/申请锁，有限等待），死亡。

多线程实现方式:1.实现 Runnable 接口 2、继承 Thread 类。因为 java 不支持多重继承，但可以实现多个接口。所有实现 Runnable 有更好的扩展性；还可以实现资源个共享，即多个线程基于一个 Runnable 对象，和共享 Runnable 的资源。
第三种是实现 Callable 接口;将实现类当做线程中运行的任务，然后用 Thread 来调用。callable 的 call 方法有返回值，通过 FutureTask 进行封装。

A.**start**()是启动一个新线程 A,进入就绪状态。start 方法只能被调用一次。

Run()方法，是就绪状态的线程获取 cpu 之后调用的，线程进入运行状态。如果手动调用，相当于执行一个普通方法。

Java 中，每个对象有且只有一个同步锁。调用 synchronized 方法就获取了对象的锁。

当一个线程访问对象的同步方法或者同步代码块，其他线程不能再访问该对象的同步方法或者同步代码块，但可以访问该对象的非同步方法或者非同步代码块。

t1.**wait**(),让当前线程进入阻塞状态，释放 t1 对象的锁，直到其他线程调用 t1.notify()方法，当前运行线程才可能被唤醒。

wait()和 notify()方法必须出现在同步代码块中，等待和唤醒是依赖于同步锁实现的，同步锁是对象持有，每个对象有且只有一个。所以 wait 要释放锁，因为之后他释放了该锁，其他对象才能获取该锁，然后进入同步代码中，notify 该对象上等待的对象。所以这也就是 wait 和 notify 方法为什么出现在 Object 对象上。

yield()：Thread 静态方法，表示当前线程从运行状态转入就绪状态，给其他线程竞争的机会。不会释放任何锁。

sleep()：Thread 静态方法，当前线程从运行状态进入等待阻塞状态，不释放任何锁，休眠一段时间后，该线程进入就绪状态。单位毫秒

A.**join()**：当前线程进入阻塞状态，等线程 A 执行完之后，当前线程从阻塞状态进入就绪状态。

线程优先级，1-10，默认 5.用户线程，守护线程，isDaemon() t1.setDaemon(true);用户线程运行时，jvm 不得关闭。gc 线程是守护线程。

中断：调用线程的 A.interrupt()方法，会设置线程 A 的中断标记为 true。如果线程 A 处于 wait(),join(),sleep()等阻塞状态时（不是 io 阻塞和锁阻塞），清除中断标记，抛出 InterruptedException,线程结束。

A.interrupted()方法，查看线程是否处于中断状态。且清除中断标记，而 isInterrupted 则查看中断标记，不会清除。

Executor 的中断，shutdown()，不再接受新任务，等待线程池中所有的任务完成。shutdownNow(),不接受新的，和未处理的。调用每个正在执行线程的 interrupt()；

中断线程池中的一个线程：使用 submit 方法提交线程，会返回一个 future 对象，调用该对象的 cancel(true)方法中断线程。

锁机制

悲观锁和乐观锁

乐观锁：每次访问数据的时候都认为其他线程不会修改数据，所以直接访问数据，更新的时候再判断在此期间其他线程是否修改数据。CAS 和版本号机制是乐观锁的实现。

乐观锁适合多读场景，悲观锁适合多写情况。

版本号机制:数据有个 version 字段，表示被修改的次数。

CAS:无琐算法，非阻塞同步，需要读写的内存值 V 和旧的期望值 A 相同时，更新为 B.一般都是自旋 CAS,不断的重试。乐观锁缺点：1、ABA 问题（加入版本号机制）。2、自旋 CAS 如果一直不成功，开销大。3、只对单变量有效，当涉及多个共享变量时，无效。

乐观锁就是，**每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁用到的机制就是 CAS，Compare and Swap。CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。CAS 是通过硬件命令保证了原子性。**

悲观锁：每次访问数据的时候都会认为其他线程会修改数据，所以先获取锁，再访问数据。synchronized 和 ReentrantLock 都是悲观锁思想的实现。

Synchronized 关键字三种实现方式：

修饰实例方法，对当前实例对象加锁，进入同步代码前要获取对象实例的锁。修饰静态方法，对当前类对象加锁，修饰代码块，指定加锁对象，给对象加锁。

具体实例，双重校验锁实现对单例模式;

Synchronized 同步的实现，是基于**进入退出监视器 Monitor 对象实现的，**无论是同步代码块还是同步方法，都是如此；**同步代码块，是根据 monitorenter 和 monitorexit 指令实现的**，同步方法，是通过设置方法的 **ACC_SYNCHRONIZED 访问标志**;监视器 Monitor 对象存在于每个对象的对象头中。

synchronized 和 ReenTrantLock 的区别:

两者都是可重入锁(自己可以再次获取自己的内部锁)，锁计数器加 1;

synchronized 只能是是非公平锁，而 ReenTrantLock 默认实现非公平锁，也支持公平锁(先等先得)

synchronized 依赖于 JVM 实现，而 ReenTrantLock 是基于 JDK 实现的;

ReenTrantLock 功能加多：1、**等待可中断**，2、**支持公平锁**，3、**基于 Condition 实现选择性唤醒;**

synchronized 经过一系列的优化，性能已得到大大的提升，和 ReenTrantLock 相差无几。

ReentrantLock,可重入互斥锁，独占锁。互斥锁：同一时间只能被一个线程持有。可重入锁：可以被单个线程多次获取。ReentrantLock 支持公平锁和非公平锁，Synchronized 只支持非公平锁。公平锁：线程依次排队获取锁。非公平锁：不管是不是队头都能获取。公平锁和非公平锁，它们尝试获取锁的方式不同：公平锁在尝试获取锁时，即使“锁”没有被任何线程锁持有，它也会判断自己是不是 CLH 等待队列的表头；是的话，才获取锁。而非公平锁在尝试获取锁时，如果“锁”没有被任何线程持有，则不管它在 CLH 队列的何处，它都直接获取锁。

公平锁要维护一个队列，后来的线程要加锁，即使锁空闲，也要先检查有没有其他线程在 wait，如果有自己要挂起，加到队列后面，然后唤醒队列最前面的线程。这种情况下相比较非公平锁多了一次挂起和唤醒。

线程切换的开销，其实就是非公平锁效率高于公平锁的原因，因为非公平锁减少了线程挂起的几率，后来的线程有一定几率逃离被挂起的开销。

JDK1.6 引入了大量的**锁优化**:偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术减少开销。

锁主要存在 4 种**状态**：无琐状 态，偏向锁状态，轻量级锁状态，重量级锁状态 。锁可升级不可降级，提供获取锁和释放锁的效率。

自旋锁:进程进入阻塞的开销很大，为防止进入阻塞状态，在线程请求共享数据锁的时候循环自旋一段时间，如果在这段时间内获取到锁，就避免进入阻塞状态了。

1.6 引入**自适应自旋锁**，自旋次数不再固定：由锁拥有者状态和上次获取锁的自旋次数决定。

锁消除：对于被检测出不可能存在竞争的共享数据的锁进行消除。(逃逸分析)

锁粗化:虚拟机探测到一系列连续操作都对同一个对象加锁解锁，就将加锁的范围粗化到整个操作系列的外部。

偏向锁：当锁对象第一次被线程获取的时候，进入偏向状态，标记为 101，同时 CAS 将线程 ID 进入到对象头的 Mark Word 中，如果成功，这个线程以后每次获取锁就不再需要进行同步操作，甚至 CAS 不都需要。当另一个线程尝试获取这个锁，偏向状态结束，恢复到未锁定状态或者轻量级状态。

轻量级锁：对象头的内存布局 Mark Word,有个 tag bits,记录了锁的四种状态:无琐状态，偏向锁状态，轻量级锁状态，重量级锁状态.轻量级锁相对重量级锁而言，使用 CAS 去避免重量级锁使用互斥量的开销。线程尝试获取锁时，如果锁处于无琐状态，先采用 CAS 去尝试获取锁，如果成功，锁状态更新为轻量级锁状态。如果有两条以上的线程争用一个锁，状态重为重量级锁。

Java 内存模型:数据主要存放在主内存中，为了加快对内存的数据处理，在 cpu 和内存中间，加入了寄存器，存储器，高速缓存等处理器缓存;从多线程角度，**可以把内存模型简化成:主内存和线程**

本地内存。线程可以把变量从主内存读取到本地缓存中，然后再本地缓存中进行读写，然后将改变结果写入到内存中。这就导致线程本地内存和主内存数据不一致的情况。即可见性。**volatile** 可解决这个问题。

Volatile 变量在汇编阶段，会多出一条 lock 前缀指令，这会导致当前处理器缓存的数据写回到系统内存中，且让其他改数据的处理器缓存失效。这就保障了可见性

Volatile 的修饰的变量，**虚拟机会使用内存屏障禁止指令重排序保障其有序性。但 volatile 变量不能保证其原子性。所以 volatile 是线程同步的轻量级实现，性能好，多线程访问 volatile 变量不会发生阻塞，volatile 变量主要用于变量在多线程之间的可见性。但并不能保障原子性，不可替代 synchronized.synchronized 解决的是多线程之间访问共享资源的同步性。**

ThreadLocal,线程本地存储，提供了一个线程局部变量，让访问某个变量的线程都拥有自己的线程局部变量，这样线程对变量的访问就不存在竞争问题，也不需要同步。每个线程都有一个 ThreadLocal.ThreadLocalMap 对象。map 的键是 ThreadLocal t = new ThreadLocal()，值是 value。ThreadLocalMap 的 Entry 节点的 key 指向 ThreadLocal 是弱引用，虚拟机只要发现就可以垃圾回收。

Thread -> ThreadLocalMap -> Entry<ThreadLocal,value> ---> ThreadLocal(key 弱引用到 ThreadLocal);

所以 ThreadLocal 设置为 null,ThreadLocal 只有弱引用，可以被回收，但 value 存在强引用，不能被回收。

如何避免？事实上，ThreadLocalMap 的 get,set 方法中，会对 key（ThreadLocal）进行 null 判断，如果为 null,value 也设置为 null.也可以手动条调用 ThreadLocal.remove()方法。

AQS 与 JUC

AQS(AbstactQueuedSynchronizer)**同步队列器，是一个构建锁和同步器的框架**，使用 AQS 能够简单有效的构造出应用广泛的大量同步器。如 ReentrantLock, Semphore>

AQS 原理：**如果被请求的共享资源空闲，则将当前请求线程设为有效的工作线程，并且将共享资源设置为锁定状态。如果请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的。即将暂时获取不到的线程放入队列中。**CLH,是虚拟的双向队列，即不存在队列实例，**仅存在节点与节点之间的 pre 和 next 关系。AQS 将每条请求共享资源的线程封装成一个 CLH 锁队列的一个节点来实现锁的分配。**

AQS 属性(Node head, Node tail, int **state**（这个是最重要的，代表当前锁的状态，0 代表没有被占用，大于 0 代表有线程持有当前锁）, Thread 持有独占锁的线程);

等待队列中每个线程被封装为一个 Node 实例（thread + waitStatus(-1：当前 node 的后继节点对应的线程需要被唤醒,) + pre + next）;

State:表示当前锁的状态，等于 0 时，表示没有被线程占用。当大于 0 时，表示被线程占用。

Node 节点的属性 watiStatus:默认为 0，当大于 0 时，表示放弃等待，ReentrantLock 是可以指定 timeouot 的。等于-1，表示当前 node 的后继节点对应的线程需要被唤醒。当等于-2 时，标志着线程在 Condition 条件上等待的线程唤醒。等于-3 时，用于共享锁，标志着下一个 acquireShared 方法线程应该被允许。

公平锁，只有处于队头的线程才被允许去获取锁。非公平性锁模式下线程上下文切换的次数少，因此其性能开销更小。公平性锁保证了锁的获取按照 FIFO 原则，而代价是进行大量的线程切换。非公平性锁虽然可能造成线程“饥饿”，但极少的线程切换，保证了其更大的吞吐量。

AQS 组件：

AQS 类别：独占锁和共享锁。

共享锁有 ReentrantReadWriteLock、CountDownLatch、CyclicBarrier、Semaphore:

ReadWriteLock，读写锁。维护了一对相关联的读取锁和写入锁。读取锁用于只读操作，共享。写入锁用于写入操作， 是独占锁。不能同时存在读取锁和写入锁。

CountDownLatch(lætʃ)是通过共享锁实现的;CountDownLatch doneSignal = new CountDownLatch(LATCH_SIZE);

CountDownLatch 对象维护一个 count,执行一次 doneSignal.countDown()时，count 减一直到 count 为 0 时， doneSignal.await()的等待线程才能运行。所以 countDownLatch 可以让一个线程等待

一组线程完成之后才执行。

CyclicBarrier('saɪklɪk 'bæriər) cb = new CyclicBarrier(SIZE);**调用线程创建 N 个齐头并进的 CyclicBarrier 对象。每个线程执行 cb.await()时，参与者数量加一，当参与者数量达到 SIZE，阻塞的参与线程继续运行。**

(01) CountDownLatch 的作用是允许 1 或 N 个线程等待其他线程完成执行；而 CyclicBarrier 则是允许 N 个线程相互等待,直到到达某个公共屏障点.

(02) CountDownLatch 的计数器无法被重置；CyclicBarrier 的计数器可以被重置后使用，因此它被称为是循环的 barrier。

Semaphore('seməfɔ:r) **sem = new Semaphore(SEM_MAX);建立 N 个信号量。sem.acquire(count);获取 count 个信号量。如果有就给予，没有就阻塞。sem.release(count);释放信号量。**

JUC 原子类 基本类型：**AtomicInteger**,**AtomicLong**,AtomicBoolean;数组类型 AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray;引用类型，对象属性修改类型。在 32 位操作系统中，64 位的 long 和 double 变量由于会被 JVM 当作两个分离的 32 位来进行操作，所以不具有原子性；

原子类基本通过**自旋 CAS** 来实现，期望的值和现在的值是否一致，如果一致就更新。

```
public final boolean compareAndSet(long expect, long update) {
    return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
}
```

主要利用 CAS+volatile + native 方法来保证操作的原子性，从而避免同步方法的高开销。CAS 原理是那期望的值和现在的值进行比较，如果相同则更新成新的值。

集合框架的多线程实现类：

CopyOnWriteArrayList(volatile 数组来保持数据、增删改会新建数组，然后 copy 到 volatile 数组。只有查找效率高)线程安全机制是通过 volatile 和互斥锁实现的。增删改时，先获取互斥锁，然后创建新数组，复制到 volatile 数组中，释放锁。

1、**CopyOnWriteArrayList** 相当于线程安全的 ArrayList，它实现了 List 接口。CopyOnWriteArrayList 是支持高并发的。

2、**CopyOnWriteArraySet** 相当于线程安全的 HashSet，它继承于 AbstractSet 类。CopyOnWriteArraySet 内部包含一个 CopyOnWriteArrayList 对象，它是通过 CopyOnWriteArrayList 实现的。

1、ConcurrentHashMap 是线程安全的哈希表(相当于线程安全的 HashMap)

2、ConcurrentSkipListMap 是线程安全的有序的哈希表(相当于线程安全的 TreeMap)

3、ConcurrentSkipListSet 是线程安全的有序的集合(相当于线程安全的 TreeSet)

1、 **ArrayBlockingQueu(kju:)**是数组实现的线程安全的**有界**的阻塞队列。

2、 **LinkedBlockingQueue** 是**单向链表实现的(指定大小)阻塞队列**，该队列按 FIFO（先进先出）排序元素。

3、 **LinkedBlockingDeque** 是双向链表实现的(指定大小)双向并发阻塞队列，该阻塞队列同时支持 FIFO 和 FILO 两种操作方式。

4、 **ConcurrentLinkedQueue** 是单向链表实现的**无界队列**，该队列按 FIFO（先进先出）排序元素。

5、 **ConcurrentLinkedDeque** 是双向链表实现的**无界队列**，该队列同时支持 FIFO 和 FILO 两种操作方式。

线程池

线程池则是为了**减少线程建立和销毁带来的性能消耗**。线程池的使用可以帮助我们**更合理的使用系统资源.**

Executors 是个静态工厂类。它通过静态工厂方法返回 ExecutorService、ScheduledExecutorService、ThreadFactory 和 Callable 等类的对象。

Executor ɪg'**zekjətər** 只提供了 execute()'**eksɪkju:t** 接口来执行已提交的 Runnable 任务的对象。

ExecutorService 接口，继承 Executor,AbstractExecutorService 抽象类实现了 ExecutorService 接口。ThreadPoolExecutor，真正线程池实现类。

ThreadPoolExecutor 是线程池类。对于线程池，可以通俗的将它理解为"存放一定数量线程的一个线程集合。线程池允许若个线程同时运行;

当添加的到线程池中的线程超过它的容量时，会有一部分线程阻塞等待。线程池会通过相应的调度策略和拒绝策略，对添加到线程池中的线程进行管理。

关闭线程池：调用线程池的 shutdown()接口时，线程池处在 SHUTDOWN 状态时，不接收新任务，但能处理已添加的任务。

调用线程池的 shutdownNow()接口时，线程池切换为 STOP 状态，不接收新任务，不处理已添加的任务，并且通过调用每个线程的 interrupt()方法尝试中断正在处理的任务。

一定要通过 ThreadPoolExecutor(xx,xx,xx...) 来明确线程池的运行规则，指定更合理的参数。

ThreadPoolExecutor 线程池的 7 大参数：

kɔ:r corePoolSize：核心池的大小：创建线程池之后，线程池中的线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到 corePoolSize 后，就会把到达的任务放到缓存队列当中；调用了 prestartAllCoreThreads()或者 prestartCoreThread()方法，可以在任务来之前预创建线程。**核心线程会一直存活，及时没有任务需要执行；**

设置 allowCoreThreadTimeout=true（默认 false）时，核心线程会超时关闭；

maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

keepAliveTime：非核心线程的最大空闲时间。

TimeUnit：空闲时间的单位。

BlockingQueue<Runnable> workQueue ：等待执行的任务队列，队列分为有界队列和无界队列。有界队列：队列的长度有上限，当核心线程满载的时候，新任务进来进入队列，当达到上限，有没有核心线程去即时取走处理，这个时候，就会创建临时线程。（警惕临时线程无限增加的风险）

无界队列：队列没有上限的，当没有核心线程空闲的时候，新来的任务可以无止境的向队列中添加，而永远也不会创建临时线程。（警惕任务队列无限堆积的风险）

ThreadFactory：线程工厂，用来创建线程

RejectedExecutionHandler：队列已满，而且任务量大于最大线程的异常处理策略

线程池属性属性：

largestPoolSize，记录了曾经出现的最大线程个数。因为 setMaximumPoolSize()可以改变最大线程数。

poolSize：线程池中当前线程的数量。

当提交一个新任务时：

（1）如果 poolSize<corePoolSize，新增加一个线程处理新的任务。无论是否有空闲的线程新增一个线程处理新提交的任务；

（2）如果 poolSize=corePoolSize，新任务会被放入阻塞队列等待。

（3）如果阻塞队列的容量达到上限，且这时 poolSize<maximumPoolSize，新增线程来处理任务。

（4）如果阻塞队列满了，且 poolSize=maximumPoolSize，那么线程池已经达到极限，会根据饱和策略 RejectedExecutionHandler 拒绝新的任务。

* 当线程空闲时间达到 keepAliveTime 时，线程会退出，直到线程数量=corePoolSize
* 如果 allowCoreThreadTimeout=true，则会直到线程数量=0

workQueue：一个阻塞队列：ArrayBlockingQueue 和 PriorityBlockingQueue 使用较少，一般使用 LinkedBlockingQueue 和 Synchronous；

workQueue 的类型为 BlockingQueue<Runnable>，通常可以取下面三种类型：

1）**ArrayBlockingQueue：**基于数组的先进先出队列，此队列创建时必须指定大小；

2）**LinkedBlockingQueue：**基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为 Integer.MAX_VALUE；

3）**synchronousQueue**：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

线程拒绝策略：
ThreadPoolExecutor.**AbortPolicy**:**丢弃任务并抛出 RejectedExecutionException 异常。**
ThreadPoolExecutor.**DiscardPolicy**：**也是丢弃任务**，但是不抛出异常。
ThreadPoolExecutor.**DiscardOldestPolicy**：**丢弃队列最前面的任务**，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.**CallerRunsPolicy**：由调用线程处理该任务

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdown()和 shutdownNow()，其中：
shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

动态调整大小：
setCorePoolSize：设置核心池大小
setMaximumPoolSize：设置线程池最大能创建的线程数目大小

其他种类的：**Executor 框架的工具类 Execuors 来实现。**
newFixedThreadPool：**建立一个线程数量固定的线程池**，规定的最大线程数量，超过这个数量之后进来的任务，会放到等待队列中，如果有空闲线程，则在等待队列中获取，遵循先进先出原则。corePoolSize 和 maximumPoolSize 要一致，Executors 默认使用的是 LinkedBlockingQueue 作为等待队列，这是一个无界队列。
newSingleThreadExecutor：**建立一个只有一个线程的线程池**，如果有超过一个任务进来，只有一个可以执行，其余的都会放到等待队列中，如果有空闲线程，则在等待队列中获取，遵循先进先出原则。使用 LinkedBlockingQueue 作为等待队列。等待队列无限长的问题，容易造成 OOM。
newCachedThreadPool：**缓存型线程池，在核心线程达到最大值之前，有任务进来就会创建新的核心线程，并加入核心线程池，即时有空闲的线程，也不会复用。达到最大核心线程数后，新任务进来，如果有空闲线程，则直接拿来使用，如果没有空闲线程，则新建临时线程**。并且线程的允许空闲时间都很短，如果超过空闲时间没有活动，则销毁临时线程。关键点就在于它使用 **SynchronousQueue 作为等待队列，它不会保留任务，新任务进来后，直接创建临时线程处理。容易造成无限制的创建线程，造成 OOM。**
newScheduledThreadPool：计划型线程池，可以设置固定时间的延时或者定期执行任务，同样是看线程池中有没有空闲线程，如果有，直接拿来使用，如果没有，则新建线程加入池。使用的是 DelayedWorkQueue 作为等待队列，这中类型的队列会保证只有到了指定的延时时间，才会执行任务。容易造成无限制的创建线程，造成 OOM。

线程池大小如何设置？
对于计算密集型的任务，一个有 Ncpu 个处理器的系统通常通过使用一个 Ncpu + 1 个线程的线程池来获得最优的利用率.
对于包含了 I/O 和其他阻塞操作的任务，不是所有的线程都会在所有的时间被调度，因此你需要一个更大的池。2*Ncpu
Nthreads = Ncpu x Ucpu x (1 + W/C)，其中
Ncpu = CPU 核心数
Ucpu = CPU 使用率，0~1
W/C = 等待时间与计算时间的比率
线程数 = Ncpu /（1 - 阻塞系数）= Ncpu x (1 + W/C)

线程数设置过大有什么缺点：

如果 maxsize 过大会占用更多资源，cpu 会频繁地进行上下文切换，会导致 cpu 缓存的数据失效和重新加载，结果就是上下文切换和 reload 的时间变多，也就是说 cpu 将更多的时间花费到对线程的管理上去了，这时候更多的线程反而更慢

MySQL

事务相关

数据库的**事务**

事务：一组数据库操作，要么全都执行，要么都不执行;

事务特性：ACID. 原子性：事务是最小的执行单位，不可分割，保证事务要么都完成，要么都不完成。

一致性：？？？执行事务前后，数据保持一致。隔离性：并发访问数据库时：一个事务不被其他事务干扰。持久性：事务一 旦提交，对数据库的改变是持久的

并发事务带来的问题：

脏读：一个事务读取了另一个事务修改但未提交的数据。

丢失修改：数据被两个事务连续修改，第一个事务的修改丢失了。

不可重复读，一个事务连续读两次数据，但结果不一样。(两次读之间，数据被其他事务修改)。

幻读：一个事务连续读两次数据，读取数据量不一样。(两次读之前，数据被其他事务删除或新增)。

事务隔离级别:

1、读未提交。可以读取尚未提交的数据。能导致脏读，不可重复读，幻读。

2、读已提交。允许读取并发事务已经提交的数据。导致不可重复读，幻读。

3、可重复读。意义在哪？？？？对同一字段，多次读取结果一致。导致幻读。

不可重复读很容易让人陷入一个思维定式那就是 我干嘛需要多次读取一个值还要保证一致

要跳出这个思维看本质：我在事务中会不会受到其他事务的影响？

举个简单的例子 数据校对（只是举个例子体现意思 不用太在意具体的业务）

我要取当前的余额 当前的账单 上个月的余额 我要检验一下数据对不对

我在事务中取了当前的账单和上个月的余额，好嘛，这时候又有新的订单提交了，我再获取余额是不是就不一致了？

4、串行化。所有事务，依次执行。没啥问题。**（这个串行化是针对行锁的，不同行的事务可以并发）**

设置隔离级别之后，并不是不能并发，而是并发的时候，一个事务的修改数据(绝对读到，提交的才能读到。提交不提交，更新的数据都读不到。提交不提交，增删的数据都读不到)，什么时候才能被另一个事务读到。但彼此的逻辑操作没有影响。

MySQL InnoDB **默认支持可重复读**，但使用了 **Next-Key Lock 算法避免了幻读的发生**。完全达到了保保证事务的隔离要求。但在分布式事务下，一般可串行化。

Innodb 和 Myisam 的区别：

1、MyISAM 不支持事务，而 Innodb 支持事务。2、Myisam 是表级锁，而 Innodb 是行级锁。3 外键支持：mysiam 表不支持外键，而 InnoDB 支持。4、count 运算：myisam 缓存有表的行数，这种缓存只是表行的总数，where 筛选无效。而 Innodb 没有。

MyISAM 适合：(1)做很多 count 的计算；(2)读密集；(3)没有事务。

InnoDB 适合：(1)要求事务；(2)写密集（3）高并发

锁机制

三种并发控制机制：悲观并发控制、乐观并发控制和多版本并发控制。悲观并发控制其实是最常见的并发控制机制，也就是锁；乐观并发控制其实也有另一个名字：乐观锁. MVCC 多版本并发控制机制，可以与前两者中的任意一种机制结合使用，以提高数据库的读性能。

乐观锁：在访问数据之前，默认不会有其他事务对此数据进行修改，所以先访问数据，然后再查找在此期间是否有事务修改数据。这不是数据库自带的，需要我们自己去实现，一般基于版本去实现。

悲观锁：

按照锁的粒度把数据库锁分为表级锁和行级锁。

表级锁: 对当前操作的整张表加锁,实现简单，加锁快，不死锁，但并发能力低。

行级锁: **只针对当前操作的行进行加锁**。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的**开销也最大，加锁慢，会出现死锁**。

Record Lock 记录锁: 锁住某一行，如果表存在索引，**那么记录锁是锁在索引上的**，如果表没有索引，那么 InnoDB 会创建一个隐藏的**聚簇索引加锁**。

Gap LocK 间隙锁: 间隙锁是一种记录行与记录行之间存在空隙或在第一行记录之前或最后一行记录之后产生的锁。间隙锁可能占据的单行，多行或者是空记录。 **对索引项之间的"间隙"加锁，锁定记录的范围,不包含索引项本身。其他事务不能在锁范围内插入数据**，这样就防止了别的事务新增幻影行。

Next-key Lock： 锁定索引项本身和索引范围。NK 是一种记录锁和间隙锁的组合锁。既锁住行也锁住间隙。即 Record Lock 和 Gap Lock 的结合。**可解决幻读问题。**

根据是否独占，锁又可以分为共享锁和排他锁。

共享锁（Share Locks，简记为 S）又被称为读锁，**其他用户可以并发读取数据，但任何事务都不能获取数据上的排他锁，直到已释放所有共享锁。**

排它锁（(Exclusive lock,简记为 X 锁)）又称为写锁，若事务 T 对数据对象 A 加上 X 锁，则只允许 T 读取和修改 A，其它任何事务都不能再对 A 加任何类型的锁，直到 T 释放 A 上的锁。

Innodb 同时支持行锁和表锁。但行锁和表锁的同时存在会发生冲突，如 A 申请了行共享锁，而 B 再申请表互斥锁。这时 B 不仅需要查看是否已经存在其他表锁，以及逐个查看是否存在行锁，效率太低。于是又引入了意向锁。意向锁是一种表级锁，用来指示**接下来的一个事务将要获取的是什么类型的锁（共享还是独占）。意向锁分为意向共享锁（IS）和意向独占锁（IX），依次表示接下来一个事务将会获得共享锁或者独占锁。**

意向共享锁（IS）：事务打算给数据行加共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。

意向排他锁（IX）：事务打算给数据行加排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。

在意向锁存在的情况下，事务 A 必须先申请表的意向共享锁，成功后再申请一行的行锁。而事务 B 发现表上有意向共享锁，说明表中有些行被共享行锁锁住了，因此，事务 B 申请表的写锁会被阻塞。而且，申请意向锁的动作是数据库自动完成的，不需要我们手动申请。

MVCC 多版本并发控制（Multiversion Concurrency Control），多版本控制：指的是一种提高并发的技术。最早的数据库系统，只有读读之间可以并发，读写，写读，写写都要阻塞。引入多版本之后，只有写写之间相互阻塞，其他三种操作都可以并行，这样大幅度提高了 InnoDB 的并发度。

每一个写操作都会创建一个新版本的数据，读操作会从有限多个版本的数据中挑选一个最合适的结果直接返回；在这时，读写操作之间的冲突就不再需要被关注，而管理和快速挑选数据的版本就成了 MVCC 需要解决的主要问题。

各数据库中 MVCC 实现并不统一，MVCC 只在 READ COMMITTED 和 REPEATABLE READ 两个隔离级别下工作;

对于使用 InnoDB 存储引擎的表来说,它的聚簇索引记录中都包含两个必要的隐藏列:(**trx_id 事务 ID、roll_pointer 上个版本指针**,其实还有一个 row_id 的隐藏列但这里用不着);

每次对记录进行改动，都会把对应的事务 id 赋值给 trx_id 隐藏列，也会把旧的版本写入到 undo 日志中；

所以在并发情况下，一个记录可能存在多个版本，通过 roll_pointer 形成一个版本链。MVCC 的核心任务就是：判断一下版本链中的哪个版本是当前事务可见的。这就有了 ReadView 的概念，这个 ReadView 中主要包含当前系统中还有哪些活跃的读写事务，把它们的事务 id 放到一个列表中，我们把这个列表命名为为 m_ids；根据 ReadView 的活跃事务 ID 列表和版本链事务 ID 进行比较找出可见的事务 ID 最大的版本：

1、如果版本的 trx_id 属性值小于 m_ids 列表中最小的事务 id，表明生成该版本的事务在生成

ReadView 前已经提交，所以该版本可以被当前事务访问。
2、如果版本的 trx_id 属性值大于 m_ids 列表中最大的事务 id，表明生成该版本的事务在生成 ReadView 后才生成，所以该版本不可以被当前事务访问。
3、被访问版本的 trx_id 属性值在 m_ids 列表中最大的事务 id 和最小事务 id 之间，那就需要判断一下 trx_id 属性值是不是在 m_ids 列表中，如果在，说明创建 ReadView 时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建 ReadView 时生成该版本的事务已经被提交，该版本可以被访问。
MVCC 只在读已提交和可重复读这两个隔离机制下运行。这两个隔离机制下 MVCC 实现方式的区别就在于：读已提交是每次读取数据前都生成一个 ReadView；而可重复读，是在第一次读取数据时生成一个 ReadView，后序的重复查询就不再生产 ReadView 了。

总结：

多版本并发控制指的就是在使用 READ COMMITTD、REPEATABLE READ 这两种隔离级别的事务在执行普通的 SEELCT 操作时访问记录的版本链的过程，这样子可以使不同事务的读-写、写-读操作并发执行，从而提升系统性能。READ COMMITTD、REPEATABLE READ 这两个隔离级别的一个很大不同就是生成 ReadView 的时机不同，READ COMMITTD 在每一次进行普通 SELECT 操作前都会生成一个 ReadView，而 REPEATABLE READ 只在第一次进行普通 SELECT 操作前生成一个 ReadView，之后的查询操作都重复这个 ReadView 就好了。

Mysql 死锁处理方式 1、等待，直到超时，事务自动回滚。2、发起死锁检测， 回滚一个事务，让其他事务执行。

死锁检测，构建一个以事务为起点，锁为边的有向图，看是否存在环。

索引相关

索引优缺点：(1) 优点：加快检索速度(2)缺点：(a)创建索引和维护索引需要耗费时间(b)索引需要占用空间 (c)进行数据的增删改时候需要动态维护索引

索引类型：主键索引，唯一索引，全文索引，普通索引，复合索引。

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，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。但是有缺点。1、不能使用范围查询。2、无法利用索引的数据来避免任何排序运算；3、不支持多列联合索引的最左匹配规则；4、任何时候都不能避免表扫描。5、存在所谓的哈希碰撞问题。

所以我们都用 **B+树**，只有叶子节点存储数据，其他的节点只是起到索引的作用。平衡，性能稳定，每次查询的次数都是树的高度。

索引是一种数据结构。**索引本身很大，不可能全部存储在内存中，因此索引以索引表的形式存储在磁盘中。这样的话，索引查找过程中就要产生磁盘 I/O 消耗**，相对于内存存取，I/O 存取的消耗要高几个数量级，所以评价一个数据结构作为**索引的优劣最重要的指标就是在查找过程中磁盘 I/O 操作次数的渐进复杂度**。换句话说，索引的结构组织要尽量**减少查找过程中磁盘 I/O 的存取次数。**

M 阶 B 树：

1、 **树中每个结点至多有 m 个子结点**（即 M 阶）；
2、 **若根结点不是叶子结点,则至少有 2 个子结点**；
3、 除根结点和叶子结点外,**其它每个结点至少有 ceil(m/2)个子结点；即中间节点最少有 ceil(m/2)个子结点。**
4、 **所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息；**
5、 **有 k 个子结点的非终端结点恰好包含有 k-1 个关键字(单节点里元素).**

每个节点中元素个数 n 必须满足： **[ceil(m / 2)-1]<= n <= m-1。**(即 M 阶树单节点最多有 M-1 个元素)
每个结点中关键字从小到大排列，并且当该结点的孩子是非叶子结点时，该 k-1 个关键字正好是 k 个孩子包含的关键字的值域的分划.

B+树的不同之处：**非叶子节点只存储键值信息。数据记录都存放在叶子节点中。所有叶子节点之间都有一个链指针。**

B+树的优点：

1、B+树中间节点不存放数据，所以同样大小的磁盘页上可以容纳更多节点元素，IO 次数更少。
2、B+树的查询必须最终找到叶子节点，而 B-树只需要找到匹配的元素即可。B+树性能稳定。
3、范围查询方便。B-树只能依靠繁琐的中序遍历，而 B+树只需要在链表上遍历即可。

磁盘数据地址：柱面号、盘面号、块号

因为普通的全表查询时间复杂度是 O(n);如果是平衡二叉树，或者红黑树，查找时间变成 O(log2N)，但他们依然不适合做索引。因为索引通常比较大，存于磁盘中，无法一次将全部的索引加载到内存中，每次只能从磁盘中读取一个页到内存中，而平衡二叉树底层实现是数组，逻辑上相邻的节点在物理结构上可能相差很远，因此磁盘 IO 次数可能很大，平衡二叉树没能充分利用磁盘预读功能。**磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。**这样做的理论依据是计算机科学中著名的局部性原理。红黑树这种结构，h 明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的 I/O 渐进复杂度也为 O(h)，效率明显比 B-Tree 差很多。

B 树的每个节点可以存储多个关键字，**它将节点大小设置为磁盘页的大小，充分利用了磁盘预读的功能。每次读取磁盘页时就会读取一整个节点。也正因每个节点存储着非常多个关键字，树的深度就会非常的小。进而要执行的磁盘读取操作次数就会非常少**，更多的是在内存中对读取进来的数据进行查找。

logm(n+1)<= h <=log(ceil(m/2)) (n+1)/2 + 1

B+树的关键字全部存放在叶子节点中，非叶子节点用来做索引，而叶子节点中有一个指针指向一下个叶子节点。做这个优化的目的是为了提高区间访问的性能。而正是这个特性决定了 B+树更适合用来存储外部数据。

二叉查找树 BST：查找最好时间复杂度 **O(logN)，最坏时间复杂度 O(N)。**插入删除的实现简单，时间复杂度一致。

平衡二叉查找树 AVL：查找的时间复杂度维持在 O(logN)，不会出现最差情况 .AVL 树在执行每个插入操作时最多需要 1 次旋转，其时间复杂度在 O(logN)左右。AVL 树在执行删除时代价稍大，一次删除操作最多需要 O(logN)次旋转,执行每个删除操作的时间复杂度需要 O(2logN)。

查找 效率最好情况下时间复杂度为 O(logN).

插入和删除操作改变树的平衡性的概率要远远小于 AVL（RBT 不是高度平衡的）。因此需要的旋转操作的可能性要小，**而且一旦需要旋转，插入一个结点最多只需要旋转 2 次，删除最多只需要旋转 3 次(**小于 AVL 的删除操作所需要的旋转次数)。**虽然变色操作的时间复杂度在 O(logN)，但是实际上，这种操作由于简单所需要的代价很小。**

聚簇索引的解释是:聚簇索引的顺序就是数据的物理存储顺序；

非聚簇索引的解释是:索引顺序与数据物理排列顺序无关；

MyISAM 使用的是非聚簇索引：非聚簇索引的数据表和索引表是分开存储的。主索引和辅助索引几乎是一样的，叶子节点存储的是指向数据的物理地址。

Innodb 使用的是聚簇索引。**聚簇索引的主键索引的叶子结点存储的是键值对应的数据本身，辅助索引的叶子结点存储的是键值对应的数据的主键键值。**

B+树有主键索引和辅助索引两种;；主键索引就是按照表中主键的顺序构建一颗 B+树，并在叶节点中存放表中的行记录数据，一个表只能有一个主键索引。**而辅助索引，叶节点并不存储行记录数据，仅仅是主键。通过辅助索引查找到对应的主键，最后在聚集索引中使用主键获取对应的行记录。（这个叫回表查询？？?）**

最左前缀原则：mysql 索引可以引用多列，叫联合索引，如果查询条件精确匹配联合索引的左边连续一列或者多列，则查询命中索引。a,b,c 的联合索引，(a,c)可以**命中 a,c 不能命中**。

联合索引并不是全部不中，或者全部中。可以只命中一部分，例如单单命中 A.

