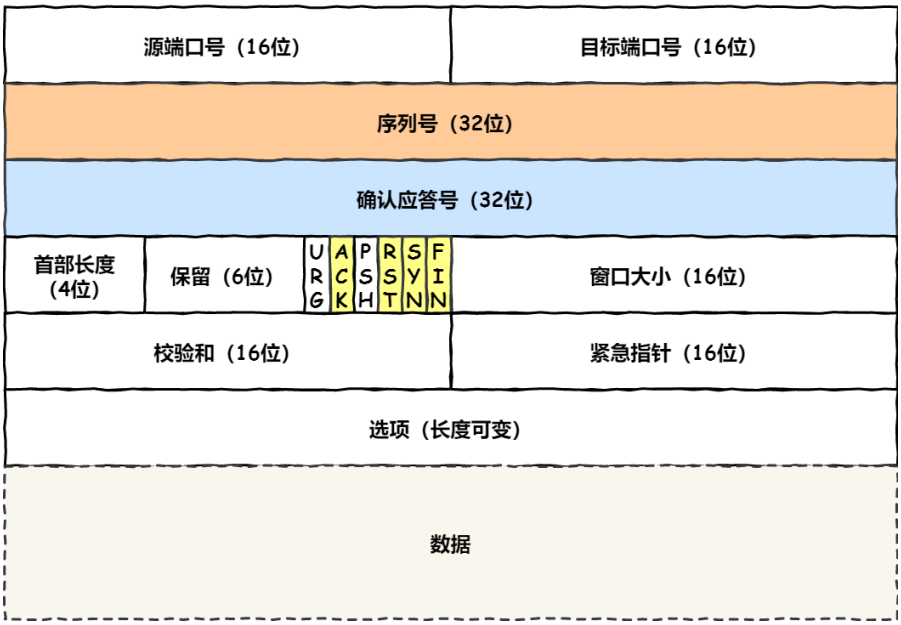
TCP

- ## TCP基本认识
  - ### TCP头格式

TCP 头部格式



带颜色的就是需要重点关注的地方

- **0. 源端口号 / 目标端口号**
  - 作用：告知数据应该发送给**哪个应用（又有说是线程）**，IP负责发送到哪个主机，而TCP对应用层负责，所以应该要知道从哪个应用发出（源端口），也要知道给哪个应用（目标端口）
- **1. 序列号**
  - 作用：解决**网络包乱序**的问题——能够找到该次TCP连接所有发送的包，且接收端能够正确计数并接收
  - 概念：建立TCP连接的时候，发送端的主机生成的**随机数**，通过**SYN包**传递给接收端主机，至此之后，在该连接中，每发送一次数据，序列号就 **+ 【数据字节数】**
- **2. 确认应答号**
  - 作用：解决不丢包问题——滑动窗口
  - 概念：**下一次希望收到的序列号**，接收端发送给发送端，发送端收到该确认应答后，可以确认该序列号之间的包均已经正确接受
- ——序列号 和 确认应答号 配合使用，序列号能够保证发送端乱序发送均能正确到接收端，且接收端可以按序排列。确认应答号能够接收端在有序排列下，告知发送端最新有序的最大序列号

- **3. 控制位（1bit），**主要是在请求连接/断开连接中用到（让主机识别那些特殊的报文），这些报文都会使连接状态发生改变
  - ACK：1，使能【确认应答号】，TCP要求最初连接的SYN包可以不为1之外，其他时候都必须设置为0——回复
  - RST：1，表示TCP连接中出现异常，强制断开连接——重新连接
  - SYN：1，表示**期待连接**（在3次握手中用到），并且发送端会在序列号中传递对应产生的随机序列值，作为序列号初始值——发起连接
  - FIN：1，表示停止传输数据，并希望断开连接。用在通信结束后希望断开连接，通信的双方主机之间就可以相互交换FIN位为1的TCP段——结束连接
- **4. 窗口大小**
  - 作用：流量控制，双方在连接的时候会商量一个窗口大小（即缓存大小），表示当前的处理能力，发送太快，窗口塞满了，会出现大量丢包现象；发送太慢了，窗口常为空，处于空闲状态

