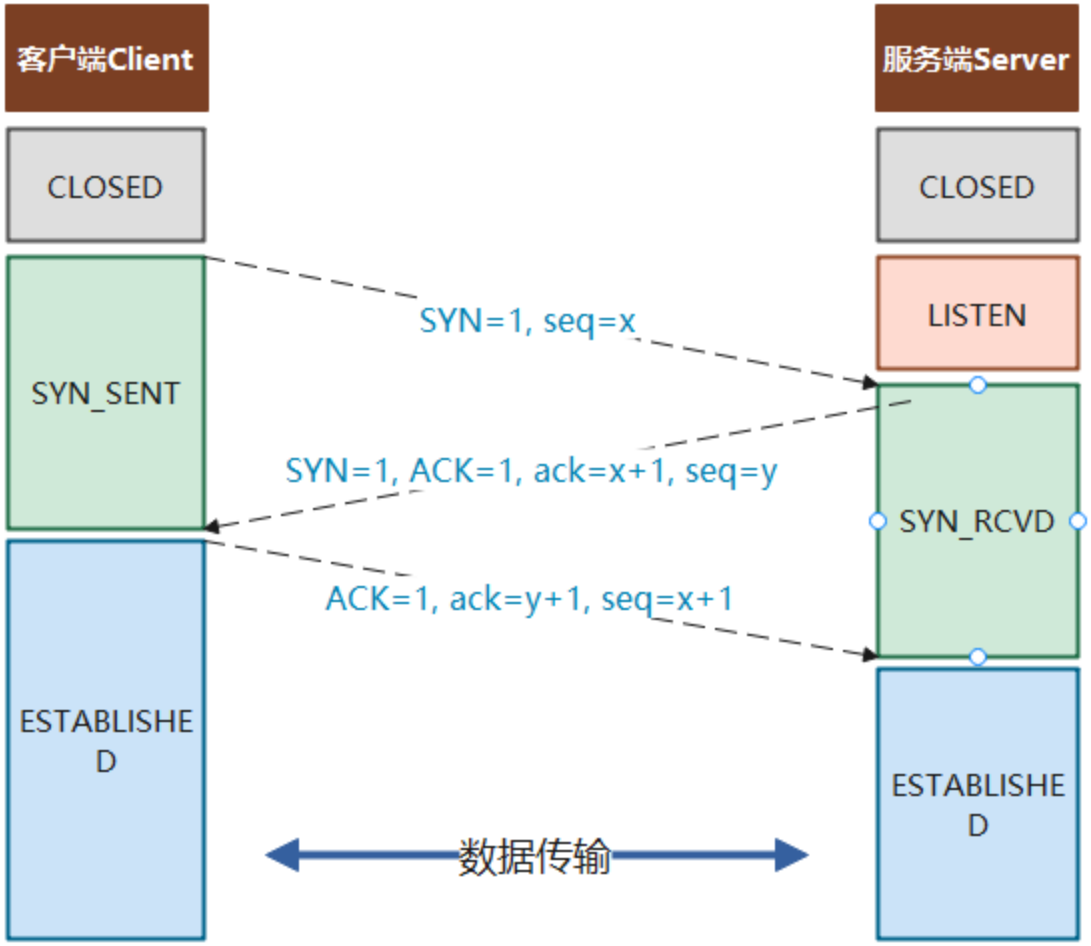
计算机网络

1. TCP 和 UDP 的区别

特性	TCP	UDP
可靠性	可靠（保证顺序，不丢包）	不可靠
连接性	面向连接	无连接
报文	面向字节流	面向报文
传输效率和速度	传输效率低，速度慢	效率高，速度快
双工性	全双工（通讯传输的术语，通信允许数据在两个方向上同时传输）	一对一、一对多、多对一、多对多
流量控制	滑动窗口	无
拥塞控制	慢开始、拥塞避免、快重传、快恢复	无
应用场景	对效率要求低，准确性要求高的场景	对效率要求高，准确性要求低
实用举例	FTP文件传输、HTTP万维网、SMTP电子邮件	DNS域名转换、SNMP网络管理、TFTP文件传输

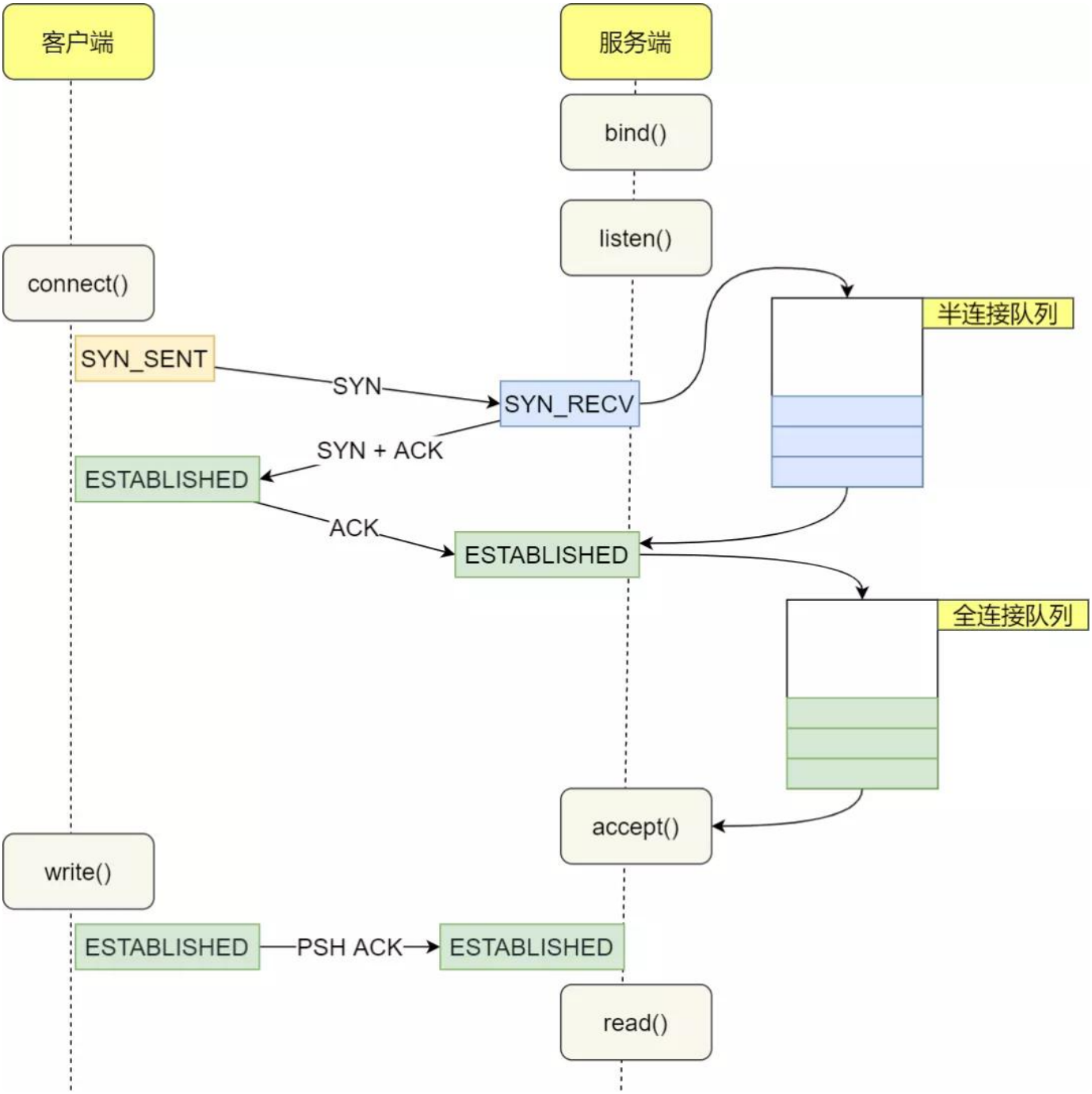
2. TCP三次握手，四次挥手

1）TCP 三次握手



- 刚开始，客户端（Client）和服务器（Server）都处于 CLOSED 状态；
- 服务端创建传输控制块（TCB），时刻准备客户进程的连接请求，处于 LISTEN 监听状态；
- **第一次握手：**客户端将 TCP 报文的**标志位 SYN 置为1**，随机产生一个序号值 seq=x 保存在 TCP 首部的序列号字段里，然后指明客户端打算连接的服务器端口，并将数据包发送给服务器端。发送完毕后，客户端进入 SYN-SEND 状态；
- **第二次握手：**服务端收到数据包后，由标志位 SYN=1 知道了客户端请求建立连接，服务端将 TCP 报文的**标志位 SYN 和确认应答号 ACK 都置为 1**，请求号 ack = x+1，再随机产生一个序号值 seq=y，并将该数据包发送给客户端以确认连接请求，服务端进入 SYN-RCVD 状态；
- **第三次握手：**客户端收到服务端的确认后，检查 ack 是否为 x+1，ACK 是否为1，如果正确则将确认应答 ACK 置为 1，请求号 ack=y+1，并将数据包发送给服务器。**服务器端检查 ack 是否为 y+1，ACK 是否为 1，如果正确则成功建立连接**。客服端和服务器都进入 ESTABLISHED 状态，三次握手结束，客户端和服务器可以开始传输数据了。