MySQL 的查询优化器会自动调整 where 子句的条件顺序以使用适合的索引，不过建议 where 后

的字段顺序和联合索引保持一致，养成好习惯。

组合索引(大于等于 2 小于等于 3)也是建立一个 B+树，只不过非叶子节点存储的是第一个列。叶子节点组合的列都有，中了第一列之后，然后安装其他的列索引搜查。
联合索引的好处：利用覆盖索引，避免回表操作。
而且对于两个单列查询返回行较多，同时查返回行较少，联合索引更高效。

Explain 语句的字段：
ID: SELECT 的查询序列号;
select_type:示查询中**每个 select 子句的类型**
(1) SIMPLE(简单 SELECT，**不使用 UNION 或子查询**等)
(2) PRIMARY(**子查询中最外层查询**，查询中若包含任何复杂的子部分，最外层的 select 被标记为 PRIMARY)
(3) UNION(UNION 中的第二个或后面的 SELECT 语句)
(4) DEPENDENT UNION(UNION 中的第二个或后面的 SELECT 语句，取决于外面的查询)
(5) UNION RESULT(UNION 的结果，union 语句中第二个 select 开始后面所有 select)
(6) **SUBQUERY(子查询中的第一个 SELECT**，结果不依赖于外部查询)
(7) DEPENDENT SUBQUERY(子查询中的第一个 SELECT，依赖于外部查询)
(8) DERIVED(派生表的 SELECT, FROM 子句的子查询)
(9) UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)
Table:显示这一步所访问数据库中表名称,
Type 对表访问方式，**表示 MySQL 在表中找到所需行的方式，又称“访问类型”。**
ALL、index、range、 ref、eq_ref、const、system、NULL（从左到右，性能从差到好）
All:全表扫描。index: full index scan,遍历索引树。range:只检索给定范围的行，使用一个索引来选择行
ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值
eq_ref: 类似 ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用 primary key 或者 unique key 作为关联条件
const、system: 当 MySQL 对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于 where 列表中，MySQL 就能将该查询转换为一个常量，system 是 const 类型的特例，当查询的表只有一行的情况下，使用 system

NULL: MySQL 在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。
possible_keys：指出 MySQL 能使用哪个索引在表中找到记录，**查询涉及到的字段上若存在索引，则该索引将被列出，**但不一定被查询使用
Key：显示 MySQL 实际决定使用的键（索引），必然包含在 possible_keys 中
key_len：表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（**key_len 显示的值为索引字段的最大可能长度**，并非实际使用长度，即 key_len 是根据表定义计算而得，不是通过表内检索出的）
ref：列与索引的比较，表示上述表的连接匹配条件，**即哪些列或常量被用于查找索引列上的值**
rows：估算出结果集行数，表示 MySQL 根据表统计信息及索引选用情况，**估算的找到所需的记录所需要读取的行数、**
xtra：含 MySQL 解决查询的详细信息,有以下几种情况：
Using where:不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示 mysql 服务器将在存储引擎检索行后再进行过滤
Using temporary：表示 MySQL 需要使用**临时表来存储结果集，常见于排序和分组查询，**常见 group by ; order by

Using filesort：当 Query 中包含 **order by 操作**，而且无法利用索引完成的排序操作称为“**文件排序**”
Using join buffer：**改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果**。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。
Impossible where：这个值强调了 **where 语句会导致没有符合条件的行**（通过收集统计信息不可能存在结果）。
Select tables optimized away：这个值意味着仅通过使用索引，**优化器可能仅从聚合函数结果中返回一行**
No tables used：Query 语句中使用 from dual **或不含任何 from 子句**

其他优化地方：
强制类型转换会全表扫描。Varchar 的 phone = ‘123344’，别用 123345 数字。
使用短索引（又叫前缀索引）来优化索引。
存在非等号和等号混合判断条件时，在建索引时，请把等号条件的列前置。范围列可以用到索引（联合索引必须是最左前缀），但是范围列后面的列无法用到索引，索引最多用于一个范围列，如果查询条件中有两个范围列则无法全用到索引。
利用覆盖索引来进行查询操作，避免回表。

什么时候要使用索引？
主键自动建立唯一索引；
经常作为查询条件在 WHERE 或者 ORDER BY 语句中出现的列要建立索引；
作为**排序的列**要建立索引；**(单纯的 order by 不会用到索引，但如果在 where 中出现，就可以用索引了。)**
查询中与其他**表关联的字段，外键关系建立索引**
高并发条件下倾向**组合索引**；
用于**聚合函数的列可以建立索引**，例如使用了 max(column_1)或者 count(column_1)时的 column_1 就需要建立索引
什么时候不要使用索引？
经常增删改的列不要建立索引；
有**大量重复的列**不建立索引；
表记录太少不要建立索引。只有当 数据库里已经有了足够多的测试数据时，它的性能测试结果才有实际参考价值。如果在测试数据库里只有几百条数据记录，它们往往在执行完第一条查询命令之后就被全部加载到内存里，这将使后续的查询命令都执行得非常快--不管有没有使用索引。**只有当数据库里的记录超过了 1000 条、数据总量也超过了 MySQL 服务器上的内存总量时，数据库的性能测试结果才有意义。**

覆盖索引：
如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称 之为“覆盖索引”。我们知道在 **InnoDB** 存储引擎中，如果不是主键索引，叶子节点存储的是主键+列值。最终还是要“回表”，也就是要通过主键再查找一次,这样就会比较慢。覆盖索引就是把要查询出的列和索引是对应的，不做回表操作！**InnoDB** 存储引擎支持覆盖索引，即从辅助索引中就可以得到查询的记录，而不需要查询聚集索引中的记录。**SQL** 只需要通过索引就可以返回查询所需要的数据，而不必通过二级索引查到主键之后再去查询数据。
解释一： **就是 select 的数据列只用从索引中就能够取得，不必从数据表中读取，**换句话说查询列要被所使用的索引覆盖。
解释二： 索引是高效找到行的一个方法，当能通过检索索引就可以读取想要的数据，那就不需要再到数据表中读取行了。如果一个索引包含了（或覆盖了）满足查询语句中字段与条件的数据就叫做覆盖索引。
解释三：是非聚集组合索引的一种形式，它包括在查询里的 Select、Join 和 Where 子句用到的所有列（即建立索引的字段正好是覆盖查询语句[select 子句]与查询条件[Where 子句]中所涉及的字段，

也即，索引包含了查询正在查找的所有数据）。

聚集索引：

聚集索引就是按照每张表的主键构造一棵 B+树，同时叶子节点中存放的即为整张表的行记录数据。

辅助索引：

辅助索引，也叫非聚集索引。和聚集索引相比，叶子节点中并不包含行记录的全部数据。

SQL 优化的一些部分：

如何避免全表扫描？

1、where 字句别用 or 链接，可以 union all.2、in 和 not in 也慎用，可以 between and. 3.避免对字段进行 null 值判断。4,where 字句别用!=和<>操作符 5、别用以通配符开头的 like 的查询。6、别在 where 子句对字段进行表达式操作和函数操作。 7 任何地方都不要使用 select * from t。8、尽量使用数字型字段。9、复合索引尽量满足最左前缀原则。10,在查找唯一一条数据的时候，使用 limit 1.10、类型不一致会导致失效，例如字符串不加单引号会导致索引失效。

索引优化：

1、根据最左前缀原则，我们一般把排序分组频率最高的列放在最左边

2、模糊查询以%为开始的查询，只能使用全文索引来进行优化。

3、使用短索引。对串列进行索引，如果可能应该指定一个前缀长度。

in 与 exists 的区别。使用上，in 后面的查询返回结果只能有一个字段。而 exists 没有限制。

本质上： A exists B;exists 相当于遍历外面 A,看 A 中数据是否存在于 B。而 in,相当于将结果集 B 分解开，用 or 相连，相当于做多次的查询。

exists 相当于查询筛选，in 则是多次查询;

1、如果查询的两个表大小相当，那么用 in 和 exists 差别不大。

2、如果两个表中一个表大，另一个是表小，那么 IN 适合于外表大而子查询表小的情况。

3、如果两个表中一个表大，另一个是表小，EXISTS 适合于外表小而子查询表大的情况。

in 不会使用索引搜索，会全表扫描。

数据量过大：

1、查询时限定数据范围。2、读写分离，主写从读。3、垂直分区 4、水平分区.

数据库相关操作语句

数据库范式：1NF:每个关系的属性都是原子的，不可能分割。每一个列只有一个值。

2NF: 如果关系模式 R 是 1NF，且每一个非主属性完全依赖(而不能部分依赖)于候选建，那么就称 R 是第二范式。

3NF：如果关系模式 R 是 2NF，且关系模式 R（U,F）中的所有非主属性对任何候选关键字都不存在传递依赖，则称关系 R 是属于第三范式。

BCNF: BC 范式（BCNF）：符合 3NF，并且，主属性不依赖于主属性

内连接 NNER JOIN：内连接是一种一一映射关系，就是两张表都有的才能显示出来

左连接 LEFT JOIN: 左连接是左边表的所有数据都有显示出来，右边的表数据只显示共同有的那部分，没有对应的部分只能补空显示.

右连接 RIGHT JOIN:右连接，右边表的所有数据都会显示出来，左边的只会出现共同的那部分，其他的空。

全连接、外连接 Outer Join：查询出左表和右表所有数据，但是去除两表的重复数据

MySQL 中一条 SQL 语句的执行过程

查询 sql 的执行语句：

1、客户端通过 TCP 连接发送连接请求到 mysql 连接器，连接器会对该请求进行权限验证及连接资源分配。

2、建立连接后客户端发送一条语句，mysql 收到该语句后，通过命令分发器判断其是否是一条 select 语句，如果是，在开启查询缓存的情况下，先在查询缓存中查找**该 SQL 是否完全匹配**，如果完全匹配，**验证当前用户是否具备查询权限**，如果权限验证通过，直接返回结果集给客户端，该查询也就完成了。如果不匹配继续向下执行。

3、如果在查询缓存中未匹配成功，则将语句交给**分析器作语法分析**，MySQL 需要知道到底要查哪些东西，如果语法不对，就会返回语法错误中断查询。

4、分析器的工作完成后，**将语句传递给预处理器，检查数据表和数据列是否存在**，解析别名看是否存在歧义等。

5、语句解析完成后，MySQL 就知道要查什么了，之后会将**语句传递给优化器进行优化**（通过索引选择最快的查找方式），**并生成执行计划。**

6、**之后交给执行器去具体执行该语句，在执行之前，会先检查该用户是否具有查询权限，如果有，继续执行该语句。**执行器开始执行后，会逐渐将数据保存到结果集中，同时会逐步将数据缓存到查询缓存中，最终将结果集返回给客户端。

group by 语法可以根据给定数据列的每个成员对查询结果进行分组统计，最终得到一个分组汇总表。
SELECT DEPT, **MAX(SALARY)** AS MAXIMUM FROM STAFF **GROUP BY DEPT ：每个部分的最高薪水**

SELECT DEPT, **sum(SALARY)** AS total FROM STAFF **GROUP BY DEPT，每个部门的总薪水**

having 字句可以让我们筛选成组后的各种数据，where 字句在聚合前先筛选记录，也就是说作用在 group by 和 having 字句前。而 having 子句在聚合后对组记录进行筛选。我的理解就是真实表中没有此数据，这些数据是通过一些函数生存。
SELECT region, SUM(population), SUM(area) FROM bbc GROUP BY region **HAVING SUM(area)>1000000**
SELECT DEPT, MAX(SALARY) AS MAXIMUM, MIN(SALARY) AS MINIMUM FROM staff GROUP BY DEPT
HAVING COUNT(*) >2 ORDER BY DEPT

查询最近 N 天(不超过 30 天)某一款产品的订单。从第 10 条开始取 5 条，ID 从大到小倒序。
select * from table limit 9,5;#从 0 开始
写代码 创建索引
CREATE (UNIQUE/FULLTEXT/) INDEX indexName ON mytable(username(length));
ALTER table tableName ADD INDEX indexName(columnName)
CREATE TABLE mytable(
ID INT NOT NULL,
username VARCHAR(16) NOT NULL,
INDEX [indexName] (username(length)));

主从复制架构相关

数据库主从复制：

复制的用途：1、**读写分离，提供数据库性能和并发能力。2、实时备灾，用于故障切换。**

复制存在的问题：1、主库宕机后，数据可能丢失。2、主库写压力大时，复制可能延时。**复制延迟怎么解决？？？**

复制原理：主从复制是 myql 内带功能，是一个异步的过程，**把主库的二进制日志文件 binlog,复制到从库上，然后从库在本地完全顺序的执行日志中的各种操作。**

复制过程：1、主节点 log dump 线程：当从节点连接主节点时，主节点会创建一个 log dump 线程，用于发送 bin-log 的内容。**在读取 bin-log 中的操作时，此线程会对主节点上的 bin-log 加锁，当读取完成，甚至在发动给从节点之前，锁会被释放。2、从节点 I/O 线程：**当从节点上执行`start slave`命令之后，从节点会**创建一个 I/O 线程用来连接主节点，请求主库中更新的 bin-log。I/O 线程接收到主节点 bin log dump 进程发来的更新之后，保存在本地 relay-log 中。3、从节点 SQL 线程：SQL 线程负责读取 relay log 中的内容，解析成具体的操作并执行，最终保证主从数据的一致性。**

从节点上的 I/O 进程连接主节点，**并请求从指定日志文件的指定位置（或者从最开始的日志）之后的日志内容；**主节点接收到来自从节点的 I/O 请求后，通过负责复制的 I/O 进程根据请求信息读取指定日志指定位置之后的日志信息，返回给从节点。返回信息中除了日志所包含的信息之外，还**包括本次返回的信息的 bin-log file 的以及 bin-log position；**从节点的 I/O 进程接收到内容后，将接收到的日志内容更新到本机的 relay log 中，并将读取到的 binary log 文件名和位置保存到 master-info 文件中，Slave 的 SQL 线程检测到 relay-log 中新增加了内容后，会将 relay-log 的内容解析成在祝节点上实际执行过的操作，并在本数据库中执行。

MySQL **主从复制默认是异步的模式：**

异步复制：主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理；主节点不会主动 push bin log 到从节点；

半同步模式：这种模式下主节点只需要接收到其中一台从节点的返回信息，就会 commit；否则需要等待直到超时时间然后切换成异步模式再提交；这样做的目的可以使主从数据库的数据延迟缩小，可以提高数据安全性。半同步模式不是 mysql 内置的，需要装插件开启半同步模式。

全同步模式：全同步模式是指主节点和从节点全部执行了 commit 并确认才会向客户端返回成功。

binlog 记录格式：1、基于 SQL 语句的复制：记录会修改数据的 sql 语句到 binlog 中，减少了 binlog 日志量，节约 IO,提高性能。某些情况：会导致主从节点中数据不一致。**2、基于行的复制：**将 SQL 语句分解为基于 Row 更改的语句并记录在 bin log 中，也就是只记录哪条数据被修改了，修改成什么样。优点：**解决了特定情况下的存储过程、或者函数、或者 trigger 的调用或者触发无法被正确复制的问题。缺点日志量太大。3、混合方式：**能语句就语句，不能语句就切换行。

数据库用到了**读写分离**，那你知不知道这样做会有什么问题：

在从库上会读到系统的一个过期状态”的现象，暂且称之为“过期读”。

强制走主库方案其实就是，将查询请求做分类，**对于必须要拿到最新结果的请求，强制将其发到主库上。**

sleep 方案：主库更新后，读从库之前先 sleep 一下。具体的方案就是，类似于执行一条 select sleep(1) 命令。

mysql 主从复制存在的问题：

主库宕机后，数据可能丢失

从库只有一个 sql Thread，主库写压力大，**复制很可能延时**

解决方法：

半同步复制—解决数据丢失的问题

并行复制—-解决从库复制延迟的问题（并行是指从库多线程 apply binlog 库级别并行应用 binlog，同一个库数据更改还是串行的(5.7 版并行复制基于事务组)设置）

Mysql 逻辑架构可以分为两层：服务层和存储引擎。服务层：Mysql 的核心服务功能，查询语句解析，缓存，词法语法分析。

Redis

常见问题

Redis,key-Value 类型的内存数据库，整个数据库系统在内存中操作，定期异步 flush 到硬盘上进行保存。常用于缓存，也可以作分布式锁。redis 提供多种数据类型，支持事务，两种持久化方式，多种集群方案。

Redis 为什么要作为缓存？ 高性能和高并发：

高性能：用户第一次访问数据库时，是从硬盘上读取的，过程比较慢，效率比较低。redis 作为缓存，将用户访问的诗句存在缓存中，下一次再访问这些数据时就可以直接从缓存中读取了，操作缓存就是直接存在内存，速度特别快。

高并发:直接操作缓存所能承受的请求远远大于直接访问数据库的。把数据库的部分数据存在缓存中，可提供并发能力。

redis/memcached **分布式缓存**和 map/guava **本地缓存的区别:**

缓存分为本地缓存和分布式缓存，使用 map 或 guava 的是本地缓存，轻量而快速，随着 jvm 的销毁而结束，**多实例情况下，每个实例都保存一份缓存，缓存不具有一致性。**

分布式缓存，多实例情况下，各实例共用一份缓存数据，缓存具有一致性。缺点，架构复杂，要保证服务的高可用。

redis 和 memcached 的区别:

1、**redis 支持更丰富的数据类型**：redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list,set,zset,hash 等数据结构的存储，memcached 仅支持简单的数据类型，string.

2、**reids 支持数据的持久化，**可以将内存中的数据保存到磁盘中，重启的时候可以再次加载使用，而 memcached 不支持持久化。

3、集群模式，memcached 没有原生的集群模式，**redis 目前是原生支持 cluster 模式的**。

4、memcached **是多线程**，非阻塞 IO 复用的网络模型;**Redis 使用单线程的多路 IO 复用模型。**

redis 常用数据结构以及场景。

string,hash,list,set,zset;key 都是 string,但 value 是多种数据结构的。

数据类型：

String: **set、get、decr、incr、**

hash(将结构化的数据，比如一个对象（前提是这个对象没嵌套其他的对象）给缓存在 redis 里);**hget,hset,hgetall**

List,粉丝列表；存储一些列表型的数据结构，类似粉丝列表了、文章的评论列表。**以通过 lrange 命令，就是从某个元素开始读取多少个元素，**可以基于 list 实现分页查询。）**lpush,rpush,lpop,rpop,lrange**

Set：无序集合，自动去重 set 玩儿交集、并集、差集的操作，比如交集吧，可以把两个人的粉丝列表整一个交集，看看俩人的共同好友是谁。**Sadd, Spop, Sunion, sinterstore(交集)**

Sorted Set，去重但是可以排序；最大的特点是有个分数可以自定义排序规则。**Zadd,Zrange,zcard**

底层实现：

简单动态字符串 SDS:char[]数组加 len 属性和 free 属性（记录数组中未使用的字节数）；直接获取长度，防止溢出，

Redis 还实现了双端链表，双端，无环，带长度属性 len;

Redis 的字典使用哈希表作为底层实现。链地址法解决冲突的哈希表实现的。

跳跃表（skiplist）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它节点的指针，从而达到快速访问节点的目的。跳跃表通常是有序集合的底层实现之一，表中的节点按照分值大小进行排序。**

1、**由很多层结构组成；**

2、**每一层都是一个有序的链表**，排列顺序为由高层到底层，都至少包含两个链表节点，分别是前面的 head 节点和后面的 nil 节点；

3、**最底层的链表包含了所有的元素；**

4、如果一个元素出现在某一层的链表中，那么在该层之下的链表也全都会出现（上一层的元素是当前层的元素的子集）；

5、链表中的每个节点都包含两个指针，一个指向同一层的下一个链表节点，另一个指向下一层的同一个链表节点；

跳跃表的操作过程：

①、搜索：从最高层的链表节点开始，**如果比当前节点要大和比当前层的下一个节点要小，那么则往下找**，也就是和当前层的下一层的节点的下一个节点进行比较，以此类推，一直找到最底层的最后一个节点，如果找到则返回，反之则返回空。

②、插入：**首先确定插入的层数，有一种方法是假设抛一枚硬币，如果是正面就累加，直到遇见反面为止，最后记录正面的次数作为插入的层数。当确定插入的层数 k 后，则需要将新元素插入到从底层到 k 层。**

③、删除：在各个层中找到包含指定值的节点，然后将节点从链表中删除即可，如果删除以后只剩下头尾两个节点，则删除这一层。

ziplist 编码的有序集合使用紧挨在一起的压缩列表节点来保存，第一个节点保存 member，第二个保存 **score**。ziplist 内的集合元素按 score 从小到大排序，score 较小的排在表头位置。

skiplist 编码的有序集合底层是一个命名为 zset 的结构体，**而一个 zset 结构同时包含一个字典和一个跳跃表**。跳跃表**按 score 从小到大保存所有集合元素。而字典则保存着从 member 到 score 的映射，这样就可以用 O(1)的复杂度来查找 member 对应的 score 值。**虽然同时使用两种结构，但它们会通过**指针来共享相同元素的 member 和 score，因此不会浪费额外的内存。跳表也是链表的一种，只不过它在链表的基础上增加了跳跃功能，正是这个跳跃的功能，使得在查找元素时，跳表能够提供 O(logN) 的时间复杂度。**

redis 可以为 key 设置过期时间,对过期的 key,采用定期删除和惰性删除的方式。

定期**删除**：每 100ms，**随机抽取一些设置过期时间的 key,检查是否过期，**如过期，则删除。

惰性删除：一些过期的 key,并没有被定性删除删除掉，**只有当系统使用并检查其过期了**，才会将其删除。

大量 key 到内存中，达到 redis 内存最大值，会进行内存**淘汰机制**。

Redis 数据淘汰策略:**noeviction,内存达到最大值，直接返回错误。allkeys-lru:所有键里最近最少使用;**

allkeys-random:所有键随机回收；volatile-lru:设置过期时间的键中，回收最近最少使用的，volatile-random:设置过期集合键中，随机回收。

volatile-ttl:设置过期时间的键中，回收存活时间较短的键。

redis 持久化机制：redis 持久化就是将内存中的数据写入到硬盘中，为了备份重用数据。支持两种持久化方式，快照 RDB 和只追加文件 AOF;

快照 RDB,通过创建快照获得某个时间点的数据副本，可以对快照进行备份，或者复制到其他机器重用数据。默认的持久化方式。**save 900 20,在 15 分钟之后，有 20 个 key 变化，触发快照。**

AOF,实时性更好，主流方案，默认没有开启。开启后，每执行一条更改 redis 数据的命令，都将该命令写入磁盘的 AOF 中。三种 AOF 方式，always,每次修改都写入 AOF; everysec,每秒钟同步一次，多个命令一次写入。 no,操作系统决定何时同步。

RDB 优缺点：每个 RDB 都代表了某一时刻 redis 的数据，适合做冷备份，且恢复数据速度快。缺点：实时性不好，可能会丢失部分数据，如果文件过大， 可能会导致提供的服务暂停几秒。

AOF 优缺点：实时性好，一般 1s 一写入，而且写入性能比较好。缺点:AOF 文件更大，开启后 QDS 相对来说比较低，恢复速度慢。

综合 AOF 和 RDB 两种持久化方式，用 AOF 来保证数据不丢失，作为恢复数据的第一选择；用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，可以使用 RDB 进行快速的数据恢复。

redis 事务。redis 使用 MULTI、EXEC、WATCH 等命令实现事务功能。MULTI 开始事务，EXEC，执行事务。DISCARD 取消事务，WATCH 监视 key;

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。

事务可以理解为一个打包的批量执行脚本，但批量指令并非原子化的操作，中间某条指令的失败不会导致前面已做指令的回滚，也不会造成后续的指令不做。

缓存雪崩和缓存穿透

缓存雪崩，缓存在同一时间内大面积失效，所有的请求都落在数据库中，数据库短时间内承受大量请求而崩掉。

缓存同一时间内失效课可能是**服务器宕机或者设置了相同时间过期的 key 同时失效**。

事前，**尽量保证 redis 集群的高可用性，key 的过期时间尽可能错开**。

事中，本地 ehcache 缓存(先查本地缓存再查 redis,然后数据库)+hystrix 限流(每秒就接受一定量的请求)&降级(多余的请求走降级组件);

事后，利用持久化机制恢复缓存。

缓存穿透，查询一个根本不存在的数据，缓存层和数据库都不命中，失去了缓存的意义。

解决方案 1、**对空结果作缓存**，意味着设置更多的键，占用更多的内存，所以要过期时间设置很短。

2、布隆过滤器：将所有可能存在数据哈希到一个 bitMap 中，拦截不存在的数据访问，缺点有一定的误识别率。

并发竞争 key 问题，多个系统同时对一个 key 进行操作,执行顺序不同导致与期望结果不同。使用分布式锁解决。

redis 实现分布式锁。

zookeeper 实现分布式锁。

如何保证缓存和数据库双写时的**数据一致性**？

当发生数据修改时，是先更新数据库还是先更新缓存。缓存的更改是更新缓存还是直让原有缓存失效。

一般来说是先更新数据库，然后让缓存失效，将失效信息发送到 mq 中，mq 不断的重试让缓存失效，保证缓存和数据库的数据一致性。

先删除缓存，再更新数据库：A 写然后删除缓存，然而 B 查询缓存不存在，查询数据库得到旧值，然后缓存旧值。然后 A 更新数据库。这就造成了数据库和缓存不一致。这时采用延时双删策略：先删除缓存，然后更新数据库，然后休眠一段时间内如 1s，再删除缓存。淘汰这一段时间的缓存脏数据。

如果 mysql 采用了读写分离：也会造成数据不一致。A 想写，先删缓存，然后写入数据库。B 读请求，没有缓存，读取从库旧值，然后缓存旧值，数据库主从同步。还可以采用延时双删的策略，延时的时间长点，完成主从同步。

采用这种同步淘汰策略，吞吐量降低怎么办？第二步的删除作为异步，启动一个线程去删除。

第二次删除,如果删除失败怎么办？

先更新数据库，再删除缓存

缓存更新套路：

失效：应用程序先从 cache 取数据，没有得到，则从数据库中取数据，成功后，放到缓存中。

命中：应用程序从 cache 中取数据，取到后返回。

更新：先把数据存到数据库中，成功后，再让缓存失效。

缓存刚好失效，**A 做查询，得到旧值，B 做更新，让缓存失效。A 将旧值缓存。但由于 B 是更新然后让缓存失效，这一耗时明显大于 A 查询然后缓存旧值，所以出现不一致的概率很低。如果非要解决，就用延时双删，再删除一次呗。**

所以，如果删除缓存失败怎么办？

提供一个保障的重试机制即可。一、将需要删除的 key 发送给消息队列，保证其删除成功。

二、降低耦合性的方案：（1）更新数据库数据（2）数据库会将操作信息写入 **binlog 日志**当中（3）

订阅程序提取出所需要的数据以及 key（4）另起一段非业务代码，获得该信息（5）尝试删除缓存操作，发现删除失败（6）将这些信息发送至消息队列（7）重新从消息队列中获得该数据，重试操作。

redis 是单线程的，为什么还那么快？

1、操作完全基于内存，速度快。

2、数据结构简单，对数据的操作也简单。

3、采用单线程，避免了不必要的上下文切换开销。

4、使用非阻塞的多路 IO 复用模型。

I/O 多路复用模型是**利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力**，在空闲的时候，会把当前线程阻塞掉，**当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流**（epoll 是只轮询那些真正发出了事件的流），**并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。**这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。**采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快**（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了 Redis 具有很高的吞吐量。

I/O 多路复用（multiplexing）的本质是通过一种机制（系统内核缓冲 I/O 数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作。与多进程和多线程技术相比，I/O 多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

select 缺点：每次调用 select，都需要把 fd_set 集合从用户态拷贝到内核态，都需要在内核遍历传递进来的所有 fd_set，且内核对被监控的 fd_set 集合大小做了限制，为 1024。

poll 缺点：poll 没有最大文件描述符数量的限制，其他缺点和 select 一样。数据结构变成了 pollfd

epoll: 最大连接数没有限制，采用事件通知方式，每当 fd 就绪，系统注册的回调函数就会被调用，将就绪 fd 放到 readyList 里面

Unix 五种 IO 模型：阻塞 IO，网络编程中，读取客户端的数据需要调用 recvfrom。在默认情况下，这个调用会一直阻塞直到数据接收完毕，就是一个同步阻塞的 IO 方式。内核准备数据，并将数据从内核拷贝到用户内存，内核返回结果，用户进程再解除阻塞状态，重新运行起来。非阻塞 IO：用户进程调用 recvfrom 之后，如果内核数据没有准备好，并不会阻塞用户进程，而是立即返回数据未准备好的结果，用户进程以后不断调用 recvfrom 来轮询内核是否准备好数据。信号驱动 IO,调用之后，不等待数据就绪立即返回，等内核准备好数据之后，发送信号给用户进程。异步 IO，读取操作(aio_read)会通知内核进行读取操作并将数据拷贝至进程中，完事后通知进程整个操作全部完成（绑定一个回调函数处理数据）。读取操作会立刻返回，程序可以进行其它的操作，所有的读取、拷贝工作都由内核去做，做完以后通知进程，进程调用绑定的回调函数来处理数据。对比信号驱动 IO，异步 IO 的主要区别在于：信号驱动由内核告诉我们何时可以开始一个 IO 操作(数据在内核缓冲区中)，而异步 IO 则由内核通知 IO 操作何时已经完成(数据已经在用户空间中)。IO 多路复用：可以处理多个连接。这里的 select 相当于一个“代理”，调用 select 以后进程会被 select 阻塞，这时候在内核空间内 select 会监听指定的多个 datagram (如 socket 连接)，如果其中任意一个数据就绪了就返回。此时程序再进行数据读取操作，将数据拷贝至当前进程内。由于 select 可以监听多个 socket，我们可以用它来处理多个连接。

redis 热 key 问题：某个 key 被大量访问，对 redis 服务器造成了很大的压力。

解决方案：

1、服务端缓存：即将热点数据缓存至服务端的内存中，利用 ehcache ，或者一个 HashMap 都可以。在你发现热 key 以后，把热 key 加载到系统的 JVM 中。针对这种热 key 请求，会直接从 jvm 中取，而不会走到 redis 层。这个可能发生缓存和 redis 数据不一致的情况。利用 Redis 自带的消息通知机制，对于热点 Key 建立一个监听，当热点 Key 有更新操作的时候，缓存也随之更新。

2、备份热 key，即将热点 Key+随机数，随机分配至 Redis 其他节点中。这样访问热点 key 的时候就不会全部命中到一台机器上了。（Redis 集群中包含了 **16384** 个哈希槽(Hash slot)，集群使用公式 **CRC16(key) % 16384 来计算 Key 属于哪个槽。**那么同一个 Key 计算出来的值应该都是一样的，如何

将 Key 分到其他机器上呢？只要再后面加上随机数就行了，这样就能保证同一个 Key 分布在不同机器上）

这都是知道热 key 是什么的情况，那么如何发现热 key?

1、经验预估。2、Redis 自带命令查询：Redis4.0.4 版本提供了 redis-cli –hotkeys 就能找出热点 Key.

3、客户端收集：在操作 Redis 之前对数据进行统计。

redis 集群数据迁移方式

redis 集群并没有使用一致性 hash,**而是使用数据分片引入哈希槽来实现。**

redis 集群共有 16384 个哈希槽，所有的键都将映射到哈希槽中，使用 CRC16(key)%16384=哈希槽，集群按槽分片，每个节点指派不同数量的槽。

一致性 hash,是**将整个 hash 值空间 0-2^32-1 组成一个虚拟圆环，key 的哈希函数对 2^32 取模得到哈希值，在圆环上顺时针转，遇到的第一个服务器就是定位到的服务器。这样即使增加或减去一个服务器，对数据的影响比较小，数据迁移也比较简单。缺点：在某些极端情况下，可能数据扎堆分布在一个服务器上，当这个服务器出现问题，对整个系统的影响依然很大。针对分配不均的情况下，提出了虚拟节点，将服务器后面的 ip 或主机名后面增加编号，生成多个虚拟节点，分配到虚拟节点的数据实际上是分配到了该服务器上。**

普通的分布式缓存策略是：hash(obj)%N。当其中一个服务器宕机或者需要新增一个服务器时，缓存策略变为 hash(obj)%(N-1);就意味着，所有的缓存都将失效。必定会造成缓存数据的丢失，会去向后端的服务器去请求。增加删除服务器时，代价比较大，所有的数据不得不根据 id 再次计算哈希值，然后%N,进行重新分配和大规模数据迁移。

分布式与集中式：

CAP:C 一致性，A 可用性，P 分区容错性。这三个基本需求，最多能满足两个。

BASE: 基本可用，软状态，最终一致性。

分布式锁：惯用关系数据库固有的排他性实现不同进程之间的互斥，但关系数据库瓶颈所在，别啥都交给人家。

redis 的 SETNX 命令可以方便的实现分布式锁。 setNX（SET if Not eXists）+ expire;

如果是为了效率(efficiency)而使用分布式锁，允许锁的偶尔失效，那么使用单 Redis 节点的锁方案就足够了，简单而且效率高；**如果是为了正确性(correctness)在很严肃的场合使用分布式锁，那么不要使用 Redlock**；应该考虑类似 Zookeeper 的方案，或者支持事务的数据库。

1.获取当前时间戳

2.client 尝试按照顺序使用相同的 key,value 获取所有 redis 服务的锁，在获取锁的过程中的获取时间比锁过期时间短很多，这是为了不要过长时间等待已经关闭的 redis 服务。并且试着获取下一个 redis 实例。

比如：TTL 为 5s,设置获取锁最多用 1s，所以如果一秒内无法获取锁，就放弃获取这个锁，从而尝试获取下个锁

3.client 通过获取所有能获取的锁后的时间减去第一步的时间，这个时间差要小于 TTL 时间并且至少有 3 个 redis 实例成功获取锁，才算真正的获取锁成功

4.如果成功获取锁，则锁的真正有效时间是 TTL 减去第三步的时间差 的时间；比如：TTL 是 5s,获取所有锁用了 2s,则真正锁有效时间为 3s(其实应该再减去时钟漂移);

5.如果客户端由于某些原因获取锁失败，便会开始解锁所有 redis 实例；因为可能已经获取了小于 3 个锁，必须释放，否则影响其他 client 获取锁

Zookeeper 分布式锁：

竞争分布式锁，在一个节点下，创建临时序列节点，找出最小的序列节点，获取分布式锁，程序

执行完成之后此序列节点消失，通过 watch 来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理，依次类推……

架构相关

主从+哨兵：

通过持久化功能，Redis 能把内存中数据保存到硬盘上，保证了即使在服务器重启的情况下也不会损失（或少量损失）数据。数据是存储在一台服务器上的，如果这台服务器出现硬盘故障等问题，也会导致数据丢失。为了避免单点故障，通常的做法是将数据库复制多个副本以部署在不同的服务器上。Redis 提供了复制（replication）功能，可以实现当一台数据库中的数据更新后，自动将更新的数据同步到其他数据库上。在复制的概念中，数据库分为两类，一类是主数据库（master），另一类是从数据库[1] （slave）。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

当从数据库启动时，会向主数据库发送 sync 命令，主数据库接收到 sync 后开始在后台保存快照 rdb，在保存快照期间受到的命令缓存起来，当快照完成时，主数据库会将快照和缓存的命令一块发送给从数据库。从数据库依据快照和命令和主节点保持同步。之后，主每受到 1 个命令就同步发送给从数据库。

当出现断开重连后，2.8 之后的版本会将断线期间的命令传给重数据库、增量复制。

master 每次接收到写命令之后，先在内部写入数据，然后异步发送给 slave node

如果采用了主从架构，那么建议必须开启 master node 的持久化！否则重启之后，认为自己没有数据，会把从节点清空，丢失数据。

slave node 主要用来进行横向扩容，做读写分离，扩容的 slave node 可以提高读的吞吐量。

Slave node 不会设置过期 key，如果主节点一个 key 过期了，或者通过 LRU 淘汰了一个 key,发送 del 命令给从节点。

master 和 slave 都要知道各自的数据的 offset，才能知道互相之间的数据不一致的情况。

哨兵：

Redis 2.8 中提供了哨兵工具来实现自动化的系统监控和故障恢复功能。

哨兵的作用就是监控 redis 主、从数据库是否正常运行，主出现故障自动将从数据库转换为主数据库。

（1）监控主数据库和从数据库是否正常运行。

（2）主数据库出现故障时自动将从数据库转换为主数据库。

主从切换过程：

（1） slave leader 升级为 master

（2） 其他 slave 修改为新 master 的 slave

（3） 客户端修改连接

（4） 老的 master 如果重启成功，变为新 master 的 slave

1、两种数据丢失的情况

（1）异步复制导致的数据丢失，数据还没复制到 slave，master 就宕机了。

（2）脑裂导致的数据丢失。集群里就会有两个 master，

解决方法：要求至少有 1 个 slave，数据复制和同步的延迟不能超过 10 秒

如果不能继续给指定数量的 slave 发送数据，而且 slave 超过 10 秒没有给自己 ack 消息，那么就直接拒绝客户端的写请求。

哨兵：sdown 是主观宕机，就一个哨兵如果自己觉得一个 master 宕机了，那么就是主观宕机

odown 是客观宕机，如果 quorum 数量的哨兵都觉得一个 master 宕机了，那么就是客观宕机。

如果一个 master 被认为 odown 了，而且 majority 哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，

接下来会对 slave 进行排序

（1）按照 slave 优先级进行排序，slave priority 越低，优先级就越高

（2）如果 slave priority 相同，那么看 replica offset，哪个 slave 复制了越多的数据，offset 越靠后，

优先级就越高
（3）如果上面两个条件都相同，那么选择一个 run id 比较小的那个 slave
集群：
主从架构+哨兵机制虽然保证了 Redis 的高可用性，但每个 Redis 实例都是全量保存，浪费内存。为了最大化的利用内存，可以使用集群，也就是分布式存储，每台 redis 存储不同的内容。
Reids 分布式数据存储算法不是一致性哈希，而是哈希槽算法。
redis cluster 有固定的 16384 个 hash slot，对每个 key 计算 CRC16 值，然后对 16384 取模，可以获取 key 对应的 hash slot。
redis cluster 中每个 master 都会持有部分 slot，比如有 3 个 master，那么可能每个 master 持有 5000 多个 hash slot。
hash slot 让 node 的增加和移除很简单，增加一个 master，就将其他 master 的 hash slot 移动部分过去，减少一个 master，就将它的 hash slot 移动到其他 master 上去。移动成本很低。
Redis 集群集成了主从复制和哨兵的功能。集群支撑 N 个 redis master node，每个 master node 都可以挂载多个 slave node
读写分离的架构，对于每个 master 来说，写就写到 master，然后读就从 mater 对应的 slave 去读。
高可用，因为每个 master 都有 salve 节点，那么如果 mater 挂掉，redis cluster 这套机制，就会自动将某个 slave 切换成 master。redis cluster（多 master + 读写分离 + 高可用）；

在 redis cluster 架构下，每个 redis 要放开两个端口号，比如一个是 6379，另外一个就是加 10000 的端口号，比如 16379。16379 端口号是用来进行节点间通信的，也就是 cluster bus 的东西，集群总线。cluster bus 的通信，用来进行故障检测，配置更新，故障转移授权。

集群是如何判断是否有某个节点挂掉：
每一个节点都存有这个集群所有主节点以及从节点的信息。它们之间通过互相的 ping-pong 判断是否节点可以连接上。如果有一半以上的节点去 ping 一个节点的时候没有回应，集群就认为这个节点宕机了，然后去连接它的备用节点。
集群进入 fail 状态的必要条件：
1、某个主节点和所有从节点全部挂掉，我们集群就进入 faill 状态。
2、如果集群超过半数以上 master 挂掉，无论是否有 slave，集群进入 fail 状态.
3、如果集群任意 master 挂掉,且当前 master 没有 slave.集群进入 fail 状态。
主节点宕机之后，从节点发起投票，投票过程是集群中所有 master 参与。选举的依据依次是：网络连接正常->5 秒内回复过 INFO 命令->10*down-after-milliseconds 内与主连接过的->从服务器优先级->复制偏移量->运行 id 较小的。

Java 框架
Sping 相关
Spring IOC:
IOC（Inversion Of Controll，控制反转）是一种设计思想，将原本在程序中手动创建对象的控制权，交由给 Spring 框架来管理。IOC 容器是 Spring 用来实现 IOC 的载体，IOC 容器实际上就是一个 Map(key, value)，Map 中存放的是各种对象。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。IOC 容器就像是一个工厂，当需要创建一个对象，只需要配置好配置文件/注解即可，不用考虑对象是如何被创建出来的，大大增加了项目的可维护性且降低了开发难度。
IOC 就是控制反转，指创建对象控制权的转移，原本创建对象的主动权和时机由自己把控，而现在这种权利转移到 Spring 容器中。Spring 根据配置文件或注解创建实例和管理实例之间的依赖关系，在程序运行时动态的创建对象以及管理对象之间的依赖应用。Spring 的 IOC 有三种注入方式：构造器注入、setter 方法注入、接口注入。

Spring AOP:

AOP（Aspect-Oriented Programming，面向切面编程）能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可扩展性和可维护性。使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样可以大大简化代码量，提高了系统的扩展性。

Spring AOP 是基于动态代理的，如果要代理的对象实现了某个接口，那么 Spring AOP 就会使用 JDK 动态代理去创建代理对象；而对于没有实现接口的对象，就无法使用 JDK 动态代理，转而使用 CGlib 动态代理生成一个被代理对象的子类来作为代理。

Spring AOP / AspectJ AOP 的区别？

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。

Spring AOP 基于代理（Proxying），而 AspectJ 基于字节码操作（Bytecode Manipulation）。

AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单。如果切面比较少，那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ，它比 SpringAOP 快很多。

AOP 相关概念：

1.通知（Advice）

就是你想要的功能，也就是上面说的安全，事物，日志等。

2.连接点（JoinPoint）

spring 允许你使用通知的地方，spring 只支持方法连接点.只要记住，和方法有关的前前后后（抛出异常），都是连接点。

3.切入点（Pointcut）

在连接点中选取几个点作为切入点，真正的放入通知。让切点来筛选连接点，选中那几个你想要的方法。

4.切面（Aspect）

切面是通知和切入点的结合。通知说明了干什么和什么时候干（什么时候通过方法名中的 before,after，around 等就能知道），而切入点说明了在哪干（指定到底是哪个方法），这就是一个完整的切面定义。

5.引入（introduction）

允许我们向现有的类添加新方法属性。把切面引用到新方法。

6.目标（target）

引入中所提到的目标类，也就是要被通知的对象，也就是真正的业务逻辑。

7.代理(proxy)

怎么实现整套 aop 机制的，都是通过代理，这个一会给细说。

8.织入(weaving)

把切面应用到目标对象来创建新的代理对象的过程。

前置通知[Before advice]：在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常。
正常返回通知[After returning advice]：在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行。
异常返回通知[After throwing advice]：在连接点抛出异常后执行。
返回通知[After (finally) advice]：在连接点执行完成后执行，不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容。
环绕通知[Around advice]：环绕通知围绕在连接点前后，比如一个方法调用的前后。这是最强大的通知类型，能在方法调用前后自定义一些操作。环绕通知还需要负责决定是继续处理 join point(调用 ProceedingJoinPoint 的 proceed 方法)还是中断执行。

Bean 的生命周期：

1、根据配置情况调用 Bean 构造方法或者工厂方法实例化 bean 对象;

2、利用依赖注入完成 Bean 中所有属性值的配置注入。

3、如果 Bean 实现了 BeanNameAware 接口，则调用 Bean 的 setBeanName()方法传递 Bean 的 ID。

4、如果 Bean 实现了 BeanFactoryAware 接口，则调用 setBeanFactory()方法传入当前工厂实例的引用。

5、如果 Bean 实现了 ApplicationContextAware 接口，则调用 setApplicationContex ()方法传入当前 ApplicationContex 的引用。

6、如果有 BeanPostProcessor 和 Bean 关联，则调用该接口的预初始化方法 postProcessBeforeInitialzation()bean 进行加工操作。

7、如果 bean 实现了 InitializingBean 接口，将调用 afterPropertiesSet()方法。

8、如果 Bean 指定了 init-method 方法，它将被调用。

9、如果有 BeanPsotProcessor 和 Bean 关联，则该接口的后置初始化方法 postProcessAfterInitialization() 方法将被调用。

10、如果 bean 的作用域 scope="prototype",则调用者管理该 bean 的生命周期。如果作用范围为单例模型，将 bean 放入 IOC 缓存池，触发 spring 对该 bean 的生命周期管理。

11、程序使用 bean

12、如果 Bean 实现了 DisposableBean 接口，Spring 会调用 destory()方法将 bean 销毁。如果配置文件中指定了 destory-method，那将调用该方法销毁 bean.