- **TCP的位置**
  - 位置：在**传输层**
  - 背景：IP协议是在网络层，IP协议是不可靠的（主要就是让主机之间通信的），只负责发送和接收，但是不能保证包的交付、按序交付、也不保证内部数据的完整性
  - 作用：主要是保证可靠传输
  - 总结：TCP是工作在**传输层**，进行可靠数据传输的，它能够保证接收到的数据是有序的、无损坏、无间隔的（？）、非冗余的
- **TCP概念**
  - 概述：是面向连接的、可靠的、基于字节流的传输层通信协议
  - 面向连接的：**TCP是【一对一】**，而不是UDP协议可以一个主机同时发送给多个主机【一对多】
  - 可靠的：TCP能够保证一个报文一定能正确到达接收端，不论中间链路发生何种变化
  - 字节流：**消息不论有多大都能进行传输**（流概念），消息是**有序的**，前一个消息没有收到之前，不论后面的消息收到多少，都会阻塞在这边，**不会丢给上层**（应用层），并且会**丢弃重复**的报文
- **如何唯一确定一个TCP连接**
  - 连接的概念：用来保证可靠性和流量控制等的某些状态信息，这些信息的组合就是连接
  - 确定连接的条件：客户端和服务器端达成3个信息的共识：（即都同意，并遵守才能够连接）
    - socket：IP地址+端口号
    - 序列号：解决乱序发送、乱序接收的问题
    - 窗口大小：流量控制

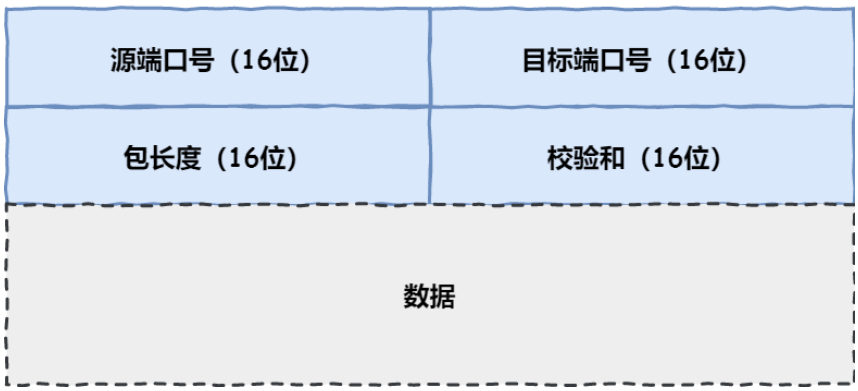
- **四元组**唯一确定一个TCP连接：
  - 源地址：32b，位于IP头部，作用是：通过**IP协议**发送报文到目的地址，告知目的地址报文来自哪里
  - 目的地址：32b，位于IP头部，作用是：通过**IP协议**发送报文到该地址，主要就是指明目的地
  - 源端口：16b，位于TCP头部，作用是：告诉TCP协议该报文来自哪个**进程**
  - 目的端口：16b，位于TCP头部，作用是：告诉TCP协议该报文应该发给哪个**进程**

- **TCP的最大连接数**

  IP的服务器监听了一个端口，
  - 背景：服务器通常在某个本地端口上监听，等待客户端的连接请求，所以收到的客户端的地址是随机的
  - 理论上最大的连接数：= **客户端 IP 数 * 客户端的端口数**，而IPv4，客户端的IP数最多为2^32，端口数最多为2^16，那么理论上的最大连接数为2^48
  - 但是实际上最大并发TCP连接数 << 理论上限
    - 1. 文件描述符的限制：连接中用到的socket，是文件，所以需要通过ulimit配置文件描述符的数目
    - 2. 内存限制：每个TCP连接都需要占用一定内存，操作系统的内存是有限的

- **UDP和TCP的区别**
  - UDP的概念：
    - 概念：UDP极简，利用IP提供面向**无连接**的通信服务
    - UDP的头部：头部只有8个字节

      UDP 头部格式

      - 源端口号/目标端口：告知UDP协议应该发送给哪个**线程**，以及从源主机的哪个线程发出的
      - 包长度：UDP首部的长度 + UDP数据的长度
      - 校验和：主要是为了可靠的传输，进行验证
  - 区别：
    - 1. 在连接上