在上述过程中，还有一些重要的概念：



- 半连接：收到 SYN 包而还未收到 ACK 包时的连接状态称为半连接，即尚未完全完成三次握手的 TCP 连接，处于第一次握手之后，第三次握手之前。
- 半连接队列：在三次握手协议中，服务器维护一个半连接队列，该队列为每个客户端的 SYN 包开设一个条目，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。

1.1 泛洪攻击（SYN Flood）

由于半连接队列有大小限制，不法分子在客户端短时间内伪造大量不存在的 IP 地址，向服务端发送连接请求，会产生两个后果：

- 导致半连接队列充斥着大量无效的连接请求。这时有新的正常请求到达服务器，可能会被丢弃或者返回 RST 重置连接报文；
- 由于是不存在的 IP，服务端长时间没有收到客户端响应，会不断超时重传数据，直到耗尽服务器的资源。

1.2 应对攻击

1. 增加 SYN 半连接队列的容量，但这不是长久办法；
2. 减少 SYN+ACK 重试次数，避免大量的**超时重发**；
3. 利用 SYN Cookie 技术，服务端收到 SYN 连接请求后不立即分配连接资源，而是根据这个连接计算出一个 Cookie，连同第二次握手回复给客户端。客户端三次握手的时候带上 Cookie 请求，服务端验证完 Cookie 以后再分配连接资源。

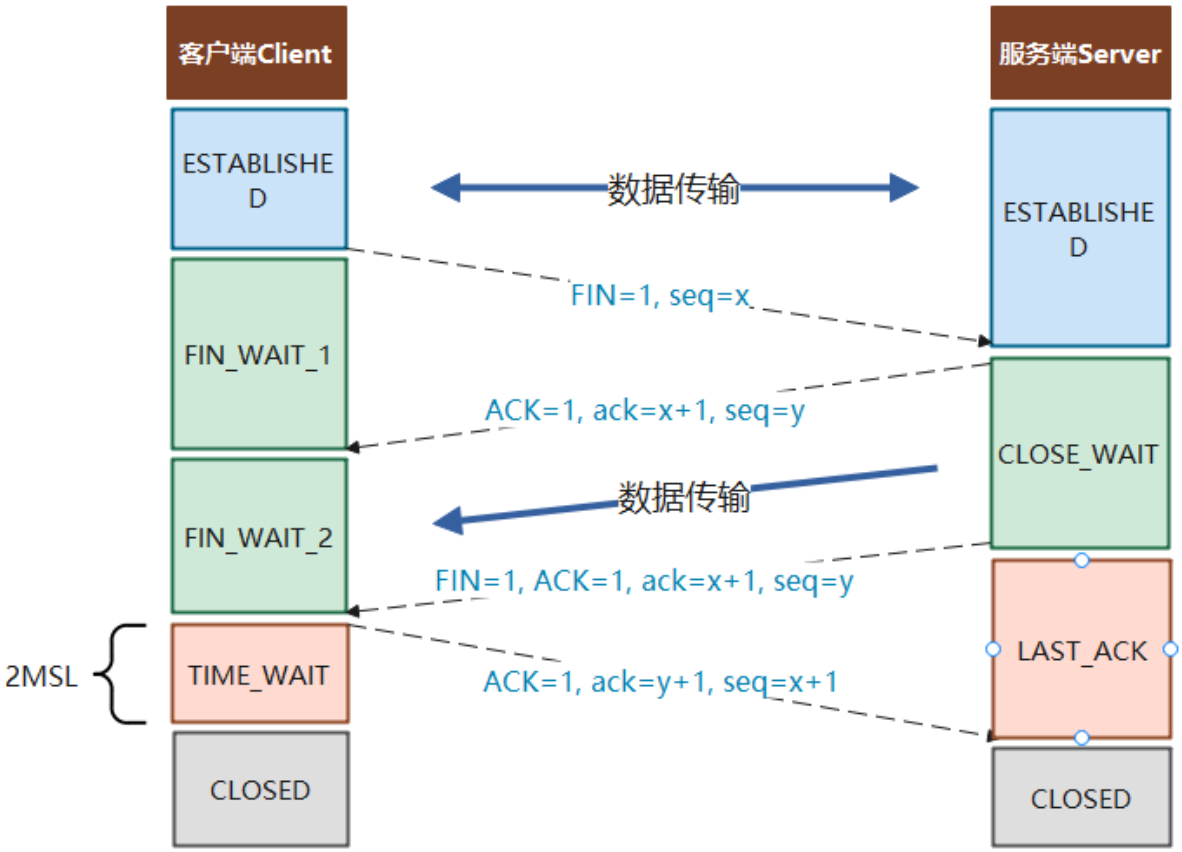
2）为什么三次握手

防止已失效的连接请求又传送到服务器端，比如以下场景：

- 当客户端发出第一次请求连接时，由于网络节点拥堵导致服务端未收到请求的报文，这时，由于服务端没有响应，客户端可能会发送多次连接请求；
- 如果没有三次握手的确认，当之前传了很久的请求到达服务端时，服务器会认为这是一个新的请求，于是建立连接，但是这个 TCP 连接一直不会通信，这样，服务端的很多资源就被白白浪费掉了。

因此，采用三次握手建立连接可以防止上述问题的出现，当客户端收到一个已失效的建立连接确认报文时，不会向服务端进行第三次握手确认。而服务端由于收不到确认，就不会和客户端建立连接了。

3）TCP 四次挥手



当客户端（Client，以下简称C端）和服务器（Server，以下简称S端）都是连接状态时：

- **第一次挥手：**C 端不想再进行数据传输了，就发起一条挥手请求，将 TCP 报文的标志位置为 FIN，设置序列号 seq 为随机数 x。此时，C 端进入 FIN_WAIT_1 状态；
- **第二次挥手：**S 端收到 C 端的 FIN 数据报，知道 C 端不再发送数据了。于是返回一条 ACK 确认消息，表示同意 C 端的关闭请求，然后 S 端进入 CLOSE_WAIT 状态。当 C 端收到 S 端的确认消息后，进入 FIN_WAIT_2 状态，等待 S 端的连接结束；
- **第三次挥手：**S 端发送数据完毕后，给 C 端发送标志位为 FIN 的报文段，请求关闭连接，并进入 LAST_ACK 状态，随机序列号为 z（图中为 y 不合适）；
- **第四次挥手：**当 C 端收到 FIN 报文段之后，再向 S 端回复标志位为 ACK 的应答消息，然后进入 TIME_WAIT 状态，当在等待 2MSL 时还没收到回复，证明 S 端已经正常关闭，于是 C 端进入CLOSED 状态。而 S 端在收到 C 端的 ACK 报文段以后，就关闭连接，直接进入 CLOSED 状态。

4）为什么四次挥手

建立连接时当 Server 收到 Client 端的 SYN 连接请求时，可以直接发送带有同步标志位 SYN 和确认应答号 ACK 的报文，所以建立连接只需要三次握手。

由于 TCP 是全双工模式，这就意味着关闭连接时，当 C 端发出 FIN 报文段时，只是表示 C 端的数据已经发送完毕了，但 S 端还是可以发送数据到 C 端的。因此，S 端很可能不会立即关闭连接，直到数据发送完毕后就需要另外发送报文段通知。

5）为什么要等待 2MSL

为什么要等待 2MSL：Max Segment Lifetime，指报文的最大存活时间，它是任何报文段被丢弃前在网络内的最长时间。

- **保证 TCP 的全双工连接能够可靠关闭：**由于 IP 协议的不可靠性或者其它网络原因，导致 S 端没有收到 C 端的 ACK 报文，那么 S 端就会在超时后重新发送 FIN，如果此时 C 端的连接已经关闭处于 CLOSED 状态，那么重发的 FIN 就找不到对应的连接了，从而导致连接错乱。因此，C 端发送完最后的 ACK 不能直接进入 CLOSED 状态，而要保持 TIME_WAIT，等待可能重传的 FIN 报文，保证对方能收到 ACK。
- **保证此处连接的重复数据段从网络中消失：**如果 C 端发送最后的 ACK 后直接进入 CLOSED 状态，然后再向 S 端发起一个新连接，这时无法保证新连接与刚关闭连接的端口号是不同的，就可能出现问题：如果前一次连接的某些数据滞留在网络中，这些延迟数据在建立新连接后到达 C 端，由于新老接口的端口号和 IP 都一样，TCP 协议就认为延迟数据是属于新连接的，新连接就会收到脏数据，导致数据包混乱。所以，TCP 连接需要在 TIME_WAIT 状态等待 2 倍 MSL，保证本次连接的所有数据在网络中消失。

6）TIME_WAIT 状态的影响

在 TCP 连接中，由于【主动发起关闭连接】的一端会进入 time_wait 状态，而此状态默认会持续 2MSL（两倍报文的最大存活时间，一般是 2*2 min），当大量的 TIME_WAIT 连接出现时，可能会造成：

- TCP 连接占用端口，无法被使用；
- TCP 端口数量过大，超过 65535（TCP 端口号为16bit，因此上限是 2^16），导致新建 TCP 连接时失败。