Spring 中的 **bean 的作用域**有哪些?

singleton:[ˈsɪŋgltən]单例模式，在整个 Spring IoC 容器中，只有一个 bean 实例，默认都是单例。

prototype:原型模式，每次从容器中请求 bean,都会产生一个新的 bean 实例。

request:每次 HTTP 请求，都会创建一个新的 bean;

session:同一个 HTTP session 共享一个 bean,不同的 session 使用不同的 bean.

globalSession:同一个全局 Session 共享一个 bean. <后面这三个只用应用到 web 应用时，作用域才有效>

Spring MVC 中的 controller 是默认单例的，为啥还线程安全？

因为 controller 基本不定义属性(成员变量)，**属于无状态 Bean，线程安全**。如果定义属性的话，最好 scope=prototype;

Controller、Service、Dao 等，这些 Bean 大多是无状态的，只关注于方法本身;

对于有状态的 bean，Spring 官方提供的 bean，一般提供了通过 ThreadLocal 去解决线程安全的方法,比如 RequestContextHolder; 它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。

spring 的事务管理方式 1、编程式事务，代码中硬编码 2、声明式事务。推荐使用。

声明式事务，有基于 xml 的声明式事务，基于注解的声明式事务。

Spring 的事务隔离级别:5 种

1:TransactionDefinition.ISOLATION_**DEFAULT**: 后端数据库默认隔离级别.Mysql-REPEATABLE_READ。Oracle-READ_COMMITTED;

2:ISOLATION_**READ_UNCOMMITTED**:读未提交，允许读取未提交的数据变更。可能导致脏读、幻读或不可重复读。

3:ISOLATION_**READ_COMMITTED**:读已提交，允许读取并发事务已经提交的数据。阻止脏读，可能导致幻读，或不可重复读。

4:ISOLATION_**REPEATABLE_READ**:可重复读：可重复读，对同一字段的读取结果是一致的。可能发生幻读。

5:ISOLATION_**SERIALIZABLE**:串行化，事务依次执行，不并发，但影响性能。

Spring 的事务传播行为：事务传播行为（propagation behavior）**指的就是当一个事务方法被另一个事务方法调用时，这个事务方法应该如何进行。**

支持当前事务的情况：

PROPAGATION_REQUIRED：**当前存在事务就加入事务**，没有就创建。

PROPAGATION_SUPPORTS： **当前存在事务就加入事务**，没有就以非事务的方式继续运行。

PROPAGATION_MANDATORY：**当前存在事务就加入事务**，没有就抛出异常。

不支持当前事务的情况：

PROPAGATION_REQUIRES_NEW：创建一个新事务，**如果当前存在事务，把当前事务挂起。**

PROPAGATION_NOT_SUPPORTED：以非事务的情况继续运行，**如果当前存在事务，挂起当前事务**。

PROPAGATION_NEVER：以非事务的方式运行，**如果当前存在事务，抛出异常**

PROPAGATION_NESTED： **如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行**；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED,**创建一个事务自己运行？**

将一个类声明 spring bean 的注解有哪些;

将类标识成可用于**@Autowired 注解自动装配的 bean,**注解的方式有：

1、**@component**:通用注解，不知道是哪一层就用这个。

2、**@Repository**：对应持久层，主要用于数据库相关操作。

3、**@Service**:对应服务层，主要涉及操作持久层，进行逻辑;

4、**@Controller**：对应控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面；

@Component 和 @Bean 的区别是什么？

两者的目的一样，都是注册 bean 到 Spring 容器中

1、作用对象不同: **@Component 注解作用于类，而@Bean 注解作用于方法。**

2、@Component 注解表明一个类会作为**组件类，并告知 Spring 要为这个类创建 bean。**

@Bean 注解告诉 Spring 这个方法将会返回一个对象，这个对象要注册为 Spring 应用上下文中的 bean。通常方法体中包含了最终产生 bean 实例的逻辑。

3、**@Bean 注解比 Component 注解的自定义性更强。**

Spring MVC 的理解

Model1 时代：jsp+javaBean,jsp 既是控制层又是表现层， 前后端依赖严重，控制逻辑和表现逻辑混杂，代码重用率第，开发效率低。

Model2 时代：JavaBean+JSP+Servlet,这就是早期的 JavaWeb MVC 开发模式,但抽象和封装程度还远远不够，程度可维护性和复用性低。

Spring MVC 可以帮助我们进行更简洁的 Web 层的开发,天然与 Spring 集成，一般把后端项目分为 Service 层（处理业务）、Dao 层（数据库操作）、Entity 层（实体类）、Controller 层(控制层，返回数据给前台页面)。

Spring MVC 工作原理：

（1）客户端（浏览器）发送请求，直接请求到 **DispatcherServlet**。(前端控制器本质是一个 servlet,是 SPringMVC 提供的所有请求的的统一入口)

（2）DispatcherServlet 根据请求信息**调用 HandlerMapping，解析请求对应的 Handler。**

（3）解析到对应的 Handler 后，开始由 HandlerAdapter 适配器处理。

（4）**HandlerAdapter 会根据 Handler 来调用真正的处理器开处理请求，并处理相应的业务逻辑**。

（5）处理器处理完业务后，会返回一个 **ModelAndView 对象，Model 是返回的数据对象，View 是个逻辑上的 View。**

（6）**ViewResolver 会根据逻辑 View 查找实际的 View。**

（7）**DispaterServlet 把返回的 Model 传给 View。**

（8）通过 View 返回给请求者（浏览器）

DispatcherServlet,前端控制器本质是一个 servlet,是 SPringMVC 提供的所有请求的的统一入口;

mybatis 中$#的区别，传表名用哪个

#{} 这种取值是编译好 SQL 语句再取值，解析成一个参数标记符?

${}会直接进行字符串替换。

#{}是编译好 SQL 中有一个占位符，然后将参数值填充到 SQL

所以：

#{}方式能够很大程度防止 sql 注入。

$方式无法防止 Sql 注入。
$方式一般用于传入数据库对象，例如传入表名.
一般能用#的就别用$.

常用的 spring 注解
@Component , @Repository , @ Controller ,@Service , @Configration，@Bean　注册类
@Autowired ，　@Resource 默认安照名称进行装配，名称可以通过 name 属性进行指定，
如果没有指定 name 属性，当注解写在字段上时，默认取字段名进行按照名称查找
@autowired 和@resource 的区别，一个接口有两个类实现时怎么指定注入哪一个
@Autowired 按 **byType** 自动注入，而**@Resource 默认按 byName 自动注入罢了**;
如果当 Spring 上下文中存在不止一个 UserDao 类型的 bean 时，就会抛出 **BeanCreationException** 异常;如果 Spring 上下文中不存在 UserDao 类型的 bean，也会抛出 **BeanCreationException** 异常。我们可以使用**@Qualifier 配合@Autowired** 来解决这些问题.
SpringMVC 注解：
@RequestMapping、
@PathVariable：用于将请求 URL 中的模板变量映射到功能处理方法的参数上
@ResponseBody ：
@RequestHeader：
@RequestParam

Java 基础知识
ThreadLoacl
ThreadLocal 类用来提供线程内部的局部变量，这种变量在多线程环境下访问时能保证各个线程里的变量相对独立于其他线程内的变量。ThreadLocal 实例通常来说都是 private static 类型，用于关联线程。static 的 ThreadLocal 变量是一个与线程相关的静态变量，即一个线程内，static 变量是被各个实例共同引用的，但是不同线程内，static 变量是隔开的。
使用方法：一个 class 中定义了一个 ThreadLocal 变量。通过重写方法 initialValue()初始化值。然后设置 get()方法得到 threadLocal.get();通过设置 set()方法设置 threadLoca.set(threadLocal.get()+10);然后一个线程任务对这个类的变量进行修改或者查找时，每个线程的变量使用的都是线程局部变量，互不影响。

每个线程对象：Thread t = Thread.currentThread();每个线程对象都有自己的 ThreadLocalMap 的变量，Map 变量，key 值为 ThreadLocal，value 为值。所以每个线程都有自己的一个 ThreadLocal 变量，且各自的值独立。
Get 的时候，先获取线程的 ThreadLocalMap 对象，如果存在的话，就获取他的值。
Set 的时候，先获取线程的 ThreadLocalMap 对象，如果存在，就更新。不存在，就新建 Map,key 为 ThreadLocal,值为 value;
总的来说，每个线程对象，都有自己的 ThreadLocalMap 变量，这个 map 变量，key 就是自己维持的 ThreadLocal 对象,值为 ThreadLocal 的值。
因为线程必然要访问 threadLocal 变量，然后调用 threadLocal.get()方法，这个方法的实现就是：先获取当前线程，然后获取当前线程的 ThreadLocalMap 对象，map 对象的 key 就是维持的 ThreadLocal 对象，value 就是变量的值。

```
public T get() {
        Thread t = Thread.currentThread();//获取当前线程
        ThreadLocalMap map = getMap(t);//获取当前线程的 ThreadLocalMap 对象
        if (map != null) {
            ThreadLocalMap.Entry e = map.getEntry(this); //当前线程 Map 对象，是否含有此TreadLocal 对象实例最为 key,如果有，取值。
                if (e != null) {
```

```
                @SuppressWarnings("unchecked")
                T result = (T)e.value;
                return result;
            }
        }
        return setInitialValue();
    }
```

内存泄露：

单纯将 ThreadLocal 置为 null，ThreadLoca 只有一个指向 Map 的 key 的虚引用，是可以回收的；但是，从 CurrentThread-----Thread------ThreadLocalMap----Entry 的 value 是存在强引用的。这时，如果线程对象不被回收，例如线程池，就可能出现内存泄露了。所以我们 Threadlocal 的时候，要用 remove() 方法。

内存泄露的场景：

使用静态的集合类：静态的集合类的生命周期和应用程序的生命周期一样长；

单例模式可能会造成内存泄露：实例对象的生命周期和应用程序的生命周期一样长，如果单例对象中拥有另一个对象的引用的话，这个被引用的对象就不能被及时回收；

数据库、网络、输入输出流，这些资源没有显示的关闭

使用非静态内部类：非静态内部类对象的构建依赖于其外部类，内部类对象会持有外部类对象的 this 引用，即时外部类对象不再被使用了，其占用的内存可能不会被 GC 回收，因为内部类的生命周期可能比外部类的生命周期要长，从而造成外部类对象不能被及时回收。解决办法是尽量使用静态内部类，

变量不合理的作用域：能声明局部变量，就不要整成成员变量。

反射

定义：在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法，这种动态获取信息以及动态调用对象方法的功能成为 Java 反射机制。

提供的功能：在运行时判断任意一个对象所属的类；在运行时构造任意一个类的对象；在运行时判断任意一个类所具有的成员变量和方法；在运行时调用任意一个对象的方法；

原理：获取类的 Class 对象，然后反向获取类或对象的各种信息。（class 对象和.class 文件的构造？？？）

获取 class 对象的三种方法：Class.forName("全类名")；类名.class；对象.getClass()；

然后根据 class 对象获取类的各种信息 ：getMethods()获取所有 public 方法，getConstructors()获取构造方法，获取 getDeclaredFields()获取所有的属性。class.newInstance() ,并强转类型，然后就可以操作对象了。对私有方法和属性，可以 setAccessible(true)，然后进行操作。所以反射是能够改变单例模式的，只需要获取私有的构造方法，然后 setAccessible(true),然后通过构造方法创建对象就行。但并不破坏封装。封装的含义和单例的含义不一样，封装是将一组程序封装起来，对外提供接口，让外部调用人员不必在意里面的实现环节，直接调用即可。而外部人员及时通过反射获取到了里面的一些私有，仍然是不能窥探其实现逻辑的，单纯的调用其私有方法， 没有任何的意义。

反射的应用场景：

我们在使用 **JDBC 连接数据库时使用 Class.forName()通过反射加载数据库的驱动程序；**②Spring 框架也用到很多反射机制，最经典的就是 xml 的配置模式。**Spring 通过 XML 配置模式装载 Bean 的过程**：1) 将程序内所有 XML 或 Properties 配置文件加载入内存中; 2)Java 类里面解析 xml 或 properties 里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)**使用反射机制，根据这个字符串获得某个类的 Class 实例; 4)动态配置实例的属性。**

String 是怎么扩容的（不知道，后来查的：2 倍扩容，超出 1M 后每次扩大 1M）

为什么会出现 4.0-3.6=0.40000001 这种现象？

2 进制的小数无法精确的表达 10 进制小数，计算机在计算 10 进制小数的过程中要先转换为 2 进制进行计算，这个过程中出现了误差。

面向对象特性

抽象：就是把现实生活中的某一类东西提取出来，用程序代码表示，我们通常叫做类或者接口。抽象包括两个方面：一个是数据抽象，一个是过程抽象。数据抽象也就是对象的属性。过程抽象是对象的行为特征。

封装：把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行封装隐藏。封装分为属性的封装和方法的封装。核心思想就是“隐藏细节”、“数据安全”：将对象不需要让外界访问的成员变量和方法私有化，只提供符合开发者意愿的公有方法来访问这些数据和逻辑，保证了数据的安全和程序的稳定。好处：良好的封装能减少耦合，对成员变量更精确的控制。

继承：子类可以继承父类的非私有方法和属性(默认属性和方法也不行)，达到复用代码的效果。Java 是单继承，一个类只能继承一个父类。

多态：**不同类的对象对同一消息作出不同的响应叫做多态。同一消息可以根据发送对象的不同而采用多种不同的行为方式。**可以用于消除类型之间的耦合关系，Spring 的核心就是多态和面向接口编程。多态的分类，编译时多态，方法的重载，运行时多态，方法的覆盖。**多态存在的条件：存在继承关系，子类重写父类的方法，父类引用指向子类。**

Java 里对象方法的调用是依靠类信息里的方法表实现的。

重载：**发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。**

重写：发生在父子类中，方法名、参数列表必须相同，返回值小于等于父类，抛出的异常小于等于父类，访问修饰符大于等于父类；如果父类方法访问修饰符为 private 则子类中就不是重写。

抽象类和接口：

抽象类可以有构造函数，接口不可以有构造函数

抽象类中可以有普通成员变量，接口中没有普通成员变量，只能有常量

抽象类中的方法可以被 static 修饰，接口中的方法不可以被 static 修饰 **---> jdk 1.8 可以接口可以有 static 修饰的方法**

抽象类中可以有普通方法和抽象方法，接口中的方法全是抽象方法 ---> jdk 1.8 接口中的方法还有 **default 和 static 修饰的方法**

一个类只能继承一个抽象类，接口可以被多实现，即一**个类只能继承一个类，可以实现多个接口**

何时使用

接口主要用于模块与模块之间的调用。主要用接口来实现多继承，因为 java 不支持类的多继承，只能用接口

抽象类主要用于当做基础类使用，即基类。如果想拥有一些方法，并且这些方法有默认实现，那么使用抽象类

1.如果要实现的类和该抽象类是同一类事物，则用继承抽象类的方法；不是同一类事物，就用接口；2.要设计较小的功能模块，用接口，要设计较大的功能单元，用抽象类；3.如果大部分方法都不确定，用接口抽象所有方法；如果只有少部分方法不确定，另有部分方法是确定的，用抽象类，实现部分确定的方法，抽象部分不确定的方法。

语法层次

抽象类和接口分别给出了不同的语法定义。

抽象类中可以有普通方法和抽象方法 public abstract，接口中的方法全是抽象方法，抽象类中可以有普通成员变量，接口

中没有普通成员变量，只能有常量 public static final。
一个重要的原则：当一个类实现了一个接口，要求该类把这个接口的所有方法全部实现
接口还可以继承多个接口。

设计层次
抽象层次不同，抽象类是对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。

跨域不同
抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a"关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的，仅仅是实现了接口定义的契约而已，"like-a"的关系。

泛型
泛型，即"参数化类型"。
创建集合时就指定集合元素的类型，该集合只能保存其指定类型的元素，避免使用强制类型转换。

Java 编译器生成的字节码是不包涵泛型信息的，泛型类型信息将在编译处理是被擦除，这个过程即类型擦除。泛型擦除可以简单的理解为将泛型 java 代码转换为普通 java 代码，只不过编译器更直接点，将泛型 java 代码直接转换成普通 java 字节码。
类型擦除的主要过程如下：
1）将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。
2）移除所有的类型参数。

非静态内部类：

1. 内部类对象的创建依赖于外部类对象；
2. 内部类对象持有指向外部类对象的引用。(静态内部类则不持有)；

Java 和 c++的区别？
static 关键字和 final 关键字
static 关键字：**static 的主要作用在于创建独立于具体对象的域变量或者方法**。类变量随着类的加载而存在于方法区中。实例变量随着对象的建立而存在于堆内存中。类变量生命周期最长，随着类的消失而消失。实例变量生命周期随着对象的消失而消失。

static 修饰方法：静态方法。不依赖于对象就可以访问，可以直接类名.静态方法访问。静态方法不可以访问对象的非静态方法和非静态变量。

static 变量：静态变量被所有的对象所共享，在内存中只有一个副本，存在方法区中，当且仅当在类初次加载时会被初始化。

static 静态代码块：在类被初次加载时执行，且执行一次，通常将只需要进行一次的初始化操作都放在 static 代码块中进行。

static 静态内部类:内部类：定义在类内部的类叫做内部类，内部类持有外部类的引用，所以能够访问外部类的所有变量和方法，内部类一般只为外部类使用，且内部类能够独立的继承接口。外部类对象通过'外部类名.this.xxx'的形式访问内部类的属性与方法。static 修饰之后就叫做静态内部类，或嵌套类 [1]要创建静态内部类的对象，并不需要其外部类的对象；也没有持有外部类的引用。 [2]不能够从静态内部类的对象中访问外部类的非静态成员。

每个类会产生一个.class 文件，文件名即为类名。同样，内部类也会产生这么一个.class 文件。内部类也会有 class 文件，文件名为外部类$内部类名称。

static 静态导包：import static myClass.*;导入类的所有静态　方法或者调用特定静态方法。使用时直接使用方法名称就可以。

final:

final 修饰变量表示常量，只能被赋值一次，赋值后值不再改变.这里的不可变，是指引用不可变，如果修饰 list,随便增删。如果修饰的类的成员变量，必须显示初始化，要么直接赋值，要么构造方法中初始化。否则就会报编译错误

final 修饰方法，表示方法不可被重写。final 方法编译的时候静态绑定，所以比非 final 方法快。private 方法其实也是 final 的。final 方法在编译阶段绑定，称为静态绑定(static binding)。

final 修饰类，表示不可被继承。final 类中的所有方法都是 final 方法。

不用 final 还可以用什么办法使得这个类不被继承、

将我们的类的构造器声明为 private 类型的。然后继承的子类，必须得调用父类的构造方法， 因为他是私有的构造函数，不能调用，出错。

重排序情况：编译器优化的重排序、指令并行重排序，内存系统重排序。

在 JMM 中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须存在 happens-before 关系。

1. **如果一个操作 happens-before 另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。**

2. 两个操作之间存在 happens-before 关系，并不意味着一定要按照 happens-before 原则制定的顺序来执行。**如果重排序之后的执行结果与按照 happens-before 关系来执行的结果一致，那么这种重排序并不非法。**

规则八种：程序次序规则（**在一个线程内一段代码的执行结果是有序的**。就是还会指令重排，但是随便它怎么排，结果是按照我们代码的顺序生成的不会变！）、锁定规则、volatile 变量规则、传递规则、线程启动规则、线程中断规则、线程终结规则、对象终结规则。

ArrayList 和 LinkedList 区别？三种方式访问集合中的元素，ArrayList 遍历删除会出的问题？Foreach 原理以及 Fail-fast 机制！

为什么会出现 4.0-3.6=0.40000001 这种现象？

为什么 HashMap 长度大于 8 才转换为红黑树，而不是 6？

HashMap 和 TreeMap 比较？

任务缓存队列及排队策略，如何自定义拒绝策略？

配置线程池大小，根据 CPU 密集和 IO 密集划分

阻塞队列以及生产者消费者的实现

设计模式的单例和工厂是面得最多的

为什么需要破坏双亲委派？自己写个 String 能加载吗？类初始化时机？

JVM 调优

Redis 的缺点？

Redis 的并发竞争问题如何解决？

集群是如何判断是否有某个节点挂掉？集群进入 fail 状态的必要条件？

减库存然后下订单，但是服务器宕机？

负载均衡实现方式，策略？

单拎出来的缓存问题，结合状态码 304 可能问到，相关头字段，**If-Modified-Since 和 Last-Modified，If-None-Match 和 ETag，它们的区别等；**

有序性的先行发生原则（happens-before）。

虚引用有哪些应用场景

介绍下 JDBC 的过程

JDBC 的 Statement 对象有哪几类

多态的实现原理

多态是指一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

Java 多态允许父类引用变量指向子类对象，引用变量发出的方法调用到底是哪个对象实现的方法，

必须在由程序运行期间才能决定。java 中的方法调用有静态绑定和动态绑定之分，静态绑定指的是我们在编译期就已经确定了会执行那个方法的字节码，而动态绑定只有在运行时才能知晓。Java 中的静态方法、私有方法以及 final 修饰的方法的调用，都属于静态绑定，对于重载的实例方法的调用，也是采用静态绑定。方法调用动作会被编译成静态调用指令，该指令对应常量池中方法的符号引用。

Java 对于方法调用动态绑定的实现主要依赖于**方法表**，但通过**类引用调用和接口引用调用**的实现则有所不同。总体而言，当某个方法被调用时，JVM 首先要查找相应的**常量池**，得到**方法的符号引用**，并查找调用类的方法表以确定该**方法的直接引用**，最后才真正调用该方法。以下分别对该过程中涉及到的相关部分做详细介绍。

动态绑定：JVM 有个方法表：记录当前类以及所有父类的可见方法字节码在内存中的直接地址。

异常体系

异常也是一种对象，java 当中定义了许多异常类，并且定义了基类 java.lang.**Throwable 作为所有异常的超类**。Java 语言设计者将异常划分为两类：**Error 和 Exception**。

1、Error（错误）：是程序中无法处理的错误，表示运行应用程序中出现了严重的错误。此类错误一般表示代码运行时 JVM 出现问题。通常有 Virtual MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）等。**StackOverflowError，OutOfMemoryError。**

2、Exception（异常）：**程序本身可以捕获并且可以处理的异常。**

Exception 这种异常又分为两类：运行时异常和编译异常。

1、运行时异常**(免检异常)**：RuntimeException 类极其子类表示 JVM 在运行期间可能出现的错误。比如说试图使用空值对象的引用（**NullPointerException**）、数组下标越界（**ArrayIndexOutBoundException**）。此类异常属于不可查异常，一般是由**程序逻辑错误引起的，在程序中可以选择捕获处理，也可以不处理。ArithmeticException，ClassCastException，NumberFormatException，IllegalArgumentException。**

2、编译异常**(受检异常)**：Exception 中除 RuntimeException 极其子类之外的异常。如果程序中出现此类异常，比如说 **IOException，必须对该异常进行处理，否则编译不通过。**在程序中，通常不会自定义该类异常，而是直接使用系统提供的异常类。**ClassNotFoundException，InterruptedException**。

排序：快速，归并，堆排序

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	O(n2)O(n2)	O(n2)O(n2)	O(n)O(n)	O(1)O(1)	稳定	简单
希尔排序	O(nlog2n)O(nlog2n)	O(n2)O(n2)	O(n)O(n)	O(1)O(1)	不稳定	较复杂
直接选择排序	O(n2)O(n2)	O(n2)O(n2)	O(n2)O(n2)	O(1)O(1)	不稳定	简单
堆排序	O(nlog2n)O(nlog2n)	O(nlog2n)O(nlog2n)	O(nlog2n)O(nlog2n)	O(1)O(1)	不稳定	较复杂
冒泡排序	O(n2)O(n2)	O(n2)O(n2)	O(n)O(n)	O(1)O(1)	稳定	简单
快速排序	O(nlog2n)O(nlog2n)	O(n2)O(n2)	O(nlog2n)O(nlog2n)	O(nlog2n)O(nlog2n)	不稳定	较复杂
归并排序	O(nlog2n)O(nlog2n)	O(nlog2n)O(nlog2n)	O(nlog2n)O(nlog2n)	O(n)O(n)	稳定	较复杂
基数排序	O(d(n+r))O(d(n+r))	O(d(n+r))O(d(n+r))	O(d(n+r))O(d(n+r))	O(n+r)O(n+r)	稳定	较复杂

//快速排序,快速排序是不稳定的，其时间平均时间复杂度是 O(nlgn)。

// 快速排序、堆排序、希尔排序等时间性能较好的排序方法都是不稳定的。稳定性需要根据具体需求选择。

// 先从序列中取出一个数作为基准数；先选择数组的最左边;

//分区过程：将把这个数大的数全部放到它的右边，小于或者等于它的数全放到它的左边；

//递归地对左右子序列进行不走 2，直到各区间只有一个数

```
private    static void quickStart(int[] arr,int left, int right){
        if(arr == null || left >= right || arr.length < 1)
        return;
        int mid = partition(arr,left,right);
        quickStart(arr,left,mid-1);
        quickStart(arr,mid+1,right);
    }
    private static int partition(int[] arr, int left, int right){
        int pivotKey = arr[left];

        while(left < right) {
            while(left < right && arr[right] >= pivotKey)
                right --;
            arr[left] = arr[right]; //把小的移动到左边
            while(left < right && arr[left] <= pivotKey)
                left ++;
            arr[right] = arr[left]; //把大的移动到右边
        }
        arr[left] = pivotKey; //最后把 pivot 赋值到中间
        return left;
```

```
    }

    private static void swap1(int[] arr, int left, int rihgt){
        int temp = arr[left];
        arr[left] = arr[rihgt];
        arr[rihgt] = temp;
    }

    //归并排序：
    //二路归并排序，让数组分成 2 组，然后递归多分几组，知道每组只有一个，然后对每 2 有序小组进行排序合并。
    //归时，要创建新数组，暂存结果
    public static int[] sortMerge(int[] array,int left,int right){
        int mid = left + (right-left)/2;
        if (left < right){
            sortMerge(array,left,mid);
            sortMerge(array,mid+1,right);
            merge(array,left,mid,right);
        }
        return array;
    }

    //将有序的 left-mid 和 mid+1 --- right 这两个有序数组排序;
    public static void merge(int[] array,int left,int mid,int right){
        //String s = "sdd";
        //s.indexOf()
        int[] temp = new int[right-left+1];
        int i = left;
        int j = mid +1;
        int k = 0;
            while (i<=mid && j <= right){
                if (array[i] < array[j])
                    temp[k++] = array[i++];
                else
                    temp[k++] = array[j++];
        }
        while (i <= mid)
            temp[k++] = array[i++];
        while (j<=right)
            temp[k++] = array[j++];

        for(int x=0;x<temp.length;x++){
            array[x+left] = temp[x];
        }
    }
```

序列化和反序列化：

java 序列化是指把 java 对象转换为字节序列的过程，而 java 反序列化是指把字节序列恢复为 java 对象的过程。

序列化是把对象转换成有序字节流，以便在网络上传输或者保存在本地文件中。反序列化：客户

端从文件中或网络上获得序列化后的对象字节流后，根据字节流中所保存的对象状态及描述信息，通过反序列化重建对象。

（利用序列化实现远程通信，可以在网络上传送对象的字节序列。在进程间传递对象，永久性保存对象）

----JDK 类库中序列化的步骤 ：只有实现了 Serializable 或 Externalizable 接口的对象才能被序列化，否则抛出异常！

如果类 a 仅仅实现了 Serializable 接口，则

ObjectOutputStream 采用默认的序列化方式，对 a 对象的非 transient 实例变量进行序列化

ObjectInputStream 采用默认的反序列化方式，对 a 对象的非 transient 实例变量进行反序列化

反序列化失败: 没有添加 serialVersionUID 可能会导致反序列化失败.

继承了一个已经实现序列化接口的父类,并且与父类有重复的属性,在反序列化的时候就会导致重复的属性数据丢失.

--1--创建一个对象输出流，它可以包装一个奇特类型的目标输出流，如文件输出流：

objectOutputStream oos=new objectOutputStream(new FileOutStream(c:\\object.out));

--2--通过对象输出流 writeObject()方法写对象：

oos.writeObject(new a("xiaoxiao","145263","female"));

----JDK 类库中反序列化的步骤

--1--创建一个对象输入流，它可以包装一个其他类型输入流，如文件输入流：

objectInputStream ois=new ObjectInputStream(new FileInputStream("object.out"));

--2--通过对象输出流的 readObject()方法读取对象：

a aa=(a)ois.readObject();

--3--为了正确读数据，完成反序列化，必须保证向对象输出流写对象的顺序与从对象输入流中读对象的顺序一致

int 和 Integer 有什么区别

为了编程方法， Java 有八大基本数据类型。又为了能将这些基本数据类型当做对象处理，又为每个基本数据类型引入了对应的包装类型。JDK1.5 之后，引入自动拆箱机制，基本数据类型和包装类可以互换。

Integer 是 int 的包装类，必须实例化为才能使用，Integer 变量实际是对象的引用，指向对象，默认值为 null;

int 是基本数据类型,不必实例化就能用，它直接存数据值，默认是 0.

Integer 变量和 int 变量比较时，只要两个变量的值是向等的，则结果为 true。因为自动拆箱。

非 new 生成的 Integer 变量和 new Integer()生成的变量比较时，结果为 false。

对于两个非 new 生成的 Integer 对象，进行比较时，如果两个变量的值在区间-128 到 127 之间，则比较结果为 true，如果两个变量的值不在此区间，则比较结果为 false。

Integer 缓存池大小：-128 ----127

RabbitMQ:

1、如何保证消息的可靠性传输（如何处理消息丢失的问题）？

1）生产者弄丢了数据：可以开启 confirm 模式，每次写的消息都会分配一个唯一的 id，然后如果写入了 rabbitmq 中，rabbitmq 会给你回传一个 ack 消息，告诉你说这个消息 ok 了。如果 rabbitmq 没能处理这个消息，会回调你一个 nack 接口，告诉你这个消息接收失败，你可以重试。**confirm 机制是异步的。**

2）rabbitmq 弄丢了数据，**开启 rabbitmq 的持久化**，就是消息写入之后会**持久化到磁盘**。**创建 queue 的时候将其设置为持久化的，发送消息的时候将**

消息的 deliveryMode 设置为 2，就是将消息设置为持久化的，此时 rabbitmq 就会将消息持久化到磁盘上去。而且持久化**可以跟生产者那边的 confirm 机制配合起来**，只有消息被持久化到磁盘之后，才

会通知生产者 ack 了。

3）消费端弄丢了数据：手动调用 rabbitmq 提供的 ack 机制。每次你自己代码里确保处理完的时候，再程序里 ack 一把。这样的话，如果你还没处理完，不就没有 ack？那 rabbitmq 就认为你还没处理完，这个时候 rabbitmq 会把这个消费分配给别的 consumer 去处理。

2、保证消息是有顺序的；

拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

3、消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？

可能你的消费端出了问题，不消费了。

先修复 consumer 的问题，确保其恢复消费速度，临时紧急扩容，将 queue 资源和 consumer 资源扩大，增加消费能力，等快速消费完积压数据之后，得恢复原先部署架构。

rabbitmq，rabbitmq 是可以设置过期时间的，就是 TTL，大量积压在 mq 里，而是大量的数据会直接搞丢。批量重导，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入 mq 里面去，把白天丢的数据给他补回来。

4、如何保证消息不被重复消费啊

保证消息队列消费的幂等性就不怕，重复消费无所谓。主要是结合业务来谈：如写数据库，先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。如果是写 redis，那没问题了，反正每次都是 set，天然幂等性。如果是下订单，让生产者发送每条数据的时候，里面加一个全局唯一的 id，根据 ID 查询是否消费过了，如果已经处理，就忽视。

5、RabbitMQ 高可用性：

rabbitmq 有三种模式：单机模式，普通集群模式，镜像集群模式。单击模式，一个节点。普通集群，多台机器上启动多个 rabbitmq 实例，每个机器启动一个。但是你创建的 queue，只会放在一个 rabbtimq 实例上，但是每个实例都同步 queue 的元数据。实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。没做到所谓的分布式。**镜像集群模式：你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，然后每次你写消息到 queue 的时候，都会自动把消息到多个实例的 queue 里进行消息同步。一，这个性能开销也太大了吧，消息同步所有机器，导致网络带宽压力和消耗很重！第二，这么玩儿，就没有扩展性可言了。**

项目相关

秒杀项目的面试问题：

秒杀项目介绍：

1：这个项目主要是针对处理高并发问题。主要考虑的问题有以下几个方面：

一、正确性。核心问题就是防止超卖，和重复下单。

二、高并发。主要是采用 Redis 进行缓存常用查询、消息队列异步下单、页面资源静态化等方面减去数据库压力。

三、安全性。主要有动态地址生成和接口放刷，双重 MD5 加密密码。

四、高可用性。一方面使用 Redis 集群的主从复制和主从切换保证 redis 的高可用性，另一方面，为防止 redsi 服务器宕机，使用限流来防止 mysql 承受过多的请求。

第二个项目介绍：

这个项目是我们实验室的项目，是和军方合作的项目。项目的流程，投标，竞标，开发和需求对接我都有参与。

项目主要是关于知识图谱的。主要是将客户的数据导入系统进行处理，定义 shcema,建立本体以及本体之间的关系，然后按照一定的将数据导入图数据中，并提供搜索展示。

数据的切割，就是将大的数据表切割成几张小表，表于表之间用外键关联，每个表生成一个实体，外键关联生成表与表之间的关系。

一如何防止超卖：

1、利用数据库自带排他锁，当减库存的时候，进位 where 判断，只有库存余量大于 0 的时候才进行进库存; update goods set num = num - 1 WHERE id = 1001 and num > 0; 2、也可以可用乐观锁 CAS 版本号机制。select version from goods WHERE id= 1001； update goods set num = num - 1, version = version + 1 WHERE id= 1001 AND num > 0 AND version = @version(上面查到的 version);

二、服务器抗压思路：

一、使用消息队列、异步生成订单；

二、redis 库存量预缓存。只将少量的请求流入到服务器。如果全部卖完，拦截请求。

三、生成订单前，进行一系列的检验：是否还有库存，是否重复下单，这些数据都可以缓存。

三、前端设计

静态资源缓存：将活动页面上的所有可以静态的元素全部静态化，尽量减少动态元素；通过 CDN 缓存静态资源，来抗峰值。在 url 后面加上？即可。

禁止重复提交：前端:用户提交之后按钮置灰，禁止重复提交；后端：在进入页面时，服务器生成 token 并存到缓存或者 session 中，form 表单使用隐藏域来存储这个 token，提交之后带有 token.后端收到这个 token,看是否与服务器生成的 token 一致，如果不一致就是重复提交。如果一致，处理完之后清除 token.

