本文内容出自：https://github.com/gzc426/Java-Interview

以后有更新内容，会在 github 更新

加入**互联网 IT** 求职、技术交流、资料共享 QQ 群 **691206173**，3T 编程资料等你来拿，

群主（群号：**691206173**）：本硕就读于**哈尔滨工业大学**，计算机专业，2019 硕士毕业，已拿**百度** java 后台开发 offer，另外还有去哪儿，华为，茄子快传，vipkid,秒针，创新工厂一点资讯这些公司的 offer。公众号中的文章有分享群主找工作的经验，java 学习/C++学习/机器学习/前端的指导路线，以及未来在百度的学习成长之路，满满都是干货，除了干货分享还有 3T 编程资料**（java/C++/算法/php/机器学习/大数据/人工智能/面试等）**等你来拿，另外还有微信交流群以及群主的**个人微信（抽空提供一对一指导意见）,当然也希望你能帮忙在朋友圈转发推广一下**

公众号

拥塞控制

慢开始

发送方维持一个叫做**拥塞窗口 cwnd（congestion window）**的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。

实时拥塞窗口大小是以字节为单位的。　　当然收到单个确认但此确认多个数据报的时候就加相应的数值。所以一次传输轮次之后拥塞窗口就加倍。这就是乘法增长，和后面的拥塞避免算法的加法增长比较。

为了防止 cwnd 增长过大引起网络拥塞，还需设置一个慢开始门限 ssthresh 状态变量。ssthresh 的用法如下：

当 cwnd<ssthresh 时，使用慢开始算法。
当 cwnd>ssthresh 时，改用拥塞避免算法。
当 cwnd=ssthresh 时，慢开始与拥塞避免算法任意。

拥塞避免

拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍。这样拥塞窗口按线性规律缓慢增长。

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（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限 **ssthresh** 设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为 1，执行慢开始算法。

快重传

快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

快恢复

①当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半。但是接下去并不执行慢开始算法。

②考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。

为什么要第三次握手

为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？

这是因为服务端的 LISTEN 状态下的 SOCKET 当收到 SYN 报文的建连请求后，它可以把 ACK 和 SYN（ACK 起应答作用，而 SYN 起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的 FIN 报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭 SOCKET,也即你可能还需要发送一些数据给对方之后，再发送 FIN 报文给对方来表示你同意现在可以关闭连接了，所以它这里的 ACK 报文和 FIN 报文多数情况下都是分开发送的。

第三次握手失败

当客户端收到服务端的 SYN+ACK 应答后，其状态变为 ESTABLISHED，并会发送 ACK 包给服务端，准备发送数据了。如果此时 ACK 在网络中丢失，过了超时计时器后，那么 Server 端会重新发送 SYN+ACK 包，重传次数根据/proc/sys/net/ipv4/tcp_synack_retries 来指定，默认是 5 次。如果重传指定次数到了后，仍然未收到 ACK 应答，那么一段时间后，Server 自动关闭这个连接。但是 Client 认为这个连接已经建立，如果 Client 端向 Server 写数据，Server 端将以 RST 包响应，方能感知到 Server 的错误。

在 S 返回一个确认的 SYN-ACK 包的时候，S 可能由于各种原因不会接到 C 回应的 ACK 包。这个也就是所谓的半开放连接，S 需要 耗费一定的数量的系统内存来等待这个未决的连接，虽然这个数量是受限，但是恶意者可以通过创建很多的半开放式连接来发动 SYN 洪水攻击 。攻击者可以通过 IP 欺骗发送 SYN 包给受害者系统，这个看起来是合法的，但事实上所谓的 C 根本不会进行 ACK 回应服务端 S 的 SYN-ACK 报文，这意味着受害者将永远不会接到 ACK 报文。 而此时，半开放连接将最终耗用受害者所有的系统资源（即使等待 ACK 包有超时限制），受害者将不能再接收任何其他的请求。

如何应对 TCP SYN Flood

第一个参数 tcp_synack_retries = 0 是关键，表示回应第二个握手包（SYN+ACK 包）给客户端 IP 后，如果收不到第三次握手包（ACK 包）后，不进行重试，加快回收“半连接”，不要耗光资源。
修改这个参数为 0 的副作用：网络状况很差时，如果对方没收到第二个握手包，可能连接服务器失败，但对于一般网站，用户刷新一次页面即可。这些可以在高峰期或网络状况不好时 tcpdump 抓包验证下。
之所以可以把 tcp_synack_retries 改为 0，因为客户端还有 tcp_syn_retries 参数，默认是 5，即使服务器端没有重发 SYN+ACK 包，客户端也会重发 SYN 握手包。

tcp_max_syn_backlog
从字面上就可以推断出是什么意思。在内核里有个队列用来存放还没有确认 ACK 的客户端请求，当等待的请求数大于 tcp_max_syn_backlog 时，后面的会被丢弃。