服务器端的解决办法有两个：

- 允许 time_wait 状态的 socket 被重用；
- 缩减 time_wait 的时间，设置为 1MSL。

3. 如何用UDP实现可靠传输

UDP 是无连接的协议，具有资源消耗少，处理速度快的优点。所以通常音频、视频和普通数据在传送时，使用 UDP 较多。因为即使丢失少量的包，也不会对接收结果产生较大的影响。

UDP 的传输层无法保证数据的可靠传输，只能通过应用层来实现了。最简单的方式是在应用层模仿 TCP 传输层的可靠性机制，比如不考虑拥塞处理时，可靠 UDP 可以这么设计：

- 添加 ack/seq 机制，确保数据发送到对端；
- 添加发送和接收缓冲区，提升传输效率；
- 添加超时重传机制。

当 C 端发送数据时，生成一个随机序列号 seq=x，然后每一片按照数据大小分配 seq。数据到达对端后放入缓存，并发送一个 ack=x+1 的包，表示已经收到了数据。

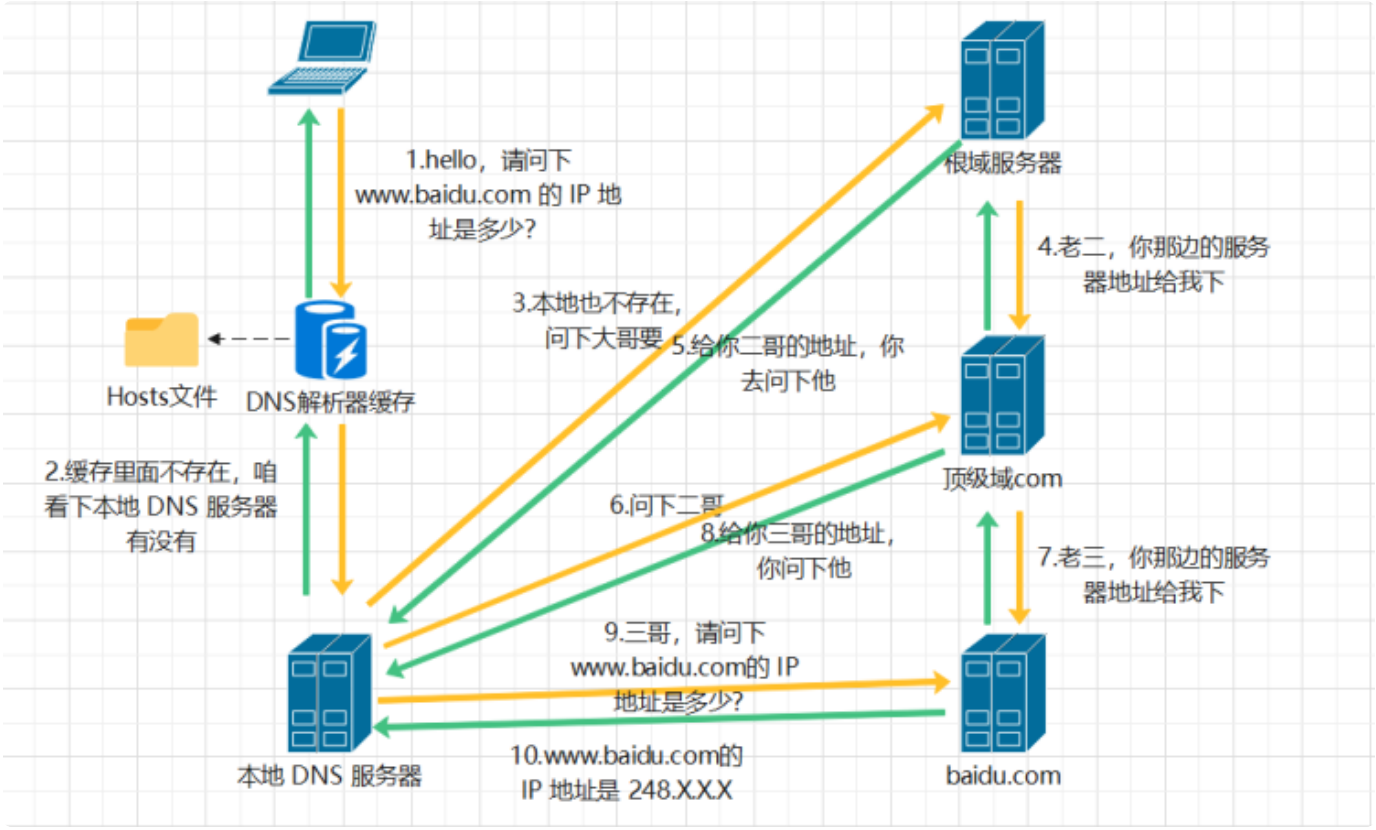
C 端收到 ack 包后，判断序号是否正确。并且 C 端在等待期间，定时任务检查是否需要超时重传。

目前，有 RUDP、RTP 等开源程序实现了 UDP 的可靠数据传输。

4. 浏览器输入公司网址，会发生什么？用到了哪些协议?

客户端获取URL – > DNS解析 – > TCP连接 – >发送HTTP请求 – >服务器处理请求 – >返回报文 – >浏览器解析渲染页面 – > TCP断开连接

1）DNS 解析



DNS 域名解析协议，负责将网址域名转换成唯一的 IP 地址，通过的路径有：DNS 解析器缓存、本地 DNS 服务器、根域、顶级域名服务器，最终获取到网址域名的 IP 地址，并缓存到本地 Hosts 文件中。

2）TCP 连接，发送HTTP 请求，服务器处理，返回报文

获取到目标主机的 IP 地址后，如果 IP 里不包含端口号，浏览器会选择一个大于 1024 的端口与目标 IP 的 80 端口发起 TCP 请求，经过 TCP 三次握手，建立 TCP 连接。

1. 客户端：

- 应用层：获取 URL，通过域名解析协议 DNS 获取 IP 地址，根据 HTTP 协议生成 HTTP 请求报文；
- 传输层：三次握手，1）客户端给服务端发送一个带 SYN （同步）标志的数据包；2）服务端回传一个带 SYN（同步）标志和 ACK （确认）标志的数据包给客户端，传达确认信息；3）客户端再传送一个带有 ACK （确认）标志的数据包给服务端，代表握手结束。连接成功后，TCP 协议再把请求报文按序号分割成多个报文段；
- 网络层：IP 协议传输数据，ARP 协议获取 MAC 地址，OSPF 协议选择最优路径，搜索服务器地址，一边中转一边传输数据；
- 数据链路层：物理层负责 0,1 比特流与物理设备电平转换，数据链路层将 0,1 序列划分为数据帧，从一个节点传输到临近节点。

2. 服务端：

通过数据链路层 ->网络层 ->传输层（TCP 协议接收请求报文，重组报文段）->应用层（HTTP 协议对请求的内容进行处理）->应用层 ->传输层 ->网络层 ->数据链路层 ->到达客户端。

3. 客户端：

接收数据：数据链路层 ->.....-> 应用层 ->浏览器渲染页面 -> 断开连接协议四次挥手。

由于 HTTP 是无状态的，一般情况下，客户端在收到服务器的响应后就会直接断开连接，然后一次 HTTP 请求就结束了。但是 HTTP1.0 有一个 keep-alive 请求字段，可以在一段时间内不断开连接（HTTP1.1 直接默认开启了 keep-alive 选项，导致服务器资源一直被占用）。这时，服务器不得不主动断开连接，而主动断开连接的一方会出现 TIME_WAIT，占用连接池，这就是 SYN-FLOOD 攻击的原因。