服务器返回表单页面时，会先生成一个 subToken 保存于 session，并把该 subToen 传给表单页面。当表单提交时会带上 subToken，服务器拦截器 Interceptor 会拦截该请求，拦截器判断 session 保存的 subToken 和表单提交 subToken 是否一致。若不一致或 session 的 subToken 为空或表单未携带 subToken 则不通过。

首次提交表单时 session 的 subToken 与表单携带的 subToken 一致走正常流程，然后拦截器内会删除 session 保存的 subToken。当再次提交表单时由于 session 的 subToken 为空则不通过。从而实现了防止表单重复提交。

用户限流：某一时间段内只允许用户提交少数次请求，IP 限流(Nginx 设置 IP 地址限流)。

中间代理层：

利用负载均衡（例如反响代理 Nginx 等）使用多个服务器并发处理请求，减小服务器压力。

(正向代理代理客户端 VPN，反向代理代理服务器。NGINX)

横向增加服务器数量，然后将请求分发到各个服务器上，将原先请求集中到单个服务器上的情况改为将请求分发到多个服务器上，将负载分发到不同的服务器，也就是我们所说的负载均衡。普通轮询算法、比例加权轮询、ip 路由负载、基于服务器响应时间负载分配、根据域名负载。

轮询（默认）：每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器 down 掉，能自动剔除。

指定权重：指定轮询几率，weight 和访问比率成正比，用于后端服务器性能不均的情况。

IP 绑定 ip_hash：每个请求按 ip 的 hash 结果分配，**这样每个访客固定访问一个后端服务器，可以解决 session 的问题。**

url_hash：按访问 url 的 hash 结果来分配请求，使每个 url 定向到同一个后端服务器，后端服务器为缓存时比较有效。

fair：按后端服务器的响应时间来分配请求，响应时间短的优先分配。

服务层：

业务分离:将秒杀业务系统和其他业务分离，单独放在高配服务器上。

采**用消息队列缓存请求：**将大流量请求写到消息队列缓存，利用服务器根据自己的处理能力主动到消息缓存队列中抓取任务处理请求。

利用缓存应对读请求：对于读多写少业务，大部分请求是查询请求，所以可以读写分离，利用缓存分担数据库压力。

数据库层：

上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。可以对数据库进行优化，减少数据库压力。

如果 redis 挂掉的话，如果提高数据库的并发能力：

业务拆分：将不同功能的模块拆分，使用不同的数据库。

MySQL 主从复制，读写分离：

分表分库：

其他策略：为请求分配成功状态或者分配秒杀资格，将没有资格的请求全部过滤，只有有资格的才能参与秒杀。说到底的秒杀这个高并发，并不是真正的处理高并发请求，而是如何应对高并发。将大量请求拦截然后放小量请求到数据库执行抢单是完全可以的，不用担心请求丢失的问题。

四、怎么保证 redis 缓存和数据库的一致性

延时双删；

存在不一致问题的，基本都是库存量。秒杀系统的设计，最重要的是不能超卖，这个问题我们已经谈过，用 mysql 排他锁或者乐观 CAS 版本号机制可以防止。而即使 redis 库存量比实际 mysql 库存量大，依然不会超卖。而 redis 库存量比 mysql 库存量小，可能发生没少卖的情况。少卖，问题不大。如果不能少卖，可以将 redis 预库存调大，他主要起到拦截请求降流的作用，一致不一致问题不大。

五、安全性问题

1、动态地址生成

2、接口防刷

六、消息队列

防止重复消费：重复消费在消息队列所存在的问题中，从来都不是一个严重的问题。如果是消息是读，那多消费一次没啥影响。如果是写，例如我们这个订单生成，消费之前，查询一下是否之前已经存在用户 ID 商品 ID 构成的订单，我们可以将生成的订单存入缓存，所以查询一次也不费劲。

消息的消费结果如何返回给消息发送方：客户端轮询订单生成结果。

消息丢失：秒杀系统中，本来就是万中选一的，丢失无所谓。如果是重要的信息，我们可以从三个角度来避免。如果是发送者丢失，开启 confirm 机制，如果队列丢失，开始 queue 持久化和消息持久化。如果是消费者丢失，关闭自动 ACK,当我们消费完之后，调用 API 给 queue 发送确认信息。

七、秒杀流程、画架构图

1、登录进入商品列表页面，静态资源缓存

2、点击进入商品详情页面，静态资源缓存，ajax 获取验证码(服务器生成三个数的预算，并将结果缓存到 redis);

3、点击秒杀，将验证码结果和商品 ID 传给后端，如果结果正确。动态生成随机串 UUID,结合用户 ID 和商品 ID 存入 redis，并将 path 传给前端。前端获取 path 后，再根据 path 地址调用秒杀服务；

4、服务端获取请求的 path 参数，去查缓存是否在；

5、如果存在，预减 redis 库存，如果还有库存，看是否已经生成订单，没有的话就将请求入消息队列。

6、从消息队列中取消息：获取商品 Id 和用户 ID,判断库存，重复下单；然后下单。

7、下单：减库存，生成订单；

8、前端轮询订单生成结果。50ms 继续轮询或者秒杀是否成功和失败;

八、优化策略

多服务器负载均衡、

简单介绍一下 Nginx

Nginx 是一款**轻量级的 Web 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器。**

Nginx 主要提供反向代理、负载均衡、动静分离(静态资源服务)等服务。下面我简单地介绍一下这些名词。

正向代理：某些情况下，**代理我们用户去访问服务器，需要用户手动的设置代理服务器的 ip 和端口号。**正向代理比较常见的一个例子就是 VPN 了。

反向代理：**是用来代理服务器的，代理我们要访问的目标服务器。代理服务器接受请求，然后将请求转发给内部网络的服务器，**并将从服务器上得到的结果返回给客户端，此时代理服务器对外就表现为一个服务器。

负载均衡

在高并发情况下需要使用，其原理就是将并发请求分摊到多个服务器执行，减轻每台服务器的压力，多台服务器(集群)共同完成工作任务，从而提高了数据的吞吐量。Nginx 支持的 **weight 轮询（默认）、ip_hash、fair、url_hash 这四种负载均衡调度算法**，感兴趣的可以自行查阅。负载均衡相比于反向代理更侧重的时将请求分担到多台服务器上去，所以谈论负载均衡只有在提供某服务的服务器大于两台时才有意义。

动静分离

动静分离是让动态网站里的动态网页根据一定规则把不变的资源和经常变的资源区分开来，动静资源做好了拆分以后，我们就可以根据静态资源的特点将其做缓存操作，这就是网站静态化处理的核心思路。

22.设计秒杀方案（从高并发、快速响应、高可用三方面回答，高并发（增加网络带宽、DNS 域名解析分发多台服务器、使用前置代理服务器 ngnix、CDN 内容分发、数据库查询优化（读写分离、分库分表）），快速响应（缓存服务器（memcached、redis）、能使用静态页面就用静态页面，减少容器解析、把常访问的图片等内容缓存）、高可用（热备，如数据库服务器的热备、集群监控（如使用 zabbix，重点关注 IO、内存、带宽和机器 load））

服务器返回表单页面时，会先生成一个 subToken 保存于 session，并把该 subToen 传给表单页面。当表单提交时会带上 subToken，服务器拦截器 Interceptor 会拦截该请求，拦截器判断 session 保存的 subToken 和表单提交 subToken 是否一致。若不一致或 session 的 subToken 为空或表单未携带 subToken 则不通过。

首次提交表单时 session 的 subToken 与表单携带的 subToken 一致走正常流程，然后拦截器内会删除 session 保存的 subToken。当再次提交表单时由于 session 的 subToken 为空则不通过。从而实现了防止表单重复提交。

缓存、降级和限流：

在开发高并发系统时，有三把利器用来保护系统：缓存、降级和限流：

缓存：缓存的目的是提升系统访问速度和增大系统处理容量

降级：降级是当服务出现问题或者影响到核心流程时，需要暂时屏蔽掉，待高峰或者问题解决后再打开

限流：限流的目的是通过对并发访问/请求进行限速，或者对一个时间窗口内的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务、排队或等待、降级等处理。

计数器法：设置一个计数器 counter，每当一个请求过来的时候，counter 就加 1，如果 counter 的值大于 100 并且该请求与第一个请求的间隔时间还在 1 分钟之内，那么说明请求数过多；如果该请求与第一个请求的间隔时间大于 1 分钟，且 counter 的值还在限流范围内，那么就重置 counter。缺点：统计的精度太低，无法处理临界问题。如果我在单位时间 **1s 内的前 10ms，已经通过了 100 个请求，那后面的 990ms，只能眼巴巴的把请求拒绝，我们把这种现象称为“突刺现象”**

```
public boolean grant() {
        long now = getNowTime();
```

```
        if (now < timeStamp + interval) {
            // 在时间窗口内
            reqCount++;
            // 判断当前时间窗口内是否超过最大请求控制数
            return reqCount <= limit;
        } else {
            timeStamp = now;
            // 超时后重置
            reqCount = 1;
            return true;
        }
    }
```

滑动窗口算法：将窗口更加细分，每个窗口都有自己的计数器，当总计算达到限定时，限流。这个滑动窗口只是将计算法变得更平滑而已。本质一样。

漏斗法：将容器比作一个漏斗，当请求进来时，相当于水倒入漏斗，然后从下端小口慢慢匀速的流出。不管上面流量多大，下面流出的速度始终保持不变。这种算法，在使用过后也存在弊端：无法应对短时间的突发流量。

在**令牌桶算法**中，存在一个桶，用来存放固定数量的令牌。算法中存在一种机制，以一定的速率往桶中放令牌。每次请求调用需要先获取令牌，只有拿到令牌，才有机会继续执行，否则选择选择等待可用的令牌、或者直接拒绝。通过 Google 开源的 guava 包，我们可以很轻松的创建一个令牌桶算法的限流器。

Google 开源工具包 Guava 提供了限流工具类 **RateLimiter**，该类基于令牌桶算法(Token Bucket)来完成限流，非常易于使用。RateLimiter 经常用于限制对一些物理资源或者逻辑资源的访问速率，它支持两种获取 permits 接口，**一种是如果拿不到立刻返回 false（tryAcquire()），一种会阻塞等待一段时间看能不能拿到（tryAcquire(long timeout, TimeUnit unit)）。**

缺点：传**统的方式整合 RateLimiter 有很大的缺点：代码重复量特别大，而且本身不支持注解方式。**

ES:

Elasticsearch 是一个近乎实时的搜索平台。这意味着从索引文档到可以搜索的时间只有轻微的延迟（通常是 1 秒）.

集群是一个或多个节点(服务器)的集合. 节点是一个单独的服务器，它是集群的一部分，存储数据，并参与集群的索引和搜索功能。

索引是具有某种相似特征的文档的集合。例如，你可以有一个顾客数据索引，产品目录索引和订单数据索引。文档是可以被索引的基本信息单元。文档用 JSON 表示,有多个 field,如年龄，性别，地址。

Elasticsearch 提供了将你的索引细分为多个碎片（或者叫分片）的能力。在创建索引时，可以简单地定义所需的分片数量。每个分片本身就是一个功能完全独立的“索引”，可以驻留在集群中的任何节点上。Shards & Replicas.每个分片又有副本。

正向索引是通过 key 找 value，反向索引则是通过 value 找 key。

首先将文本分割成一系列被称为语汇单元(token)的独立原子元素，此过程即为文档分析，然后建立倒排索引，也就是每个 term 关键词出现在哪些文档之中。ID TERM DOCUMENT List.，Elasticsearch 分别为每个 field 都建立了一个倒排索引。

Elasticsearch 为了能快速找到某个 term，将所有的 term 排个序，二分法查找 term，logN 的查找效率，就像通过字典查找一样，这就是 Term Dictionary。又有一个 Term Index，就像字典里的索引页一样，A 开头的有哪些 term，分别在哪页，可以理**解 term index 是一颗树**：这棵树不会包含所有的 term，它包含的是 term 的一些前缀。通过 term index 可以快速地定位到 term dictionary 的某个 offset，然后从这个位置再往后顺序查找。再结合 FST(Finite State Transducers)**的压缩技术，可以使 term index 缓存到内存中。从 term index 查到对应的 term dictionary 的 block 位置之后，再去**

磁盘上找 term，大大减少了磁盘随机读的次数。
用 FST 压缩 term index 外，对 posting list 也有压缩技巧，如 bitmap；
联合索引直接利用跳表(Skip list)的数据结构快速做“与”运算，或者利用上面提到的 bitset 按位“与”。

shard = hash(document_id) % (num_of_primary_shards)

字典树：
根节点不包含字符，除根节点外每一个节点都只包含一个字符。
从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
每个节点的所有子节点包含的字符都不相同。

系统设计问题：

1、让你系统的设计一个高并发的架构，你会从哪几个方面考虑？

2、一个千万级的 APP，你要搞定关注和粉丝列表，你用什么来做。要求最后一个关注的在最前面。新增和取关都要比较快的反馈你怎么做？如果一个人关注了之后，服务器宕机了怎么办？

3、设计一个榨汁机类，面向对象怎么设计

4、OOD design：计费停车场

5、多个服务器间共享 session 的解决方案

问了 new 一个对象的加载顺序，答了从父类到子类的加载过程 静态变量和静态块，哪个先加载，答了静态变量。

假设有这么一个场景，有一条新闻，新闻的评论量可能很大，如何设计评论的读和写

你如果写用 InnoDB，读用 Myisam 的话，主从同步怎么做

假设如果有同一时间海量数据入库，你怎么做（期间扯到了鹿晗关晓彤，这种微博大 V 给他安排上，还提了消息队列做削峰）

你对 Elasticsearch 有什么了解

有一个巨大的 ip 白名单池子，判断一个 ip 是否命中白名单,散列表 hash 一下或者字典树.

登录怎样校验密码，海量用户同时登录怎么优化

Cookie 和 Session 的区别，怎样存海量 Session

分布式 Session 问题

显示网站的用户在线数的解决思路

大数据相关：1 亿 = 10^8; 1KB = 2^10 = 1024 = 10^3; 1MB = 2^20 = 10 ^ 6 ; 1GB = 2^30 = 10^9;
IB = 8 bit;
2^32 = 40 亿;

URL 黑名单问题：判断值是否已经存在于集合中：hash(key) % m 哈希函数实现 MD5;
布隆过滤器解决：用长度为 M 的位数组 bitMap;将所有的值经过 K 个哈希函数计算之后对 m 取余，得到在数组中的位置。将 K 个位置取 1。对于新到的 url,也经过 K 个哈希函数取余计算数组中的值是否为 1，如果全为 1，可能存在，有一个为 0 说明一定不存在。
m 和 k 的大小问题以及容错率的计算：
m = - n * ln p / (ln2)^2 ; m 为位数组长度，n 为样本数量，p 为容错率 0.01%;
如：n = 100 亿，p = 0.0.1; m = 20n = 2000 亿 bit = 25 GB; k =ln2 * m / n = 14; ln2 = 0.7

20 亿个 32 位整数中出现次数最多的数：
用哈希表记录每个整数出现的次数<key,value>,一个记录是 8B;
如果有 20 亿个，就是 16GB;

所以，我们先将 20 亿的大文件，经过哈希函数，分割为 16 个小文件。相同的数会分配到同一个文件之中。然后在用哈希表记录每个数出现的次数。然后求每个文件出现最多的次数或者每个文件出现最多的 K 次，top 可以用小顶堆实现。

40 亿个非负整数没出现的数：

40 亿个，如果用哈希表记录每个出现的次数，一个 8B，16GB，太大。

可以使用位数组，40 亿长度的位数组，内存占用为 500MB。将 40 亿个数，一个个的 bitmap[indxe] = 1;然后将所有为 0 的 index 找出，就是没有出现的数。

如果说是 10 内存找出一个不存在的数呢？

500MB 变成 10 内存，原本的 40 亿长度除以 64，就是分割成 64 个长度为 2^22 的位数组。
int[] count = new int[64];将 40 亿个数，每个数 / 2^22，看看他落入哪个区间，count[i]++;
计算每个数的 count,看看装满没有，2^22 次方，谁大。找到之后，再申请 8MB 的空间；看看落入这个空间的哪个数少了。

找到 100 亿个 URL 重复的 URL

分流：用哈希函数，将 100 亿的大文件，转化成小文件，如 100 个，那么每个近似 100 兆。然后对 url 进行哈希表统计，是否出现重复，每个记录 8B,那么需要 800MB。如果依然是变大了，那就再使用哈希函数分流。

TOPK

也是先用哈希函数将大文件转变成小文件，然后用哈希表统计每个数出现的次数。然后对每个小文件的哈希表，使用小顶堆计算 topk。然后合在一起再计算 topK.

40 亿非负整数出现两次的数：

40 亿*2 长的位数字，1GB 的内存。每两位表示这个树出现的情况。[2*num][2*num+1]
第一次出现，标记为 01，第二次出现标记为 10，第三次出现标记为 11.求所有两次出现即标记为 10 的 num;

大数据的中位数，桶排序：

长度为 2MB（2^21）的整型数组占地 8M;分割成大约 2148 个区间。申请一个 arr[2148]的数组，遍历这么多数， num /2^21 看落到了哪个区间，然后 count[index]++;从左到右开始数数，找到中位数落在的区间位置。然后再次遍历，对这个位置的数进行哈希统计。然后找到中位数。

看到上述知乎用户 iammutex 关于如何正确生成短地址 URL 的探讨，我们知道了，可以通过发号器的方式正确的生成短地址，生成算法设计要点如下：
（1）利用放号器，初始值为 0，对于每一个短链接生成请求，都递增放号器的值，再将此值转换为 62 进制（a-zA-Z0-9），比如第一次请求时放号器的值为 0，对应 62 进制为 a，第二次请求时放号器的值为 1，对应 62 进制为 b，第 10001 次请求时放号器的值为 10000，对应 62 进制为 sBc。
（2）将短链接服务器域名与放号器的 62 进制值进行字符串连接，即为短链接的 URL，比如：t.cn/sBc。
（3）重定向过程：生成短链接之后，需要存储短链接到长链接的映射关系，即 sBc -> URL，浏览器访问短链接服务器时，根据 URL Path 取到原始的链接，然后进行 302 重定向。映射关系可使用 K-V 存储，比如 Redis 或 Memcache。
七、生成短地址之后如何跳转哪？
对于该部分的讨论，我们可以认为他是整个交互的流程，具体的流程细节如下：
（1）用户访问短链接：http://t.cn/RuPKzRW；
（2）短链接服务器 t.cn 收到请求，根据 URL 路径 RuPKzRW 获取到原始的长链接（KV 缓存数据库中去查找）：https://blog.csdn.net/xlgen157387/article/details/79863301；

（3）服务器返回 302 状态码，将响应头中的 Location 设置为：https://blog.csdn.net/xlgen157387/article/details/79863301；
（4）浏览器重新向 https://blog.csdn.net/xlgen157387/article/details/79863301 发送请求；
（5）返回响应；
八、短地址发号器优化方案
1、算法优化
采用以上算法，如果不加判断，那么即使对于同一个原始 URL，每次生成的短链接也是不同的，这样就会浪费存储空间（因为需要存储多个短链接到同一个 URL 的映射），如果能将相同的 URL 映射成同一个短链接，这样就可以节省存储空间了。主要的思路有如下两个：
方案 1：查表
每次生成短链接时，先在映射表中查找是否已有原始 URL 的映射关系，如果有，则直接返回结果。很明显，这种方式效率很低。
方案 2：使用 LRU 本地缓存，空间换时间
使用固定大小的 LRU 缓存，存储最近 N 次的映射结果，这样，如果某一个链接生成的非常频繁，则可以在 LRU 缓存中找到结果直接返回，这是存储空间和性能方面的折中。
2、可伸缩和高可用
如果将短链接生成服务单机部署，缺点一是性能不足，不足以承受海量的并发访问，二是成为系统单点，如果这台机器宕机则整套服务不可 用，为了解决这个问题，可以将系统集群化，进行“分片”。在以上描述的系统架构中，如果发号器用 Redis 实现，则 Redis 是系统的瓶颈与单点，因此，利用数据库分片的设计思想，可部署多个发号器实例，每个实例负责特定号段的发号，比如部署 10 台 Redis，每台分别负责号段尾号为 0-9 的发号，注意此时发号器的步长则应该设置为 10（实例个数）。
另外，也可将长链接与短链接映射关系的存储进行分片，由于没有一个中心化的存储位置，因此需要开发额外的服务，用于查找短链接对应的原始链接的存储节点，这样才能去正确的节点上找到映射关系。

Jmeter 进行压力测试。
分布式 seesion:
用户登录之后,随机生成 UUID，作为 token 以 Cookie 的方式返回给浏览器。同时将 token 作为 redis key 的后缀，用户对象作为 value 存入 redis 中。下次浏览器再发送 HTTP 请求的时候，带有这个 Cookie。服务器获取 http 中的 cookie 并解析，得到 token，然后再 redis 中查询，如果命中，则返回对象。

MD5 双重加密: 用户填写的表单，在发送给服务器之前，先进行一次 MD5 加密。然后以 Post 的形式发送给服务器。服务器获取这个用户 ID 和密码之后，随机生成 Sault，并进行 MD5 加密。将加密之后的存入数据库，随机 salut 也存入数据库。

如何解决卖超问题：
//当库存大于 0 的时候才成功下订单，数据库本身会有锁，那么就不会在数据库中同时多个线程更新一条记录，使用数据库特性来保证超卖的问题；
数据库加唯一索引防止用户重复购买；
redis 预减库存减少数据库访问　内存标记减少 redis 访问　请求先入队列缓冲，异步下单，增强用户体验；

页面级缓存 thymeleafViewResolver，手动渲染　使用模板引擎；将渲染的 html 页面直接存到 redis 缓存中。

对象缓存 Reids
redis 永久缓存对象减少压力
redis 预减库存减少数据库访
内存标记方法减少 redis 访问

订单处理队列 rabbitmq
请求先入队缓冲，异步下单，增强用户体验
请求出队，生成订单，减少库存
客户端定时轮询检查是否秒杀成功

下订单的过程:

1、将商品库存数量缓存到数据库，
2、获取缓存中的商品库存量
3，判断减去 1 之后的库存量
4、判断这个秒杀订单形成没有
5、正常请求，入队，发送一个秒杀 message 到队列里面去，入队之后客户端应该进行轮询。

消息队列：利用 Mqsender 将用户 ID 和商品 ID 的对象作为 Json 字符串发送给 queue,然后消息队列发给指定的接受者。
获取对象 ID 和商品 ID, 判断库存是否不足，判断是否已经下单成功。如果没有， //原子操作：1.库存减 1，(只有减库存成功，说明还有剩余的，才能下单)2.下订单(生成普通订单，生成秒杀订单，将秒杀订单缓存)，3.写入秒杀订单(如果没有生成，秒杀失败，说明卖完了，将外卖的状态写入缓存。)--->是一个事务。

商品的库存量写入缓存，预减库存，查找是否还有足够的量，判断是否重复秒杀，然后订单请求入队；然后请求出队，获取商品 ID 和用户 ID，判断库存是否不足，是否已经重复下单，如果没有，真正的减库存，然后下单同时将订单写入缓存。如果没有成功，将卖完的状态写入缓存。客户端轮询是否完成下单。
setTimeout(function() {getMiaoshaResult(goodsId);}, 50);//50ms 之后继续轮询

总之，生产者将消息按照转换模式，以交换机的模式选择，将消息发送给指定类型的 queue，消费者从监听的 queue 获取消息;
比如有 10 件商品要秒杀，可以放到缓存中，读写时不要加锁。 当并发量大的时候，可能有 25 个人秒杀成功，这样后面的就可以直接抛秒杀结束的静态页面。进去的 25 个人中有 15 个人是不可能获得商品的。所以可以根据进入的先后顺序只能前 10 个人购买成功。后面 15 个人就抛商品已秒杀完。

假设我们的秒杀场景

比如某商品 10 件物品待秒. 假设有 100 台 web 服务器(假设 web 服务器是 Nginx + Tomcat),n 台 app 服务器,n 个数据库

第一步 如果 Java 层做过滤, 可以在每台 web 服务器的业务处理模块里做个计数器 AtomicInteger(10)=待秒商品总数,decreaseAndGet()>＝0 的继续做后续处理, <0 的直接返回秒杀结束页面，这样经过第一步的处理只剩下 100 台*10 个=1000 个请求。

第二步, memcached 里以商品 id 作为 key 的 value 放个 10, 每个 web 服务器在接到每个请求的同时, 向

memcached 服务器发起请求，利用 memcached 的 decr(key,1)操作返回值>=0 的继续处理，其余的返回秒杀失败页面，这样经过第二步的处理只剩下 100 台中最快速到达的 10 个请求。

第三步，向 App 服务器发起下单操作事务。

第四步，App 服务器向商品所在的数据库请求减库存操作(操作数据库时可以 "update table set count=count-1 where id=商品 id and count>0;" update 成功记录数为 1，再向订单数据库添加订单记录，都成功后提交整个事务，否则的话提示秒杀失败，用户进入支付流程。

看看淘宝的秒杀

一、前端

面对高并发的抢购活动，前端常用的三板斧是【扩容】【静态化】【限流】

扩容：加机器，这是最简单的方法，通过增加前端池的整体承载量来抗峰值。

静态化：将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过 CDN 来抗峰值。

限流：一般都会采用 IP 级别的限流，即针对某一个 IP，限制单位时间内发起请求数量。或者活动入口的时候增加游戏或者问题环节进行消峰操作。

有损服务：最后一招，在接近前端池承载能力的水位上限的时候，**随机拒绝部分请求来保护活动整体的可用性。**

二、那么后端的数据库在高并发和超卖下会遇到什么问题呢

首先 MySQL 自身对于高并发的处理性能就会出现问题，一般来说，MySQL 的处理性能会随着并发 thread 上升而上升，但是到了一定的并发度之后会出现明显的拐点，之后一路下降，最终甚至会比单 thread 的性能还要差。

其次，超卖的根结在于减库存操作是一个事务操作，需要先 select，然后 insert，最后 update -1。最后这个-1 操作是不能出现负数的，但是当多用户在有库存的情况下并发操作，出现负数这是无法避免的。

最后，当减库存和高并发碰到一起的时候，由于操作的库存数目在同一行，就会出现争抢 InnoDB 行锁的问题，导致出现互相等待甚至死锁，从而大大降低 MySQL 的处理性能，最终导致前端页面出现超时异常。

针对上述问题，如何解决呢？ 淘宝的高大上解决方案：

I：关闭死锁检测，提高并发处理性能。

II：修改源代码，将排队提到进入引擎层前，降低引擎层面的并发度。

III：组提交，降低 server 和引擎的交互次数，降低 IO 消耗。

解决方案 1：**将存库从 MySQL 前移到 Redis 中，所有的写操作放到内存中，由于 Redis 中不存在锁故不会出现互相等待，并且由于 Redis 的写性能和读性能都远高于 MySQL，这就解决了高并发下的性能问题。然后通过队列等异步手段，将变化的数据异步写入到 DB 中。**

优点：解决性能问题

缺点：没有解决超卖问题，同时由于异步写入 DB，存在某一时刻 DB 和 Redis 中数据不一致的风险。

解决方案 2：**引入队列，然后将所有写 DB 操作在单队列中排队，完全串行处理。当达到库存阀值的时候就不在消费队列，并关闭购买功能。这就解决了超卖问题。**

优点：解决超卖问题，略微提升性能。

缺点：性能受限于队列处理机处理性能和 DB 的写入性能中最短的那个，**另外多商品同时抢购的时候需要准备多条队列。**

解决方案 3：将写操作前移到 MC 中，同时利用 MC 的轻量级的锁机制 CAS 来实现减库存操作。

优点：读写在内存中，操作性能快，引入轻量级锁之后可以保证同一时刻只有一个写入成功，解决减库存问题。

缺点：没有实测，基于 CAS 的特性不知道高并发下是否会出现大量更新失败？不过加锁之后肯定对并发性能会有影响。

解决方案 4：将提交操作变成两段式，先申请后确认。然后利用 Redis 的原子自增操作，同时利用 Redis 的事务特性来发号，保证拿到小于等于库存阀值的号的人都可以成功提交订单。然后数据异步更新到 DB 中。

优点：解决超卖问题，库存读写都在内存中，故同时解决性能问题。

缺点：由于异步写入 DB，可能存在数据不一致。另可能存在少买，也就是如果拿到号的人不真正下订单，可能库存减为 0，但是订单数并没有达到库存阀值。

总结

Java 用到的线程调度算法：
抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

默认升序，数据库排序；ASC
无法使用比较运算符来测试 NULL 值，比如 =, <, 或者 <>。
我们必须使用 IS NULL 和 IS NOT NULL 操作符。

```
CREATE TABLE Person
(
LastName varchar(30),
FirstName varchar,
Address varchar,
Age int(3)
)
```

分布式全局唯一 ID:雪花算法得出的 ID,有序，唯一，快速。**64 位：最高位预留+毫秒级时间 41 位+机器 ID 10 位+毫秒内序列 12 位；41 位时间能用 69 年。机器 ID 支持 2^10=1024 台机器分布。**
与之相对的数据库自增 ID:缺陷：**高并发下插入数据需要事务机制，对数据库压力大。大表不能做水平分表**，否则增删容易出错。

UUID：**无序，太长、没法排序、使数据库性能降低 s；**

有一个无序整型数组，如何求出该数组排序后的任意两个相邻元素的最大差值？（桶排序，尽量分散，桶的数量为 （最大 - 最小）/ 元素个数）

.讲一下 IOC 和 AOP。
Nginx
AVA 异常，分类，具体都有哪些？

线程与进程的区别？为什么要有线程？
GMS 与 G1 的区别、应用场景
反射机制（原理、使用场景、Spring 注入）
消息队列，消费者运行速度不一，如何控制数据的同步（保证插入先于删除）

Tomcat 的双亲模型，破坏双亲的的三种出现过的，OSGI 的破坏双亲的模型
手写线程池需要哪些参数
核心 10 队列 1000 最大 100 的情况线程池怎么处理

JDK 8 与先前版本的区别

讲迭代的区别顺势还讲了快速失败和安全失败

线程间的通信方式（操作系统层面，信号量等等）

List 有哪些，怎样做到线程安全：
ThreadLocal：
ThreadLocal 是安全的吗？讲讲原理分析与使用场景、内存泄漏、弱引用

Java 怎么做到跨平台的
java 怎么把.class 文件进行反编译的

安装 Java 都需要安装哪些东西
Java 虚拟机是如何找到类方法中的 Main 函数的

1. hashmap 和 hash table 和 concurrent hashmap
2. jmm 模型
自己聊 volatile synclock reentreenlock aqs 组件 cas automic 原子类
线程池 7 大参数 我给了个场景说明了参数的作用

corePoolSize：核心池的大小：
maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
keepAliveTime：空闲的线程保留的时间。
TimeUnit：空闲线程的保留时间单位。
BlockingQueue<Runnable>：阻塞队列
ThreadFactory：线程工厂，用来创建线程
RejectedExecutionHandler：队列已满，而且任务量大于最大线程的异常处理策略

redis 数据类型 zset 怎么用
nginx 集群怎么做 负载均衡算法有哪些，单点故障怎么办
普通轮询算法、比例加权轮询、ip 路由负载、基于服务器响应时间负载分配、根据域名负载。
Nginx+keepalived 双机实现 nginx 反向代理服务的高可用,一台 nginx 挂掉之后不影响应用也不影响内网访问外网。
用 Keepalived 搭建双 Nginx server 集群 防止单点故障

项目的 QPS 是多少：
讲讲 HashSet：

Spring AOP 有切点，切面，还有哪些
图的最短路径算法：
深度或广度优先搜索算法，
费罗伊德算法，最开始只允许经过 1 号顶点进行中转，接下来只允许经过 1 号和 2 号顶点进行中转.....允许经过 1~n 号所有顶点进行中转，
迪杰斯特拉算法，
Bellman-Ford 算法。

MySQL 的 MVCC：

Integer 缓冲机制：
mybatis 怎么防止 sql 注入
spring 解决循环依赖
（1）构造器的循环依赖 构造器的循环依赖问题无法解决，只能抛出异常。
（2）setter 注入构成的循环依赖，spring 设计的机制主要就是解决这种循环依赖
3、prototype 作用域 bean 的循环依赖。无法解决
步骤一：**beanA 进行初始化，并且将自己进行初始化的状态记录下来，并提前向外暴露一个单例工程方法，**从而使其他 bean 能引用到该 bean（可能读完这一句，您仍然心存疑惑，没关系，继续往下读）
步骤二：beanA 中有 beanB 的依赖，**于是开始初始化 beanB。**
步骤三：初始化 beanB 的过程中又发现 beanB 依赖了 beanA,于是又进行 beanA 的初始化，这时发**现 beanA 已经在进行初始化了，程序发现了存在的循环依赖，然后通过步骤一中暴露的单例工程方法拿到 beanA 的引用**（注意，此时的 beanA 只是完成了构造函数的注入但为完成其他步骤），从而 beanB 拿到 beanA 的引用，完成注入，完成了初始化，如此 beanB 的引用也就可以被 beanA 拿到，**从而 beanA 也就完成了初始化。**
spring 进行 bean 的加载的时候，首先进行 bean 的初始化（调用构造函数），然后进行属性填充。在这两步中间，spring 对 bean 进行了一次状态的记录

不要使用基于构造函数的依赖注入，可以通过以下方式解决：
1.在字段上使用@Autowired 注解，让 Spring 决定在合适的时机注入
2.用基于 setter 方法的依赖注入。
聚簇索引和非聚簇索引
对于聚簇索引存储来说，行数据和主键 B+树存储在一起，辅助键 B+树只存储辅助键和主键，主键和非主键 B+树几乎是两种类型的树。对于非聚簇索引存储来说，主键 B+树在叶子节点存储指向真正数据行的指针，而非主键。

1 spring aop

3. spring mvc 流程

4.mybatis 分页

sql 分页 limit + offset 分页：

pagehelper 分页插件：拦截器分页。

6. mysql 支持大小写查询

MySQL 查询是不区分大小写的！这可真的是惊呆我了，虽然知道一般情况下，关键字是不区分大小写的，但是没想到连**要查询的参数都是不区分大小写的：**

Mysql 默认的字符检索策略：utf8_general_ci，表示不区分大小写；utf8_general_cs 表示区分大小写，utf8_bin 表示二进制比较，同样也区分大小写

7. innodb 和 myisam 的区别

8 solr 或者 es 介绍一下（倒排索引；有哪些索引结构）

项目中用到的算法（排序 查找）， 设计模式

http 常见请求头

mybatis 传 list 时 xml 文件里怎么写，用哪个标签

线程池参数，什么情况下会达到最大线程数量

cookie 包括哪两种

SpringMVC 工作原理、内部流程

MyBatis （缓存机制，一级、二级原理和作用，映射是怎么实现的，XML 怎么对应到 Mapper 代理模式）

防止 sql 注入：MyBatis 启用了预编译功能，在 SQL 执行前，会先将上面的 SQL 发送给数据库进行编译；执行时，直接使用编译好的 SQL，替换占位符“?”就可以了。因为 SQL 注入只能对编译过程起作用，所以这样的方式就很好地避免了 SQL 注入的问题。

mybatis 的查询缓存分为一级缓存和二级缓存，一级缓存是 SqlSession 级别的缓存，二级缓存时 mapper 级别的缓存，二级缓存是多个 SqlSession 共享的;

当在同一个 SqlSession 中执行两次相同的 sql 语句时，第一次执行完毕会将数据库中查询的数据写到缓存（内存）中，第二次查询时会从缓存中获取数据，不再去底层进行数据库查询，从而提高了查询效率。如果 SqlSession 执行了 DML 操作（insert、update、delete），并执行 commit（）操作，mybatis 则会清空 SqlSession 中的一级缓存,避免脏读现象。

Mybatis 的二级缓存是 Mapper 级别的缓存，默认不开启，需手工配置。其存储作用域为 Mapper，也就是同一个 namespace 的 mappe.xml; 当一个 sqlseesion 执行了一次 select 后，在关闭此 session 的时候，会将查询结果缓存到二级缓存。当另一个 sqlsession 执行 select 时，首先会在他自己的一级缓存中找，如果没找到，就回去二级缓存中找，找到了就返回，就不用再去数据库了。

MySQL 怎么分页：

用 limit + offset: SELECT * FROM table LIMIT 5,10; // 检索记录行 6-15

Linux 相关

查看进程 PID

根据名称匹配：Ps -ef | grep xxxx

根据端口号 ： lsof -i:5000

关闭 xxx 的进程 ： kill -9 xxxx; 9：SIGKILL，强制中断一个进程的进行;

top 命令是 Linux 下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于 Windows 的任务管理器;

chmod 命令, chown,chgrp

cat 用途是连接文件或标准输入并打印。这个命令常用来显示文件内容，或者将几个文件连接起来显示，或者从标准输入读取内容并显示.more,less，按页查看，前翻后翻。

tail: 从指定点开始将文件写到标准输出.使用 tail 命令的-f 选项可以方便的查阅正在改变的日志文件,tail -f filename 会把 filename 里最尾部的内容显示在屏幕上,并且不但刷新,使你看到最新的文件内容.

diff 命令用于比较两个文件或目录的不同

scp 命令是 secure copy 的简写，用于在 Linux 下进行远程拷贝文件的命令;

ssh 远程登录;

ping; ifconfig

grep 命令:该命令常用于分析一行的信息，若当中有我们所需要的信息，就将该行显示出来

grep -n '要查找的字符串' 被查的文件 file:返回行数;

cp 复制，rm 删除，mv 移动，tar 解压缩，mkdir，rmdir,

正则表达式

^匹配输入字符串开始的位置，$匹配输入字符串结尾的地方。

*，匹配前面字符 0 次或者多次。+，一次或多次匹配。？，零次或一次匹配。

{n},匹配前面 n 次，{n,}至少匹配 n 次。{n,m}至少匹配 n 次，至多匹配 m 次。

？：如果跟在其他通配符后面，表示非贪心搜索。其他的默认贪心匹配。O+?,表示只匹配一次 o.而默认 o+匹配所有 o;

.，匹配任意字符。 X | Y，表示或，x 或者 y. [xyz]:字符集中的任意一个。[^xyz],反向字符集，匹配不包含字符的一个。

[a-z],匹配指定范围的任意字符。[^a-z]匹配非范围内的字符。 \d，数字字符匹配，[0-9]; \D,非数字字符匹配[^0-9];

\s,匹配任何空白字符。\S，匹配非空白字符。\w 字母数字下划线，\W，非字符数字下划线。

常用正则表示式：

任意个数字：^[0-9]*$; n 位数字：^\d{n}$; 中文：[\u4e00-\u9fa5]；非零开头的最多带两位小数的数字：^([1-9][0-9]*)+(.[0-9]{1,2})?$；带 1-2 位小数的正数或负数：^(\-)?\d+(\.\d{1,2})?$；由数字、26 个英文字母或者下划线组成的字符串：^\w+$ 或 ^\w{3,20}$ 。**Email 地址：**

^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$

密码(以字母开头，长度在 6~18 之间，只能包含字母、数字和下划线)：^[a-zA-Z]\w{5,17}$

强密码(必须包含**大小写字母和数字的组合**，不能使用特殊字符，长度在 8-10 之间)：^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$

IP 地址：\d+\.\d+\.\d+\.\d+ (提取 IP 地址时有用)

IP 地址：((?:(?:25[0-5]|2[0-4]\\d|[01]?\\d?\\d)\\.){3}(?:25[0-5]|2[0-4]\\d|[01]?\\d?\\d))

设计模式

创建型模式、结构型模式和行为型模式 3 种。

创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，

工厂方法（Factory Method）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。

抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。

适配器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

装饰（Decorator）模式：动态的给对象增加一些职责，即增加其额外的功能。

策略（Strategy）模式：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

职责链（Chain of Responsibility）模式：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

观察者（Observer）模式：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

UML 图： 三格，上面类名，中间属性变量 -+#~表可见范围。下面是方法。接口有个 O,没有正方形圈住。

依赖关系：------→ A 中引用了 B 的对象。关联关系 ——>如老师教学生这类关系

聚合关系：——白菱形。整体和部分关系，has-a;只不过部分可脱离整体存在，菱形的那边是整体。如汽车和轮子。

组合关系：——实体黑菱形。整体和部分的关系，部分不可脱离整体。黑菱形是整体。如头和嘴。

继承关系：——白三角，继承关系，老师继承人。

实现关系：------白三角：类实现了接口。

单例模式：生成单例的四种方式：懒汉模式，饿汉模式，多线程双重检验模式，静态内部类？

1、当一个类只要求生成一个对象的是场景。2、当对象需要被共享的场合。3、类对象需要频繁实例化又被频繁销毁。

关键点在于：隐藏构造方法， 开放一个静态方法，返回实例对象。有懒汉模式，饿汉模式，和多线程双重检验模式。

工厂模式：定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体工厂实现类中。抽象工厂模式是工厂模式方法的升级版本，工厂方法模式只能产生一种产品，而抽象工厂模式可以生产多个等级的产品。

代理模式:在访问对象和目标对象之间创建一个代理对象，代理对象持有对代理对象的引用，访问对象通过代理对象间接访问目标对象。而且代理对象可以扩展目标对象的功能，还一定程度了降级了

系统的耦合度。spring AOP 是基于代理实现的，当被代理对象实现了某个接口，那么 Spring AOP 会使用 JDK Proxy，去创建代理对象，而对于没有实现接口的对象， Spring AOP 会使用 Cglib 生成一个被代理对象的子类来作为代理.

原型（Prototype）模式的定义如下：用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。用 Clone()方法直接赋值对象。
工厂模式:定义一个创建产品对象的**工厂接口**,将产品对象的**实际创建工作推迟到具体子工厂类当中**。可以随时增加新的具体产品类和对应的具体工厂类。
抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。
代理模式的定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。代理对象可以扩展目标对象的功能；客户端与目标对象分离，在一定程度上降低了系统的耦合度；代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
动态代理：

适配器模式（Adapter）包含以下主要角色。
目标（Target）接口：当前系统业务所期待的接口，它可以是抽象类或接口。
适配者（Adaptee）类：它是被访问和适配的现存组件库中的组件接口。
适配器（Adapter）类：它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。
装饰（Decorator）模式的定义：指在不改变现有对象结构的情况下，**动态地给该对象增加一些职责**（即增加其额外功能）的模式，它属于对象结构型模式。
装饰模式主要包含以下角色。
抽象构件（Component）角色：定义一个抽象接口以规范准备**接收附加责任的对象。**
具体构件（Concrete Component）角色：实现抽象构件，**通过装饰角色为其添加一些职责。**
抽象装饰（Decorator）角色：**继承抽象构件，并包含具体构件的实例**，可以通过其子类扩展具体构件的功能。
具体装饰（ConcreteDecorator）角色：实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

模板方法模式：
模板方法（Template Method）模式的定义如下：**定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中**，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

Spring 中用到的设计模式：
工厂设计模式
Spring 使用工厂模式可以通过 BeanFactory 或 ApplicationContext 创建 bean 对象。
BeanFactory 采用了工厂设计模式，负责读取 bean 配置文档，管理 bean 的加载，实例化，维护 bean 之间的依赖关系，负责 bean 的生命周期。BeanFactroy 采用的是**延迟加载形式来注入 Bean 的**，即只有在使**用到某个 Bean** 时(调用 getBean())，**才对该 Bean 进行加载实例化**。
ApplicationContext 除了提供上述 BeanFactory 所能提供的功能之外，还提供了更完整的框架功能：国际化支持、资源访问，比如访问 URL 和文件、事件机制，同时加载多个配置文件等。ApplicationContext 在解析配置文件时对配置文件中的所有对象都初始化了，getBean() 方法只是获取对象的过程。
单例设计模式
Spring 中 bean 的默认作用域就是 singleton(单例)的。单例的好处：对于频繁使用的对象，可以

省略创建对象所花费的时间；new 操作的次数减少，这将减轻 GC 压力，缩短 GC 停顿时间。

Spring 通过 ConcurrentHashMap 实现单例注册表的特殊方式实现单例模式。

简单来说使用单例模式可以带来下面几个好处:

对于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级对象而言，是非常可观的一笔系统

开销；

由于 new 操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻 GC 压力，缩短 GC 停顿时间。

```
class Singleton{
    private volatile static Singleton instance;
    private Singleton(){
    }
    public static Singleton getInstance(){
        if (instance == null){
            synchronized (Singleton.class){
                if (instance == null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}

//懒汉模式，等你调用的时候才开始创建实例；懒加载启动快，资源占用小，使用时才实例化，无锁。非线程安全
class Singleton1{
    private static Singleton1 instance;
    private Singleton1(){

    }
    private static Singleton1 getInstacne(){
        if (instance == null)
            instance = new Singleton1();
        return instance;
    }

}

//饿汉模式：
//优点：饿汉模式天生是线程安全的，使用时没有延迟。
//缺点：启动时即创建实例，启动慢，有可能造成资源浪费。
class Singleton2{
    private static Singleton2 instance = new Singleton2();

    private Singleton2(){}

    public Singleton2 getInstance(){
        return instance;
    }
```

```
}

// Effective Java 第一版推荐写法

class Singleton23 {

    private static class SingletonHolder {

        private static final Singleton23 INSTANCE = new Singleton23();

    }

    private Singleton23 (){}

    public static final Singleton23 getInstance() {

        return SingletonHolder.INSTANCE;

    }

}
```

代理设计模式

AOP 能够将那些与业务无关，**却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来**，便于**减少系统的重复代码，降低模块间的耦合度**，并**有利于未来的可拓展性和可维护性**。

Spring AOP 就是基于动态代理的, 如果要代理的对象，实现了某个接口，那么 Spring AOP 会使用 **JDK Proxy**，去创建代理对象，而对于没有实现接口的对象， Spring AOP 会使用 **Cglib** 生成一个被代理对象的子类来作为代理.