所以，适当增大这个值，可以在压力大的时候提高握手的成功率。手册里推荐大于 1024。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。
当半连接的请求数量超过了 tcp_max_syn_backlog 时，内核就会启用 SYN cookie 机制，不再把半连接请求放到队列里，而是用 SYN cookie 来检验。

启用 3
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。当然，前文的高级攻击中有 SYN 混合 ACK 的攻击方法，则是对此种防御方法的反击，其中优劣由双方的硬件配置决定

客户端收到一个窗口为 0 的包怎么处理

TCP 长连接与短连接

TCP 短连接的情况，client 向 server 发起连接请求，server 接到请求，然后双方建立连接。client 向 server 发送消息，server 回应 client，然后一次读写就完成了，这时候双方任何一个都可以发起 close 操作，不过一般都是 client 先发起 close 操作。为什么呢，一般的 server 不会回复完 client 后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作
client 向 server 发起连接，server 接受 client 连接，双方建立连接。Client 与 server 完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

为什么要有 time_wait

1.可靠的终止 TCP 连接。
2.保证让迟来的 TCP 报文段有足够的时间被识别并丢弃。

1.可靠的终止 TCP 连接，若处于 time_wait 的 client 发送给 server 确认报文段丢失的话，server 将在此又一次发送 FIN 报文段，那么 client 必须处于一个可接收的状态就是 time_wait 而不是 close 状态。
2.**保证迟来的 TCP 报文段有足够的时间被识别并丢弃**，linux 中一个 TCPport 不能打开两次或两次以上。当 client 处于 time_wait 状态时我们将无法使用此 port 建立新连接，假设不存在 time_wait 状态，新连接可能会收到旧连接的数据。

1）可靠地实现 TCP 全双工连接的终止

TCP 协议在关闭连接的四次握手过程中，最终的 ACK 是由主动关闭连接的一端（后面统称 A 端）发出的，如果这个 ACK 丢失，对方（后面统称 B 端）将重发出最终的 FIN，因此 A 端必须维护状态信息（TIME_WAIT）允许它重发最终的 ACK。**如果 A 端不维持 TIME_WAIT 状态，而是处于 CLOSED 状态，那么 A 端将响应 RST 报文，B 端收到后将此报文解释成一个错误（在 java 中会抛出 connection reset 的 SocketException)。**

因而，要实现 TCP 全双工连接的正常终止，必须处理终止过程中四个报文任何一个报文的丢失情况，主动关闭连接的 A 端必须维持 TIME_WAIT 状态 。

2）允许老的重复报文在网络中消逝

TCP 报文可能由于路由器异常而“迷途”，在迷途期间，TCP 发送端可能因确认超时而重发这个报文，迷途的报文在路由器修复后也会被送到最终目的地，这个迟到的迷途报文到达时可能会引起问题。**在关闭“前一个连接”之后，马上又重新建立起一个相同的 IP 和端口之间的“新连接”**，“前一个连接”的迷途重复分组在“前一个连接”终止后到达，而被“新连接”收到了。为了避免这个情况，**TCP 协议不允许处于 TIME_WAIT 状态的连接启动一个新的可用连接**，因为 TIME_WAIT 状态持续 2MSL，就可以保证当成功建立一个新 TCP 连接的时候，来自旧连接重复分组已经在网络中消逝。

客户端和服务器都可能会进入 time_wait

高并发 TCP 服务器中进行主动关闭的一方最好是客户端：因为对于高并发服务器来说文件描述符资源是很重要的资源，如果对于每一个连接都要经历 TIME_WAIT 这个 2MSL 的时长，势必造成资源不能立马复用的浪费。虽然对于客户端来说 TIME_WAIT 状态会占用端口和句柄资源，但是客户端一般很少有并发资源限制，所以客户端执行主动关闭是比较合适的。