3）浏览器渲染页面

客户端接收响应后，根据 accept 报文头，解析接收到的数据类型。

4）断开 TCP 连接

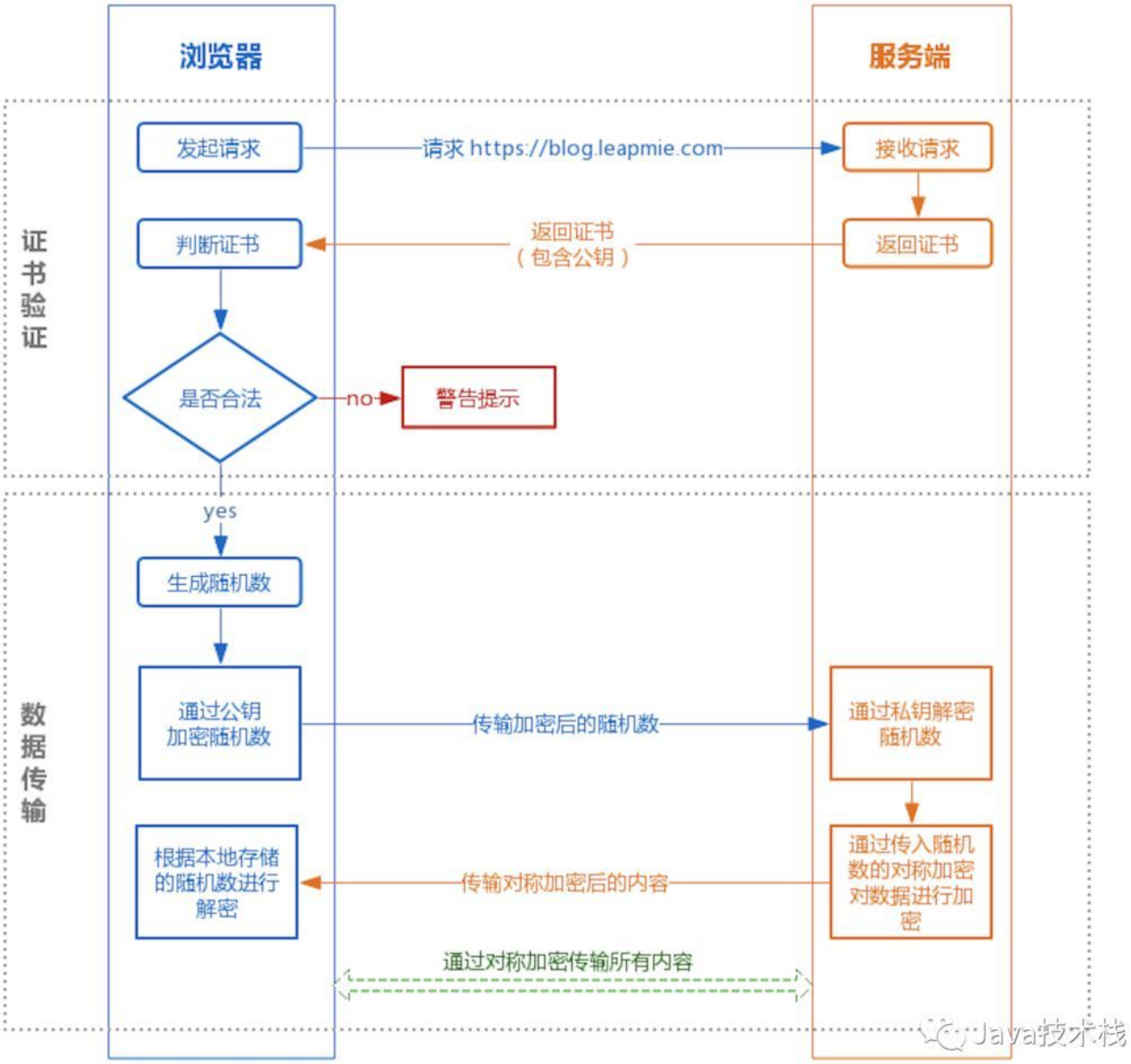
TCP 四次挥手断开连接

5. HTTP和HTTPS区别

1）HTTPS原理

前者问题在于明文通道，无法验证对方的身份及无法验证报文的完整性。故，HTTPs = HTTP + 数据加密 + 身份认证 + 完整性保护。改进：HTTP 协议中的部分通信接口被 ssl/tls 协议代替，通常，HTTP 直接与 TCP 进行通信，而使用了 ssl 协议后，则先与 ssl 通信，再由 ssl 和 TCP 通信。

2）加密过程



2.1 证书验证过程

1）浏览器发起 HTTPs 请求；

2）服务端生成一对公私钥，私钥由服务端自己存储；公钥放在 HTTPs 证书里返回给客户端， 证书内容还包含网站地址、证书颁发机构、失效日期等；

3）客户端验证证书的合法性（比如证书中的网址和当前网址是否一致，证书是否过期），若不合法则提示告警；

2.2 数据传输阶段

1）当证书验证合法后，客户端生成一个随机数作为对称算法的密钥；

2）客户端通过公钥加密随机数，并传送到服务端；

3）服务端接收到加密后的随机数以后，通过自己的私钥对随机数进行解密；

4）服务端拿到对称密钥（随机数）以后，对返回的结果数据进行对称加密。

3）为什么传输用对称加密，证书验证阶段使用非对称加密?

1）非对称加解密的效率低，传输数据过程中端到端可能有大量的交互，会影响传输的效率；

2）另外，在 HTTPs 场景下只有服务端保存了私钥，而一对公私钥只能实现单向的加解密，因此 HTTPs 采用对称加密传输；

6. HTTP响应码有哪些

- 301 Moved Permanently，永久重定向，今后任何新的请求都应使用新的URI代替；
- 302 Found，临时重定向，资源只是临时被移动，客户端应继续使用原有URI；
- 304 Not Modified // 未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源
- 400 Bad Request // 客户端请求有语法错误，不能被服务器所理解
- 401 Unauthorized // 当前请求要求用户的身份认证
- 403 Forbidden // 服务器成功解析请求，但是客户端没有访问该资源的权限
- 404 Not Found // 请求资源不存在，输入了错误的URL
- 405 Method Not Allowed // 方法禁用。客户端请求的方法被禁止，比如 POST 接口用 GET 请求
- 406 Not Acceptable // 无法响应。请求资源的内容特性无法满足请求头中的条件，因而无法生成响应实体，比如客户端请求头设置为：Accept: application/xml，和服务端所接受的 Accept 字段不同。
- 500 Internal Server Error // 服务器发生不可预期的错误
- 502 Bad Gateway // 网关错误，服务器作为网关或代理，从上游服务器收到无效响应
- 503 Server Unavailable // 由于超载或系统维护，服务器暂时无法处理客户端的请求，一段时间后可能恢复正常。

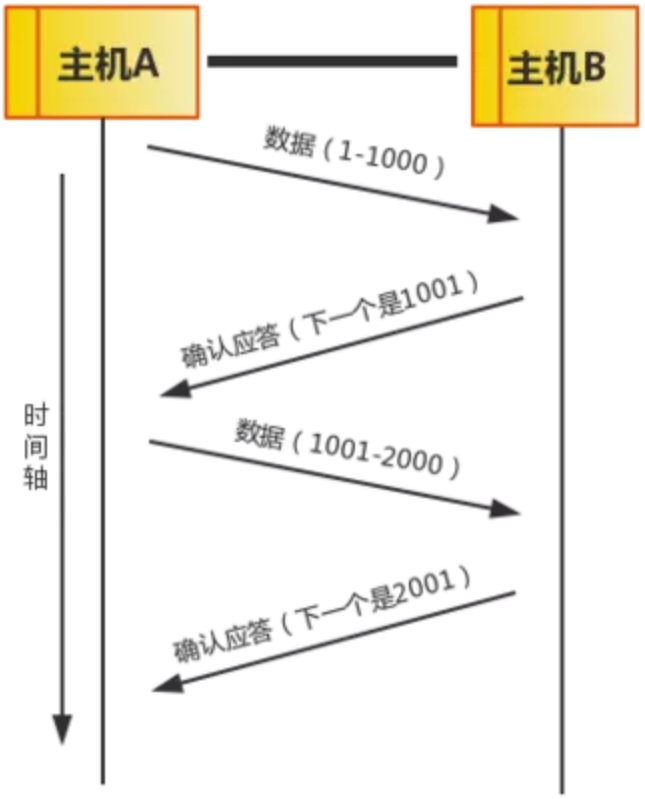
7. TCP 怎么保障可靠性

TCP协议在实现传输可靠性上面做了很多：

- 通过**序列号和确认应答信号**确保了数据不会重复发送和重复接收
- 同时通过**超时重发控制**保证即使数据包在传输过程中丢失，也能重发保持数据完整
- 通过三次握手，四次挥手建立和关闭连接的**连接管理**保证了端对端的通信可靠性
- TCP还使用了**流量控制（滑动窗口）及拥塞控制**提高了数据传输效率，保证传输过程中数据不丢失

1）序列号和确认应答信号

TCP 传输中，当发送端的数据到达接收主机时，接收端主机会返回一个已收到消息的通知，这个消息叫做**确认应答（ACK）**。



当数据从主机A发送到主机B时，主机B会返回给主机A一个确认应答。

当发送端发出数据以后，就会等待接收端的确认应答。收到应答后，就表示数据已经成功到达对端，但如果在一定时间内发送端都没有收到确认应答 ACK，发送端就会认为数据丢失，并进行**数据重发**。所以，即便产生了丢包，TCP 仍然能够保证数据到达对端，实现可靠传输。

除了确认应答，TCP 还通过**序列号**机制来保证接收端不会消费同样的数据包。

具体流程是，当数据包已经正常发送到接收端后，由于网络阻塞等原因，发送端没有收到接收端的应答信号。于是，发送端进行数据重发，这时接收端会根据数据包的序列号来判断数据包是否已经消费过了。如果数据包重复，则丢弃；否则，接收这个新的数据包。

通过序列号和确认应答号，TCP 能够识别是否已经接收数据，也可以判断是否需要接收，从而实现可靠传输。

2）超时重传机制

重发超时是指在重发数据之前，等待确认应答的那个时间间隔。如果超过这个时间仍未收到应答，发送端将进行数据重发。

TCP 要求不论在何种网络环境下都要提供高性能通信，并且无论网络的拥堵情况发生何种变化，都必须保持这一特性。为此，它会在每次发包时计算往返时间（RTT，Round Trip Time）以及偏差（RTT波动的时间，也叫抖动）。**将这个往返时间和抖动时间相加，每次的重发超时时间就比这个总和稍大一点**。