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。 Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作. Spring AOP 已经集成了 AspectJ , 如果我们的切面比较少,那么两者性能差异不大。但是，当切面太多的话，最好选择 AspectJ ，它比 Spring AOP 快很多。

模板方法

模板方法模式是一种行为设计模式，它定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。 模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤的实现方式。

Spring 中 jdbcTemplate、hibernateTemplate **等以 Template 结尾的对数据库操作的类，**它们就使用到了模板模式。一般情况下，我们都是使用继承的方式来实现模板模式，但是 Spring 并没有使用这种方式，而是**使用 Callback 模式与模板方法模式配合，既达到了代码复用的效果，同时增加了灵活性。**

观察者模式:

观察者模式是一种对象行为型模式。它表示的是一种对象与对象之间具有依赖关系，当一个对象发生改变的时候，这个对象所依赖的对象也会做出反应。Spring 事件驱动模型就是观察者模式很经典的一个应用。比如我们每次添加商品的时候都需要重新更新商品索引，这个时候就可以利用观察者

模式来解决这个问题。
Spring 事件驱动模型中的三种角色：事件角色、事件监听者角色、事件发布角色、

事件角色：ApplicationEvent (org.springframework.context 包下)充当事件的角色,这是一个抽象类。
事件监听者角色：ApplicationListener 充当了事件监听者角色，它是一个接口，里面只定义了一个 onApplicationEvent（ApplicationEvent E）方法来处理 ApplicationEvent。
ApplicationEventPublisher 充当了事件的发布者，它也是一个接口。publishEvent(ApplicationEvent event)

流程：
定义一个事件: 实现一个继承自 ApplicationEv ent，并且写相应的构造函数；
定义一个事件监听者：实现 ApplicationListener 接口，重写 onApplicationEvent() 方法；
使用事件发布者发布消息: 可以通过 ApplicationEventPublisher 的 publishEvent() 方法发布消息

适配器模式: 适配器模式(Adapter Pattern) 将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。

spring MVC 中的适配器模式: 在 Spring MVC 中，DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的 Handler。解析到对应的 Handler（也就是我们平常说的 Controller 控制器）后，开始由 HandlerAdapter 适配器处理。HandlerAdapter 作为期望接口，具体的适配器实现类用于对目标类进行适配，Controller 作为需要适配的类。

Spring AOP 的增强或通知(Advice)使用到了适配器模式，与之相关的接口是 AdvisorAdapter。通知有很多类型：前置通知、后置通知、环绕通知、返回通知、异常通知。每种通知都有相对应的拦截器。Spring 预定义的通知要通过对应的适配器，适配成 MethodInterceptor 接口(方法拦截器)类型的对象（如：MethodBeforeAdviceInterceptor 负责适配 MethodBeforeAdvice）。
装饰者模式:
装饰者模式可以动态地给对象添加一些额外的属性或行为。相比于使用继承，装饰者模式更加灵活。简单点儿说就是当我们需要修改原有的功能，但我们又不愿直接去修改原有的代码时，设计一个 Decorator 套在原有代码外面。其实在 JDK 中就有很多地方用到了装饰者模式，比如 InputStream 家族，InputStream 类下有 FileInputStream (读取文件)、BufferedInputStream (增加缓存,使读取文件速度大大提升)等子类都在不修改 InputStream 代码的情况下扩展了它的功能。Spring 中配置 DataSource 的时候，DataSource 可能是不同的数据库和数据源。我们能否根据客户的需求在少修改原有类的代码下动态切换不同的数据源？这个时候就要用到装饰者模式(这一点我自己还没太理解具体原理)。

集合重复问题:
位图法：集合的最大元素 max,创建 max+1 的数组，如果遇到元素 5，设 nums[6] = 1,下次还这样时，如果 nums[i]=1,表示已经存在。还可以排序。
布隆过滤器：检测一个元素是否属于一个集合。当他不属于这个集合时，那一定数不属于的。如果判断属于，那不一定。实现原理:位数组和 Hash 结合。M 位的位数组，初始化为 0.定义 k 个不同的 hash 函数，每个函数都将集合中的元素映射到位数组的一位。向集合中插入元素时，k 个哈希得到 k 位。将这些位设为 1.
新的被检测元素时，看这 k 位是否全为 1.有一个为 0，说明不在集合中。

TOPK 问题:
最小堆，建立一个最小堆。一个个遍历。中序遍历最终的最小堆；mlogm 建堆 m 大小。 时间复杂度 nmlogm
分治：将大文件分成小文件，每个小文件用快排算出 topk, 然后合并一块求 topk.
频率 TOPK, 将数据集按照 hash 的方法分解成多个小数据集，然后用 trie 或者 hash 统计每个小数据

集的频率，然后用小顶堆算出每个数据集中出现频率最高的 K 个，然后算出所有的 topK.
去重：
位图法： 9 位数的话，99999999+1 大小的位数组。
排序问题：
数据库排序法：文本文件导入数据库，数据库索引排序。
分治法：分治排序，然后集合。
位图法：形成 max+1 的位数组，填入，然后从大到小开始排

Spring 框架相关问题总结：

Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方法。主要是简化 spring 的使用难度，简化了配置复杂度，使开发者快速上手。优点有：独立运行，简化配置，自动配置，上手容易。

Spring Boot 的核心配置文件有哪几个？它们的区别是什么？

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

.properties 和 .yml

Spring Boot 的核心注解是哪个：

@SpringBootApplication，他包含了三个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

从 Spring3.0，@Configuration 用于定义配置类，可替换 xml 配置文件，被注解的类内部包含有一个或多个被@Bean 注解的方法，这些方法将会被 AnnotationConfigApplicationContext 或 AnnotationConfigWebApplicationContext 类进行扫描，并用于构建 bean 定义，初始化 Spring 容器。

@SpringBootConfiguration 也是来源于 @Configuration，二者功能都是将当前类标注为配置类，并将当前类里以 @Bean 注解标记的方法的实例注入到 srping 容器中，实例名即为方法名。

@ComponentScan：Spring 组件扫描。

用于将一些标注了特定注解的 bean 定义批量采集注册到 Spring 的 IoC 容器之中，这些特定的注解大致包括：@Controller@Entity@Component@Service@Repository

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项。

@EnableAutoConfiguration 注解启用自动配置，其可以帮助 SpringBoot 应用将所有符合条件的 @Configuration 配置都加载到当前 IoC 容器之中，

6、开启 Spring Boot 特性有哪几种方式？

1）继承 spring-boot-starter-parent 项目 2）导入 spring-boot-dependencies 项目依赖

SpringBoot 内置 Tomcat/Jetty 容器，可以独立运行。

8、运行 Spring Boot 有哪几种方式？

1）打包用命令或者放到容器中运行

3）直接执行 main 方法运行

2）用 Maven/ Gradle 插件运行

Spring Boot 自动配置原理是什么？

扫描所有具有 META-INF/spring.factories 的 jar 包。spring-boot-autoconfigure-x.x.x.x.jar 里就有一个这样的 spring.factories 文件。

这个 spring.factories 文件也是一组一组的 key=value 的形式，其中一个 key 是 EnableAutoConfiguration 类的全类名，而它的 value 是一个 xxxxAutoConfiguration 的类名的列表，这些类名以逗号分隔。

这个@EnableAutoConfiguration 注解通过@SpringBootApplication 被间接的标记在了 Spring Boot 的启动类上。在 SpringApplication.run(...)的内部就会执行 selectImports()方法，找到所有 JavaConfig 自动配置类的全限定名对应的 class，然后将所有自动配置类加载到 Spring 容器中。

Spring Boot 启动的时候会通过@EnableAutoConfiguration 注解找到 META-INF/spring.factories 配置文件中的所有自动配置类，并对其进行加载，而这些自动配置类都是以 AutoConfiguration 结尾来命名的，它实际上就是一个 JavaConfig 形式的 Spring 容器配置类，它能通过以 Properties 结尾命名的类中取得在全局配置文件中配置的属性如：server.port，而 XxxxProperties 类是通过 @ConfigurationProperties 注解与全局配置文件中对应的属性进行绑定的。

Starters

Starters 可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。Starters 包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到支持的管理传递性依赖。

Spring Boot 读取配置的几种方式:

@Value 注解读取方式: @Value("${info.address}")

@ConfigurationProperties 注解读取方式: @ConfigurationProperties(prefix = "info")

读取指定文件资源目录下建立 config/db-config.properties:

@PropertySource+@ConfigurationProperties 注解读取方式]

@PropertySource+@Value 注解读取方式

Spring Boot 支持 Java Util Logging,Log4j2,Lockback 作为日志框架，如果你使用 starters 启动器，**Spring Boot 将使用 Logback 作为默认日志框架**。无论使用哪种日志框架，Spring **Boot 都支持配置将日志输出到控制台或者文件中。**

spring-boot-starter 启动器包含 spring-boot-starter-logging **启动器并集成了 slf4j 日志抽象及 Logback 日志框架。**

SpringBoot 热部署：

Spring Boot 提供了一个名为 spring-boot-devtools 的模块来使应用支持热部署，提高开发者的开发效率，无需手动重启 Spring Boot 应用。

添加依赖模块 devtools,在配置文件中自定义。

```
# 热部署开关，false 即不启用热部署
spring.devtools.restart.enabled: true

# 指定热部署的目录
#spring.devtools.restart.additional-paths: src/main/java

# 指定目录不更新
spring.devtools.restart.exclude: test/**
```

pring Boot 可以兼容老 Spring 项目吗，如何做？

可以兼容，使用 @ImportResource 注解导入老 Spring 项目配置文件

Mybatis

Mybatis 是一个半 ORM（对象关系映射）框架，它内部封装了 JDBC，**开发时只需要关注 SQL 语句本身**，不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。MyBatis 可以使用 XML 或注解来**配置和映射原生信息，将 POJO 映射成数据库中的记录**。

通过 xml 文件或注解的方式将要执行的各种 statement 配置起来，并**通过 java 对象和 statement 中 sql 的动态参数进行映射生成最终执行的 sql 语句**，最后由 **mybatis 框架执行 sql 并将结果映射为 java 对象并返回**。（从执行 sql 到返回 result 的过程）。

Mybaits 的优点：基于 SQL 语句编程，相当灵活；与 JDBC 相比，减少了 50%以上的代码量。与 Spring

很好的集成。
MyBatis 框架的缺点：**SQL 语句的编写工作量较大**；SQL 语句**依赖于数据库**，导致数据库移植性差，不能随意更换数据库。
MyBatis 与 Hibernate 有哪些不同？半 ORM 框架，MyBatis 需要程序员自己编写 Sql 语句。编写原生态 sql，可以严格控制 sql 执行性能，灵活度高。Hibernate **对象/关系映射能力强，数据库无关性好**，对于关系模型要求高的软件，如果用 hibernate 开发可以**节省很多代码**，提高效率。
#{}和${}的区别是什么？**#{}是预编译处理，${}是字符串替换**。Mybatis 在处理#{}时，会将 sql 中的#{}替换为?号，调用 PreparedStatement 的 set 方法来赋值；使用#{}可以有效的防止 SQL 注入，提高系统安全性。$方式一般用于传入数据库对象，例如传入表名.
当实体类中的属性名和表中的字段名不一样 ，怎么办
1、sql 语句中定义字段名的别名 2、通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。
模糊查询：

```
string wildcardname = “%smi%”;
list<name> names = mapper.selectlike(wildcardname);

<select id=”selectlike”>
  select * from foo where bar like #{value}
</select>
```

2、可能引起 sql 注入。

```
<select id=”selectlike”>
        select * from foo where bar like "%"#{value}"%"
</select>
```

MyBatis 插入数据返回插入对象的主键属性设置：
useGeneratedKeys="true" keyProperty="userId"

Dao 接口即 Mapper 接口。接口的全限名，就是映射文件中的 namespace 的值；接口的方法名，就是映射文件中 Mapper 的 Statement 的 id 值；接口方法内的参数，就是传递给 sql 的参数。Mapper 接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为 key 值，可唯一定位一个 MapperStatement。在 Mybatis 中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个 MapperStatement 对象。
Mapper 接口里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略。Mapper 接口的工作原理是 JDK 动态代理，Mybatis 运行时会使用 JDK 动态代理为 Mapper 接口生成代理对象 proxy，代理对象会拦截接口方法，转而执行 MapperStatement 所代表的 sql，然后将 sql 执行结果返回。

Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页。**分页插件的基本原理是使用 Mybatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。**

使用 sql 的别名，或者 ResultMap,逐一定义数据库列名和对象属性之间的映射关系。有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

动态 sql：执行原理是根据表达式的值 完成逻辑判断并动态拼接 sql 的功能。：trim | where | set | foreach | if | choose | when | otherwise | bind。

Xml 映射文件中，除了常见的 select|insert|updae|delete 标签之外，还有哪些标签？

答：<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态 sql 的 9 个标签，其中<sql>为 sql 片段标签，通过<include>标签引入 sql 片段，<selectKey>为不支持自增的主键生成策略标签。

不同的 Xml 映射文件，如果**配置了 namespace，那么 id 可以重复；如果没有配置 namespace，那么 id 不能重复；**
原因**就是 namespace+id 是作为 Map<String, MapperStatement>的 key 使用的**，如果没有 namespace，就剩下 id，那么，id 重复会导致数据互相覆盖。有了 namespace，自然 id 就可以重复，namespace 不同，namespace+id 自然也就不同。

MyBatis 实现一对多有几种方式,怎么操作的？
有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在 resultMap 里面的 collection 节点配置一对多的类就可以完成；嵌套查询是先查一个表,根据这个表里面的 结果的外键 id,去再另外一个表里面查询数据,也是通过配置 collection,但另外一个表的查询通过 select 节点配置。

接口绑定，**就是在 MyBatis 中任意定义接口,然后把接口里面的方法和 SQL 语句绑定, 我们直接调用接口方法就可以,**这样比起原来了 SqlSession 提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,**一种是通过注解绑定，就是在接口的方法上面加上 @Select、@Update 等注解，里面包含 Sql 语句来绑定；**另外一种就是**通过 xml 里面写 SQL 来绑定, 在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名。**当 Sql 语句比较简单时候,用注解绑定, 当 SQL 语句比较复杂时候,用 xml 绑定,一般用 xml 绑定的比较多。

ResultMap 和 ResultType：
两者都是表示查询结果集与 java 对象之间的一种关系，处理查询结果集，映射到 java 对象。
resultMap：表示将查询结果集中的列一一映射到 bean 对象的各个属性。
ResultType:表示的是 bean 中的对象类，此时可以省略掉 resultMap 标签的映射，但是必须保证查询结果集中的属性 和 bean 对象类中的属性是一一对应的，此时大小写不敏感，但是有限制。
parameterType 直接将查询结果列值类型自动对应到 java 对象属性类型上，不再配置映射关系一一对应。

4、Spring MVC 的主要组件？
（1）**前端控制器 DispatcherServlet**（不需要程序员开发）
作用：**接收请求、响应结果，相当于转发器**，有了 DispatcherServlet 就减少了其它组件之间的耦合度。
（2）**处理器映射器 HandlerMapping**（不需要程序员开发）
作用：**根据请求的 URL 来查找 Handler**
（3）处**理器适配器 HandlerAdapte**r
注意：在编写 Handler 的时候要按照 HandlerAdapter 要求的规则去编写，这样适配器 HandlerAdapter 才可以正确的去执行 Handler。
（4）**处理器 Handler**（需要程序员开发）
（5）**视图解析器 ViewResolver**（不需要程序员开发）
作用：进行视图的解析，**根据视图逻辑名解析成真正的视图**（view）
（6）视图 View（需要程序员开发 jsp）
View 是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf 等等）
SpringMVC 常用的注解有哪些？
@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。

@RequestBody：注解实现接收 http 请求的 json 数据，将 json 转换为 java 对象。
@ResponseBody：注解实现将 conreoller 方法返回对象转化为 json 对象响应给客户。
控制器注解：一般用@Controller 注解，也可以使用@RestController,@RestController 注解相当于@ResponseBody ＋ @Controller。方法无法返回到 JSP 页面，视图解析器不起作用。
13、如果在拦截请求中，我想拦截 get 方式提交的方法,怎么配置？
答：可以在@RequestMapping 注解里面加上 method=RequestMethod.GET。
14、怎样在方法里面得到 Request,或者 Session？
答：直接在方法的形参中声明 request,SpringMvc 就自动把 request 对象传入。
15、如果想在拦截的方法里面得到从前台传入的参数,怎么得到？
答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。
16、如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象？
答：直接在方法中声明这个对象,SpringMvc 就自动会把属性赋值到这个对象里面。
17、SpringMvc 中函数的返回值是什么？
答：返回值可以有很多类型,有 String, ModelAndView。ModelAndView 类把视图和数据都合并的一起的，但一般用 String 比较好。
18、SpringMvc 用什么对象从后台向前台传递数据的？
答：通过 ModelMap 对象,可以在这个对象里面调用 put 方法,把对象加到里面,前台就可以通过 el 表达式拿到。
19、怎么样把 ModelMap 里面的数据放入 Session 里面？
答：可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key。

注解原理：
注解本质是一个继承了 Annotation 的特殊接口，其具体实现类是 Java 运行时生成的动态代理类。我们通过反射获取注解时，返回的是 Java 运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用 AnnotationInvocationHandler 的 invoke 方法。该方法会从 memberValues 这个 Map 中索引出对应的值。而 memberValues 的来源是 Java 常量池。

Spring

3、Spring 的 AOP 理解：
OOP 面向对象，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。
AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。
AOP 实现的关键在于 代理模式，AOP 代理主要分为静态代理和动态代理。静态代理的代表为 AspectJ；动态代理则以 Spring AOP 为代表。
（1）AspectJ 是静态代理的增强，所谓静态代理，就是 AOP 框架会在编译阶段生成 AOP 代理类，因此也称为编译时增强，他会在编译阶段将 AspectJ(切面)织入到 Java 字节码中，运行的时候就是增强之后的 AOP 对象。
（2）Spring AOP 使用的动态代理，所谓的动态代理就是说 AOP 框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个 AOP 对象，这个 AOP 对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。
Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理：
①JDK 动态代理只提供接口的代理，不支持类的代理。核心 InvocationHandler 接口和 Proxy 类，InvocationHandler 通过 invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy 利用 InvocationHandler 动态创建一个符合某一接口的的实例，生成目标类的代理对

象。

②如果代理类没有实现 InvocationHandler 接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现 AOP。CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

（3）静态代理与动态代理区别在于生成 AOP 代理对象的时机不同，相对来说 AspectJ 的静态代理方式具有更好的性能，但是 AspectJ 需要特定的编译器进行处理，而 Spring AOP 则无需特定的编译器处理。

InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy 是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

4、Spring 的 IoC

（1）IOC 就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到 Spring 容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。DI 依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖 IoC 容器来动态注入对象需要的外部资源。

（2）最直观的表达就是，IOC 让对象的创建不用去 new 了，可以由 spring 自动生产，使用 java 的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

（3）Spring 的 IOC 有三种注入方式 ：构造器注入、setter 方法注入、根据注解注入。

（1）BeanFactory：**是 Spring 里面最底层的接口，包含了各种 Bean 的定义，读取 bean 配置文档，管理 bean 的加载、实例化，控制 bean 的生命周期，维护 bean 之间的依赖关系。**ApplicationContext 接口作为 BeanFactory 的派生，除了提供 BeanFactory 所具有的功能外，还提供了更完整的框架功能：

①继承 MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册 bean 的事件。

④同时加载多个配置文件。

⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层。

（2）**①BeanFactroy 采用的是延迟加载形式来注入 Bean 的**，即只有在使用到某个 Bean 时(调用 getBean())，才对该 Bean 进行加载实例化。而 ApplicationContext 是启动时创建所有的 bean;

6、请解释 Spring Bean 的生命周期？

首先说一下 Servlet 的生命周期：实例化，初始 init，接收请求 service，销毁 destroy；

Spring 上下文中的 Bean 生命周期也类似，如下：

（1）实例化 Bean：

对于 BeanFactory 容器，**当客户向容器请求一个尚未初始化的 bean 时，或初始化 bean 的时候需要注入另一个尚未初始化的依赖时，容器就会调用 createBean 进行实例化。**对于 ApplicationContext 容器，当容器启动结束后，**通过获取 BeanDefinition 对象中的信息，实例化所有的 bean。**

（2）设置对象属性（依赖注入）：

实例化后的对象被封装在 BeanWrapper 对象中，紧接着，Spring 根据 BeanDefinition 中的信息 以及 通过 BeanWrapper 提供的设置属性的接口**完成依赖注入。**

（3）处理 Aware 接口：

接着，**Spring 会检测该对象是否实现了 xxxAware 接口，并将相关的 xxxAware 实例注入给 Bean：**

①如果这个 Bean 已经实现了 **BeanNameAware 接口，会调用它实现的 setBeanName(String beanId)方法**，此处传递的就是 Spring 配置文件中 Bean 的 id 值；
②如果这个 Bean 已经实现了 **BeanFactoryAware 接口，会调用它实现的 setBeanFactory()方法**，传递的是 Spring 工厂自身。
③ 如 果 这 个 Bean 已 经 实 现 了 **ApplicationContextAware 接 口 ， 会 调 用 setApplicationContext(ApplicationContext)方法**，传入 Spring 上下文；
（4）BeanPostProcessor：
如果想对 Bean 进行一些**自定义的处理，那么可以让 Bean 实现了 BeanPostProcessor 接口**，那将会调用 **postProcessBeforeInitialization(Object obj, String s)方法。**
（5）InitializingBean 与 init-method：
如果 Bean 在 Spring 配置文件中配置了 **init-method 属性，则会自动调用其配置的初始化方法。**
（6）如果这个 Bean 实现了 **BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法；**由于这个方法是在 Bean 初始化结束时调用的，**所以可以被应用于内存或缓存技术；**
以上几个步骤完成后，Bean 就已经被正确创建了，之后就可以使用这个 Bean 了。
（7）DisposableBean：
当 Bean 不再需要时，会经过清理阶段，**如果 Bean 实现了 DisposableBean 这个接口，会调用其实现的 destroy()方法；**
（8）destroy-method：
最后，如果这个 Bean 的 Spring 配置中配置了 **destroy-method 属性，会自动调用其配置的销毁方法**

Spring 的自动装配：
（1）no：默认的方式是不进行自动装配的，通过手工设置 ref 属性来进行装配 bean。
（2）**byName：通过 bean 的名称进行自动装配**，如果一个 bean 的 property 与另一 bean 的 name 相同，就进行自动装配。
（3）**byType：通过参数的数据类型进行自动装配。**
（4）constructor：利用构造函数进行装配，并且构造函数的参数通过 byType 进行装配。
（5）autodetect：自动探测，如果有构造方法，通过 construct 的方式自动装配，否则使用 byType 的方式自动装配。
@Autowired 和@Resource 之间的区别
(1) @Autowired **默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在**（可以设置它 required 属性为 false）。
(2) @Resource **默认是按照名称来装配注入的，只有当找不到与名称匹配的 bean 才会按照类型来装配注入。**

Spring 事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring 是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过 binlog 或者 redo log 实现的。声明式事务管理建立在 AOP 之上的。其本质是通过 AOP 功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

AOP 通知类型：前置通知，后置通知，返回后通知，抛出异常通知，环绕通知。
头条代码问题：
1、数组的逆序数　　剑指

```
public int InversePairs(int [] array) {
        int count = 0;
        for(int i=0;i<array.length-1;i++){
            for(int j=i+1;j<array.length;j++){
```

```
                    if(array[i] > array[j])
                        count++;
                }
            }
            return count;
        }
```

真正的优化要用归并排序；

归并排序：先 int mid = left+(right-left)/2;然后

mergeSort(nums,left,mid,right);

mergerSort(nums,left,mid,right);知道归到还剩每组一人，咱再并。

并的时候，之前是有序数组排序。这里面要加个：从后往前遍历，然后移动到临时数组。比较的时候，如果 nums[left] > nums[right]那边的，那么 count += right – mid + 1;这些都是逆序数组对。

2、LRU　　leetcode:146

```
class LRUCache {

  Node head = new Node(0, 0), tail = new Node(0, 0);

  Map<Integer, Node> map = new HashMap();

  int capacity;

  public LRUCache(int _capacity) {

     capacity = _capacity;

     head.next = tail;

     tail.prev = head;

  }

  public int get(int key) {

     if(map.containsKey(key)) {

        Node node = map.get(key);

        remove(node);

        insert(node);

        return node.value;

     } else {

        return -1;

     }

  }

  public void put(int key, int value) {

     if(map.containsKey(key)) {
```

```
        remove(map.get(key));
    }
    if(map.size() == capacity) {
        remove(tail.prev);
    }
    insert(new Node(key, value));
}
private void remove(Node node) {
    map.remove(node.key);
    node.prev.next = node.next;
    node.next.prev = node.prev;
}
private void insert(Node node){
    map.put(node.key, node);
    Node headNext = head.next;
    head.next = node;
    node.prev = head;
    headNext.prev = node;
    node.next = headNext;
}
class Node{
    Node prev, next;
    int key, value;
    Node(int _key, int _value) {
        key = _key;
        value = _value;
    }
}
```

```
}
```

3、最长回文序列　leetocde 5

```
class Solution {
    int left = 0;
    int max =    0;
    public String longestPalindrome(String s) {
        if(s == null || s.length() < 1)
            return "";
        for(int i =0;i<s.length();i++){
            expend(s,i,i);
            expend(s,i,i+1);
        }
        return s.substring(left,left+max);
    }
    public void expend(String str, int i,int j ){
        while(i>=0 && j < str.length() && str.charAt(i) == str.charAt(j)){
            i--;
            j++;
        }
        if(max < j-i-1){
            max = j-i-1;
            left = i+1;
        }
    }

}
```

4、矩阵中的最长递增路径，可以上下左右一起都走；　leetcode**329**

```
class Solution {
    private static final int[][] dirs={{0,1},{1,0},{-1,0},{0,-1}};
    public int longestIncreasingPath(int[][] matrix) {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
        int row = matrix.length;
        int col = matrix[0].length;
        int[][] nums = new int[row][col];
        int max = 0;
        for(int i=0;i<row;i++){
            for(int j=0;j<col;j++){
                int len = helper(matrix,i,j,nums,row,col);
                max = Math.max(max,len);
            }
        }
        return max;
    }
    public int helper(int[][] matrix, int i, int j,int[][] nums,int row, int col){
        if(nums[i][j] != 0) return nums[i][j];
        int max = 1;
        for(int[] dir : dirs){
```

```
                int x = i+dir[0];
                int y = j+dir[1];
                if(x<0 || x >= row || y<0|| y>= col || matrix[x][y] <= matrix[i][j] ) continue;
                    int len = 1 + helper(matrix,x,y,nums,row,col);
                    max = Math.max(max,len);
            }
            nums[i][j] = max;
            return max;
        }
    }
```

5、判断一个二叉树是另一个二叉树的子树 剑指

先判断一个树是不是包含另一个树。

```
contains(TreeNode root1,TreeNode roo2){
    if(root2 == null) return true;
    if(root1 == null || root1.val != root2.val) return false;
    return contains(root1.left,root2.left) && contains(root1.right,root2.right);
}
```

然后判断 contains(root,root1) || check(root.left,root1) || check(root.right,root2);

6、归并排序的时间复杂度 NlogN;

```
Public void mergerSort(int[] nums,int left,int right){
    if(left < right){
        int mid = left+(right-left)/2;
        mergerSort(nums,left,mid);
        mergerSort(nums,mid+1,right);
        merger(nums,left,mid,right);
    }
}
Public void merger(int[] nums,int left,int mid, int right){
    Int[] temp = int[right-left+1];
    Int I = left,j = mid+1;
    Int k = 0;
    While(i<=mid && j<=right){
        if(nums[i] < nums[j])
            temp[k++] = nums[i++];
        else
            temp[k++] = nums[j++];
    }
    While(i<=mid)
    Temp[k++] = nums[i++];
    While(j<=right)
    Temp[k++] = nums[j++];
    For(int i=0;i<temp.length;i++){
        Nums[i+left] = temp[i];
    }
}
```

7、海量数据的前 K 个。 海量

8、求给出 01 矩阵中的最大正方形面积（全为 1） lc221

动态规划， 当 m[i][j] == 1 时，dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1]) + 1;

```
public int maximalSquare(char[][] matrix) {
```

```
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;
        int r = matrix.length;
        int c = matrix[0].length;
        int[][] dp = new int[r+1][c+1];
        int max = 0;
        for(int i =1;i<=r;i++){
            for(int j=1;j<=c;j++){
                if(matrix[i-1][j-1] == '1'){
                    dp[i][j] = Math.min(dp[i-1][j-1],Math.min(dp[i-1][j],dp[i][j-1]))+1;
                    max = Math.max(max,dp[i][j]);
                }
            }
        }
        Return max*max;
    }
```

9、求二叉树中距离最远的节点　leetcode543？？(任意两个节点之间最长的距离？)

递归求二叉树深度的时候，求每个节点左子树深度加右子树深度和的最大。

```
class Solution {
    int max = 0;
    public int diameterOfBinaryTree(TreeNode root) {
        getDepth(root);
        return max;
    }
    public int getDepth(TreeNode root){
        if(root == null) return 0;
        int left = getDepth(root.left);
        int right = getDepth(root.right);
        max = Math.max(max,left+right);
        return Math.max(left,right)+1;
    }
}
```

10、判断字符串是否为合法 IPV4 地址　　lc**468,是否为合法的 ipv4 或 ipv6 地址**

```
    String[] tokens = ip.split("\\.");
    if(tokens.length != 4 ) return false;
    for(String token : tokens){
        if(!isValidIP4(token)) return false;
    }

    public boolean isValidIP4(String token){
        if(token.startsWith("0") && token.length()>1) return false;
    try {
        int parsedInt = Integer.parseInt(token);
        if(parsedInt<0 || parsedInt>255) return false;
        if(parsedInt==0 && token.charAt(0)!='0') return false;
    } catch(NumberFormatException nfe) {
        return false;
    }
    return true;
```

```
    }
```
11、数组值为 1-n，各出现一次，先加入 x（x 也是 1-n 的范围），找出 x

12、给定 n，计算 15n，不用+*/，想了好久最后根据提示做出来了，n<< 4-n

13、给定字符数组 chars，将其右移 n 位，空间复杂度最低，先整体反转，再反转 0~n 对应的字符串，和 n+1~len-1 对应的字符。

```
    public String LeftRotateString(String str,int n) {
        if ( str == null || str.length() == 0 ||n    < 0 )
            return "";
        char[] chas = str.toCharArray();
        int len = chas.length;
        swap(chas,0,len-1);
        n = n % len;
        swap(chas,0,len-n-1);
        swap(chas,len-n,len-1);
        return String.valueOf(chas);
    }
    public void swap(char[] chars,int start, int end){
        char temp;
        while (start < end){
            temp =    chars[start];
            chars[start] = chars[end];
            chars[end] = temp;
            start++;
            end--;
        }
}
```

14、100 层楼，只有两个鸡蛋，找出鸡蛋会在哪一层楼被摔碎

动态规划：dp(a,b)表示 a 块石头扔 b 次，共能测出多少层楼。dp(x,1)=1，表示扔一次就能测一个楼。dp(x,y) = dp(x,y-1)+dp(x-1,y-1)+1 就是状态装换方程。

15、reverse linked list in a group of k

如果是翻转链表，那就是头插入翻转。如果是

```
public ListNode reverseKGroup(ListNode head, int k) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        int len =0;
        while(head != null){
            len++;
            head = head.next;
        }
        ListNode pre = dummy,cur = pre.next;
        for(int j=len;j>=k;j -= k){
            for(int i=1;i<k;i++){
                ListNode next = cur.next;
                cur.next = next.next;
                next.next = pre.next;
                pre.next = next;
            }
```

```
            pre = cur;
            cur = pre.next;
        }
        return dummy.next;
    }
```

16、给十桶乒乓球(每桶中乒乓球数量无限),有一个桶的球重 9g,其余桶均为 10g。找到 9g 的那桶要测几次

17、如何空间 O(0)实现两个数的互换

```
a = a+b;b=a-b;a=a- b;
a = a^b;b=a^b;a=a^b;
```

18、IP 地址的 Regex

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
^((2(5[0-5]|[0-4]\d))|[0-1]?\d{1,2})(\.((2(5[0-5]|[0-4]\d))|[0-1]?\d{1,2})){3}$
```

19、Linux 的基本系统指令

```
top,ps,grep,tar,lsof,find,
```

20、the longest path in a binary tree lc124:

```
class Solution {
    int maxValue = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        MaxPathDown(root);
        return maxValue;
    }
    public int MaxPathDown(TreeNode root){
        if(root == null) return 0;
        int left = Math.max(0,MaxPathDown(root.left));
        int right = Math.max(0,MaxPathDown(root.right));
        maxValue = Math.max(maxValue,left+right+root.val);
        return Math.max(left,right)+root.val;
    }
}
```

21、the largest consecutive sum in an array

```
public int consectiveSum(int[] nums){
    if(nums == null || nums.length == 0) return 0;
    int[] dp = new int[nums.length];
    dp[0] = nums[0];
    int max = dp[0];
    for(int i=1;i<nums.length;i++){
        dp[i] = Math.max(nums[i],dp[i-1]+nums[i]);
        max = Math.max(max,dp[i]);
    }
    Return max;
}
```

22、给一个很大的 log file，形式是 user id, login time, logout time，如何找到峰值

23、LeetCode 41 Find missing positive,

将符合统计的正整数(大于 0，小于 n)放入正确的位置，A[I] 放入 A[A[i]-1]如，5 放入 A[4]；然后头从遍历，遇到 i+1 != A[i]的 i+1 就是缺失的。

```
public int firstMissingPositive(int[] nums) {
        for(int i=0;i<nums.length;i++){
            while(nums[i] > 0 && nums[i] <= nums.length && nums[i] != nums[nums[i]-1])
                swap(nums,i,nums[i]-1);
```

```
        }
        for(int i=0;i<nums.length;i++){
            if((i+1) != nums[i])
                return i+1;
        }
        return nums.length+1;
    }
    public void swap(int[] nums,int i,int j){
        int temp = nums[i];
        nums[i]  = nums[j];
        nums[j]  = temp;
    }
```

这个地方一定要用 while,因为一个将当前遍历的人换到正确的地方之后，保证换过来人也要去正确的地方。

24、给一个小于一亿的中文数字字符串,转化成数字格式。

25、一个数组,把所有的 0 都移动到末尾,还要保持顺序不乱，问了其时间和空间复杂度, O(n)和 O(1)

从头遍历数组，且维护一个非 0 的指针。遍历时，如果数字不是 0，那么这个数字和非 0 指针的数字进行互换。

```
public void moveZeros(int[] nums){
    int pos = 0;
    for(int i=0;i<nums.length;i++){
        if(nums[i] != 0){
            int temp = nums[i];
            nums[i] = nums[pos];
            nums[pos] = temp;
            pos++;
        }
    }
}
```

优化的时候，如果遍历指针和非 0 指针不一样，说明存在 0 了，且 pos 指的就是 0。直接进行替换就行。

```
public void moveZeros(int[] nums){
    int pos = 0;
    for(int i=0;i<nums.length;i++){
        if(nums[i] != 0){
            if(i != pos){
                nums[pos] = nums[i];
                nums[i] = 0;
            }
            pos++;
        }
    }
}
```

26、java 的设计模式

27、每 n 个反转一次链表。

28、罗马数字转整数 leetcode13

罗马数字的特点:遍历数组，如果当前的大于等于后面的字母代表的数，res += map.get(str.charAt(i));否则 res -= map.get(str.charAt(i)); result += map.get(chars[i]);

```
    public int romanToInt(String s) {
        Map<Character,Integer> map = new HashMap<>();
        map.put('I',1);
        map.put('V',5);
```

```
        map.put('X',10);
        map.put('L',50);
        map.put('C',100);
        map.put('D',500);
        map.put('M',1000);
        int i=0,j=1;
        int res = 0;
        for(;j<s.length();i++,j++){
            if(map.get(s.charAt(i)) >= map.get(s.charAt(j)))
                res += map.get(s.charAt(i));
            else
                res -= map.get(s.charAt(i));

        }
        res += map.get(s.charAt(i));
        return res;
    }
```

29、二叉树中的最大路径和，leetcode124

```
class Solution {
    int maxValue = Integer.MIN_VALUE;
    public int maxPathSum(TreeNode root) {
        MaxPathDown(root);
        return maxValue;
    }
    public int MaxPathDown(TreeNode root){
        if(root == null) return 0;
        int left = Math.max(0,MaxPathDown(root.left));
        int right = Math.max(0,MaxPathDown(root.right));
        maxValue = Math.max(maxValue,left+right+root.val);
        return Math.max(left,right)+root.val;
    }
}
```

30、二叉树的序列化和反序列化

sqrt(2)约等于 1.414，要求不用数学库，求 sqrt(2)精确到小数点后 10 位

如何实现一个高效的单向链表逆序输出？

31、输入一个数组，输出数组中满足条件的数字，条件为：数组中当前元素的值大于等于它前面所有的元素，小于等于它后面所有的元素

首先，双层遍历将符合条件的选出来 N^2;

优化：整一个数组 cand[],记录当前符合条件的数，同时记录当前遍历过的最大数。

遍历数组：如果当前 nums[i] > max,说明这老哥符合条件，直接入 cand[]数组。index++,表示符合条件的个数加 1。如果 < max，说明前面存在比他大的，这个不合适。而且检验当前数 nums[i]出现之后，cand 数组不合格的去除。

```
public int findNums(int[] nums){
    if(nums == null || nums.length == 0) return 0;
    int[] cand = new int[nums.length];
    int index = 0;
    cand[index++] = nums[0];
```

```
        int max = nums[0];
        for(int i=1;i<nums.length;i++){
            if(nums[i] >= max){
                cand[index++] = nums[i];
                max = nums[i];
            }else{
                while(index > 0 && cand[index-1] > nums[i] )
                    index--;
            }
        }
        return index;
}
```

32、给出一个数字，对数字的两位进行交换，只能交换一次，输出可能结果中的最小数字

```
public int maximumSwap(int num) {
        char[] chas = String.valueOf(num).toCharArray();
        int[] buckets = new int[10];
        for(int i=0;i<chas.length;i++){
            buckets[chas[i]-'0'] = i;
        }
        for(int i=0;i<chas.length;i++){
            for(int k=9;k > chas[i]-'0';k--){
                if(buckets[k] > i){
                    char c = chas[i];
                    chas[i] = chas[buckets[k]];
                    chas[buckets[k]] = c;
                    int res = Integer.valueOf(new String(chas));
                    return res;
                }
            }
        }
        return num;
 }
```

33、输入一个字符串，字符串中字符全部为数字，在字符串中插入 '.' 使得结果为合法的 ip 地址，输出全部可能的结果

34、输入一个矩阵，矩阵中元素为 0 或 1，找出满足条件的正方形的最大边长，条件为正方形中的元素全部为 1

35、求出当天直播间内同时在线人数的最大值

36、基数排序

37、对于一个链表，请设计一个时间复杂度为 O(n),额外空间复杂度为 O(1)的算法，判断其是否为回文结构。

先 fast 和 slow 指针双走，找到中间的节点。然后头插法将后面的节点逆序。然后比较是否回文。

```
    public boolean isPalindrome(ListNode head) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode fast = dummy,slow = dummy;
        while(fast != null && fast.next != null){
            fast = fast.next.next;
            slow = slow.next;
        }

        ListNode l1 = head;
        ListNode l2 = reverse(slow.next);
```

```
        while(l2!=null){
            if(l1.val != l2.val)
                return false;
            l1 = l1.next;
            l2 = l2.next;
        }
        return true;
    }
    public ListNode reverse(ListNode head){
        if(head == null)
            return null;
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode cur = head,pre = dummy;
        while(cur.next != null){
            ListNode next = cur.next;
            cur.next = next.next;
            next.next = pre.next;
            pre.next = next;
        }
        return dummy.next;