        - TCP：**面向连接**的传输层协议，必须要先连接后才能传输数据
        - UDP：不需要连接，有需要就立刻传输
    - 2. 协议的服务对象
        - TCP：一对一的服务，一个TCP连接有且只能有2个端点
        - UDP：可以一对一、一对多、多对多的多种形式的交互
    - 3. 是否有可靠性
        - TCP：**可靠性交付的**，数据可以是**无差错、不丢失、不重复、按需到达**
        - UDP：尽最大努力交付的，不保证可靠交付数据
    - 4. 拥塞控制、流量控制
        - TCP：有拥塞控制和流量控制，能够保证网络传输的安全性——能够顺利到达，不会发生丢包
        - UDP：无控制操作，即不关注网络拥堵情况，即使很拥堵也照样发送
    - 5. 首部开销
        - TCP：为了确保可靠性，所以首部开销较大，普通是20字节，如果增加选项，长度还会增加
        - UDP：固定为8字节
    - 6. 传输方式
        - TCP是流式传输，没有上限，但是能够保证有序和可靠
        - UDP是按包为单位发送，有边界，但是可能会丢包和乱序（包之间的乱序）
    - 7. 分片
        - TCP：TCP会设置一个MSS，如果数据长度 > MSS，就会在传输层进行分片，目标主机收到后，在传输层进行重组，如果丢失其中一个分片，那么就重传该丢失的分片即可
        - UDP：向下给IP是整个包，如果该包的长度 > IP控制的MTU，那么会在网络层（IP层）进行分片，然后分开传送，目标主机收到之后，在IP层组装完毕后上传给传输层，如果丢失其中一个分片，那么整个UDP包都需要重发——传输效率很差，所以要求UDP向下丢的包 < MTU
- TCP和UDP的分别应用场景：
    - TCP：根据可靠性交付的特性
        - FTP文件传输
        - HTTP/HTTPS
    - UDP：根据非连接、非一对一、随时性和简单高效
        - 包总量较小的通信，eg：DNS、SNMP
        - 视频、音频等多媒体
        - 广播
- **UDP和TCP的【首部长度】字段的区别**

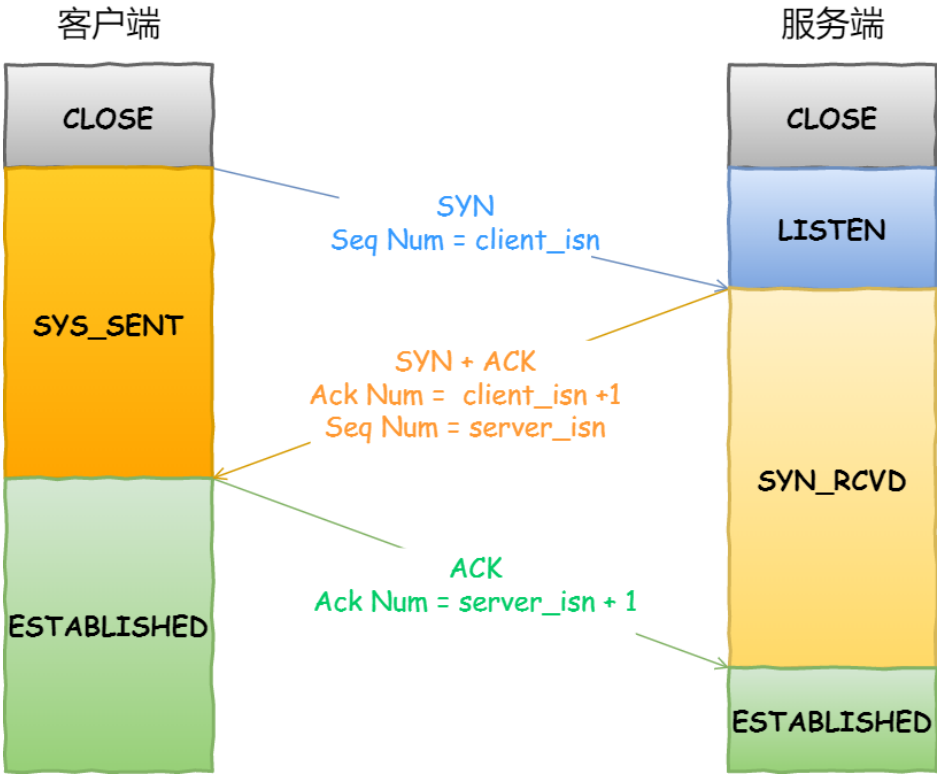
- TCP有一个可变的【选项】字段，UDP的首部长度是固定的

- **UDP有【包长度】字段，而TCP没有**
  - TCP的包长度可以计算得到：TCP数据长度 = IP总长度 - IP头部 - TCP首部长度
  - UDP的包长度：其实也可以计算得到，但是网络和硬件等对数据的要求，首部长度要求是**4字节的倍数**——所以为了UDP首部长度满足要求，所以增加了该字段（反正不用白不用）