3）连接管理

TCP 是面向连接的通信协议，在连接前，会通过 TCP 首部发送一个 SYN 包作为建立连接的标识，如果对端返回确认应答 ACK，则认为可以通信。

另外，通信完毕后需要发送 FIN 包来关闭连接，它的过程就是我们常说的**三次握手建立连接和四次挥手关闭连接。**

4）滑动窗口机制

建立 TCP 连接的时候，可以确认发送数据包的单位，就是**最大消息长度**（MSS，Max Segment Size），也就是一个段。TCP 三次握手的时候，会在两端主机之间计算得出这个值，并写入 TCP 首部告诉对方，然后两端在数据传输时会选取一个较小的 MSS 值使用。

但以 MSS 段进行数据传输有一个缺点，那就是当包的往返时间（RTT）变长时，通信的性能就会很低。类似于**无缓冲的阻塞等待**，数据段只能一个一个地处理，确认应答，然后再进行发送，没法

充分利用 TCP 通信双端的性能。

于是，TCP 引入了**窗口**的概念。**确认应答不再以每个分段来确认，而是以更大的单位进行确认**。这样，发送端就不必发送一个段就等待应答，而是可以继续发送数据。

窗口大小就是发送端无需等待确认应答 ACK，而可以继续发送数据的最大值。滑动窗口的实现，采用了缓冲机制，可以对多个段同时进行确认。

比如，当发送 100,101,102 时，如果收到了 101 的 ACK 应答，那么 102 之前的数据就没必要重发了。避免了发送端可能会因为网络阻塞等问题，重复发送对端已收到数据包。

并且，当网络出现丢包时，接收端的主机会一直发送某个序列号的 ACK，当发送端收到三次确认应答后，就会认为数据丢失了，并进行数据重发。这种机制比之前的超时重发更加高效，被称作**高速重发控制**。因此，滑动窗口的这些可靠性机制更充分利用了通信双端的效率，达到 TCP 高性能传输的目的。

8. TCP流量控制和拥塞控制

1）流量控制（滑动窗口）

若发送方数据发的过快，那接收方可能来不及接收，造成数据丢失。流量控制就是让发送方不要发太快，这是通过滑动窗口机制来实现的。

- ACK 是确认应答号，0/1；
- ack 是期望对方继续发送的数据序列号（请求序列号）；
- seq 是数据包本身的序列号，一般是随机数生成。
- 死锁避免：当 TCP 一方 A 收到 B 的零窗口通知时，会等到**超时计时器**到期时发送探测报文段。如果探测到 B 的窗口仍然是 0，就重置计时器；如果不再是 0，就开始发送数据，以免出现死锁的局面。

2）拥塞控制

如果流量控制是控制发送链路的速度，那么拥塞控制就是控制链路的车辆数。

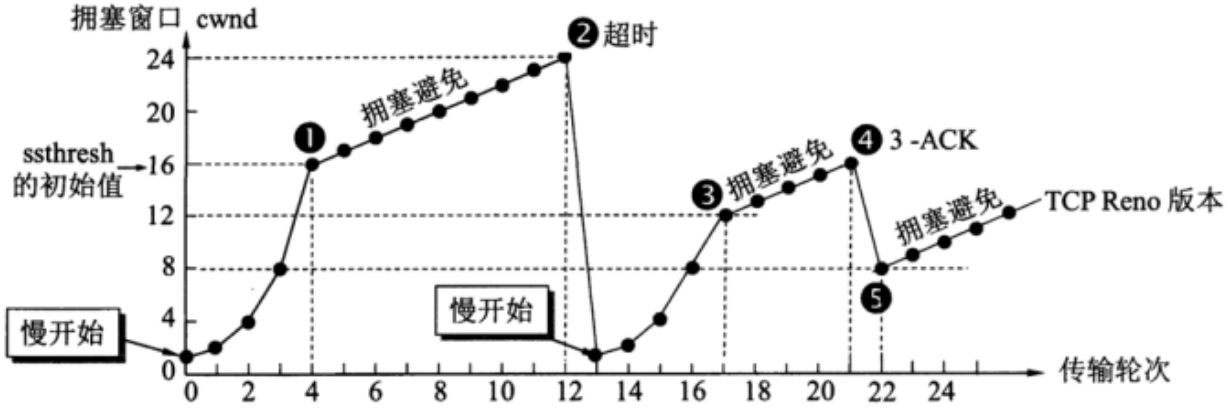


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

- 首先，TCP 窗口的实际单位是字节数，我们为了叙述方便，用报文段的个数来作为窗口大小的单位；
- **慢开始算法**：没到达传输阈值时，每经过一个传输轮次，传输的报文段就加倍（指数增长）；
- **拥塞避免算法**：到达传输阈值时，每经过一个传输轮次，传输的报文段加 1；
- **快恢复算法**：当发送方需要进行快重传时，TCP 会把拥塞阈值设置为最大窗口的一半（图中序号4），然后再把拥塞窗口设置为拥塞阈值的一半（也就是最大窗口的 1/4），开始执行拥塞避免算法。

什么是快重传

- 接收方：如果接收方收到一个失序的报文段，就立即回送一个 ACK 给发送方。比如收到了 m2, m4, m5 三条报文，说明乱序发生，那么需要对 m2 进行 3 次确认；
- 发送方：收到 3 个重复的 ACK 时，发送方就不必等待 m3 的重传计时器到期，而快速重传丢失的包，这便是快重传。

当不使用快重传时：如果发送方在超时计时器到期时，还未收到接收方的确认，那很可能是传输的报文段已经达到窗口最大值，需要把阈值设置为最大窗口值的一半。然后，拥塞窗口置为 1，重新执行慢开始算法。

使用快重传时：需要用快恢复算法，将拥塞阈值设置为最大拥塞窗口的一半（图中序号4），然后，再把拥塞窗口设置为拥塞阈值的一半（最大窗口的1/4），开始执行拥塞避免算法（图中序号5）。

9.GET和POST的区别

1) 请求时的区别：

操作	GET	POST
后退按钮/刷新	无害	数据会被重新提交
缓存	浏览器默认缓存	需手动设置，不会默认缓存
数据类型的限制	只允许 ASCII 字符	无限制，允许二进制数据
安全性	较差，发送的数据是 URL 的一部分	相对较好，数据放在 Body 中

2）其它区别：

- GET/POST 只是 HTTP 请求的两种方式，所以在传输上没有区别，因为 HTTP 是基于 TCP/IP 协议；
- 报文格式上，仅仅是第一行请求名不同。带参数时，一个在 URL，一个在 BODY；
- 从传输的角度，二者都不是安全的。POST 只是数据在地址栏不可见，HTTP 依旧是明文传输，HTTPS 才是安全的。

10. 加密与数据安全

1）对称，非对称加密

对称加密，即信息的双方用同一个密钥去加解密信息，使用了对称密码编码技术。由于算法公开，所以密钥不能对外公开。它的计算量小，加密速度快。缺点是不安全，密钥管理困难，如 AES，IDEA。

非对称加密，只能由成对的公私钥进行加解密，一般是公钥加密，私钥解密。过程：甲方生成一对密钥，并将其中一把作为公钥公开出去；乙方拿到公钥，对数据加密后发送给甲方，甲方用专用私钥进行解密。安全，但加密速度较慢，如 RSA 算法（RSA 支持私钥加密，公钥解密）。