    }
```

38、当数据量较大时，快速排序和堆排序性能比较

39、1~100，每轮依次划掉奇数位置上的数，最后会剩下哪个数？

40、A,B 两个人赌博，胜率各自为 0.5，现在设定获胜规则：A 只要获胜 2 局以上就获胜，B 要 3 局以上才会获胜，问 A,B 双方获胜概率多少

41、区间最大最小值。两个长度为 n 的序列 a,b,问有多少区间[l,r] 满足 max(a[l,r])<min(b[l,r])即 a[l,r]的最大值小于 b[l,r]的最小值

42、最大不重复子串

43、字符串 s="0123456789101112..."返回 s[i]，让采用逐步缩小范围的方法，分为三类：一位数、二位数、三位数，给定一个 m 直接可以判断它落在哪个区间，然后从区间起始开始填数字串。

44、反转单链表。

45、复杂链表复制。

46、数组 a,先单调地址再单调递减，输出数组中不同元素个数。要求：O(1)空间复杂度，不能改变原数组

这个应该是先求出递减开始的地方。然后前面半段从后往前遍历，后面半段往后遍历。这都是有序的数列。然后求重复的数有几个。然后总的减去重复的数就行。

47、.64 匹马，8 个赛道，找最快的 4 匹马。

先分为 8 组，每组进行赛跑。把每组的后四个淘汰。然后取每组的第一赛跑。将后四位所在的组排布淘汰。9 次之后，还剩 16 匹马。A1 是必然第一名。D2D3D4,B4,C3C4 必然被淘汰。其他 9 个需要确定前三。将 A234B23C12D1 八个进行排位。然后看 B2 和 C1 能否进前三。如果能，这次确定。如果不能 A234B1 排位。确定前三。

48、64 匹马，8 个赛道，找最快的 8 匹马。

分八组，然后排序。然后每组第一排位，能去掉一些。然后进行其他的排位。很复杂，但次数也不是那么多。

49、二叉搜索树找第 k 小数据，两种方法实现。

中序遍历的递归方法和非递归？

50、长度为 n 的数组，元素大小是 0~n-1,判断数组元素是否有重复的，要求：O(1)时间空间复杂度。

```
public boolean duplicate(int[] nums,int length,int[] duplication) {
```

```
        if(nums == null) return false;
        for(int i=0;i<nums.length;i++){
            while(i != nums[i]){
                if(nums[i] == nums[nums[i]]){
                    duplication[0] = nums[i];
                    return true;
                }
                int temp = nums[i];
                nums[i] = nums[nums[i]];
                nums[nums[i]] = temp;
            }

        }
        return false;
    }
```

51、求一个数组连续子数组的最大区间和。

52、连续子序列最大的和

53、list1/list2 交替打印元素

54、36 进制加法:

```
static Character[] nums = { '0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z' };
  static List<Character> list = Arrays.asList(nums);
public String add(String str1,String str2){
        StringBuffer sb = new StringBuffer();
        char[] chas1 = str.toCharArray();
        char[] chas2 = str.toCharArray();
        int i = chas1.length-1;
        int j = chas2.length-1;
        int crray = 0;
        while(i >= 0 || j >= 0){
            int index1 = j < 0 ? 0 : getIntFromChar(chas1[i]);
            int index2 = j < 0 ? 0 : getIntFromChar(chas2[j]);
            int sum = index1+index2 + carry;
            carry = sum / 36;
            sb.append(list.get(sum%36));
            i--;
            j--;
        }
        if(carry == 1)
            sb.append(1);
        return sb.reverse().toString();
}

public int getIntFromChar(char c){
        if(c >= '0' && c<='9')
            return c-'0';
        if(c>='a' && c<=z")
            return c-'a'+10;
        return -1;
```

}

55、拜占庭问题　　分布式一致性问题

56、对一个八位数有三种操作：　加一、减一、反转　。　至少多少次操作可以把一个八位数 A 变成八位数 B。

57、合并区间

58、快排

59、生产者-消费者　模型

60、排序一个字符串时间要求 O(n)

61、最大正方形面积(不会换题)

62、　空间要求 O(1)

63、给一个有重复数字的数组，求集合{(a,b,c) | a+b+c=0}

3sum，先排序。然后遍历 0-n-2;然后 left = i+1,right = n-1;sum= -nums[i];

如果 nums[left]+nums[right] = sum;就将集合加入。并且去重。

64、两个栈实现队列

入栈，出栈的时候，如果有就出，没有就入栈的全部倒入出栈，再出。

65、二叉树转化为双端链表

中序遍历，或者先序遍历；

66、手写线程池

67、海量数据迅速查询包含关键字的所有文章 B+树？

68、LRUcache

69、给定一个整型数组，查找三个元素相加等于 0 的所有三元组，要求去重

70、之字形打印二叉树

层序遍历，resList+flag.如果 flag 是 false,linkedList 采用头插法；

71、给定一个数组，调整该数组，使其满足堆的性质

72、给定 n 个单词，如果单词组成一致但是元素顺序不一致，该对单词为同位词，例如：abc,bca 为同位词．求所有同位词的集合输出

遍历字符串，将字符串转化成字符数组，然后排序。然后转化成字符串。将排序后的字符串当做 key。如果没有 key,那就 put(key,new ArrayList<String>());如果有 key,取出 list 然后加入。

73、链表，两个链表的公共点

两种方法，分别求链表的长度，然后长的走两步。

或者，遍历两个链表，while(cur1!=cur2),如果链表走到底了，就转身另一个链表。直到遇到公共链表或者到空。

74、10000 万行 iP 地址，求出倒数第二列的所有 IP

75、将 0-n 的整数放到一个长度为 n 的数组中，找出缺失的那个数

遍历数组：

76、在一亿个数中找出最大的 10 个数

77、在一亿个数中找出中间的 10 个数

78、在内存中多叉树树和二叉树哪个更快？二叉树

79、n 条直线可以将空间划分为多少个区域

(n^3+5*n+6)/6

80、二叉树的最长路径长度。

81、二叉树的后续遍历非递归形式

82、SQL 题　596.超过 5 名学生的课

83、买卖股票的最佳时机，只能一次买入和一次卖出

遍历，保留目前遍历过程中最小值。如果当前值大于最小值，算算卖出去利润多少；和保留的最大取大的那个。如果当前值小于最小值，最小值改变。

```
public ing getMax(int[] nums){
    int min = nums[0];
    int max = 0;
    for(int i=1;i<nums.length;i++){
        if(nums[i] < min){
```

```
                min = nums[i];
            }else{
                max = Math.max(nums[i] – min, max);
            }
        }
        return max;
}
```

84、可以进行多次交易的结果，求赚取的最大利润。

多次交易，遍历，如果当前值比昨天的小，那就昨天买今天卖。累加利润

```
public int getMax(int[] nums){
        int count = 0;
        for(int i=1;i<nums.length;i++){
            if(nums[i] > nums[i-1]) count += nums[i] – nums[i-1];
        }
        return count;
}
```

85、一天有 24 小时，一个工人在某一天里短短续 续的工作了几个小时 例如 1 ～ 2 点，4～8 点 5～10 点。。。，求这个工人这一天中一共工作了几个小时。(其实就是区间合并问题)

下面的是将区间进行合并。主要是将数组进行排序，按照数组的第一个开始点进行排。遍历原数组。维护一个 start 和 end，如果当前遍历数组的开始节点小于等于 end,那么。将 end 更新成 end 和结束时间的最大值。否则就将以前的 start、end 整合，然后弄新的 start、end;

```
        public int[][] merge(int[][] intervals) {
            List<int[]> resList = new ArrayList<>();
            if(intervals == null || intervals.length == 0)
                return resList.toArray(new int[0][]);
            Arrays.sort(intervals,(a,b)->(a[0]-b[0]));
            int start = intervals[0][0];
            int end = intervals[0][1];
            for(int[] i : intervals){
                if(i[0] <= end){
                    end = Math.max(end,i[1]);
                }else{
                    resList.add(new int[]{start,end});
                    start = i[0];
                    end = i[1];
                }
            }
            resList.add(new int[]{start,end});
            return resList.toArray(new int[resList.size()][]);
        }
```

86、Redis 热 key 问题

87、(A,B)(A,C)(B,D)(D,A)判断是否有循环引用，提示用拓扑排序

88、蛇形打印二叉树

89、数组找是否存在和为 M 的两个数

90、分布式生成唯一 ID 的方法

雪花算法。

91、KMP

92、实现一个阻塞队列（生产者消费者模型）

93、找出 10000 个数据中第 k 大的数

94、输入一个字符串，包含数字、加减乘除和括号，输出结果，编程

95、给定一个数字 x，要求使用 k 个数字求和可以得到 x，数字从 1-9 中选择，不能重复。

96、手写一个线程池，要提现出复用的思想（一开始没想到，后来提醒我用队列，硬写了一个😅）

97、数组里找两数之和（要求调到最优时间复杂度）。剑指原题。

98、输入一个正整数 N，返回 N 个 '(' 和 N 个 ')' 的所有可能情况。如 N=2，输出)((，(())等。

只返回上题中括号合法的情况，比如 (())，()() 即为合法。

99、返回数组中不存在的最小正整数，要求时间 O(n) 空间 O(1)。如 [-1, 0, 5, 2] 返回 1，[7, 8, 9, 10] 返回 1，[0, 1, 2, 3] 返回 4。

100、在 1～9 里不重复地选择 k 个数，返回所有相加等于 x 的情况。(leetcode 216)

101 变形的二分查找

```
public ListNode mergeKLists(ListNode[] lists) {
        if (lists==null||lists.length==0) return null;
        ListNode dummy = new ListNode(0);
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>(lists.length,new    Comparator<ListNode>(){
            @Override
            public int compare(ListNode o1,ListNode o2){
                if (o1.val<o2.val)
                return -1;
            else if (o1.val==o2.val)
                return 0;
            else
                return 1;
            }
        });

        for(ListNode node:lists){
            if(node != null)
                minHeap.add(node);
        }

       ListNode cur = dummy;
        while(!minHeap.isEmpty()){
            cur.next    = minHeap.poll();
            cur = cur.next;
            if(cur.next != null)
                minHeap.add(cur.next);
        }
        return dummy.next;
    }
```

102、子串匹配问题

76.minimum-window-substring、30.substring-with-concatenation-of-all-words、4.trapping-rain-water，

103、求树的最左下节点

104、用正反面概率不相等的硬币，凑出 50%

两次均为正面：p * p、

第一次正面，第二次反面：p * (1 - p)

第一次反面，第二次正面：(1 - p) * p

两次均为反面：(1 - p) * (1 - p)

连续抛两次，如果两次结果一样，重新抛两次，如果不一样，第一个为正面的概率就是 50%；

105、无序数组中第 k 大的数（quick select）

106、求旋转数组找最小值（二分）

107、判断二叉树是否镜像（递归）

108、反转链表按 k

109、最长重复子串

110、抛硬币，先抛者赢得概率

等比数列求和，a1 = 1/2 , q = 1/4 , Sn = a1 * (1 - q ^n) / (1 - q) , n 趋于无穷 Sn = 1/2 / 3/4 = 2 / 3

111、在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

112、给定一个数组，存储着按照时间排序的股票价格，第 n 个位置的元素为第 n 次交易时的股票价格；现假设只允许你进行一次买，然后在某一时刻卖出（单只股票），请设计算法，求解你可能获得的最大收益，如果股价是非增的，则收益为 0。

113、我们可以用 21 的小矩形横着或者竖着去覆盖更大的矩形。请问用 n 个 21 的小矩形无重叠地覆盖一个 2*n 的大矩形，总共有多少种方法

114、100 亿个无符号整数，取最大的 100 个，内存有限。（BitMap）

115、给定一个矩阵，从左上角开始只能往下或者右走，求到达右下角的最小权值路径

116、一道大数据量的题目，A 文件有 3T，里面放的是 uid+uname，B 文件 2T，里面放的是 uid+unage，找出相同的 uid 并写成 uid+uname+uage 的样子，限制内存 2G

117、求一棵树的镜像，给我一个 List<User>，User 有自己的 id 和父亲的 id，要我转成一棵树

118、有序链表的合并

119、手写 hashmap 的 put 方法

120、罗马数字字符串转阿拉伯数字

121、递增重复数组找 target 出现的范围

```
public int findLeft(int[] nums,int target){

}
```

122、java 的 static 讲下，然后有 6 种用法

123、类方法，代码，静态变量，实例，实例变量分别存到哪里

124、字符串转 Int，如果越界就返回 0

```
public int strToInt(String str){
    if(str == null || str.length == 0) return 0;
    char[] chas = str.toCharArray();
    int index =0;
    int signle = 1;
    if(chas[index] == '+')
        index++;
    else if(chas[index] == '-'){
        index++;
        signle = -1;
    }int res =0;
    for(int i=index;i<str.length();i++){
        char c = chas[i];
        if(c-'0' > =0 && c-'0' <= 9 ){
            res = res * 10 + c-'0';
        }else break;
    }
    return signle * res;
```

}

125、设计一个短域名服务：短信存不了太长网站，需要弄成短域名，你该如何设计一个服务，可以为全国的网址服务。我说的将所有网址存数据库，用 hashcode 来当短域名，如果冲突就拼接随机数，表大就分表，当时忘记说再加缓存了，笨

126、将 List 转成 tree：（当前节点 id，节点名字，父节点 id），（当前节点 id，节点名字，父节点 id），（当前节点 id，节点名字，父节点 id）=》转成一个树，说自己定义数据结构，设计用例，要能编译运行，我用 hashmap 存，然后写到差不多了他说不用写了

127、查看占用了 80 端口的进程

lsof -i:80

128、红黑树和平衡二叉树的区别

129、B 树和 B+树的区别？ B+树的应用查询复杂度和插入复杂度

130、lc400

131、单向链表实现加法

132、求一个整数二进制表示中 1 的个数

```
public int getOne(int x){
    int count = 0;
    while(x != 0){
        x = x & x-1;
        count++;
    }
    return count;
}
```

133、打家劫舍

134、收到礼物最大值

135、五张牌，其中大小鬼为癞子，牌面为 0，判断这五张牌是否能组成顺子，要求不排序只遍历一次

用一个数组 nums[14]来保存遍历到的数。如果出现重复，说明不是顺子。记录这 5 个数的最大值和最小值，为大小鬼不管他们。如果最大值-最小值>4,说明怎么都不可能是顺子。

```
public boolean isContinuous(int[] nums){
    int[] map = new int[14];
    int max = 0,min = 13;
    if(nums == null || nums.length != 5) return false;
    for(int i=0;i<nums.length;i++){
        if(nums[i] == 0 ) continue;
        if(++map[nums[i]] > 1) return false;
        int temp = nums[i];
        if(temp > max)
            max = temp;
        if(temp < min)
            min = temp;
    }
    if(max – min > 4)
        return false;
    return true;
}
```

136、给定一个字符串和一个字符，比如 "abcabca" 和 'a'，随机打印出其中一个 'a' 的下标，保证每个下标输出的概率是一样的，不能开辟额外存储，字符串只能遍历一次

137、 给定一个字符串打印所有的子串，要求不重复，重点是不重复，使用字典树判重

```
Public Set<String> getAllSub(String str){
    int len = str.length();
```

```
    Set<String> set =new HashSet<>();
    for(int i=0;i<len;i++){
        for(int j=i+1;j<len;j++){
            set.add(str.substring(i,j));
        }
    }
    return set;
}
```

138、数组子区间的最大和？知道你肯定刷过这个题，再加一个条件：找到数组中的两个数 A 和 B，要求将 A 和 B 交换之后自区间和是最大的，输出 A、B 和 最大自区间和

139、桌子上有一副牌，循环进行以下操作：（1）将顶部的牌放到桌上 （2）再将当前顶部的牌放入底部，循环到所有牌都放到桌上，假设最后放到桌子上的牌顺序是 13 12 11 .... 1，问初始的牌堆是怎么放的

140、自然数 1-n,排一块组成的字符串，求第 k 位是什么，12345678910，如何第 10 位是 0；

首先，我们先判断第 K 位落到了几位数上了。一位数是 1-9 这个就位，二位数是 10-99 这 90*2；

三位数是 100-999 这 900*3；四位数是 1000-9999，这 9000*4；

```
  public int findNthDigit(int n) {
        int start = 1;
        int len = 1;
        long count = 9;
        while( n > (int)len * count){
            n -= (int)len*count;
            len++;
            count = count * 10;
            start = start * 10;
        }
         start += (n - 1) / len;
         String s = Integer.toString(start);
         return Integer.valueOf(s.charAt((n - 1) % len)+"");
    }
```

将中缀表达式转为后缀表达式，输入 a+b*c/d-a+f/b 输出 abc*d/+a-fb/+

后缀表达式的计算方法：将数据入栈，如果是操作符，就弹出两个，注意顺序，然后计算，然后再压入。直到弹出 n 个之后，最后的结果在栈底。

```
    public int evalRPN(String[] tokens) {
        if(tokens == null) return 0;
        if(tokens.length == 1) return Integer.valueOf(tokens[0]);
        Stack<Integer> stack = new Stack<>();
        String strs = "+-*/";
        for(int i=0;i<tokens.length;i++){
            if(!strs.contains(tokens[i]))
                stack.push(Integer.valueOf(tokens[i]));
            else{
                int a = stack.pop();
                int b = stack.pop();
                int result;
                if(tokens[i].equals("+"))
                    result = a + b;
                else if(tokens[i].equals("-"))
```

```
                result = b-a;
            else if(tokens[i].equals("*"))
                result = a * b;
            else
                result = b / a;
            stack.push(result);
        }
    }
    return stack.pop();
}

public static void main(String[] args){
    String s = "2 * ( 3 + 5 ) + 7 / 1 - 4";
    System.out.println(toPostfix(s));
    System.out.println(calPostfix(toPostfix(s)));
}

public static List<String> toPostfix(String str){
    //假设中序表达式有空格作为连接
    String[] strs = str.split(" ");
    List<String> resList = new ArrayList<>();
    Stack<String> stack = new Stack<>();
    String express = "+-*/()";
    for (int i=0;i<strs.length;i++){
        if (!express.contains(strs[i]))
            resList.add(strs[i]);
        else{
            if (strs[i].equals("("))
                stack.push("(");
            else   if (strs[i].equals("+") || strs[i].equals("-")){
                while (!stack.isEmpty() && !stack.peek().equals("("))
                    resList.add(stack.pop());
                stack.push(strs[i]);
            }
            else if(strs[i].equals("/") || strs[i].equals("*")){
                while (!stack.isEmpty() && (stack.peek().equals("/") || stack.peek().equals("*")))
                    resList.add(stack.pop());
                stack.push(strs[i]);
            }else if (strs[i].equals(")")){
                while (!stack.isEmpty() && !stack.peek().equals("("))
                    resList.add(stack.pop());
                stack.pop();
            }
        }
    }
    while (!stack.isEmpty())
        resList.add(stack.pop());
    return resList;
}
```

```
public    static int calPostfix(List<String> list){
        Stack<Integer> stack = new Stack<>();
        String express = "+-/*";
        for (int i=0;i<list.size();i++){
            String e = list.get(i);
            if (!express.contains(e))
                stack.push(Integer.valueOf(e));
            else {
                int result;
                int b = stack.pop();
                int a = stack.pop();
                if (e.equals("+"))
                    result = a+b;
                else if (e.equals("-"))
                    result = a-b;
                else if (e.equals("*"))
                    result = a * b;
                else
                    result = a / b;
                stack.push(result);
            }
        }
        return stack.pop();
    }
```

链表

19. Remove Nth Node From End of List **删除倒数第 K 个节点。快先走 k,然后一起走，删除第 K 个**

21. Merge Two Sorted Lists **合并两个有序链表，可以递归，可以非递归。**l1.next = mergeTwoLists(l1.next,l2);

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1 == null) return l2;
        if(l2 == null) return l1;
        if(l1.val < l2.val){
            l1.next = mergeTwoLists(l1.next,l2);
            return l1;
        }else{
            l2.next = mergeTwoLists(l1,l2.next);
            return l2;
        }
    }
```

24. Swap Nodes in Pairs **成对翻转链表：**pre.next!=null &&pre.next.next!=null 时，将 cur.next 插入到 pre 之后。

61. Rotate List **链表右移 K 位**:遍历求 len,从 dummy 走 len-k 步，后面的直接放 dummy 后，尾指头，真尾空

82. Remove Duplicates from Sorted List II:**将重复的全部删掉。**

83. Remove Duplicates from Sorted List，**将重复多余的删掉，保留一个。**

```
while (cur!=null){
    if(cur.val == pre.val){
        pre.next = cur.next;
    }else{
        pre = pre.next;
```

```
        }
        cur = cur.next;  }
```

86. Partition List：将链表比 target 小的左移放前面，后面的都是大于等于的，两边相对位置不变。

建立两个头结点，遍历原链表，小的放在第一后面，大的放在第二个后面。然后将大的放入小后。

92.Reverse Linked List II,**将 m-n 之间的链表逆转。在 m-n 之间进行头插法。**

先走 m 步，然后头插法反转 m-n 次。

109. Convert Sorted List to Binary Search Tree：**链表中间节点作为根节点，前面的左子树，右边的右子树。**

一快一慢指针，走到中间。生成根节点，左边的断开，生成左子树，右边的生成右子树

```
public TreeNode helper109(ListNode head){
 if(head == null)
      return null;
 ListNode dummy = new ListNode(0);
 dummy.next = head;
 ListNode pre = dummy;
 ListNode slow = head;
 ListNode fast = head;
 while(fast.next!=null && fast.next.next!=null){
      fast =fast.next.next;
      slow = slow.next;
      pre = pre.next;
 }
 TreeNode root= new TreeNode(slow.val);
 pre.next = null;
 root.left =helper109(dummy.next);
 root.right = helper109(slow.next);
 return root;
}
```

138. Copy List with Random Pointer:链表有 next,有 random;复制他， 用 hashmap;

复杂链表复制，用 hashMap<Node,new Node(val)>;然后遍历原链表；get(node).next =

141. Linked List Cycle:链表是否有环，fast 走两步，slow 走一步。如果相遇或者到 null,出结果。

142. Linked List Cycle II，链表入口节点位置。Fast，slow 相遇，然后 fast 从 head 出发，再相遇。

143. Reorder List：第一个-倒 1-第 2-倒 2；这个比较麻烦！

快慢指针，找到中间的节点。然后将中间之后的节点采用头插法反转链表。然后前面的和后面的进行遍历，将后面的一个个的插入前面的中间。

```
while (cur2!=null){
    ListNode next1 = cur1.next;
    ListNode next2 = cur2.next;
    cur1.next = cur2;
    cur2.next = next1;
    cur1 = next1;
    cur2 = next2;
}
```

147. Insertion Sort List:插入排序重排链表

插入排序，while(cur.next!=null){将 cur.next 的值和 cur 比较，如果比他大不用排，继续。否则 p 从 dummy 遍历，找到 p.next>next.val 的，将 next 插入 p 后面。整个就是插入排序的思路。}

148. Sort List，NlogN 给链表排序：分治：将链表切成两部分，然后递归求这个。然后 merge;

归并算法：首先检查是否链表长度为 null 或者为 1；如果是直接返回，不是就切割成两半，用快慢指针快慢指针。不停的且一半有一半切割 l1 = sortList(head); l2 = sortList(solw);然后并，并的过程就是有序链表合并;

160. Intersection of Two Linked Lists,第一个长度，第二个长度，然后走齐，然后一起走到一样.

是否有共同点，看最后是否一致。

找共同点，l1 遍历，空继续 l2. l2 遍历，空找 l1.直到两人一样。

234：链表是否是回文：

快慢找到中间，然后逆转后面的，前后齐步走，看看是否始终值一样

328. Odd Even Linked List：偶数位的位于前面，奇数位的后面。一个 odd 开头，一个 even 开头，分别找 next.next；

```
ListNode odd = head,even = head.next, last = even;
while (even!=null && even.next!=null){
    odd.next = odd.next.next;
    even.next =even.next.next;
    odd = odd.next;
    even = even.next;
}
odd.next = last;
return head;
```

二叉树：

94. Binary Tree Inorder Traversal：**中序遍历，非递归。一路左，弹出，cur=pop.right;**

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> resList = new ArrayList<>();
    if(root == null) return resList;
    Stack<TreeNode> stack = new Stack<>();
    while(root != null || !stack.isEmpty()){
        while(root!=null){
            stack.push(root);
            root = root.left;
        }
        root = stack.pop();
        resList.add(root.val);
        root = root.right;
    }
    return resList;
}
```

96. Unique Binary Search Trees:**N 个节点，有几种 BST,动态规划规律可解**

98. Unique Binary Search Trees：**是不是 BST,中序遍历是递增的，保持前序的节点；然后比较大小**

```
中序遍历，维护一个 pre 表示上一个节点，比较当前和之前，如实当前不大于之前，不是 BST;
if(pre == null) pre = cur;
else if(pre.val >= cur.val) return false;
pre = cur;
```

100. Same Tree：**俩树是否一致。递归。都空返回 true,一个空或者值不一样 false;然后比较左子树和右**

101. Symmetric Tree：**是否对称，和上个一样，转化成递归(root.left,root.right);**

```
Symmetric(root1.left,root2.right) && Symmetric(root1.right,root2.left);
```

102. Binary Tree Level Order Traversal:**分层，分开打印。Size**

103. Binary Tree Zigzag Level Order Traversal 之字形，**size+flag**

```
if(!flag) Collections.reverse(tempList);
resList.add(new ArrayList(tempList));
flag = !flag;
```

104. Maximum Depth of Binary Tree:**递归，或者层序 size+level**

105. Construct Binary Tree from Preorder and Inorder Traversal:中序放入 map(in[i],i);

106. Construct Binary Tree from Inorder and Postorder Traversal:中序放入 map(in[i],i);

107. Binary Tree Level Order Traversal II:**从下层到顶层的层序；size + res.add(0,levelList);**

108. Convert Sorted Array to Binary Search Tree：**有序数组转化成二叉树，中间元素作根节点，其他的递归**

```
public TreeNode helper(int[]nums,int left,int right){
    if(left > right)    return null;
    int mid = left + (right -left)/2;
    TreeNode root = new TreeNode(nums[mid]);
    root.left = helper(nums,left,mid-1);
    root.right = helper(nums,mid+1,right);
    return root;
}
```

110. Balanced Binary Tree：求深度的时候，作 left 和 right 的差值比较；

111. Minimum Depth of Binary Tree：**最小深度，层序，size+遇到叶节点返回 level；**

112. Path Sum：根节点到叶节点是否之和是否存在等于 target 的，直接 hasPathSum，递归；

```
public boolean hasPathSum(TreeNode root, int sum) {
    if(root == null)    return false;
    if(root.val == sum && root.left == null && root.right == null) return true;
    return hasPathSum(root.left,sum-root.val) || hasPathSum(root.right,sum-root.val);
}
```

113. Path Sum II：给出所有的根到叶之和为 target 的组合。helper113(root,sum,resList,temList);

```
if(root == null) return;
tempList.add(root.val);
if(root.left == null && root.right == null && sum == root.val)
    resList.add(new ArrayList(tempList));
helper(resList,tempList,root.left,sum-root.val);
helper(resList,tempList,root.right,sum-root.val);
tempList.remove(tempList.size()-1);
```

114. Flatten Binary Tree to Linked List：所有节点飘向右方

遍历根节点，如果有左子树，那么将 root.right = root.left;root.left =null;就是将整个左子树移到右子树上。
但原本的右子树和根的左子树如何连一块？将右子树放到左子树最右边的那个节点的右边。

```
while(root != null){
    if(root.left != null){
        TreeNode prev = root.left;
        while(prev.right != null){
            prev = prev.right;
        }
        prev.right = root.right;
        root.right = root.left;
        root.left = null;
    }
    root = root.right;
}
```

116. Populating Next Right Pointers in Each Node：完全二叉树，整个右指针指向层序的右边那个。

117. Populating Next Right Pointers in Each Node II：普通二叉树，右指针指向右边那个。

129. Sum Root to Leaf Numbers：根到叶拼接成数字，所有的加一块之和。递归 sumRootToLeaf(root,0);

```
if(root == null )    return 0;
if(root.left == null && root.right == null)
    return s*10+root.val;
return sumRootToLeaf(root.left,s*10+root.val) + sumRootToLeaf(root.right,s*10+root.val);
```

144. Binary Tree Preorder Traversal：前序遍历

145，后序遍历；

```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> results = new ArrayList<Integer>();
    Deque<TreeNode> stack = new ArrayDeque<TreeNode>();
    while (!stack.isEmpty() || root != null) {
        if (root != null) {
            stack.push(root);
            results.add(root.val);
            root = root.right;
        } else {
            root = stack.pop().left;
        }
    }
    Collections.reverse(results);
    return results;
}
```

199. Binary Tree Right Side View：层序的最后一个的节点列表。Size+level==0;

```
Queue<TreeNode> que = new LinkedList();
que.add(root);
while(!que.isEmpty()){
    int size = que.size();
    while(size>0){
        TreeNode node = que.poll();
        if(size==1)
            result.add(node.val);
        if(node.left != null)
            que.add(node.left);
        if(node.right != null)
            que.add(node.right);
        size--;
    }
```

222. Count Complete Tree Nodes:完全二叉树的节点个数；

因为是完全二叉树，所以可以查看最左节点深度和最右节点深度，如果一样，说明是满了，2^n-1;

否则，1+递归左子树+递归右子树。时间复杂度？lgn ^ 2;

```
public int countNodes(TreeNode root) {
    int leftDepth = leftDepth(root);
    int rightDepth = rightDepth(root);
    if(leftDepth == rightDepth)
        return (1<<leftDepth) -1;
    return 1+countNodes(root.right)+countNodes(root.left);
}
public int leftDepth(TreeNode root){
    int depth =0;
    while (root!=null){
        depth++;
        root = root.left;
    }
    return depth;
}
public int rightDepth(TreeNode root){
```

```
        int depth =0;
        while (root!=null){
            depth++;
            root = root.right;
        }
        return depth;
    }
```

230. Kth Smallest Element in a BST：BST 第 K 小的节点，**中序遍历,**

235. Lowest Common Ancestor of a Binary Search Tree

二叉查找树的特点，根大于所有的左子树，根小于所有的右子树。所以只有根节点值和 p,q 比较就行。

如果 pq 都小于根，说明最近祖先在左子树。如果都大于根，说明最近祖先在右子树。否则根就是。

```
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root.val > p.val && root.val > q.val)
            return lowestCommonAncestor(root.left,p,q);
        if(root.val < p.val && root.val < q.val)
            return lowestCommonAncestor(root.right,p,q);
        return root;
    }
```

236. Lowest Common Ancestor of a Binary Tree：

从头到下递归，递归返回条件时：如果 root 遇到 p 或者 q，返回。或者为 null;如果 left 为空，说明都在右边，right 就是，如果 right 为空，那就在 left.如果都不空，说明俩边都有，返回 root;

```
if(root == null || root == p || root == q) return root;
TreeNode left = lowestCommonAncestor(root.left,p,q);
TreeNode right = lowestCommonAncestor(root.right,p,q);
return left == null ? right : right == null ? left : root;
```

543. Diameter of Binary Tree,两个节点的最长路径长度

```
    public int getDepth(TreeNode root){
        if(root == null) return 0;
        int left = getDepth(root.left);
        int right = getDepth(root.right);
        max = Math.max(max,left+right);
        return Math.max(left,right)+1;
    }
```

剑指 offer 里面的相关题目：

No3.数组中重复的数字 n 个数，范围 0 到 n-1,找到任意一个重复的即可

将值为 i 的数，放入 i 的位置。从位置遍历，将 A[i]和 A[A[i]]的互相交换，直到 A[i] = I;

重复的时候

空间复杂度为 N 的好办，boolean 数组即可。如果没有额外空间，就交换。

No4,二维数组查找，从左到右递增，从上到下递增。

右上开始，比 target 小，向下。比他大，就像左。

如果层铺开，有序。可以二分查找。算 mid 在二维数组中的位置

No5,空格替换，如果有空格，就将他替换成%20

(StringBuffer str)遍历得到长度 2*空格+原来的长度，str.setLength(newLength);

str.setCharAt(j--, '0');替换原来的。更新

//str.insert(0,str)插入。

No6 从尾到头打印链表

入栈，出栈；

No7 重建二叉树,前序遍历和中序遍历重构二叉树

首先将中序遍历的结果存到 map 中，map.put(in[i],i);key 为值，value 为位置。

前序第一个是根节点。然后找到中序数组中根节点的位置，然后算出左节点个数，右节点个数，递归。

!!!!! No8 二叉树的下一个节点，中序遍历的二叉树下一个节点，这个树的节点，有指向父指针;

如果二叉树该节点的右子树不为空，那么右子树的最左节点就是下一个节点。

如果为空，那么下一个节点就是右边的父节点。

```
TreeLinkNode parent = pNode.next;
 while (parent!=null && parent.left!=pNode){
      pNode = parent;
      parent = parent.next;
 }
 return parent;
```

No9,两个栈实现队列

一个入，一个出。出的不够入的入，然后出。

No10 斐波那契数列;

f1 =1,f2 = 1, fn = fn-1 + fn-2;

No 10-1,青蛙可以挑一阶，也可以 2 阶，跳到 n 阶，几种做法

No10-2,变态跳，可以跳 1-n 阶，随便跳。求 n 阶几种方法;

return 1 << (target - 1);

No11,旋转数组最小的数字：非减排序的数组，可能存在重复 // if nums[mid] > nums[right] left = mid + 1;
else if(nums[mid] < nums[right]) right = mid; else right--; return left;

No12,矩阵中的路径，矩阵有字符，可以从任意开始，上下左右任意走，但不能走走过的格子，求是否存在一条包含给定字符串的路径

矩阵，双层遍历：对每个位置进行 dfs 搜索。dfs 时，维护一个矩阵，表示是否已经走过。先筛选，如 ij 边界问题和当前遍历字母和 target 路径是否一致。成功的标志是，相同的已经到头，k= nums.length-1;
成功之后，这个遍历过。然后 dfs 向四周扩散。有一个返回 true,那就是 true.然后没遍历过,返回 fasle.

No13,机器人的运动范围，从(0,0)开始，上下左右移动，但不能进入行坐标和列坐标数位之和大于 k 的格子，给定 k,能打几个格子。

从 0，0 开始 dfs.dfs 时，维护一个矩阵，看看是否曾遍历过。判断是否越界，是否遍历过，是否行列之和小于等于 k.如果是，返回 0。然后遍历过，然后四处 dfs+1;

```
if (sRow < 0 || Scol < 0 || sRow >= rows || Scol >= cols || flag[sRow][Scol] || count(sRow,Scol) > k)
    return 0;
flag[sRow][Scol] =true;
return   helper13(sRow-1,Scol,rows,cols,flag,k)//甚至可以删了后退的两个。
          +helper13(sRow+1,Scol,rows,cols,flag,k)
          +helper13(sRow,Scol-1,rows,cols,flag,k)
          +helper13(sRow,Scol+1,rows,cols,flag,k)+1;
```

No14 剪绳子，长为 n，剪 m 段（m,n 都是整数，且 m>1），所有的长度的乘积最大为多少。

```
public int cutRope(int n) {
    if(n == 2 || n == 3)    return n-1;
    int[] dp = new int[n+1];
    dp[1] = 1;dp[2] = 2;dp[3] = 3;
    for (int i = 4; i<=n;i++){
        int max = 0;
        for (int j =1;j <= i /2 ;j++){
            max = Math.max(dp[j] * dp[i-j],max);
        }
```

```
            dp[i] = max;
        }
        return dp[n];
    }
```

No15,二进制中 1 的个数;

```
        while (n != 0) {
            n = n & (n - 1); //  让二进制的 n 最右边的 1 变成 0；
            count++;
        }
```

No16, 数值的整数次方

```
    public double Power(double base, int exponent) {
        //如果指数为负，为 0，为正。
        //如果 double 的基数为 0
        //正常情况下，不能对 base 乘以 n 次。而是折半。exponent 分奇偶，也分正负。
        if (exponent == 0 && base != 0)
            return 1;
        if (exponent == 1)
            return base;
        if(base ==0 && exponent > 0)
            return 0;
        if(base == 0 && exponent <= 0)
            throw    new RuntimeException();
        double result = 1;
        int n = exponent;
        if (exponent < 0 )
            n = -exponent;
        result    = Power(base,n >> 1);
        result = result * result;
        if ((n & 1) == 1)
            result = result * base;
        if (exponent < 0)
            result = 1 /result;
        return result;
    }
```

数组：

No18-1,删除节点。给定一链表头结点和指定节点，O(1)时间内，删除指定节点

No18-2.删除重复的节点。存在的重复节点，全都删了。Cur 走到重复节点的最后一个，看是否是 pre.next;

```
            while (cur.next != null && cur.val == cur.next.val){
                cur= cur.next;
            }
```

No19,实现函数来匹配包含‘.’和‘*’的正则表达式.

No20 实现一个函数来判断字符串是否表示数值。

```
    return string.matches("[\\+\\-]?\\d*(\\.\\d+)?([eE][\\+\\-]?\\d+)?");
```

No20-2 将字符串转换成整数；不合法返回 0；

从头开始遍历，先看是否为+-，标记正负号。然后一直遍历，如果出现为的字符在 0-9 之间。

```
    for(int i=index;i<str.length();i++){
        if(str.charAt(i) - '0' > 9 || str.charAt(i)-'0' < 0) return 0;
        sum = result * 10 + str.charAt(i)-'0';
        result = sum;
```

```
        }
        return symbol*result;
```

累积相加，并判断是否越界。

No21,调整数组顺序使得奇数位于偶数之前。冒泡排序；if(array[j]%2==0 && array[j+1]%2==1){交换}

No22,返回链表的倒数第 k 个节点。

快慢指针，快的走 k 个，然后慢快一起，到尾。

No23,链表中环的入口节点;

从头开始，快慢节点，如果相遇，那就有环。不相遇就没有环。相遇之后，快的从头出发，一起走，相遇的地方就是入口节点。

```
    ListNode fast = pHead,slow = pHead;
        while(fast.next!=null){ fast =fast.next.next;slow =slow.next; if(fast == slow) break;}
         if(fast == null)
            return null;
        fast = pHead;
        while(fast != slow){
            fast = fast.next;
            slow = slow.next;
        }
        return fast;
```

No24,反转链表,并返回反转之后的头结点。头插法；while(cur.next!=null){将 cur.next 插入 dummy 后面}；

No25,合并两个有序的链表，重建一个新的，包含所有的两个链表。--->单调不减 l1.next = merge(l1.next,l2)

No26,输入 A 和 B 两个树，判断 B 是 A 的子树。空树不是子树 先看是否马上包含

No27,二叉树的镜像。输入一个二叉树，将他变成他的镜像 递归，交换左右子树，然后递归左右子树。

No28,输入一个二叉树，判断二叉树是不是对称的。

主要看左子树和右子树是否对称

```
    helper(root1.left,root2.right) && helper(root1.right,root2.left);
```

No29,顺时针打印矩阵。

```
    while(rowStart<=rowEnd && colStart <= colEnd)
            helper(resList,matrix,rowStart++,rowEnd--,colStart++,colEnd--);
```

No30,包含 min 函数的栈 一个正常栈，一个 min 函数栈。

No31,栈的压入弹出序列,第一个表示压入序列，判断第二个是不 是弹出序列。

维护 j 表示弹出序列的位置，将压入序列压入栈中，如果栈顶和弹出的一样，弹出。最后如果栈为空，T.