TCP连接建立

在传输之前，需要请求建立连接——3次握手完成

- **TCP三次握手过程**

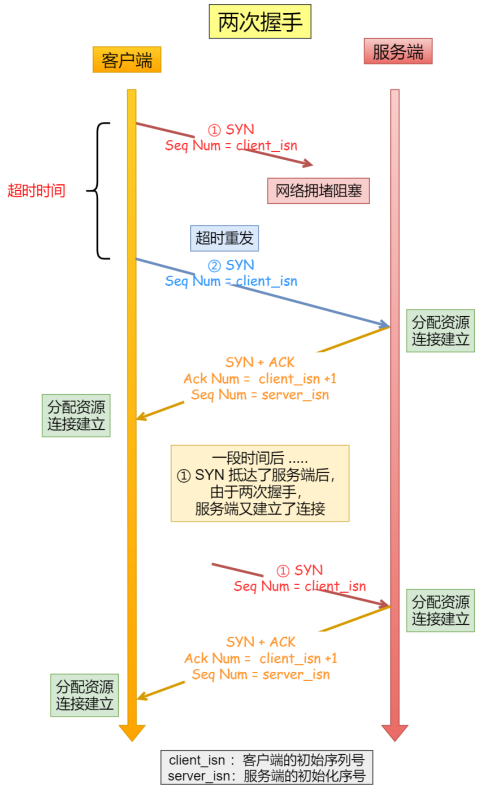


  - 0. 初始，两端都是关闭状态，服务端是先开始**监听**的——即，等待接收连接请求，服务端处于**LISTEN状态**（而不是收到请求后才打开）
  - 1. 客户端需要建立连接，所以首先发出请求报文：
    - 客户端会随机初始化序号**client_isn——放在首部的序列号**中，
    - 同时把标志位**SYN设置为1**，表示是SYN报文。
    - 然后将此报文发送给服务端，表示请求TCP连接，该报文是不包含应用层的数据的，发出该报文后，客户端是处于**SYS_SENT状态**（服务器在收到该报文并做出响应之间都是LISTEN状态）
  - 2. 服务端在收到SYN报文后，进行响应报文：
    - 也随机初始化序号**server_isn——放在首部的序列号**中，
    - 同时把首部的确认应答号设置为：client_isn + 1
    - 再将，**SYN 和 ACK设置为1**

    - 将该报文发给客户端，该报文也不包含应用层的数据，发出该报文后，服务端处于**SYS_RCVD状态**（客户端在发出SYN报文到收到回复之前，一直都是SYN_SENT状态）
  - 3. 客户端在收到服务端的响应后，还需要发送一个响应报文：
    - ACK设置为1（SYN为0）
    - 确认应答号设置为server_isn + 1
    - 将该报文发送给服务端，**该报文可以包含应用层的数据**，发出该报文后，客户端处于ESTABLISHED状态（服务端在作出响应到收到回复之前，一直都是SYN_RCVD状态）
  - 4. 服务端收到报文后，也处于ESTABLISHED状态
  - ——至此，3次握手完成，TCP连接建立，之后就可以发送数据了
  - ps：Linux可以通过指令查看TCP的连接状态：
    - netstat -napt

- **why是三次**
  - 笼统的说法：3次握手才能保证双方都有接收和发送数据的能力——3个报文，那么会进行4次传输，那么表明双方都能发送数据和接收数据
  - 主要原因：
    - 1. 阻止重复历史连接的初始化**（首要原因）——防止旧的重复连接初始化造成混乱**
      - 主要原因是：网络情况复杂，数据包接收的顺序可能和发送数据不一样
      - 场景：客户端发送SYN请求，由于网络阻塞一直收不到ACK，那么客户端超时重传，此时client_isn重新设置了，而前一个client_isn被认为是无效的，而服务端收到旧SYN后，发出了响应，客户端收到ACK报文后，对比确认应答号，发现不是最新SYN的预期应答，所以客户端会发送RST给服务端，表示中断此次连接
      - 而如果是二次握手，那么客户端无法告知服务端是历史连接，而三次握手可以在第三个报文中告知服务端是否是预期连接
        - 如果是历史连接，即确认应答号是过期的，那么第三次报文是RST
        - 如果不是历史连接，那么第三次报文是ACK报文，连接会建立
    - 2. 同步双方的初始序列号
      - 双方都有一个独立的序列号，用来保证传输的可靠性：去除重复数据，按照序列号的顺序使得报文有序排列，标记哪些数据包已经被正确接收到了（通过确认应答返回）