混合加密，结合前两者的优缺点实现，将对称加密的公钥通过非对称加密传输（见 HTTPS 加密过程），如 TLS/SSL 算法。

2）公钥和私钥加密的区别

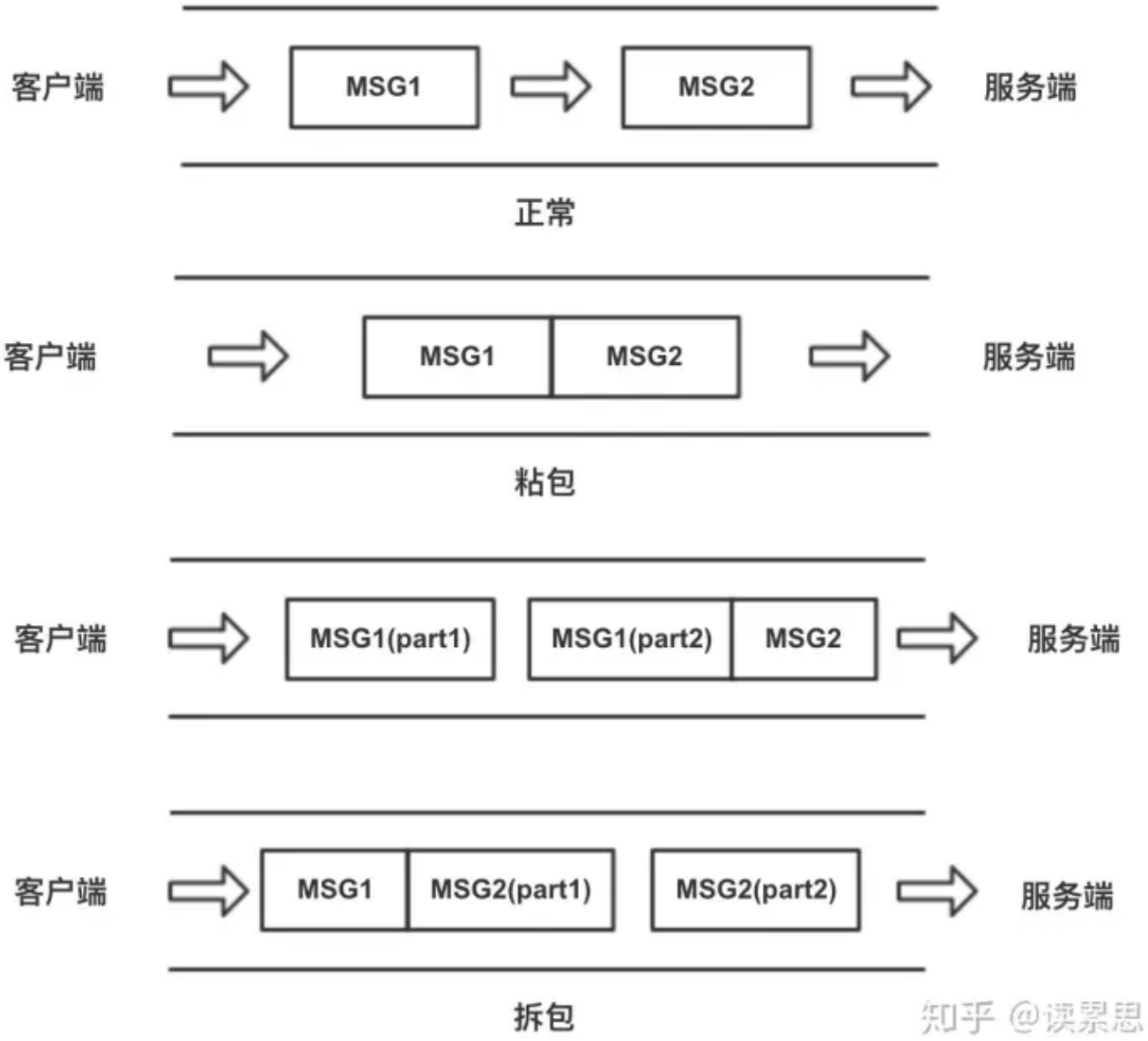
公钥加密，一般是为了保证数据的机密性；私钥加密，一般是为了保证数据的不可篡改性和不可抵赖性；

3）如何保证信息的机密性、不可抵赖性

比如甲向乙发送一段保密数据，那么甲乙各自需要有一个私钥且都拥有对方的公钥。首先使用甲方的私钥对摘要数据进行加密，再用乙方的公钥来加密这段数据，最后再发给乙。

11. TCP 粘包问题

TCP 粘包和拆包问题常常出现在基于 TCP 协议的通讯中，比如 RPC 框架、Netty 等。



粘（zhan）包是指 TCP 协议中，通讯的一端一次性连续发送多个小的数据包，TCP 会将这多个数据包打包成一个 TCP 报文发送出去。从接收缓冲区看，后一个包数据的头紧挨着前一个包数据的尾。

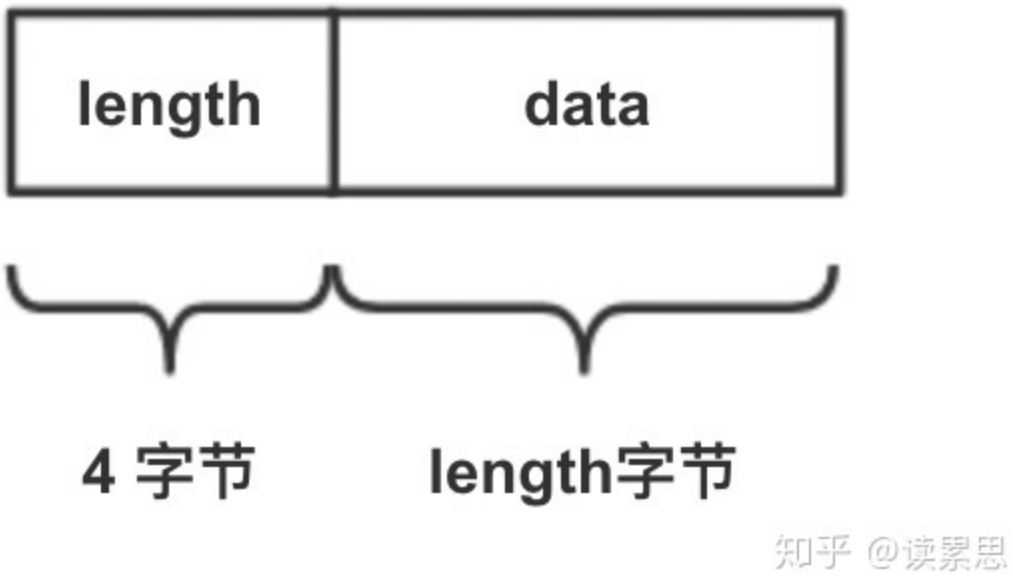
而通讯的一端发送数据包超过 TCP 报文一次传输的最大报文段长度（MSS，Max Segment Size）时，就会将一个数据包拆成多个 MSS 长度的 TCP 报文多次传输，这就是拆包。

粘包的原因有：

- 发送方要发送的数据包大小总和小于 MSS，将这多个数据包一次发送；
- 接收端的应用层**没有及时读取**接收缓冲区的数据；
- 数据发送过快，导致缓冲区堆积多个数据包后一次性发送（如果发送端每发送一条数据包就 Sleep 一段时间，就不会发生粘包）。

解决方案：

对于粘包的情况，要在接收端进行拆包；对于拆包的情况，要对其进行粘包。比较通用的做法就是每次发送的应用数据包前加上四个字节的包长度：



实现方式有两种：

- 引入 Netty 库，它封装了多种拆包粘包的方式，比如消息头包含消息长度的协议，只需要调用接口即可；
- 自己写协议封装和解析，相当于实现 netty 库拆包粘包的简易版本，比如每个包都封装成固定长度，或者在每个包的末尾使用固定的分隔符 \r\n 等。

TCP 是面向字节流的协议，没有消息边界，它本没有包的概念，粘包和拆包只是一种有助于形象理解的现象。由于 UDP 有消息边界，因此不会发生粘包现象。

12. TCP 连接中一端异常

一个 TCP 连接，没有打开 Keepalive 选项，没有数据交互。现在一端突然断电或者进程崩溃，两者有何区别?

问题中包括：

- 未开启 keepalive
- 一直没有数据交互

- 主机断电
- 进程崩溃

1）TCP keepalive

Keepalive 是 TCP 的保活机制，由客户端打开。一个时间段以后如果 TCP 双端没有消息交互，TCP 保活机制会开启：客户端每隔一段时间，发送一个探测报文，如果连续几个探测报文都没有响应，说明服务器端出现异常，TCP 连接就会断开。详细定义：

定义一个时间段，在这个时间段内，如果没有任何连接相关的活动，TCP 保活机制会开始作用，每隔一个时间间隔，发送一个探测报文，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

在 Linux 内核可以有对应的参数可以设置保活时间、保活探测的次数、保活探测的时间间隔，以下都为默认值：

```
net.ipv4.tcp_keepalive_time=7200
net.ipv4.tcp_keepalive_intvl=75
net.ipv4.tcp_keepalive_probes=9
```

- tcp_keepalive_time=7200：表示保活时间是 7200 秒（2小时），也就 2 小时内如果没有任何连接相关的活动，则会启动保活机制
- tcp_keepalive_intvl=75：表示每次检测间隔 75 秒；
- tcp_keepalive_probes=9：表示检测 9 次无响应，认为对方是不可达的，从而中断本次的连接。

也就是说在 Linux 系统中，最少需要经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接。

```
tcp_keepalive_time + (tcp_keepalive_intvl * tcp_keepalive_probes)
```



```
7200 + (75 * 9) = 7875 秒（2 小时 11 分 15 秒）
```

不过，考虑到检测时间太长，且缩短时间又很难体现设计初衷。现状是大部分应用并没有默认开启 TCP 的 keep–alive 选项。

2）一端主机崩溃/断电

服务端主机崩溃/断电时，TCP 连接可以通过客户端的 keepalive 保活机制来探测对方是否存活；

客户端主机崩溃/断电时，由于服务端没有开启 TCP keepalive，根据有无数据交互来决定接下来的状态：

- 若没有数据交互的情况下，服务端的 TCP 连接会一直处于 ESTABLISHED 连接状态，直到服务器重启进程；
- 若有数据交互，服务端向客户端发送的数据段得不到响应，一定时长后，服务端触发**超时重传**机制，重传未响应的数据段：
  - 服务端重传报文且一直未收到对方响应情况下，如果达到最大重传次数或者最大超时时间后，就会停止重传；
  - 如果这时候客户端重启了，客户端会发送 RST（Reset the connection）报文，重置该 TCP 连接。

3）一端主机宕机重启

当客户端宕机后，服务端向客户端发送的报文得不到响应，一定时长后，服务端触发**超时重传**机制，重传未响应的报文。

服务端重传报文的过程中，若客户端主机重启完成，客户端的内核就会接收重传的报文，然后根据报文信息传递给对应的进程：

- 如果主机没有进程监听该 TCP 报文的目标端口号，那么客户端会回复 RST 报文，断开该 TCP 连接；
- 如果有进程监听该 TCP 报文的目标端口号，由于主机重启后，之前 TCP 连接的数据结构已经丢失了，因此客户端内核里的协议栈会找不到该 TCP 连接的 socket 结构体，于是回复 RST 报文，重置 TCP 连接。

所以，只有有一方重启完成，收到之前 TCP 连接的报文，都会回复 RST 报文，以断开/重置连接。

4）一端进程崩溃

当应用进程被 kill -9 杀掉以后（模拟进程崩溃），服务器会发送 FIN 报文，与客户端进行 TCP 四次挥手。

同样地，客户端进程崩溃时也会发送 FIN 报文到服务端。

13. HTTP鉴权

浏览器如何验证服务端的身份是不是仿冒的?