```
        for(int j=0;j<pushA.length;j++){
            stack.push(pushA[j]);
            while(!stack.isEmpty() && stack.peek() == popA[i]){
                stack.pop();
                i++;
            }
        }
```

No32,从上到下打印二叉树 创建队列，先根入队，然后当队非空{弹出，操作，将左右子树入队}

之字形打印二叉树

```
          LinkedList<TreeNode> queue = new LinkedList<>();
            queue.add(pRoot);
            TreeNode cur = null;
            boolean flag = true;
            while (!queue.isEmpty()){
                int size = queue.size();
                ArrayList<Integer> list = new ArrayList<>();
                for (int i=0;i<size;i++){
                    cur = queue.poll();
```

```
                if(cur.left!=null)
                    queue.add(cur.left);
                if(cur.right!=null)
                    queue.add(cur.right);
                list.add(cur.val);
            }
            if(!flag)
                Collections.reverse(list);
            resList.add(list);
            flag = !flag;
        }
        return resList;
```

No33,输入一个整数数组，判断是不是二叉搜索树的后序遍历序列

No34,二叉树中和未某一值得路径

```
public void helper(ArrayList<ArrayList<Integer>> resList,ArrayList<Integer> tempList,TreeNode root,int target){
        if(root == null || root.val > target)    return;
        tempList.add(root.val);
        if(root.val == target && root.left == null && root.right==null)
            resList.add(new ArrayList(tempList));
        helper(resList,tempList,root.left,target-root.val);
        helper(resList,tempList,root.right,target-root.val);
        tempList.remove(tempList.size()-1);
}
```

No35 复杂链表的复制;链表不仅有 val,next,还有一个指向任意节点

Map<RandomListNode,RandomListNode> map = new HashMap<>();

创建一个 map,key，value 都是节点。将原本的链表节点压入为 key,value 为值一样的链表；

将

map.put(cur,new RandomListNode(cur.label));

cur = pHead;

```
        while (cur != null){
            map.get(cur).next = map.get(cur.next);
            map.get(cur).random = map.get(cur.random);
            cur = cur.next;
        }
        return map.get(pHead);
```

No36 二叉搜索树和双向链表;二叉搜索树转换成一个排序的双向链表，不能创建新的节点，只能调整指针顺序。中序遍历，保持头结点，以及 pre 节点，从栈中弹出的节点 cur 和 pre 建立联系。

```
            cur = stack.pop();
            if(realHead==null){
                realHead = cur;
                temphead = cur;
            }else{
                temphead.right = cur;
                cur.left = temphead;
                temphead = cur;
            }
            cur = cur.right;
```

No37 实现函数序列化二叉树和反序列二叉树

```
public class Solution {
    public int index = -1;
    String Serialize(TreeNode root) {
        StringBuffer sb = new StringBuffer();
        if(root == null){
            sb.append("#,");
            return sb.toString();
        }
        sb.append(root.val + ",");
        sb.append(Serialize(root.left));
        sb.append(Serialize(root.right));
        return sb.toString();
    }
    TreeNode Deserialize(String str) {
        index++;
        int len = str.length();
        if(index >= len){
            return null;
        }
        String[] strr = str.split(",");
        TreeNode node = null;
        if(!strr[index].equals("#")){
            node = new TreeNode(Integer.valueOf(strr[index]));
            node.left = Deserialize(str);
            node.right = Deserialize(str);
        }

        return node;
    }
}
```

No38 输入一个字符串，给出所有的排列;

首先这是个排列问题，每次遍历从字符串取值的时候，都要从 0 开始，且维护一个 boolean 数组表示第 i 位是否已经选中。

其次，我们要进行去重判断，也就是 `if (used[i] || (i >0 && str[i] == str[i-1] && !used[i-1])) continue;`

```
        if(sb.length() == str.length){
            resList.add(sb.toString());
            return;
        }
        for (int i=0;i<str.length;i++){
            if (used[i] || (i >0 && str[i] == str[i-1] && !used[i-1])) continue;
            used[i] = true;
            sb.append(str[i]);
            helper38(resList,sb,str,used);
            used[i] = false;
            sb.deleteCharAt(sb.length()-1);
        }
```

No39,数组中出现超过一半的数字,如果不存在，返回 0;

摩尔投票：int count =0,result = 0;遍历数组，如果等于 result，count++,否则如果 count==0,换人。否则 count--;最后再遍历看等于 reslut 的人有几个，是否过了一半，

No40，最小的 k 个数，然后先进去 K 个，如果比较和堆顶大小，符合条件出来一个，再进去

首先建立一个大小为 K 的优先队列最大堆，默认最小堆。

```
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(k, new Comparator<Integer>() {
        @Override
        public int compare(Integer o1,Integer o2){
            return o2.compareTo(o1);
        }
    });
    for(int i=0;i<input.length;i++){
        if(maxHeap.size()!=k){
            maxHeap.offer(input[i]);
        }else if(maxHeap.peek()>input[i]){
                maxHeap.poll();
                maxHeap.offer(input[i]);
            }
    }
```

No41，数据流中的中位数，奇数个，中间那个。偶尔个，中间俩的平均

No42,连续子数组的最大和，数组中有正数有负数，求所有连续子数组的最大和。

```
dp[i] = Math.max(dp[i-1]+array[i],array[i]);
```

No43,1-n 所有数中，求所有十进制位出现 1 个总数

No44，从 0-n,所有的从前到后排到一块，实现一个函数，求第 k 位的数字是几？

No45,给定一个数组，求组合到一块的最小数字

首先将整数数组变成字符串数组，然后排序，排序方法就是 o1+o2.compareTo(o2+o1)

```
String[] strs = new String[numbers.length];
for(int i=0;i<numbers.length;i++)
    strs[i] = String.valueOf(numbers[i]);
Arrays.sort(strs,new Comparator<String>(){
    @Override
    public int compare(String o1,String o2){
        return (o1+o2).compareTo(o2+o1); //默认都是升序
    }
});
  String res = String.join("",strs);
```

No46,0-25 翻译成 a-z,给一个数字，求有几种翻译方法

No47,礼物的最大价值，矩阵，每一步都有一个值，从左上到右下，只能向右或向下移动，求最大和

No48，最长不含重复字符的子字符串，

No49,只含 2,3,5 因子的数是丑数，1 也是，求第 n 个丑数

这个是非常规动态规划：维持 235 的因子个数 i,j,k

丑数的排序：index[1] = 1,第一个丑数，

```
int i=j=k=1;
for(int m=2;m<=n;m++){
    dp[i] = min(dp[i]*2,dp[j]*3,dp[k]*5);
    if(dp[m]= dp[i]*2) i++;
if(dp[m] == dp[j]*3) j++;
if(dp[m] == dp[k]*5) k++;
}
```

No50，第一个只出现一次的字符,返回的是位置

```
int[] nums = new int[128];
for(int i=0;i<str.length();i++){
```

```
            nums[str.charAt(i)]++;
        }
        然后遍历，找到第一个值为 1 的。
```

No51，数组中的逆序对,归并排序！！！！

归并的时候，从大到小比较，如果左边的比右面的大，那么 count += right – mid +1;

No52,两个链表的第一个公共节点。

要么算两个长度，然后长的走两步，然后同时走，走到遇见相等的。

要么 l1 走完走 l2,l2 走完走 l1。如果存在肯定有相同的。

```
while(l1 != l2){
    l1 = (l1 == null) ? headB : l1.next;
    l2 = (l2 == null) ? headA : l2.next;
}
return l1;
```

如果是比较两个链表是否有交点，直接看俩链表的最后一个节点是否一样。

No53,在排序数组中查找数组，一个数字出现了多少次

```
public int getLeft(int[] nums, int k){
    int left = 0;int right = nums.length-1;int res = -1;
    while(left <= right){
    int mid = left+(right-left)/2;
     if(nums[mid] == k ) res = mid;
     if(nums[mid] >= target) right =mid-1;
     else left =mid+1;}
}
```

No54,二叉搜索树中，第 K 小节点

No55-1，二叉树深度　递归，空为 0，求左子树，求右子树。返回左右最大的+1；

No55-2,是不是平衡二叉树　递归求深度，然后加一步，左的深度-右的深度绝对值是否大于 1；

No56-1,一个数组，一个出现一次，其他出现两次，求一次的。

```
        int diff =0;
        for(int n : array){
            diff = diff^n;
        }
        diff =diff & -diff;
        for(int n:array){
            if((diff & n) == 0){
                num1[0] = num1[0]^n;
            }else{
                num2[0] = num2[0]^n;
            }
        }
```

No56-2,数组，一个出现一次，其他出现三次。求一次

No57-1，递增排序的数组，和 target.找出一对和为 target 的数字

双指针，一个头一个尾，看结果移动。

No57-2,正数 s,给出连续正数序列，其和未 s.所有的序列。序列最少有俩。

```
        双指针，如果区间比 target 大，small++,f 否则 big++;
        int small =1,big =2;
        int mid = (1 + sum) / 2;
        while ( small <= mid){
            int cur = (small+big) * (big - small +1)/2;
            if (cur > sum)
```

```
                small++;
            else if (cur < sum)
                big++;
            else {将这个结果加入最终的结果集中}
```

No58-1,翻转单词顺序

用 API 快点，

```
public String ReverseSentence(String str) {
    if(str==null||str.trim().equals(""))// trim 掉多余空格
        return str;
    String[] words = str.split(" ");// 以空格切分出各个单词
    还他妈不如对每个字符串进行翻转呢。先以字符串建 StringBuffer,然后 reverse()方法；
    strs.join(“ ”, words);
}
```

No58-2,左旋转字符串，三次翻转。

```
        swap(chas,0,len-1);
        n = n % len;
        swap(chas,0,len-n-1);
        swap(chas,len-n,len-1);
```

No59-1 给出数组和窗口大小，求滑动窗口的最大值。

No59-2,实现带有 max 函数的队列，出入队和 max 都是 O(1)

No60,n 个色子，点数和为 s，s 所有可能出现的值得概率

No61,扑克牌抽 5 个数，是不是顺子。大小王可以为任何;

int[] map = new int[14];遍历这五个数，如果是大小王，不理他们。如果出现重复了，直接返回 false.

同时遍历过程中，维护一个最大值一个最小值。最后看看 max-min>4?

No62,0 到 n-1,围一圈，从 0 开始，删除第 m 个数字。求最后的数字;

维持几个数，boolean[] falg =new boolan[n];表示是否存活。

count = n,表示存活个数。 step = 0 表示这一轮的第几个人, inde=-1 表示遍历数组的位置。

```
while(count > 0){
    index ++;
    if(index == n) index =0;
    if(flag[index]) continue; //遍历直到找到还健在的人
    step++;if(step == m){ count--;flag[i] = true; step = 0;}}
```

No63,数组，值为股票当时的价格，求最大利润

No64,求 1+2+3+...+n,不能乘除，for,while,if,else 等。

1.需利用逻辑与的短路特性实现递归终止。 2.当 n==0 时，(n>0)&&((sum+=Sum_Solution(n-1))>0)只执行前面的判断，为 false，然后直接返回 0；3.当 n>0 时，执行 sum+=Sum_Solution(n-1)，实现递归计算 Sum_Solution(n)。

```
    public int Sum_Solution(int n) {
        int sum = n;
        boolean ans = (n>0)&&((sum+=Sum_Solution(n-1))>0);
        return sum;
    }
```

No65,不用加减乘除做加法

第一步：相加各位的值，不算进位，得到 010，二进制每位相加就相当于各位做异或操作，101^111。

第二步：计算进位值，得到 1010，相当于各位做与操作得到 101，再向左移一位得到 1010，(101&111)<<1。

```
        public int Add(int num1,int num2) {
            while(num2 != 0){
                int temp = num1;
                num1 = num1^num2;
                num2 = (temp & num2)<<1;
```

```
        }
        return num1;
    }
```

No66，构建乘积数组

先创建数组，从头遍历 A,每个 B[i]保存前 i-1 个数的乘积。

然后从尾到头遍历数组 A,用 temp 保存 i 后面的乘积。然后 B[i] = temp * B[i],更新 temp;

```
        B[0] = 1;
        for(int i=1;i<A.length;i++){
            B[i] = B[i-1] *A[i-1];
        }
        int temp=1;
        for(int i=A.length-1;i>=0;i--){
            B[i] = B[i] * temp;
            temp = A[i] * temp;
        }
        return B;
```

字符流中第一个不重复的数：用 StringBuffer 老保存字符，用 128 个 map 保存每个字符的个数。让之前保存的数组，从头开始遍历，找到第一个个数统计为 1 的值。

```
    StringBuffer sb = new StringBuffer();
    int[] map = new int[256];
    public void Insert(char ch)
    {
        sb.append(ch);
        map[ch]++;
    }
    public char FirstAppearingOnce()
    {
        char[] chars = sb.toString().toCharArray();
        for(char c : chars){
            if(map[c] == 1)
                return c;
        }
        return '#';
    }
```

总结猿辅导手撕算法题

1.栈排序

申请一个辅助栈，将数据栈弹出，如果辅助栈为空或者 cur>=辅助栈元素，直接压入辅助栈。

否则的话，辅助栈弹出，并压入数据栈。直到 cur >=辅助栈顶。

2.链表实现队列

入队进入列表，出队，去除头结点，返回头结点值。

3.最长连续递增序列

If(nums[j] > nums[j-1]) dp[j] = dp[j-1]+1;

4.最长不连续序列

遍历序列 i,同时维持 j,和 set,如果能放入就好，不能的话，就慢慢将 j 一个个移除。

5.二维数组回行打印

还是那一套。

6.无序数组构建一棵二叉排序树,先排序，再构建。递归构建；

7.一个数组实现两个栈

维护 size=0，size2 = length-1;栈 1 进栈，size++;出栈--，设为 null;

栈 2 进栈，size--,出栈，++；

8.二叉树宽度 层序遍历，求每层的 size 最大

9.二叉树是否对称， helper(root.left,root.right);

10.链表 m 到 n 反转 ，双指针，快的先走 m 步，头插法将 m 后面的插入再走 n-m 次。

11.一个 n 位数，现在可以删除其中任意 k 位，使得剩下的数最小

12.实现有符号大数链表加法，靠近头结点位置为高位

13.字符串横向改纵向

14.八皇后的问题

15.找出来数组中每个元素后边第一个比它大的值

16.给你一个二叉树，从上往下看，然后左往右顺序输出你能看到节点，同一个竖直方向上上面的节点把下面的节点遮挡住了

17.链表反转，分别用遍历与递归实现

```
public Node reverNode(Node head){
    if (head == null || head.next == null)
        return head;
    Node newHead = reverNode(head.next);
    head.next.next = head;
    head.next = null;
    return newHead;
}
```

18.完全二叉树的最大深度与节点个数 //求根节点左子树高度，和右子树高度，如果一致，左子树满。左子树加上根节点 2^left,然后递归求右子树的个数。如果不一致，右子树满，然后公式求右子树，然后递归左子树。

```
public int getWanQuan(TreeNode root){
    if (root == null)
        return 0;
    TreeNode left = root.left;
    int lcount = 0;
    while (left != null){
        lcount++;
        left = left.left;
    }
    int rcount = 0;
    TreeNode right = root.right;
    while (right != null){
        rcount++;
        right = right.right;???? //  应该为 left,这个算法还是不如上一个清晰。
    }
    int res;
    //左子树和右子树高度相同，左子树是满的，右子树再递归求点数
    if (lcount == rcount){
        res = (int)Math.pow(2,lcount) + getWanQuan(root.right);
    }else {
        res = (int)Math.pow(2,rcount) + getWanQuan(root.left);
    }
    return res;
}
```

19.两个栈实现队列 入队进栈，出队的时候，如果出栈不空就出，否则入栈的所有都倒入出栈，然后出。

20.两个有序数组交集、并集

新数组指针 k=0;

两个数组分别进行遍历 i=0,j=0.如果相等，num[k++] = a[i++],j++;如果不等，a 小，i++,否则，j++;

并集：如果不等，小的入 K++,如果相等，入 k++,两人都走。

21.给定一个有序存在重复的值链表，使得每个元素只出现一次

删除链表，pre 和 cur,如果 cur 和 pre 一样了，删除 cur,如果不一样，同步走。

22.leetcode 200

```
private int m,n;
private int[][] directions = {{0,1},{0,-1},{1,0},{-1,0}};
public int numIslands(char[][] grid) {
    if(grid == null ||grid.length == 0)
        return 0;
    m = grid.length;
    n = grid[0].length;
    int result =0;
    for(int i =0;i<m;i++){
        for(int j =0;j<n;j++){
            if(grid[i][j] == '1'){
                dfs(grid,i,j);
                result++;
            }

        }
    }
    return result;
}
public void dfs(char[][] grid,int i,int j){
    if(i <0 || j<0 || i >= m || j >=n || grid[i][j] == '0')
        return;
    grid[i][j] = '0';
    for(int k=0;k<directions.length;k++){
        dfs(grid,i+directions[k][0],j+directions[k][1]);
    }   }
```

23.二叉搜索树转有序双向链表(中序遍历)

24.字符串全排列 ，可能有重复的，要去重

```
 public void helper38(ArrayList<String> resList, StringBuffer sb,char[]    str,boolean[] used){
    if(sb.length() == str.length){
        resList.add(sb.toString());
        return;
    }
    for (int i=0;i<str.length;i++){
        if (used[i] || (i>0 && str[i] == str[i-1] && !used[i-1])) continue;
        used[i] = true;
        sb.append(str[i]);
        helper38(resList,sb,str,used);
        used[i] = false;
        sb.deleteCharAt(sb.length()-1);
    }
}
```

25.二叉搜索树第 k 个节点，不用中序遍历，还是中序遍历吧，非递归。

26.有序数组查找重复元素个数

二分查找：先求左边的， 再求右边的。

```
While(left <right){
Mid = left+(right-left)/2;
If(nums[mid] == target) res = mid;
If(nums[mid] >= target) right = mid-1;
Else left = mid+1;
}
Return res;
```

27.定长数组实现队列

28.用二分法对一个数字开根号

```
public static double sqrt(double x){
    double left = 0;
    double right = x;
    double mid = left + (right-left)/2;
    while (Math.abs(right-left) >= 1e-10){
        mid = left + (right-left)/2;
        if (Math.abs(mid * mid - x) < 1e-10)
            return mid;
        else if (mid * mid > x)
            right = mid;
        else
            left = mid;
    }
    return mid;
}
```

29.判断一颗树是不是二叉搜索树（中序遍历是否有序；记录 pre，和当前比较，是否 pre 一直小于当前）

30.Excel 表的列字母转换，输入第几列，输出列字母组合

31.链表第 k-1 个节点

32.手撕快排

33.二分查找

34.一个无序有正有负数组，求乘积最大的三个数的乘积

35.求二叉树的深度，不使用递归 ：层序遍历和递归

36.实现链表，无序链表，对链表值奇偶分离并排序，空间复杂度 O(1)

37.单调不递减数组，给一个 target，找出大于等于 target 的下标 index

38.单调不递减链表，删除掉重复值

39.无序数组构建一棵二叉排序树(先排序，然后取中间的节点生成根节点，左边的生成左子树，右边的右子树)

40.行和列都是有序的二维矩阵找一个 target 值

得到 mid 之后，要算出他在哪行哪列。

41.是否是回文链表(快慢指针，找到中间节点，然后后面的入栈，弹出、遍历对比)；

42.打印出根节点到叶子节点的最长路径

43.双链表按照奇偶顺序分成两个链表，要求不要复制链表

44.不严格递增数组，要求删除出现次数大于 k 的数字，要求不要新建存储空间

还是维持一个符合条件的指针的 j;

45.链表相邻元素交换

两个一换，while（cur!=null && cur.next != null）,调整 pre,cur,cur.next,cur.next.next 之间的 cur 和 next 的顺序。

46.二叉树的最小公共祖先

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root == null || root == p || root == q)
            return root;
        TreeNode left = lowestCommonAncestor(root.left,p,q);
        TreeNode right = lowestCommonAncestor(root.right,p,q);
        return (left!=null && right!=null)?root:(left == null ? right : left);
    }
```

47.字符串形式自定义进制大数相加

48.链表每隔 k 个反转

要么用 pre,start,end,next。Cur 每走一步，计算，到 count 时，pre,pre.next,cur,cur.next 做链表翻转。然后 count 清零。继续遍历 cur,知道 cur 为空。

要么，先走一遍，得到长度。然后 lenht> k，也用 pre,pre.next,cur,cur.next 做链表翻转。然后 len = len-k;

49.输出根节点到叶子节点路径之和为 target 的路径列表（helper(resList,tempList,root,sum);

50.一些数，任意排列求可形成最小的值

```
public String PrintMinNumber(int [] numbers) {
        String[] strs = new String[numbers.length];
        for(int i=0;i<numbers.length;i++)
            strs[i] = String.valueOf(numbers[i]);
        Arrays.sort(strs,new Comparator<String>(){
            @Override
            public int compare(String o1,String o2){
                return (o1+o2).compareTo(o2+o1);
            }
        });
        String res = String.join("",strs);
        return res;
    }
```

51.LeetCode　1038.

52.数组题，任意一个整型数组，判断是否可以将数组分为三个区间，每个区间中数值的和相同

53.已排序的整数数组去重

一个遍历指针，一个去重之后的位置指针。如果重就++，不重就将当前遍历放入去重指针 j++;

54.错位的全排列（第一位不能是 1，第二位不能是 2）

55.k 路链表归并

```
public ListNode mergeKLists(List<ListNode> lists) {
        if (lists==null||lists.size()==0) return null;

        PriorityQueue<ListNode> queue= new PriorityQueue<ListNode>(lists.size(),new Comparator<ListNode>(){
            @Override
            public int compare(ListNode o1,ListNode o2){
                if (o1.val<o2.val)
                    return -1;
                else if (o1.val==o2.val)
                    return 0;
                else
                    return 1;
            }
        });
```

```
        ListNode dummy = new ListNode(0);
        ListNode tail=dummy;

        for (ListNode node:lists)
            if (node!=null)
                queue.add(node);

        while (!queue.isEmpty()){
            tail.next=queue.poll();
            tail=tail.next;

            if (tail.next!=null)
                queue.add(tail.next);
        }
        return dummy.next;
    }
```

56.非降序数组，找与 target 最相近的数的下标

57.二叉树逆时针打印最外层节点

先左边排到第二，然后先序或中序得到叶子节点，然后右边的跟二到倒二。

58.输入一个数字 n，构建一个完全二叉树并输出、

层序遍历构建二叉树。

59.输入一个矩阵，起始点和目标点，判断是否存在可达路径

60.无向图最短路径

61.第 K 层叶子节点个数 （层序遍历，level = k，弹出 k 层的节点，统计叶节点个数）

迪杰斯特拉(Dijkstra)， **Dijkstra 算法可以计算任意节点到其他节点的最短路径**

贪心策略是每次选可达的点中距离源点最近的点进行扩展，即贪心选取最短距离的点

指定一个节点，例如我们要计算 'A' 到其他节点的最短路径

引入两个集合（S、U），**S 集合包含已求出的最短路径的点**（以及相应的最短长度），**U 集合包含未求出最短路径的点**（以及 A 到该点的路径，注意 如上图所示，**A->C 由于没有直接相连 初始时为∞**）

初始化两个集合，**S 集合初始时 只有当前要计算的节点，A->A = 0，**

U 集合初始时为 A->B = 4, A->C = ∞, A->D = 2, A->E = ∞，敲黑板！！！ 接下来要进行核心两步骤了

从 U 集合中找出路径最短的点，加入 S 集合，例如 A->D = 2

更新 U 集合路径，if ('D 到 B,C,E 的距离' + 'AD 距离' < 'A 到 B,C,E 的距离') 则更新 U

循环执行 4、5 两步骤，直至遍历结束，得到 A 到其他节点的最短路径

库鲁斯卡尔(Kruskal) :每次选择最小权的边，加入树中，但不能生成环。

贪心策略是每次选最短的边（刨除成环的边）来作为最小生成树，**即贪心最短边**

普里姆算法(Prim)，从选中的一个点开始，找到和这些点相连的其他集合的边中选取最短的那条，然后将那个点加入已选集合。**从已选点集合中，选取和已选集合边最小的点，加入集合。**

贪心策略是每次选可达的点中距离曾经扩展过的点中任意点的最短距离，类似 Dij，只是不是找距离源点的最短距离

KMP

贪心策略 0.0 不是贪心

是动态规划，动态规划的是当前状态失败之后上一次匹配的位置（求的是最长的与前缀子串匹配的左子串）

1. Two Sum:给定数组和 target,返回两个元素之和为 target 的元素 index;

遍历数组时，将 target-当前元素，index 放入 map。如果存在结果，map.get(nums[i])!=null，说明已经合格了。

2. Add Two Numbers：他是一个反转的链表，求真正链表代表数字的和的链表的翻转；

其实就是俩链表一一相加，进位就进位。

其中技巧是 while(l1 != null || l2 != null){cur1 = l1==null ? 0: l1.val; l1 = l1 == null ? null :l1.next};

字符串 **3. Longest Substring Without Repeating Characters：字符串最长的不重复子序列长度：**

遍历字符串，同时又整个 j 指针，维持个 set，将当前字符，放入 set,如果放不进去，就是有重复的，j 位置的支付删除，直到放进去。这时候，以 i 为结尾的不重复序列长度就是 i-j+1;然后对出个最大的。

```
While(set.contains(c)){
        Set.remove(str.charAt(j++));
} set.add(c);    max = Math.max(max,j-i+1);
```

字符串 5. Longest Palindromic Substring：字符串最长的回文序列。

从头到尾遍历字符串：对遍历的进行从里到外扩展，扩展有两个形式，一是 expend(i,i),二是 expend(i,i+1)，**扩展过程中，直到碰到不回文的。然后计算出回文的最大长度和 left;**

```
void  Expend(String str, int left, int right){
    while(left >=0 &&right <str.length()    && str.charAt(left) == str.charAt(right) )
    left--;right++;}
    if(max < right-left-1){
    max = right-left-1; leftMin = left+1;
}
```

字符串 6. ZigZag Conversion：一个字符串，一个行数。然后字符竖折竖折来排序。然后一层一层的组合字符。

M 行：**整个 StringBuffer[m]数组**;遍历字符串，先向下走 m 个，每个分到一个 sb 中，然后上反走 m-2 个，每个分到一个 sb 中。完事之后，m 个 sb 都加到 sb[0]之后，然后 toString();

7. Reverse Integer：将整数值反转，如 120---21，-21---- -12，如果范围大于整数范围，就返回 0；

为了防止溢出，long = result; 对输入的整数值 x,循环到 x 不等于 0：**result = result *10 + x %10;X = x/10;同时判断 result 是否溢出，大于 Integer.MAX_VALUE 小于 MIN_VALUE。即对 x 不断取余，获取个位数。然后作为 result 的高位。在这个过程中正负无所谓**。-123%10=-3;-123/10=-12;

8. String to Integer (atoi)：前面的空字符不理，非空之后，如果遇到非 s 数字，直接不再搭理后面的，结果定型。如果遇到+-计正负，然后遇到整数就记结果,直到结束或者遇到非数字。

首先，去除前面的空。然后，对第一个字符进行是否+-判断，标记住。然后，继续遍历，先看是否非数字，还是如何判断是否越界的问题，如果是数字，就一步步高位取结果。如果遇到非数字，直接跳出，返回结果。

判断是否越界，可以用 long result = 0;每位遍历之后，**看 result 是否大于 int 最大值和 int 最小值。**

如果是，则越界。返回特殊处理，否则继续进行。

9. Palindrome Number，一个数，正反来是否一样。负数不一样

当 x<0，不回文。然后反转这个树，取余作高位。看看这俩结果是否一样。

12. Integer to Roman

```
public String intToRoman(int num) {
        String M[] = {"", "M", "MM", "MMM"};
        String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
        String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
        String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
        return M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];
    }
```

14. Longest Common Prefix；N 个字符串，求这 N 个字符串最长公共前缀

把第一个字符串当做前缀。遍历这剩下来的字符串：如果 while(strs[i].indexOf(pre)!=0),说明 pre 减去一位，substing(0,length()-1);或者，一个个 I,每个字符串遍历看是否和第一个一致。如果一直，i++。不相等就 break;

```
String pre = strs[0];
```

```
        for(int i=1;i<strs.length;i++){
            while(strs[i].indexOf(pre) != 0)
                pre = pre.substring(0,pre.length()-1);
        }
        return pre;
```

15. 3Sum,数组中，找 3 个和为 0 为三个数，每个数不能用两次。要考虑重复的问题。

先排序。然后从 0 到 n-3 遍历。Sum=0-num[i];然后进行二分查找，一头 i+1,一尾。如果等于 sum，结果保存，同时如果左边的有重复的，一直++，如果右边的重复的，一直--。然后再 left++,right--;如果不等于 然后调整 left 和 right;

```
For(int i=0;i<len-2;i++){
If(i>0 && nums[i-1] == nums[i] ) continue;
Int left = i+1,right = len-1,sum = -nums[i];
While(left <right){
Int temp = nums[left] + nums[right];
If(temp == sum){
resList.add(Arrays.asList(nums[i],nums[left],nums[right]));
while(left<right && nums[left+1] == nums[left]) left++;
while(left<right && nums[right-1]==nums[right-1]) right--;
left++;right--;
}else if( temp < sum) lef++;else right--;
}
}
```

16. 3Sum Closest:一个数组，找出 3 个和最进阶 target 的。

首先，排序。然后先随便三个数相加得到 res;开始遍历，从 0 到 n-2,然后对每个 i，left=i+1,right 为尾部。While(left<right){sum=这三个相加,根据 sum 和 target 的大小调整两个指针，res 和 target 的差与 sum 与 target 的差距保存最小的.}

17. Letter Combinations of a Phone Number

```
Public void combation(List<String> resList, String prefix,int offset, String digits){
If(offset == digits.length()){
    resList.add(prefix);return}
String letters = keys[digits.charAt(offset)-‘0’];
For(char c:letters.toCharArray()){//相当于遍历一下
    Combation(resList,prefix+c,offset+1,digits);}
}
```

18. 4Sum 数组中 4 个数之和为 0，考虑重复的数，但一个数不能用两次，组合也不能太一致。

先排序。然后遍历第一层，如果重复 continue，然后遍历第二层：继续考虑重复。Sum = target-那俩。然后双指针，如果等于 sum,加入，然后去重。左右指针移动，不等的话就根据大小调整。

19,删除倒数第 k 个节点

Dummy,然后 fast 走 k 步，且 fast 不能为 null,如果走了 k 步，就说明长度大于 k,还可以删。然后 low 和 fast 一块走，fast 到末尾。

20、括号序列是否匹配：

弄个栈，如果遇见左括号类型的，就将该类型的右括号入栈。如果是右括号，弹出看是否一致。若一致且最后为为空，返回 true：

21、合并两个有序序列；

非递归，两个链表，一次比较入队。然后剩余的入队。

递归，链表为空，返回另外一个。然后比较俩链表大小，小的 l1.next = mer(l1.next,l2);

22、n 对小括号的全正确排序：

正常的排列组合问题。这是两个变量，一个 left,一个 right。辅助函数是 resList,string 代替 tempList,左括号数量和右括号数量；为了使得左括号始终再右括号前面，辅助函数要走两边，第一遍加（，第二遍加）；

```
public void helper(List<String> resList,int left,int right, String s){
```

```
        if(left==0&&right==0){
            resList.add(s);
            return;
        }
        if(left>0)
            helper(resList,left-1,right+1,s+"(");
        if(right>0)
            helper(resList,left,right-1,s+")");
    }
```

24：成对的翻转链表

while(pre.next!=null && pre.next.next!=null){

将 cur 后的 next 利用头插法出入到 pre 的后面。然后 pre 和 cur 再后移；

}

Dummy;当 dummy 的 next 和 next.next 都不为空时；cur 作为 pre，next 作为第一个节点，next.next 作为第二个节点，然后将第二个节点插入到 pre 的后面。然后 cur 移动两个节点。

26、有序数组，删除多余的重复数组，in-place 删除，然后返回长度。

遍历数组，再加一个不重复的指 j 针。当 num[j] =当前的时候，继续遍历，不等于时，将当前的移动到 nums[j]并加加；

27、数组，删除指定的数字；

同上。遍历数组，并维持一个处理后指针；

28、**实现 indexOf(String s1,String s2);**

往死里遍历：外层遍历：从 0-（s1 的长-s2 的长）：里层遍历：从 0 到 s2 的长度，如果出现了不一致，break;如果 j 走到了最后完全一致，就可以直接返回了。

33、左移一部分的有序数组中，查询 target

二分查找的变形；

34、有序数组中，查询一个 target 第一次和最后一次出现的位置；

二分查找的变形。

35、将一个数插入到有序数组中，求位置

二分查找

39：非重复数组，找出和为 target 的数组，一个不能用两次；

排列组合类似的回溯问题；

40，带有重复的数组，找出和未 target 的数组组合，一个不能用两次；

回溯+组合。

43，两字符串的数字，相乘，得到结果，输出字符串。

一个字符串的 indexI 为 0-n-1,一个字符串的 indexJ 为 0-n-1;这两个相乘的位置是放在 index[i+j,i+j+1];

46,不重复数组的，所有全排列；

回溯+排列；

47，带有重复的数组，全排列；

回溯+排列；

48：n x n 矩阵，顺时针转动 90 度。

和那个顺时针打印矩阵一样，这个也是一圈圈的转动，先外圈，再内圈。Temp+置换的形式。

49，一群字符串，有些字符串字母组成一样，有的不一样，将一样的放一块；

遍历数组：将字符串转化成数组，然后排序。将排序的作为 key,真正的值作为放入 list 作为 value; 这样 hashmap 的 values 就是结果；

```
public List<List<String>> groupAnagrams(String[] strs) {
        List<List<String>> resList = new ArrayList<>();
        Map<String,List<String>> map = new HashMap<>();
```

```
        for(String str: strs){
            char[] chas =str.toCharArray();
            Arrays.sort(chas);
            String key = String.valueOf(chas);
            if(!map.containsKey(key))
                map.put(key,new ArrayList<>());
            map.get(key).add(str);
        }
        return new ArrayList<>(map.values());
    }
```

50：计算 power(x,n); x 在正负 100，n 在正负 2^31;

```
public double myPow(double x, int n) {
        if(n == 0)
            return 1;
        if(n == Integer.MIN_VALUE)
            return myPow(x*x, n/2);
        if(n < 0){
            x = 1/x;
            n = -n;
        }
        return n % 2 == 0 ? myPow(x*x,n/2) : x * myPow(x*x,n/2);
    }
}
```

54:一圈圈顺时针打印数组。

先整外圈，再里圈。

58：最后一个单词的长度

从后遍历，计数，知道遇见空格。

59：给一个 N,将 1-N^2,整成上面外圈顺时针到里圈的矩阵。

差不多。先建立 NN 数组，然后从外圈到里圈遍历：每圈的时候，顺时针赋值。

60：1-N 这 n 个数字的全排列，求第 K 小的；

主流思想我一下子理解不了啊

61、翻转链表，链表右移 K 为，可以循环多移；

第一、首先遍历链表 fast,得到链表的长度。然后整 slow 指针，移动 len-K 个，然后将后面的 k 个移动到 dummy 的后面，fast 作为最后的指针指向本来的头结点，然后 slow 指向 null;

62、矩阵移动，mxn 矩阵从左上角移动到右下角，只能横移和竖移，共有多少方法；

第一行只有一个方法， 第一列只有一种。然后 dp[i][j] = dp[i-1][j] + dp[i][j-1];

63、矩阵移动，从左上移动到右下：横竖移动，矩阵值为 1 的不能走；求多少种移动方式。

同上，只不过在判断的时候，先判断值是否为 1，为 1 表示 0；然后其他的一样.

64、矩阵，从左上到右下，最短的路径和多少：

双层遍历，都为 0 为等于左上角。一行和一列，等于前面的那个加自身。其他的，等于上面和左边最小的那个加自身。

66、数组表示的数+1，然后返回之后的

数组从后遍历：如果数字不为 9，直接加，然后返回原数组。如果为 9，设为 0，然后下一个加一。如果最后一只都是 999，然后新建数组，长度加 1，初始为 1；

67、两个二进制字符串相加，得到的二进制字符串

两个字符串，只要有一个不空，从尾到头相加，加的结果封二进 1，整一个进位标记 carry，每位的加人 StringBuffer,最后有进位，咱也加入。最后再反转。

69. Sqrt(x) 实现 int sqrt(int x)；

```
            long r = x;
        while (r*r > x)
```

```
        r = (r + x/r) / 2;
    return (int) r;
```

70、n 层楼梯，爬上去的方式有几种？只能一层或两层。

Fn = Fn-1 + Fn-2;

73、mxn 的数组，如果有一个为 0，那么就让他的所有列和行都为 0；in-place,求最后的数组；

双层遍历遍历数组，如果值为 0，横向第一列为 0，竖向的第一行为 0；然后有第一行或者第一列为 0，比较下 fc,fr 为 0。然后再次遍历第一行和第一列的，如果遇见 0，再竖向和横向的全设为 0。如果存在 fc 和 fc 为 0，一行和一列全设置为 0；

74、mxn 的矩阵，层序遍历就是有序的数组。在矩阵中搜索 target;

二分查找；

75：一个 2/1/0 的数组，将 0 分到左边，2 移动右边。

遍历整个数组，维持两个指针：left 和 right。当遍历到 0 是，将当前的 cur 和 left 互换。Left++;

如果遍历到 2，就将当前的 2 与 right 互换，right--;

77:1-N，挑选 K 个进行排列；

回溯+排列；

78：求数组的所有子集合。

回溯+组合；

80：删除有序数组中过于多余的数字，最多每个出现两次。

遍历数组，维持一个 left 指针，当前元素与 left-2 相等时，开始复制。否则继续走。

81：翻转部分的有序数组，可能存在重复，搜索 target;

二分查找的变形；

82：删除有序链表，将重复的全都删了；

Dummy，pre ,cur,将 cur 一直走一直走走到同一节点的最后面，如果 pre.next = cur;

就说明没有重复的，直接走，如果不是，是重复的，然后全都删了。

83：删除有序链表多余的节点，重复的保留一个。

如果当前和 pre 相等的话，直接将 cur 删除。否则继续双走。

86：链表，给定 target，将比 target 小的节点移动到左边。大的和等于的移动到后面。其他相对位置不变。

维持两个链表：一个比 target 小，其他的往第二个链表上挂。然后将第二个链表挂在第一个之后。

88：将两个有序数组合到第一个数组中；

两个数组之和为 len,将两个数组，两个指针，从尾到头，比较，大的放到 len一那。

89：题都暂时一眼看不懂

90：可能有重复元素的数组，求所有的组合。

回溯+组合。

91：将数字解码成 A-Z;1-26;,给出数字字符串，有几种解码方式：

Dp[n],以 index 为 n 结尾的子串，有几种编码方式，dp[0] 如果等于 0 就是 0，否则是 1；

然后倒数第一个在 0-9，dp[i]=dp[i-1];如果倒数俩在 10-26 之间，再加上个 dp[i-2]。

92：逆转链表的 m-n 这部分；

逆转链表基本试用头插法。将 cur 的 next 都放到 pre 之后。而逆转中间一部分。是指针先走两步，m--,n--;直到 m=1;然后头插逆转。逆转过程中，只逆转 n-m 个。

93：给定一数字字符串，输出可能的 IP 地址列表。

三重遍历，将字符串分割为(0，i)、(i、j)、(j,k)、(k,len)四部分，然后对每部分都进行合格性检查：既:长度小于 4，数小于 256，第一位等于 0 的话长度只能是 1;

94,二叉树中序遍历：

Stack,当 root 不为 null 或者栈非空时，root!=null 时。一路将左子树全部压入。然后弹出，走右子树。前序遍历，压入的时候就处理，中序，弹出的时候处理。后续，压入的时候处理，但先压右，然后压左。然后反转、

96：n 个节点，共有多少种二叉搜索树。

Dp[n] = dp[n-i]*dp[i-1];(1-n);

98:判断一个树，是否是二叉搜索树。

中序遍历，会得到一个有序数组。所以必然要当前节点的值大于前个节点。

```
 if(pre != null && root.val <= pre.val) return false;
```

pre = root;

100：判断两个树是否一致。

递归：如果都为空，一致；如果有一个为空或者值不一样，不一致。然后递归俩的左子树和右子树。

101：判断一个树是否对称。

将 is(root)转化成 is(root.left,root.right);递归调用：都为空，真。一空或值不同为假。然后 left.left 和 right.right 递归。

102：二叉树层序遍历。

加入队列，当队列不为空的时候：弹出时，先标记下队列的宽度。然后这是一层。弹出 n 个，并处理将子树加入。

103,之字形层序比那里

队列，整个 falg 正反标记位，正时候，list 正着加入，反时候，list 加入后 reverse.