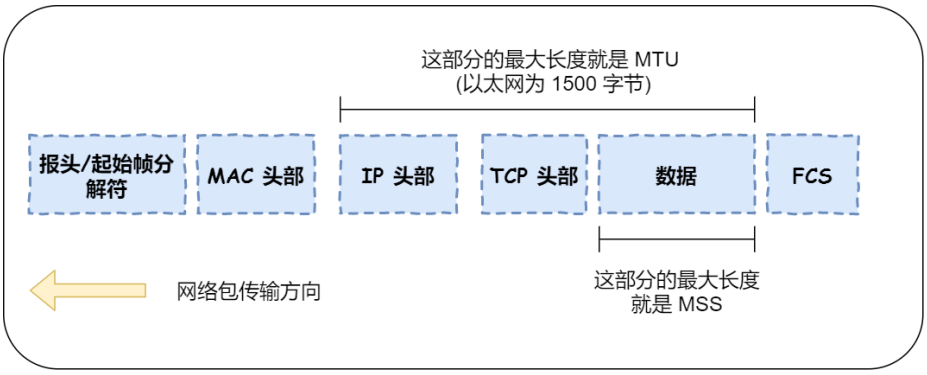
        - 客户端发送client_isn给服务端，服务端收到SYN报文后，会发送一个ACK应答报文表示正确收到；而服务端在ACK报文中会带有server_isn，所以客户端还需要在收到ACK报文后发出响应报文，表示正确收到——那么client_isn 和 server_isn都已经被可靠的同步了
        - ——而二次握手，无法做到可靠的同步，无法保证服务端的初始化序列号被正确同步了
    - 3. 避免资源浪费
        - 如果是两次握手，那么服务端在收到SYN报文后，就需要建立连接了，那么如果此时网络中是存在阻塞的，客户端一直收不到服务端的ACK报文，那么就会超时重发SYN，而服务端只能对收到的每次SYN后，建立连接，而连接的建立是需要分配资源的——服务端建立多个冗余连接，造成资源浪费



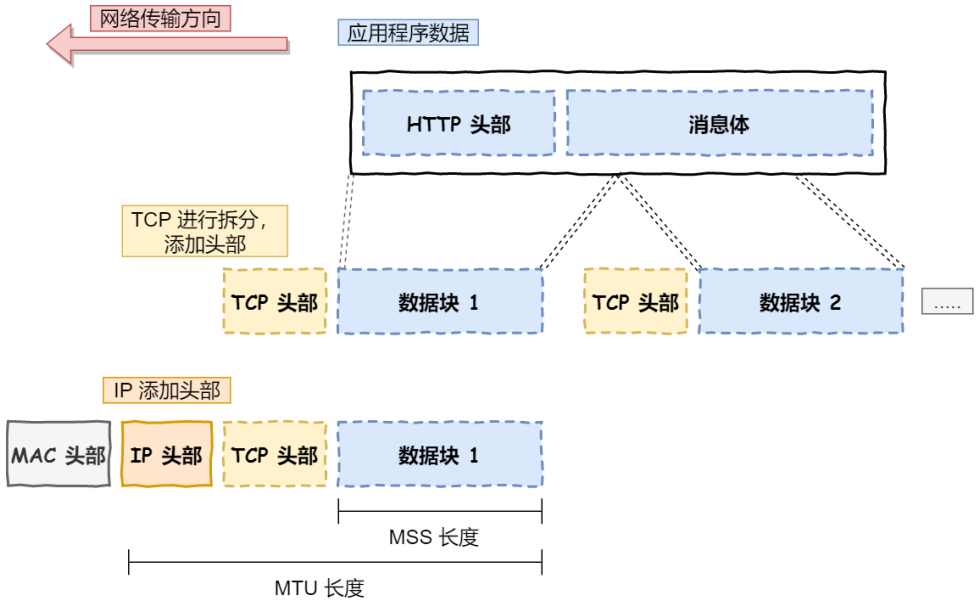
- 总结：三次握手是在保证可靠连接的基础上的最小连接次数
- 【两次握手】：不能防止历史连接的建立，也会出现重复连接造成资源的浪费，也不能确保可靠的同步
- 【四次握手】：可以压缩为三次握手