为什么密钥，需进行对称加密和非对称加密?

14. HTTP 1.0/1.1/2.0 之间的区别

1）HTTP1.0

- **无状态**：服务器不跟踪也不记录请求过的状态；
- **无连接**：浏览器每次请求都需要建立 TCP 连接。

每个请求建立一个 TCP 连接，请求完成后立马断开连接。这会导致两个问题：

- 连接无法复用，每次请求都需要进行一次 TCP 连接，而 TCP 的连接释放过程又比较耗费资源，所以这种无连接的特性会导致网络利用率很低；
- 队头阻塞（head of line blocking），HTTP1.0 规定下一个请求必须在前一个请求响应到达之前才能发送。如果当一个请求响应迟迟不到达，后续的请求就会阻塞。

2）HTTP1.1

为了解决 HTTP1.0 的痛点，HTTP1.1 带着这些特性出现了：

- **长连接**。HTTP1.1 新增了一个 connection 字段，通过默认设置为 Keep-alive 可以保持连接不断开，避免了每次客户端和服务器请求都要重复建立和释放 TCP 连接，提高了网络的利用率。如果客户端想关闭 HTTP 连接，可以在请求头中携带 connection:close 来告知服务器连接将关闭。
- **支持请求管道化（pipelining）**。基于 HTTP1.1 的长连接，使得请求管道化成为可能。管道化使得下一次请求不必在前一个请求响应返回后才能进行，而是可以同时发送多个请求，服务器

按照先进先出（FIFO）的原则来处理它们。

3）HTTP2.0

- **二进制分帧**：HTTP1.x 的请求和响应报文段都是由起始行、首部和正文组成，各部分之间以文本换行符进行分隔；而 HTTP2.0 将请求和响应数据分割为**更小的帧**，并且采用**二进制编码**，解析起来更为高效。
- **多路复用**：在 HTTP1.1 协议中，浏览器客户端在同一时间针对同一域名下的请求有一定的数量限制，超过限制数量的请求会被阻塞。这也是为什么有些站点会有多个静态资源 CDN 域名的原因之一；而 HTTP2.0 的多路复用允许同一个连接发起多次请求-响应消息，这些消息以二进制帧的形式组成，可以乱序发送。因此，单个连接上通信双端可以并行交错地请求和响应，并互不干扰，很容易地实现了**多流并行**。
- **头部压缩**：在 HTTP1.x 中，头部元数据都是以纯文本的形式发送，通常会给每个请求增加 500~800 字节的负荷。HTTP2.0 使用头部压缩的方式来减少需要传输的 header 大小，并且通讯双方各自缓存了一份 header fields 表，既避免了重复 header 的传输，又减小了需要传输的大小。使得通讯更为快捷，高效。
- **服务器推送**：在 HTTP2.0 之前，客户端浏览器在请求一个页面时，需要浏览器解析到相应的位置，再多次发送请求；而 HTTP2.0 是一种**在客户端请求之前发送数据的机制**，相当于在一个 HTML 页面内集合了所有的资源，更为高效。除此之外，服务器推送还有一个更大的优势：可以缓存！也让在遵循同源策略的情况下，不同页面之间可以共享缓存资源成为可能。

15. 套接字详解

https://www.51cto.com/article/608725.html?u_atoken=024d6ff8-df0d-4e84-bee8-4098b79317a5&u_asession=01x9RTKp8mT92M6EHNwyJF9J5OW0X-h_CBPHQLLTSmBqkWs1qtVYeuJrUWzKCdtR2-X0KNBwm7Lovlpxjd_P_q4JsKWYrT3W_NKPr8w6oU7K9q1jFgrjQiHBtoU9C_ePCJtXQuvWAUwqvTq5ErmvVKrmBkFo3NEHBv0PZUm6pbxQU&u_asig=05GK_J6a25ce9sXtA6ObOh7G_XfRqW3Utq75VHWV8BeY0CKFF_E-GbZlSx0W8VjXgNavepxPbtmigD8ksTl6ReSLzGit8DzpVHkN9V7YSyoR6Oe5YmbXML74rLCkQCUEuDEClUnNceOgCD0yyt5F0j8tcNiYgdPHJTSwp2Wh7Kdfn9JS7q8ZD7Xtz2Ly-b0kmuyAKRFSVJkkdwVUnyHAlJzeJv3_bflhFkdF2mcAGDrVNu2bA4J8RJVYoOlFnoTRYUlCLXF00RedlaK-_vtvWG2-3h9VXwMyh6PgyDlVSG1W-ddWbi683AJqtmavttULZt2t06Kw4WXjL8pvgPxmcQytATvR3iilobqMjqnDQHPQMpSHcPL5ZqKshm0Ki_Hl6YmWspDxyAEEo4kbsryBKb9Q&u_aref=j7kVkqAkteDBxAtepgPYglme3fQ%3D

https://blog.csdn.net/wumenglu1018/article/details/54019755

1）进程间通信

本地进程可使用多种方式来通信，比如常见的四类：

- 消息传递（管道、FIFO、消息队列）；
- 同步（互斥量、信号量、读写锁）；
- 共享内存（分配一段多进程可见的内存）；
- 远程过程调用（RPC）

进程通信首要解决的问题是如何唯一标识一个进程，本地进程可以通过进程 PID 来唯一标识，但是在网络中是行不通的。但是，TCP/IP 协议族帮我们解决了这个问题：

- 网络层的“IP 地址”可以唯一标识一台网络中的主机；
- 传输层的“协议+端口”可以唯一标识主机中的应用程序（进程）。

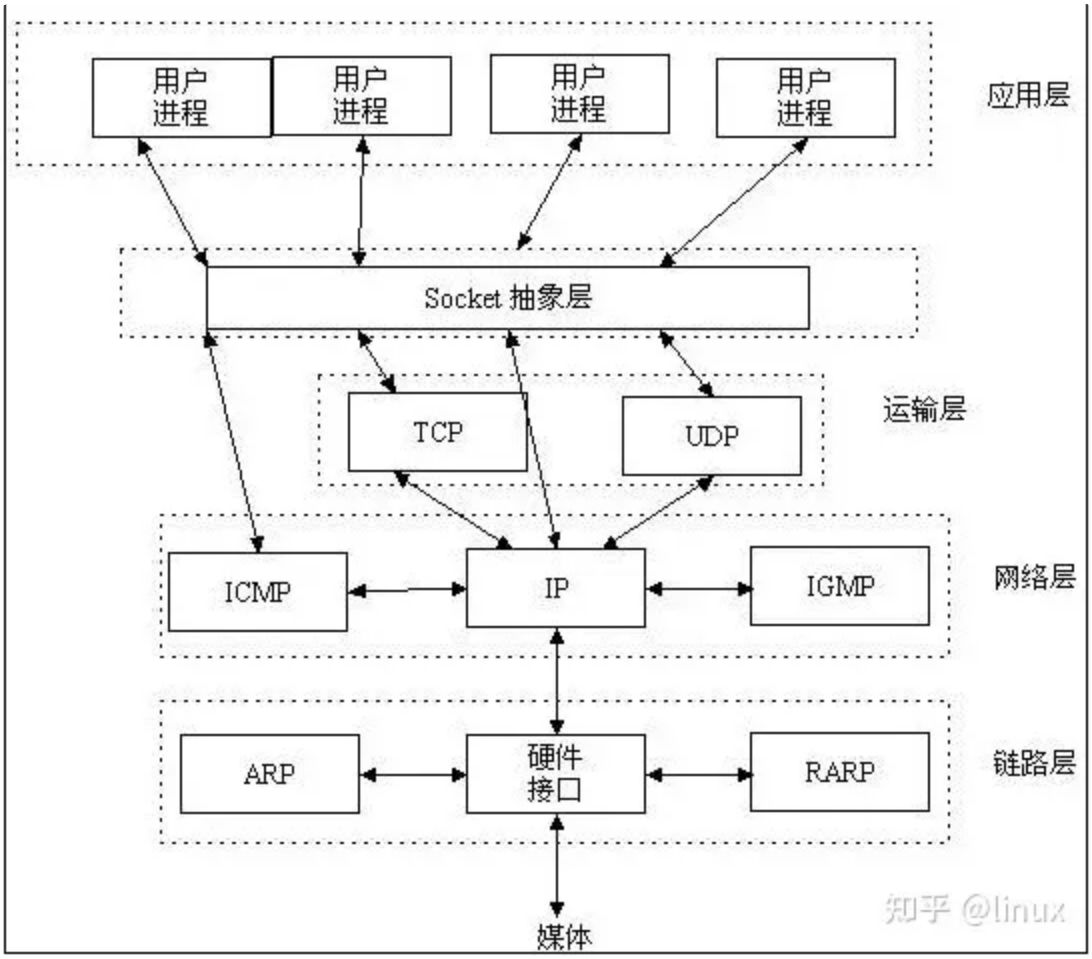
这样，通过三元组（IP 地址，协议，端口）就可以标识网络中的进程了。

使用 TCP/IP 协议的应用程序通常采用应用编程接口：套接字（socket）和 TLI（已经被淘汰）来实现进程间通信，目前几乎都是采用 socket，所以网络之中一切皆“socket”。