104:二叉树的深度:

递归，或者层序 size 遍历。

105：前序和中序构成二叉树。

将中序加入 map(in[i],i);然后递归(pre,sp,ep,in,is,id,map);主要是要判断左子树有几个，右子树几个的位置。

106：中序和后序构成二叉树

一样；

107：层序遍历：从底层到上层；

一样的层序遍历：只不过每层都要直接放入队前。

108: 将有序数组转化成二叉搜索树

数组的中间(start+end)/2;这个作为根节点，然后左边的递归作为左子树，后边的递归作为右子树、

109：将有序链表转化成二叉搜索树：

理念和上面一样，链表的中间节点作为根节点，前面的作为左子树递归，后面的作为右子树递归。而找中间节点，就是 pre,fast,slow,fast 走两步，slow 走一步。Slow 就是根节点。但要将 pre 节点和后面的节点断开，pre.next = null;

110:是否平衡二叉树：

Getdepth，在递归求深度的时候，要计算是否左右节点的高度是否大于 1；

111：叶节点到根节点的最短路径。

非递归的层序遍历：也是循环的是时候整个 size，但是如果弹出的时候，左子树且右子树都为 null，那证明叶节点来了，return 那个 level;

递归：

```
if (root == null)   return 0;
if (root.left == null)      return minDepth(root.right) + 1;
if (root.right == null) return minDepth(root.left) + 1;
return Math.min(minDepth(root.left),minDepth(root.right)) + 1;
```

112：二叉树是否存在从根节点到叶节点的路径和为 target:

因为这个比较简单，只需要判断是否有，而不是找出所有的路径。递归判断：root == null,不对。如果 root 没有左子树和右子树，且值等于 target,那直接返回 true。然后返回是否左右子树（left，target-val）；

113:找出所有路径和为 target 的序列

helper113(TreeNode root,int sum,List<List<Integer>> resList,List<Integer> tempList)

templist.add(root.val);如果是叶节点，且值为剩余的，就直接加入结果集。否则继续往下走。先左，后右，然后删除。

114：将二叉树往一边偏；

前序遍历？pre 和当前是左子树的关系；右子树都为空；

116：满二叉树，将为每个节点加上 next，就是每层的右边那个。

117：二叉树，将为每个节点加上 next，就是每层的右边那个。

125：字符串是否回文，只考虑数字和字母，大小写不论。

双指针，头尾，先将非字母和字符淘汰，然后转化为小写看看是否一致。不一样，直接 false：

129：二叉树，从头结点到尾节点：拼接成数字，然后所有的加一块。

递归求 sumRootToLeaf(root,sum);sum 为所有和拼接数字之和。

当 root null 直接返回 0!；当左子树和右子树都为 null 时，直接返回 sum*10+val;否则，左子树的递归加上右子树的递归。

136,一个数组，一个出现一次，其他的出现两次，求一次的那个

遍历：res = res ^ i;

137：数组，有人出现三次，有人出现一次，求一次的。

```
int a=0,b=0;
for (int i : nums) {
    b = b^i & ~a;
    a = a^i & ~b;
}
return a|b;
```

138：带有 random 指针的复杂链表的复制：

HaspMap,每个都带值复制一下：放 key,value 里面。然后根据 key，value 的 next,random，这些都补上。

139：

举个例子：字符串为 applepen 和 wordDict 为 apple，pen。当我们遍历到字母 n 的时候我们从 n 开始向前找看看有没有 apple：即字符串被分为 app 和 lepen（因为 apple 的长度为 5，我们期待 lepen 可能为 apple，若 lepen 确实为 apple，app 也能够被 break（即 dp[2] = true)，那么我们将 dp[7] = true,但 lepen 不为 apple 而且 dp[2]为 false，因此我们循环到下一个 wordDict 即 pen，将字符串划分为前面的 apple 和后面的 pen，而后面的 pen 确实存在，dp[4] = true，所以确实可以划分，因此 dp[7]= true。

141. 链表是否有环：

快慢指针从头结点出发， 快的走两步，慢的走一步。当俩人走到一起时，就真的有环。

142：链表环入口节点

走到一起时，快从头开始，然后来人都齐步走，再次相遇时，为入口头结点。

143：重排链表：1-2-3-4-5；先第一，再最后一个，先第二个，再倒二，就这样。

这个先记住，过会再说，看剧呢，分心

144：前序遍历二叉树：

栈，先左子树，入栈，一直再左。然后弹出，当 cur = pop.rightl;入栈的时候打印。

146. LRU Cache

147：对链表进行插入排序，

148：NlongN 的时间排序链表。

150:逆波兰

151：单词逆序：

s.trim().split(" +");分割成数组：然后数组 Arrays.asList(strs),然后 Collections.reverse();

然后 String.join(“ ”,strs);

155:最小值的栈

一个栈正常走，另一个保存最小值。保持栈顶为最小，往下就变大。

160：两个链表的相交点

非递归就是，长度，然后长的走两步，然后齐步走。

165：比较版本号打大小，大于就返回 1，小于返回-1，相等返回 0；有些版本 1.01 是等于 1.001 的；

将字符串按.切割成数组。遍历到最大的数组长度。比较短的就当成 0，不为 0 的 Integer.parseInt();比较大小。

167：有序数组，找出两数之和为 target 的索引。

二分查找、

168：数字转化为 26 进制的 A-Z；

```
while(n != 0){
    n--;
    sb.insert(0, (char)('A' + n % 26));
    n /= 26;
}
```

169. Majority Element，超过一半的数字

摩尔投票；

171：26 进制 A-Z 转化为十进制数字

172：Given an integer n, return the number of trailing zeroes in n!.

```
while(n!=0){
    n = n/5;
    res+=n;
}
```

179：整数数组，求组合到一块的最大值。

将整数数组转化为字符串数组，然后 Arrays.sort(strs,new Comparator){};

然后 join;

187:DNA 序列，ACGT；求 10 长度的连续序列出现超过一次的序列。

字符串从第一个字母进行遍历，每次就截取 10 个字符。长度为 10 的字符串放入 set 中，如果放不进去，说明这个字符串以及有重复了，将这个放入结果集中。

189：数组左移 K 位

和字符串左移 K 为一样，更简单。翻转函数 Swap(int[] array,int s, int e);

先打翻转，然后 k%len,再进行两次翻转

199：二叉树，从右边看到的节点数组。也就是每层最右边那个。

还是层序遍历吧。每次循环开始的时候，先取 queue 的数量 K，然后弹出 K,最后一个 K 就是每层最右面那个。

200：dfs 求岛屿的个数。1 表示陆地，0 表示岛屿。给定一个矩阵，求岛屿的个数。岛屿的周围都是水，则是独立的一个。

因为陆地是相连的。我们遍历这个矩阵，第一次遇见 1 的时候，说明 count++;然后我们采用 dfs,将周围的 1 全部变成 2 或 0；这个 dfs 然后向四周扩散，当然边界也得出来，遇到不是 1 的就停止。这样，遇见一个 1 就把所有能连上的都变成 2，然后继续下一块岛屿。

201：N-M 数组的连续数，逐个&；求最后的结果。

也可以一个个遍历，从 M 干到 M;但也有规律。就是看着连续的数组，也就是 N 和 M 最右边有几位是相等的。而不等的，说明有 0 有 1；那就说明中间夹着的后面的位置都有 0；所以&的时候结果为 0；当 m!=n 的时候： m >>1 ,n >> 1; count++;然后再 m << count;

209:最短的连续子序列之和大于等于 target;

这玩意就得遍历； 维持两个指针，一个是当前走的指针，一个是计算开始的连续指针。还有一个是 sum 之和。I=j=0,sum=0;然后 i 先出发，然后 sum += nums[i];当和大于 target 时，min = Min(min.i-j+1);同时也让 j 开始走，并 sum-nums[j],看是否 j-i 这一点能否大于 target;

215：无序数组中找出最 K 大的数字：

最小堆 PriorityQueue(默认为最小堆)；

216：1-9 的数，选出 n 个，之和为 k。求所有 N 的组合。貌似一个数不能用两次

回溯+组合

220：一个数组，是否存在不同的数 i 和 j,使得 nums[i]-nums[j]的绝对值最大是 t,而 i-j 的绝对值最大是 k;

桶排序之类的，不太会。

221：矩阵，有 0 有 1；找出最大的正方形，其里面数值都 1；

```
dp[i][j] = Math.min(dp[i-1][j-1],Math.min(dp[i-1][j],dp[i][j-1]))+1;
result = Math.max(result,dp[i][j]);

    return result * result;
```

这里面 dp[i][j]是对角线和上面和左面那个，最小的 dp+1； 如果周围三个为 0，所以只能以他开始构建正方形，其他人已经不靠谱。如果其他人都为 1，那么他可以作为边长为 1 的正方形的右下角。如果周围都是 2，说明周围都有扩展，他可以为三。

222：给定一完全二叉树，求节点总个数。

要么直接遍历求个数。这个显然没有用到完全二叉树的概念。

要么对根节点，求最左的高度，最右的高度。如果他来一样。1<<leftDepth -1;如果不一样，递归求 1+count（左子树）+count(右子树)

223：以(A,B)和（C,D）为对角顶点，构建矩形，以 EF，GH 构建，求纵的矩形面积。

求 ABCD 面积和 EFGH 面积，求 AE，BF 的最大作为左底层，CG,DH 的最小作为上油层，如果左底小于上右，那么求这个重复面积，然后总减去这个得到总面积

227：非负整数，有+-*/和空格数字的字符串组成的表达式，求表达式的值；

整个栈。字符串从 0 开始遍历，如果是数字，继续进行，记得累加。如果遇到了非数字或者到头了，就要对上面的那个数字进行总结了，即现在遇到的 sign = “+-*/”;最开始时是+，所以：如果遇到+，就直接 push num,如果遇到-,就 push-num,如果遇到*，就将前面那个弹出，然后 push*num,如果遇到/就弹出然后 push 除去的结果。然后所有的弹出，然后求和；

228：给出排序好的数组，没有重复，将连续的整成 0->2 之类的形式，求所有的连续范围段。

也是数组从头遍历，如果下一个的值减去当今==1，那么证明这是个连续的，所以一直往下走，直到走到断开的地方。然后让当今的 cur 和 i 一直走的 nums[i]，看看是否相等，如果相等，说明这个人没跟人连续，直接加入结果集。如果不相等，就转化为 cur---nums[i]的方式,加入。

229，数组中超过 n/3 的次数的数。

摩尔投票变种：int countA = 0 ,countB = 0, res1 = 0 , res2 =0;对数组进行遍历，如果 res1，count1++；如果等于 res2,count2++;如果都不是，在 A=0 的情况下，对 A 进行转换成当前的。否则 B=0 的情况下，对 B 进行转换。如都不是，count1--,count2--;

然后再次遍历，看数组中 res1,res2 出现的次数是否超过了 n/3;

230:BST 二叉搜索树中，第 K 小的节点

中序遍历就是一个有序数组，遍历的时候记着，然后到 k 时，直接返回。2

236：二叉树中，两个节点最近的公共祖先节点。

递归的看的难受，看不懂

238：构建乘积数组

第一遍遍历，存前面的连城。第二遍遍历。存后面的连城，然后再相乘。

240：在矩阵中遍历，每行都增加，每列都增加。

从右上角开始遍历，如果小就下移，大就右移动。

241：数字和操作符，+-*,请随便加括号改变计算顺序，然后得出所有可能的结果。

这个有意思：这个就相当于分治；

我们首先遍历这个字符串。遇见加减乘号，将字符串分割成两部分，直到将所有的数字全部分成小段。

然后根据分段的符号，进行双层遍历，左边的取一个，右边的取一个，然后进行操作。结果放入 resList,进行返回。

260：数组，只有两个出线一次，其他的出现两次，求出现一次的那俩

首先遍历 diff ^= n； diff = diff & -diff; 然后再遍历，diff^n == 0 的一组，其他的一组。分别得出 res1,res2;

264：第 K 个丑数，因子只有 2,3,5 的数；

Dp[m] = min(dp[i]*2,dp[j]*3,dp[k]*5);维持 2,3,5 的指针，从 1 开始。一旦采用，就++;

274：if h of his/her N papers have at least h citations each, and the other N − h papers have no more than h citations each.

就是作者一共 N 个论文，其中 h 篇的引用都要大于等于 h，剩余的都不大于 h。这 h 就是他的引用数。

分成 N+1 个桶。0-N;将值大于等 N 的都放入 N 桶中，其他的都放在自己的序号中。

然后从 N 桶倒数 if(count >= i) return i;

这个 count 的计算方式:将大于等于 H 的都统计了，剩余的都在小 h 桶，也符合规矩。

275：同上，也是求 H。但这个数组是有序的；

有序的，就整二分查找。主要是看 citations[mid]<index 的关系，index 就是右半边的大数有几个。

287：N+1 数，都是 1-n 之间，求重复的那个数。

```
 int slow = 0,fast = 0;
while (true){
    slow = nums[slow];
    fast = nums[nums[fast]];
    if(slow == fast)
        break;
}
fast = 0;
while (fast != slow){
    fast = nums[fast];
    slow = nums[slow];
}
return slow;
```

300：最长递增序列，可以不连续。但是前面的 index 比后面的小。

求 dp[n]:当然，整个遍历。然后再遍历比当前 index 小的， 当 nums[i]>nums[j]时，dp[i] = Max（dp[i],dp[j]+1）;然后求 dp[n]中最大的。

306：数字字符串，我们将字符串分割最少三个数字，然后 Fn = Fn-1+Fn-2;

一下子看不懂了。

310：一个图，给的形式是 n 个节点，然后节点间关系相连的[][].以一个节点为根节点，求得到的最小高度的树的根节点。

313：超级丑数，求第 N 个丑数，并且给出数组因子，只有只包括该数组因子的数才是超级丑数

Dp[i] = max(dp[i],dp[index[j]]*pre[j]);

318:字符串数组，求两个没有公共字母的字符串长度乘积的最大值；

看不懂，

319：开关转换，刚开始有 N 的灯，最初都是灭的，第一轮，每一个都按一个。第二轮，每二个按一个开关。第 N 轮之后，亮着的灯的个数。

return (int)Math.sqrt(n);

322：给一堆零钱，可以重复使用，然后给整钱。将整钱换成零钱，求最小的零钱数，如果不能换，返回-1；

324：无序数组重排序：nums[0] < nums[1] > nums[2] < nums[3]....

他这是，重新复制一个数组，然后排序。然后遍历原数组，将排序好的前一半放到偶数位，后一半放到奇数位。

328：链表重排序，将奇数位 index,而不是奇数值的放到前面。偶数 index 的放后面：

跳着来，将 odd.next = odd.next.next;even.next = even.next.next;这样就将 odd 的串一块，even 的串一块。然后将 old 指向奇数的开头。

331：将一个二叉树，他的空子节点记成#，给定一个字符串，问这个是不是前序遍历；

332:航班信息[from,to],一些列的这种数组信息，这人从 JFK 出发，求他的真正路线一次经过的城市；

334：数字数组，是否存在三个子序列，可以不连续，是递增的；

Return true if there exists i, j, k

such that arr[i] < arr[j] < arr[k] given 0 ≤ i < j < k ≤ n-1 else return false.

```
int m1=Integer.MAX_VALUE,m2=m1;
        for(int num:nums){
            if(num <= m1)
                m1 = num;
            else if(num <= m2)
                m2 = num;
            else
                return true;
        }
        return false;
```

338：给定一 num.求 0-num 这些数每个的二进制中 1 的个数。

res[i] = res[i>>1]+(i&1);

347，TopK 出现频率最高的数字，一个数组，每个数字可能出现多次，求最 K 多出现的那 k 个

先用 hashmap 统计每个数字出现的次数，key 是数字，value 是出现的次数。

然后弄个优先队列，

343：给定一个整数，将他分成最小二段，和为这个整数，求这些段乘积的最大值。

```
dp[i] = Math.max(dp[i],Math.max(j, dp[j]) * Math.max(i - j, dp[i - j]));
```

357:n 位数的数，求没有重复数字的总个数。11,121 这种就不行。

```
if (n==0)
            return 1;
        int sum    =10;
        int q = 9;
        for (int i =1; i<n; i++){
```

```
            q *= (10-i);
            sum += q;
        }
        return sum;
```

头条：

1、数组的逆序数　//归并排序时，都从右到左比那里，比较大小时，如果左边大，逆序数加；

2、LRU　//hashMap 加双向链表，双向链表有头尾节点，key,value 封装成 Node,hashMap 的 key 为 key,value　为 node 节点。node 节点又在双向链表中

3、最长回文序列　leetocde 5 ：从左到右遍历时，并扩展，看能扩到哪。记录最大和最左

4、矩阵中的最长递增路径，可以上下左右一起都走；　leetcode329　//双层遍历进行 dfs,维持一个二维矩阵。如果矩阵不为 0，max=1,往四个方向遍历，如果符合条件，进行扩展。len = 1+ dfs();然后求 max,最后将 max 赋值给数组。

5、判断一个二叉树是另一个二叉树的子树 剑指　//2 是否为空，1 是否为空或者是否相等，左子树和右子

6、归并排序的时间复杂度　// NlongN

7、求给出 01 矩阵中的最大正方形面积（全为 1）　lc221　dp[i-1][j-1] = min{dp[i-1][j-1],dp[i][j-1],dp[i-1][j]}+1;

8、求二叉树中距离最远的节点　leetcode543　// 递归求根的深度，max = Math.max(max,left+right);

9、判断字符串是否为合法 IPV4 地址　　lc468//对.切割，是否长度为 4，且每个大于 0 小于 256

10、数组值为 1-n，各出现一次，先加入 x（x 也是 1-n 的范围），找出 x //将 A[i]放到 i 位置，如果不是，

while(A[i] !=　i) if(A[i] = A[A[i]) 这个重复，否则，i 和 A[i]位置的交换

11、给定 n，计算 15n，不用+*/　　n<<4 – 15;

12、给定字符数组 chars，将其右移 n 位 先将整个数组翻转，然后将 len-n 翻转，n 翻转

13、100 层楼，只有两个鸡蛋，找出鸡蛋会在哪一层楼被摔碎//x+x-1+x-2+….1 = 100;

14、reverse linked list in a group of k //先求长度，然后 for(int j = len;j>=k;j = j-k) 然后遍历一段时间，头插。

15、如何空间 O(0)实现两个数的互换 a = a +b; b = a-b; a = a-b;

16、IP 地址的 Regex // 25[0-5]|2[0-4]/d{1} | [0-1]?\d{0,2}

17、the longest path in a binary tree　lc124

```
        if(root == null) return 0;
        int left = Math.max(0,MaxPathDown(root.left));
        int right = Math.max(0,MaxPathDown(root.right));
        maxValue = Math.max(maxValue,left+right+root.val);
        return Math.max(left,right)+root.val;
```

18、the largest consecutive sum in an array dp[i] = max(dp[i-1]+ nums[i],nums[i]);

19、LeetCode 41 Find mis sing positive将符合条件的，即大于 0 小于 n 的，进行转换。换到该去的地方 A[i]放到 i-1 地方。

20、给一个小于一亿的中文数字字符串,转化成数字格式

21、一个数组,把所有的 0 都移动到末尾,还要保持顺序不乱 维持临界点 j,如果当前遍历不是 0，就和 j 互换

22、罗马数字转整数 leetcode13 //罗马数字特点，如果这个比后面的小，减去，否则加上。

23、二叉树的序列化和反序列化//层序遍历

24、输入一个数组，输出数组中满足条件的数字，条件为：数组中当前元素的值大于等于它前面所有的元素，小于等于它后面所有的元素。// 维持一个数组，以及长度，以及目前的 max.如果当前大于 max，入队。否则，将队里面的慢慢退。

25、给出一个数字，对数字的两位进行交换，只能交换一次，输出可能结果中的最小数字//桶排序，table = new int[10];算出每个数字最后出现的位置。从前面遍历，从 9 开始遍历，如果 table[i]存在，就交换。

26、输入一个字符串，字符串中字符全部为数字，在字符串中插入 '.' 使得结果为合法的 ip 地址，输出全部可能的结果

27、基数排序

28、是否为回文结构。一快一慢，走到中间，然后逆转链表。然后走，看是否回文

29、最大不重复子串 set 以及前面的 j;往前遍历，如果 set 不重复，加入，否则将 j++的一个个弹出去。

30、复杂链表复制//

31、长度为 n 的数组，元素大小是 0~n-1,判断数组元素是否有重复的

32、list1/list2 交替打印元素

33、36 进制加法

34、合并区间

35、快排
36、生产者-消费者 模型
37、排序一个字符串时间要求 O(n)
38、给一个有重复数字的数组，求集合{(a,b,c) | a+b+c=0}
39、两个栈实现队列
40、二叉树转化为双端链表
41、手写线程池
42、之字形打印二叉树
43、给定一个数组，调整该数组，使其满足堆的性质
44、给定 n 个单词，如果单词组成一致但是元素顺序不一致，该对单词为同位词，例如：abc,bca 为同位词．求所有同位词的集合输出
45、链表，两个链表的公共点
46、二叉树的后续遍历非递归形式
47、买卖股票的最佳时机，只能一次买入和一次卖出
48、可以进行多次交易的结果，求赚取的最大利润
49、(A,B)(A,C)(B,D)(D,A)判断是否有循环引用，提示用拓扑排序
50、数组找是否存在和为 M 的两个数
51、KMP
52、实现一个阻塞队列（生产者消费者模型）
53、找出 10000 个数据中第 k 大的数
54、输入一个字符串，包含数字、加减乘除和括号，输出结果，编程
55、给定一个数字 x，要求使用 k 个数字求和可以得到 x，数字从 1-9 中选择，不能重复。
56、输入一个正整数 N，返回 N 个 '(' 和 N 个 ')' 的所有可能情况
57、76.minimum-window-substring、30.substring-with-concatenation-of-all-words、4.trapping-rain-water，
58、求树的最左下节点
59、无序数组中第 k 大的数（quick select）
60、求旋转数组找最小值（二分）
61、判断二叉树是否镜像（递归）
62、给定一个矩阵，从左上角开始只能往下或者右走，求到达右下角的最小权值路径
63、字符串转 Int，如果越界就返回 0
64、lc400
65、单向链表实现加法
66、打家劫舍
67、收到礼物最大值
68、五张牌，其中大小鬼为癞子，牌面为 0，判断这五张牌是否能组成顺子
69、给定一个字符串打印所有的子串，要求不重复
70、自然数 1-n,排一块组成的字符串，求第 k 位是什么，12345678910，如何第 10 位是 0；

算法问题的一些总结：

1、位运算：

交换 AB ; A = A^B; B = A^B; A = A^B;

位运算实现加法：a^b 是不进位的相加，(a&b)<<1 是相加的进位；

while(b != 0){sum = a ^b; b = (a&b)<<1; a = sum;} return sum;

位运算实现减法：a-b = a + (-b); b 变成-b : negNum(int b){return add(~b,1)};

二进制有多少个 1， n & (n-1)使二进制最右的 1 变为 0；

异或 a^0 = a ; a^a = 0;

所以一个数出现奇数次，一个数出现偶数次，那么奇数次那个就位 x^n = x;

如果两个树出现奇数次，其他数出现偶数次。那么所有的异或结果为 a^b,因为这俩不一样，所以

一定不是 0，某一位存在一。

x & (~x + 1)的结果就是 x 中最右第一位为 1，其他为 0 的数。

然后第二次遍历，只取 n & x != 0 的数，进行异或，将其中的一个奇数位淘汰，结果就是其中的一个 a。 然后 a ^ x = b;

头条的再总结：

http https (方式，get post update 那些？区别)介绍(还插问了状态码)，它们用的是 tcp 还是 udp，用 udp 行不行，ssl 怎么加密的，怎么确保安全的，还插问了对称加密非对称加密。这里追问的有些多

判断子树问题

SOCKET 编程？？？

讲一下 B+树，红黑树，红黑树的使用场景

段页式内存管理，和段式，页式的比较

5.算法，逆序输出单向链表，不能用额外空间，不能链表逆序，写了个递归，又问我链表很长栈溢出怎么办……

6.算法，一个矩阵中找一条升序路径，写了逐点用 DFS

操作系统虚拟内存换页的过程

虚拟内存换页的算法有哪些？

换页算法里面，FIFO 有什么缺点？怎么改进？

为什么公平锁效率低？

滑动窗口编程题

4G 内存，40 亿整数，全排序该怎么做

设计一个 MaxStack

N 个能随机等概率生成 1-N 的数字的骰子，对其做 distinct 之后的剩下元素个数的数学期望

比方说三个骰子，如果投出 1 1 1，那么 distinct 之后只剩下 1 个元素

若干个线程，有执行的开始时间和结束时间，如果两个线程时间有重叠，那么就需要并行计算，问最多会有多少个并行计算

一棵树，如果从右边看这个棵树，返回从上到下的输出

HashMap 怎么实现？除了链表法还有什么办法解决冲突？

如果有若干个线程同时执行 HashMap 的 put 操作，会有什么后果？

协程和线程的区别？

概率题：如果一直抛硬币，直到：如果出现反反正就 A 赢，如果出现正反反就 B 赢，那么这是公平的游戏吗？谁获胜的概率大？

算法题： 如果 a[0]<a[1],a[n-2]>a[n-1]，那么请找出任意一个点使得 a[i-1]<a[i]>a[i+1] 要求 logN

描述堆的实现

如果有一组数字，按照“拿出第一个数在桌上并然后将下一个数放到队尾”一直操作直到数字全部放在桌子上，给你最后在桌子上的数字，请返回最开始数字的顺序

一个数组，找最大的连续子数组和

linux 中如果读取文件"/xx/yy/z"请描述过程

XML 格式解析

有序数组找到第一个小于 0 的数和第一个大于 0 的数

fork 和循环结合后打印多少个字符

实现一个 string 类

矩阵左上角到右下角的最短路径和
实现一个智能指针
合并两个排序数组并去重
最长无重复子串
两个排序数组找中位数
string 转 float

- 用过 socket 吗？常用函数有哪些？
- TCP 与 UDP？打电话、发短信、广播分别适合用什么？
- HTTP 和 HTTPS？公钥私钥？私钥加密的数据能用公钥解密吗？如果能，安全性怎么保证？
- 进程和线程？进程间通信？允许多个进程读，单个进程写，锁该怎么设计？
- 把[1, 10^15]的数映射到[1, 10^6]，哈希函数怎么设计？如果数据分布不均匀，1 出现 1 次，2 出现 2 次，n 出现 n 次，哈希该怎么设计？哈希冲突怎么解决？
- 两个超大整数的字符串做减法运算。

``

```
public String addStrings(String num1, String num2) {
        StringBuilder sb = new StringBuilder();
        int carry = 0;
        for(int i = num1.length() - 1, j = num2.length() - 1; i >= 0 || j >= 0 || carry == 1; i--, j--){
            int x = i < 0 ? 0 : num1.charAt(i) - '0';
            int y = j < 0 ? 0 : num2.charAt(j) - '0';
            sb.append((x + y + carry) % 10);
            carry = (x + y + carry) / 10;
        }
        return sb.reverse().toString();
    }
```

- 1~10000 中 7 出现的次数，如 17 算 1 个，77 算 2 个。

```
//No43,1-n 所有数中，求所有十进制位出现 1 个总数
    public int NumberOf1Between1AndN_Solution(int n) {
        //
//          1. 如果第 i 位（自右至左，从 1 开始标号）上的数字为 0，则第 i 位可能出现 1 的次数由更高位决定（若没有高位，视高位为 0），等于更高位数字 X 当前位数的权重 10i-1。
//          2. 如果第 i 位上的数字为 1，则第 i 位上可能出现 1 的次数不仅受更高位影响，还受低位影响（若没有低位，视低位为 0），等于更高位数字 X 当前位数的权重 10i-1+（低位数字+1）。
//          3. 如果第 i 位上的数字大于 1，则第 i 位上可能出现 1 的次数仅由更高位决定（若没有高位，视高位为 0），等于（更高位数字+1）X 当前位数的权重 10i-1。

        int count = 0;//1 的个数
        int i = 1;//当前位
        int current = 0,after = 0,before = 0;
        while((n/i)!= 0){
            current = (n/i)%10; //高位数字
            before = n/(i*10); //当前位数字
```

```
            after = n-(n/i)*i; //低位数字
            //如果为 0,出现 1 的次数由高位决定,等于高位数字 * 当前位数
            if (current == 0)
                count += before*i;
                //如果为 1,出现 1 的次数由高位和低位决定,高位*当前位+低位+1
            else if(current == 1)
                count += before * i + after + 1;
                //如果大于 1,出现 1 的次数由高位决定,//（高位数字+1）* 当前位数
            else{
                count += (before + 1) * i;
            }
            //前移一位
            i = i*10;
        }
        return count;
}
```

如何保证 int 加加（加号打不出来）线程安全。

cpu 调度方式有哪些。

- 100 级台阶，每次上一步或两步，有多少种走法。
- 如果 200 级，你估计有多少种走法（不用编程）。
- 给出一个数组，找出两个数[a,b]和为 n，不存在则返回[-1,-1]。
    - 字符串实现减法
    - 快速排序

以 O(1)时间复杂度取得最小值的栈，要求有 pop push getMin 方法

tcp 三次握手加一次变成四次握手有什么问题？

数据库行锁表锁，什么时候会加锁?

查询语句是否用到索引的分析。

rocketmq 消息队列原理，单机性能多少？ 为什么这么快？

rocketmq 如何支持事务的？

1.给一个字符串数组，统计每一个字符串出现的次数，要求不能用 set,map.时间复杂度 O(n).

实现一个阻塞队列，考虑到多线程并发的情况，要求有 put,get,isEmpty, isFull 方法。

手写算法: 反转二叉树

场景题: 1 索引设计:一个表有三个字段 A,B,C

2 设计一个短链接服务，短信中的短网址点开之后变成完整的 url，完整的 url 转成短网址发送到用户短信中。整体流程设计。

3. 给两个文件 a,b a 大小为 3t, b 大小为 2t，a 中存储的是 id 和 name ，b 中存储的是 id 和 title，计算机内存 2g，要求用最快的方法找出 a 和 b 的 id 重合的部分，输出文件 c，c 中存储的是 id,name,title。 注 id 是 varchar 32.

unicode、utf8 的区别

Unicode 把所有语言都统一到一套编码里，这样就不会再有乱码问题了。Unicode 标准也在不断发展，但最常用的是用两个字节表示一个字符。如果要用到非常偏僻的字符，就需要 4 个字节；UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用 1~4 个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8 的编码规则很简单，只有两条：

对于单字节的符号，字节的第一位设为 0，后面 7 位为这个符号的 Unicode 码。因此对于英语字母，UTF-8(Unicode)编码和 ASCII 码是相同的。

Z 遍历二叉树，递归?

SYN 洪范攻击?如何避免

lc 33
1-n-2-n-1;链表变形；
8.http 状态码
9.http1.0/1.1/2.0 了解吗，有啥不同
10.hashmap 内部结构，详细讲讲，何时扩容，线程安全吗，哪些线程安全，hashtable concurrenthashmap 有啥不同
手撕最大堆，实现对应的 push 和 pop 操作
智力题：门外 3 个开关，门内三盏灯，门外看不到门内的状况，只能进去一次，怎么确定所以开关对应哪个灯
开灯会热!!!!!!!!
towsum，找出给定数组里所有和为 target 两个数的组合，数字不能重用
判断字符串是否为合法 IPV4 地址
讲讲 md5，md5 加密的可以解密出来吗？
.http 协议中有一个 range 字段，是什么作用，如果是使用到这个字段，返回码是什么？
8.如何遍历 arraylist，删除值为 n 的数，即假设是{1,2,1,1,3,4,1}，遍历删除值为 1 的。
9.了解迭代器吗，能不能用迭代器做上面的删除
syn 攻击
4、跳表的数据怎么判断是排在第几位
5、联合索引是只建了一个树吗，查询数据过程
ping 是哪个协议
ping 程序是用来探测主机到主机之间是否可通信，如果不能 ping 到某台主机，表明不能和这台主机建立连接。
ping 使用的是 ICMP 协议，它发送 icmp 回送请求消息给目的主机。ICMP 协议规定：目的主机必须返回 ICMP 回送应答消息给源主机。如果源主机在一定时间内收到应答，则认为主机可达。

traceroute 协议
Windows 下是纯粹使用 ICMP 报文。直到找到目的地址，并记录经过的路径
而 linux 下是发出一个 UDP 报文。且同样也设置 TTL 时长，但选择一个不可能的值作为 UDP 端口(大于 30000)，使目的主机的任何一个应用程序都不可能使用这个端口

内存泄漏是什么，怎么检测
写一个 list 删除目标元素的函数，然后写个测试用例测试一下能不能通，为什么不能正向遍历
倒序遍历，别用 foreach;
或者用迭代器的 remove()方法

leetcode 93

```
public List<String> restoreIpAddresses(String s) {
    List<String> resList = new ArrayList<String >();
    int len = s.length();
    for(int i=1;i<4 && i<len-2;i++){
        for(int j = i+1;j<i+4 && j<len-1;j++){
            for(int k = j+1;k<j+4 && k<len;k++){
                String a = s.substring(0,i);
                String b = s.substring(i,j);
                String c = s.substring(j,k);
                String d = s.substring(k,len);
                if(isVaildIP(a)&&isVaildIP(b)&&isVaildIP(c)&&isVaildIP(d))
                    resList.add(a+"."+b+"."+c+"."+d);
            }
        }
```

```
        }
        return resList;
    }
    public boolean isVaildIP(String s){
        if(s.length() > 3||Integer.parseInt(s)>255||s.length()==0||(s.charAt(0)=='0'&&s.length()>1))
            return false;
        return true;
    }
```

找出一个字符串中所有的回文子串

死锁的四个条件

互斥条件、请求与保持、不可剥夺、循环等待

预防死锁 避免死锁

死锁避免：分配资源之前，进行动态检查，如何分配之后处于安全状态，那就分配，否则不分配。

服务器是怎么找到客户端的，通过 ip 就直接找到了吗？

二叉树两个节点间的最长距离

重复次数最多的最长连续子串（即找到重复次数最多的子串，若有多个，输出最长的）

1. 10G 数据，1G 内存，如何快速找到重复出现的数据？
2. 10G 数据，1G 内存，如何快速找到重复出现次数最多的数据？

1.输入一个数组，输出数组中满足条件的数字，条件为：数组中当前元素的值小于等于它前面所有的元素，大于等于它后面所有的元素

3. 2.给出一个数字，对数字的两位进行交换，只能交换一次，输出可能结果中的最小数字
4. 二面：
5. **1.输入一个字符串，字符串中字符全部为数字，在字符串中插入 '.' 使得结果为合法的 ip 地址，输出全部可能的结果**
6. 2.输入一个矩阵，矩阵中元素为 0 或 1，找出满足条件的正方形的最大边长，条件为正方形中的元素全部为 1
7. 三面：
8. 1.解一个一元三次方程
9. 2.日志文件中记录着一个直播间中用户进入和退出的日志，字符串格式为 id:in:time 或 id:out:time，求出当天直播间内同时在线人数的最大值

2.平衡二叉树的结构是什么样的，插入数据最多会旋转几次。2 次

7.手撕代码。大概就是公司要搞团建，组成两两一组，但是大家只愿意和熟人组队，给了一堆输入，第一行是 N 个人，第二行分别是 该人的序号，有几个伙伴，伙伴的序号。问最多可以组成多少队伍。

2.ping 的过程用到了什么协议？ICMP，是哪层的？网络层。

有 1,2,5,10 个零钱，要组成 N 元，总共有多少种组合方式。

每个任务需要 100ms，现要达到 1000tps，需要 4 核 8G 的服务器多少台，每台需要开多少线程？（完全不会。。。）

2. 有一个 10GB 的文件，里面存储了 uint32 型的数字，现需要找出哪些数字是重复的（1G 内存）？

说了每次读取一小片数据，然后构建位图判别是否重复出现，然后被问了位图使用多大的空间？

进阶：找到重复次数最多的前 k 个数字？

读取策略相同，使用 hashmap 进行频率统计，然后问 hashmap 占用内存大，怎么使用分片的方式进行统计，要细节。。。

3. 编程是给定 unix 文件路径，对其进行简化，主要考虑 /，./，../。

我使用了双端队列，对分割后的路径元素进行入队，当出现../时进行尾端出队，最终拼接队列元素。

4. 输入 url 跳转到另一个 url，在后端怎么做？状态码 301,302 的区别？

根据前序遍历和中序遍历还原二叉树，根据获得的前序遍历和中序遍历结果输出一颗二叉树（输出后续遍历）

单链表操作：

输入：奇数位升序，偶数位降序

- 假定一张表的数据格式为 id,name,parentId，表的数据不大，1000 条以内，得到这些数据的树型结构 输入：List ， 输出：Node（手写代码）
- 有 2 个文件，分别是 A(3t 大小)，B(2t)大小，A 文件的组织形式为 uid, username,B 文件的组织形式为 uid, age,找出 A、B 文件交集的数据放入一个文件，文件的数据格式为 uid,username,age（讲讲思路）

对 uid 进行 hash 将大文件拆分成小文件，小文件之间取交集，再合并结果

7. 算法题：打印根节点到所有叶子节点的路径，挺简单的 dfs 三分钟搞定~
8. 算法题：给定一个数组 height[n] 表示山的高度，然后下雨了，水量够大，把山都淹了，问存了多少水？

5. 介绍一下页表？
6. 页表的数据结构？除了有地址相关的参数还有什么？（我只说了多级页表，其他没说了，后来意识到他可能再说 TLB？自闭了。。

多级页表：页目录+页表+页内偏移。

7. 伙伴系统了解吗？介绍一下？
8. 分段和分页有什么区别？各自适合什么场景？
9. 伙伴算法晓得吗？介绍一下
10. 进程调度算法了解什么？介绍一下 O(1)和 FCFS？

8. linux 的一些常用命令：查看 cpu 网络 磁盘 内存的命令都知道哪些？
9. gdb 用过吗，说一些 gdb 的命令？
10. 单向链表快排？

hash 冲突的解决方法:

了解进程间的通信吗？

我说，进程间的通信包括管道，命名管道，消息队列，共享内存等。一般管道比较慢，消息队列做同步，共享内存快。

tcp 本身通过 ack 确认机制来保证确认信息收到了，并通过重传机制对丢失信息进行重传，也就是说，应用端将数据交给 TCP，自己啥都不用管了，他一定能传到，所以称之为可靠性，udp 没有这样的机制，直接 最大化传输，但是，并不意味着他就不可靠，只是可靠性的保证交给了应用端去做。

8.数据库 sql 语句查询，跨表查询有哪几种方式

10.算法题一，长度为 n 的数组，有一个长度为 k 的滑动窗口，询问各个滑动窗口内的中位数。

11.算法题二，二维数组从上到下从左到右单调递增，要求 O(n)复杂度数量级内查找出二维数组内是否存在该数

两个链表相加，类似于大数加法的形式。

最大连续子序和，

最大的两个不连续的串和

二叉搜索树找第 k 小数据，两种方法实现。中序遍历？递归？非递归？

8.代码题：反转单链表。

9.代码题：复杂链表复制。

10.代码题：数组 a,先单调地址再单调递减，输出数组中不同元素个数。要求：O(1)空间复杂度，不能改变原数组

区间最大最小值。两个长度为 n 的序列 a,b,问有多少区间[l,r] 满足 max(a[l,r])<min(b[l,r])即 a[l,r]的最大值小于 b[l,r]的最小值

18.代码题：最大不重复子串。给定一个字符串，找出其中无重复字符的最长

子字符串的长度。

19.代码题：字符串 s=“0123456789101112...”返回 s[i]，让采用逐步缩小范围的方法，

分为三类：一位数、二位数、三位数，给定一个 m 直接可以判断它落在

哪个区间，然后从区间起始开始填数字串。

写一个代码提现继承，泛型，多态。

写一个代码，在数组找三个数的和距离目标值最近。

手写单例模式。

线程池的实现

多线程和多进程

http 和 https,ca 证书

tcp 和 udp

tcp 协议需要连接，有没有知道其他的协议能改善它的缺点。

设计一个协议，规避 tcp 的缺点，拥有 tcp 和 udp 的优点。面试官有提到一个谷歌设计的协议，具体叫什么没听清。

QUIC 协议

MapReduce 的 Map 和 Reduce 怎么做的

8 皇后问题共有多少种解法

Epoll 的水平触发和边缘触发的区别，如何处理边缘触发模式下数据的读取

1. redis 有多少种数据类型，**有序集合的底层实现，插入和查找的复杂度**
2. **一个数字串删除指定个数的数字字符，剩下的组成一个最大的数**（3051，若删除 1 个，则为 351，删除 2 个则为 51，删除 3 个则为 5）

对链表进行排序，我给出归并排序的方法，又问不使用递归如何做归并排序，还有其他方法可以对它排序吗，回答冒泡排序，又问如何优化冒泡排序，如果链表部分有序如何优化？

100 个小球，其中红色 50 个，蓝色 50 个。有两个抽屉，每个抽屉都足以放下 100 个球，将这 100 个球放到这两个抽屉中，另一个人选其中一个抽屉并从里面随机拿一个球，如何分配才能使拿到红球概率最大。

有两个 string A 和 B，判断 A 是否为 B 的字串，然后我用了最简单的方法实现后，问我是否有优化的可能（一开始脑抽说二分，说了两句戛然而止，后面也没有想到很好的方法，就不献丑了，好气哟前两天还跟荣鹅讨论说了正则，面试时一个字也没想起来 = =）

设计一个资源池，以下三个操作满足 O(1)复杂度：（1）从未分配的资源中分配一块资源；（2）从已分配的资源中释放指定资源；（3）随机访问一块已分配的资源。大致意思是这样。

2. 有 A，B，C，D 和 E 共 5 所学校。在一次检查评比中，已知 e 肯定不是第二名或第三名，他们相互进行推测，A 说，E 一定是第一名；B 说，我校可能是第二名；C 说，A 校最差；D 说，C 不是最好；E 说，D 是第一名。结果只有第一和第二名的学校猜对了。编写一个程序，求出这 5 所学校的名次。

1.N 个长度为 K 的有序链表合并，时间复杂度，空间复杂度（用的归并，当时面试官还问时间复杂度和空间复杂度还能优化吗，想了一下想不到，然后面试官说他也不知道怎么优化😂）

输出一个有序数组中一个数字的第一次出现的位置或者返回-1.

```
for (int i=0;i<n;++i)
    {
        dp_max[i][i] = a[i];
        dp_min[i][i] = b[i];
        for (int j=i+1;j<n;++j)
        {
            dp_max[i][j] = max(dp_max[i][j-1], a[j]);
            dp_min[i][j] = min(dp_min[i][j-1], b[j]);
        }
    }
```

寻找峰值： 最左和左右都是负无穷大。

```
    public int findPeakElement(int[] nums) {
        int left = 0;
        int right = nums.length-1;
        while(left <right){
            int mid = left + (right-left)/2;
            if(nums[mid] <nums[mid+1])
                left = mid + 1;
            else
                right = mid;
        }
         return left;
    }

}
```