TIME_WAIT 状态到底会占用什么？

被占用的是一个五元组：（协议，本地 IP，本地端口，远程 IP，远程端口）。对于 Web 服

务器，协议是 TCP，本地 IP 通常也只有一个，本地端口默认的 80 或者 443。只剩下远程 IP 和远程端口可以变了。如果远程 IP 是相同的话，就只有远程端口可以变了。这个只有几万个，所以当同一客户端向服务器建立了大量连接之后，会耗尽可用的五元组导致问题。

客户端断开连接造成 time_wait 影响

客户端：
客户端与服务端进行短连接的 TCP 通信，如果在同一台机器上进行压力测试模拟上万的客户请求，并且循环与服务端进行短连接通信，那么这台机器将产生 4000 个左右的 TIME_WAIT socket，后续的短连接就会产生 address already in use : connect 的异常。
如果是客户端发起了连接，传输完数据然后主动关闭了连接，这时这个连接在客户端就会处于 TIMEWAIT 状态，同时占用了一个本地端口。如果客户端使用短连接请求服务端的资源或者服务，客户端上将有大量的连接处于 TIMEWAIT 状态，占用大量的本地端口。最坏的情况就是，本地端口都被用光了，这时将无法再建立新的连接。

客户端断开连接造成 time_wait 解决

客户端：
1. 使用长连接，如果是 http，可以使用 keepalive
2. 增加本地端口可用的范围，比如 Linux 中调整内核参数：net.ipv4.ip_local_port_range
3. tcp_tw_reuse 参数用来设置是否可以在新的连接中重用 TIME_WAIT 状态的套接字。注意，重用的是 TIME_WAIT 套接字占用的端口号，而不是 TIME_WAIT 套接字的内存等。这个参数对客户端有意义，在主动发起连接的时候会在调用的 inet_hash_connect()中会检查是否可以重用 TIME_WAIT 状态的套接字。如果你在服务器段设置这个参数的话，则没有什么作用，因为服务器端 ESTABLISHED 状态的套接字和监听套接字的本地 IP、端口号是相同的，没有重用的概念。但并不是说服务器端就没有 TIME_WAIT 状态套接字。

服务器：
不像客户端有端口限制，处理大量 TIME_WAIT Linux 已经优化很好了，每个处于 TIME_WAIT 状态下连接内存消耗很少，

而且也能通过 tcp_max_tw_buckets = 262144 配置最大上限，现代机器一般也不缺这点内存。

tcp_timestamps 参数用来设置是否启用时间戳选项，tcp_tw_recycle 参数用来启用快速回收 TIME_WAIT 套接字。tcp_timestamps 参数会影响到 tcp_tw_recycle 参数的效果。如果没有时间戳选项的话，tcp_tw_recycle 参数无效

客户端收到 ConnectionReset

server 端主动发起了断连
导致“Connection reset”的原因是服务器端因为某种原因关闭了 Connection，而客户端依然在读写数据，此时服务器会返回复位标志“RST”，然后此时客户端就会提示“java.net.SocketException: Connection reset”。
服务器返回了“RST”时，如果此时客户端正在从 Socket 套接字的输出流中读数据则会提示 Connection reset”；

服务器返回了“RST”时，如果此时客户端正在往 Socket 套接字的输入流中写数据则会提示“Connection reset by peer”。

UDP 可靠传输

实现一个最基础的可靠 udp 通讯协议，我们只需要提供一个重传机制即可。在这我实现了一个简单的可靠 udp 协议，这个协议为每一个发送出去的 udp 数据包分配一个包 id，每次接收方收到一个数据包时，都要回应发送方一个 ack 对应这个包 id。协议通过这种确认机制来保证接收方能收到发送方发出的 udp 数据包，在发出的时候，发送方应该设置一个计时器，超时的话会重传数据包。

具体来说它没做这些事情：

它没有保证包的有序性。发送方连续发送几个 udp 数据包，接收方可以以任何顺序收到这几个数据包。如果想要做到有序性，必须由应用层来完成。
它没做流量控制。发送方连续大量发送数据包会导致网络性能变差，丢包次数增大。
它没对数据包大小做控制。为了避免 IP 层对数据包进行分片，应用层应该要保证每个数据包的大小不超过 MTU。如果这个数据包会经过广域网（一般情况下）这个值应该不超过 576。考虑到 IP 头的 20 字节，udp 头的 8 个字节，以及这个协议头的字节。最好每次发送的数据大小在 512 以内。
技术面试中常见的网络通信细节问题解答
1. TCP/IP 协议栈层次结构
2. TCP 三次握手需要知道的细节点
3. TCP 四次挥手需要知道的细节点(CLOSE_WAIT、TIME_WAIT、MSL)
4. TCP 与 UDP 的区别与适用场景
5. linux 常见网络模型详解(select、poll 与 epoll)
6. epoll_event 结构中的 epoll_data_t 的 fd 与 ptr 的使用场景
7. Windows 常见的网络模型详解(select、WSAEventSelect、WSAAsyncSelect)
8. Windows 上的完成端口模型(IOCP)
9. 异步的 connect 函数如何编写
10.select 函数可以检测网络异常吗？
11.你问我答环节一
12. epoll 的水平模式和边缘模式