2）socket是什么

socket 起源于 UNIX，而 UNIX/Linux 的基本哲学之一是“一切皆文件”，文件的三种基本操作是：打开文件（open），读写数据（write/read），关闭文件（close）。

而 socket 可以看做是一种特殊的文件，它自然也提供了这些基本操作的函数接口，例如 socket() 、write()、read()、close() 函数。



Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议.如今大多数基于网络的软件，如浏览器，即时通讯工具甚至是P2P下载都是基于Socket实现的。

3）socket 常用函数

3.1 socket() 函数

```
1   // domain为协议域，type是socket的类型，protocol指定协议
2   int socket(int domain, int type, int protocol);
```

socket 函数相当于普通文件的打开操作，在 UNIX/Linux 系统下打开一个普通文件时会返回一个文件描述符（file descriptor，简称 fd）。socket() 也是类似的，主机上某个进程调用 socket() 函数后会生成一个唯一标识符，即 socket 描述符（socket descriptor），后续进程的读写操作都会用到它。

3.2 bind() 函数

```
// sockfd是socket描述符
// *addr是一个地址指针，存放了待绑定给socketfd的协议地址（比如IP+端口号）
// addrlen是地址长度
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

bind() 函数就是给 socketfd 描述符绑定一个对外的网络地址。

通常，服务器启动的时候都会通过 bind() 函数去绑定一个对外开放的地址（IP地址+端口号），客户端可以通过这个地址来访问服务器；而客户端就不用调用 bind() 手动指定，而是在系统调用时自动分配一个端口号和自身 IP 组合。

3.3 listen()、connect() 函数

```
// 指定socket监听，backlog为最大连接个数
int listen(int sockfd, int backlog);
// 指定服务器端的地址，发送connect请求
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
```

listen 函数指定某个 socket，设置最大连接个数进行监听，并且，listen 函数会将 socket 的操作由主动变为被动类型，来等待客户的连接请求。

connect 函数是客户端调用的，调用时会指定客户端主机的某个 socket，和服务器端对应的 socket 建立网络连接。

如果主机是服务器，在调用 socket()、bind() 函数之后会调用 listen() 来监听这个 socket，如果有客户端调用 connect() 指定地址发出请求，服务器就会收到这个请求。

3.4 accept() 函数

```
// sockfd是服务器的socket描述符，*addr返回客户端的协议地址，addrlen是地址长度
// accept函数成功以后，会返回一个全新的sockfd，代表与客户端的连接套接字
int accept(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
```

服务器端依次调用 socket()、bind()、listen() 函数之后，就会监听指定的 socket 地址；这时客户端调用 socket()、connect() 之后向服务器端发送了一个连接请求。

当服务器端监听到这个请求之后，就会调用 accept() 函数接收请求，当请求接收以后，就可以进行进程之间的网络数据传输了，和本地进程传输数据是一样的。

3.5 read()、write() 函数

```
// fd是已连接的套接字，*buf是需要读取/写入的数据，count是数据长度
ssize_t read(int fd, void *buf, size_t count);

ssize_t write(int fd, void *buf, size_t count);
```

当客户端与服务器建立连接之后，就可以调用 read() 和 write() 函数进行读写操作了：read 函数负责从 fd 中读取内容，有三种情况：

- 读取成功后，会返回实际读取的字节数；
- 读取完成时，说明文件已经结束了；
- 读取失败时，返回值会小于 0，可能产生了读中断（EINTR）或者网络连接出现问题（ECONNREST）

write 函数将 buf 中的数据写入到文件描述符 fd，有两种情况：

- 写入成功后，返回实际写入的字节数；
- 写入失败时，返回 -1，可能产生了写中断（EINTR）或者对方已经关闭了连接（EPIPE）。

除了 read()/write() 函数，还有其它的 I/O 函数也有着类似的作用，比如：recvmsg()/sendmsg() 函数，是非常通用的网络 I/O 函数。

3.6 close() 函数

```
// fd是已连接的套接字
int close(int fd);
```

当服务器和客户端数据完成读写操作以后，就需要关闭对应的 socket 描述符，相当于文件读写完成后都需要用 close() 关闭文件一样。

注意：在 TCP 连接中， close 函数只是将 socket 描述符的引用计数 -1，只有当 socket 上的引用计数为 0 时，才会触发客户端和服务器的“四次挥手”终止连接。

4）socket 和 HTTP 区别

1. socket

了解了上面的介绍之后，我们知道 socket 套接字就是通信的基石，是一个接口而非协议，它是 TCP/IP 协议的封装。在进行 socket 网络通信时必须的五种信息：连接使用的协议、本地主机 IP、端口号、远程主机 IP、端口号。

所以，当服务端实现了 socket 接口的时候就指定好传输层协议了，可能是 TCP 或者 UDP，当用 TCP 时，该 socket 就是个 TCP 连接，反之就是 UDP 连接。远程客户端请求时，指定当前主

机的协议、IP 地址和端口号，进行网络连接。一般是长连接，比如网络消息的通信，在建立连接之后一般不会轻易断开，除非遇到双方宕机，网络故障，或者长时间没有通信等原因。

2. http

HTTP（超文本传输协议）是建立在 TCP 协议之上，用于在客户端和服务器之间传输应用数据的协议。每次客户端进行 HTTP 请求时，服务器会返回一个响应，在 HTTP1.1 中当客户端请求结束后通过 connection:false 主动释放连接。交互方式为请求–响应，过程无状态，相当于短连接。

16. Cookie 和 Session 机制

1）产生背景：

HTTP 是一个无状态协议，无状态是指服务端不会跟踪和记录请求，即对请求处理没有记忆能力，这意味着每个请求都是独立的。它的优缺点分别是：

- 优点：服务器处理请求时不需要上下文信息，因此应答很快，每一次请求都是“点到为止”，提升了请求处理的效率；
- 缺点：缺少访问状态意味着如果后续请求和之前相关，比如 APP 登录功能，每次都需要重新登录，就必须重传请求，导致每次请求会传输大量重复的信息。

在一些 Web 交互场景下，比如：

- 登陆某网站时，需要记住登陆用户名密码信息，避免每次都进行用户名密码输入操作；
- 登陆某网站时，需要记住用户登陆的状态，避免每次都进行重复操作；
- 购物车添加商品时，需要标识和跟踪某个用户，才能知道购物车里面有几本书。

于是，两种用于保持 HTTP 连接状态的技术应运而生，分别是 Cookie 和 Session。

2）Cookie

Cookie是存储在客户端的一小段文本信息，由服务器颁发，在请求的 Response 中返回给浏览器然后进行存储。当浏览器再次请求网站时，会把 cookie 一同发送至服务器，服务器检查该 cookie，以此来记录用户状态。

如果浏览器禁用 cookie，就需要在 URL 后面加上 sid=xxxx 这样的参数来让服务端识别此用户状态。

3）Session

Session 是存储在服务器中的状态信息。当要记录用户的登录状态时，服务器会在用户初始登录的时候生成一个 sessionID，并把它放在返回给用户浏览器的 cookie 里面。只要用户继续访问，服务器就会更新 session 的最后访问时间，从而保持用户的会话状态。

服务器一般把 session 放在内存或者数据库存储，为了优化存储空间，会删除长时间没有访问的 session。

4）Cookie 和 Session 的区别

- **访问机制**：Cookie 通过检查客户端的用户“通行证”来确定用户身份，Session 检查服务器的“客户档案表”来确认用户状态。
- **安全程度**：不法分子可能会分析存放在本地的 Cookie 进行 Cookie 欺骗，而 Session 是有人登陆或者启动某个会话时才会产生，且 SessionID 是加密和定时失效的。所以 Session 安全系数更高。
- **会话机制**：Cookie 是由服务器生成，在 HTTP 请求的 Response 中返回给浏览器，浏览器解析 cookie 后将其保存为本地文件，该文件会自动将请求同一个服务器的信息绑定到一起（单个 cookie 保存的数据不能超过 4k）。Session 用类似哈希表的结构来保存用户信息，相当于一个“客户档案表”。SessionID 生成以后会放在一个名为 JSEESIONID 的 cookie 里面，返回浏览器。当用户带着保存了 SessionID 的 cookie 文件访问服务器时，服务器会解析出 sessionID，进行用户状态的判断，并更新 session 的最后访问时间。