13. 如何将 socket 设置成非阻塞的(创建时设置与创建完成后设置)，非阻塞 socket 与阻塞的 socket 在收发数据上的区别
14. send/recv(read/write)返回值大于 0、等于 0、小于 0 的区别
15.如何编写正确的收数据代码与发数据代码
16.发送数据缓冲区与接收数据缓冲区如何设计
17.socket 选项 SO_SNDTIMEO 和 SO_RCVTIMEO

socket 选项 TCP_NODELAY

在网络拥塞控制领域，有一个非常有名的算法叫做 Nagle 算法（Nagle algorithm），这是使用它的发明人 John Nagle 的名字来命名的，John Nagle 在 1984 年首次用这个算法来尝试解决福特汽车公司的网络拥塞问题（RFC 896）。

该问题的具体描述是：如果我们的应用程序一次产生 1 个字节的数据，而这个 1 个字节数据又以网络数据包的形式发送到远端服务器，那么就很容易导致网络由于太多的数据包而过载。比如，当用户使用 Telnet 连接到远程服务器时，每一次击键操作就会产生 1 个字节数据，进而发送出去一个数据包，所以，在典型情况下，传送一个只拥有 1 个字节有效数据的数据包，却要发费 40 个字节长包头（即 ip 头 20 字节+tcp 头 20 字节）的额外开销，这种有效载荷（payload）利用率极其低下的情况被统称之为愚蠢窗口症候群（Silly Window Syndrome）。可以看到，这种情况对于轻负载的网络来说，可能还可以接受，但是对于重负载的网络而言，就极有可能承载不了而轻易的发生拥塞瘫痪。

针对上面提到的这个状况，Nagle 算法的改进在于：如果发送端欲多次发送包含少量字符的数据包（一般情况下，后面统一称长度小于 MSS 的数据包为小包，与此相对，称长度等于 MSS 的数据包为大包，为了某些对比说明，还有中包，即长度比小包长，但又不足一个 MSS 的包），则发送端会先将第一个小包发送出去，而将后面到达的少量字符数据都缓存起来而不立即发送，直到收到接收端对前一个数据包报文段的 ACK 确认、或当前字符属于紧急数据，或者积攒到了一定数量的数据（比如缓存的字符数据已经达到数据包报文段的最大长度）等多种情况才将其组成一个较大的数据包发送出去。

设置 NODELAY 会立即发送，不会延迟。

19.socket 选项 SO_REUSEADDR 和 SO_REUSEPORT（Windows 平台与 linux 平台的区别）
20.socket 选项 SO_LINGER
21.shutdown 与优雅关闭
22.你问我答环节二
23.socket 选项 SO_KEEPALIVE
24.关于错误码 EINTR

如何解决 tcp 粘包问题

面向流的协议

1）数据包固定大小，每收到该大小字节视为一个包
2）分隔符，比如\r\n
3）自定义数据包，header 中指定 body 的长度（最常使用）

26.信号 SIGPIPE 与 EPIPE 错误码
27.gethostbyname 阻塞与错误码获取问题
28.心跳包的设计技巧（保活心跳包与业务心跳包）

断线重连机制如何设计

客户端与服务器断开，客户端重连
分布式中下流服务与上流服务重连

1）每隔多久重连一次，间隔时间逐渐增长
2）监听网络变化，网络变化时立即重连

30.如何检测对端已经关闭

如何清除无效的死链（端与端之间的线路故障）

比如四次挥手时的 TIME_WAIT
比如心跳超时
定时器！
数据结构：小顶堆/红黑树

32.定时器的不同实现及优缺点
33.你问我答环节三
34.http 协议的具体格式
35.http head、get 与 post 方法的细节
36.http 代理、socks4 代理与 socks5 代理如何编码实现
37.ping
38.telnet
39.你问我答环节四
40.总结