- **初始序列号ISN的产生**
    - 每次连接都需要重新初始化序列号ISN，主要是为了：
        - 能够通过序列号区分旧的连接和新的连接，将旧连接的报文丢弃，防止出现连接混乱
        - 为了安全性，防止黑客伪造相同的序列号的TCP报文被接收
    - 如何产生随机ISN：
        - 初始时：利用系统的时钟，每4ms增加1

    - 后面确定了一个较好的ISN随机生成算法：
        - ISN = M + F (localhost, localport, remotehost, remoteport) M：是计时器，每过4ms增加1；F是hash算法，参数是源IP，源端口，目的IP，目的端口计算出一个哈希值，并且该哈希值不能被轻易计算出来：MD5算法是较优的hash值算法
- **why TCP层还需要MSS**
    - 背景：IP层存在MTU，如果包超过MTU，会自动进行IP分片，但是TCP层还是会对TCP报文进行TCP分片（UDP不会进行分片，全权交给IP分片）
    - TCP设置MSS原因：主要是为了重传的效率
        - 如果TCP不进行分片，都交给IP层来分片，那么如果从传输层得到的TCP包 + IP首部 > MTU，那么就要进行分片，分片发送，如果在传输过程中，IP部分分片丢失，就要全部重传所有IP分片，而IP层没有超时重传机制，需要TCP层来负责超时重传——所以，变成了，服务端收到TCP报文，发现存在丢失，那么不会向客户端发送ACK，而客户端超时等待之后会重新发送整个TCP包（IP层还会分层），那么该效率是很低的
        - 所以，**在建立TCP连接的时候，需要协商双方的MSS值**，TCP层发现TCP数据超过MSS后，就会进行TCP分片，从而形成的IP包也不会超过MTU，**不用进行IP分片**。如果TCP分片其中一个丢失，超时重传是MSS为单位的——提高效率，TCP的机制能够保证确认丢失哪个分片
    - ps：MTU：IP首部 + IP数据；MSS：TCP的数据（不包括TCP首部）

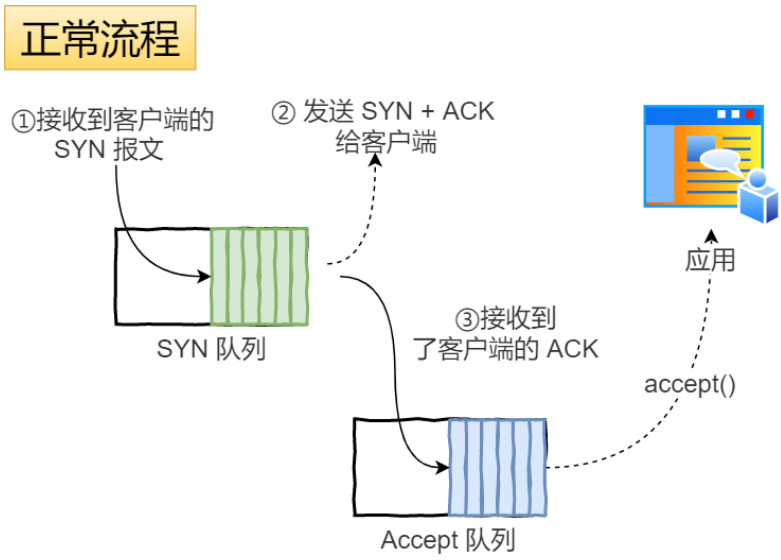


    - TCP层的分片：



- 将HTTP层丢下来的HTTP报文=HTTP头部 + 消息体，直接分片——所以**HTTP的头部只会出现在第一个TCP数据报**中，并且每片都添上TCP的头部——每片都会有一个头部，并且头部的部分参数是不同的（用来标识TCP数据报的顺序）
- 然后每个TCP数据报到IP层，都会增加一个IP头部
- 到MAC层，又会增加一个MAC首部

- **SYN攻击**
  - 就是攻击者在短时间内伪造多个不同的IP地址的SYN报文，发送给服务端，而服务端就进入到SYN_RCVD状态，但是由于不存在真实的IP地址，所以服务端发出的响应等不到回答，那么会IP地址会**占满服务端的SYN接收队列**，而服务器无法为正确的客户端建立连接
  - 解决方法：
    - 1. 修改Linux参数，控制队列大小和当前队列满时做出适合的处理
    - 2. tcp_syncookies 可以应对SYN攻击：
      - Linux的SYN队列和ACK队列的工作流程：

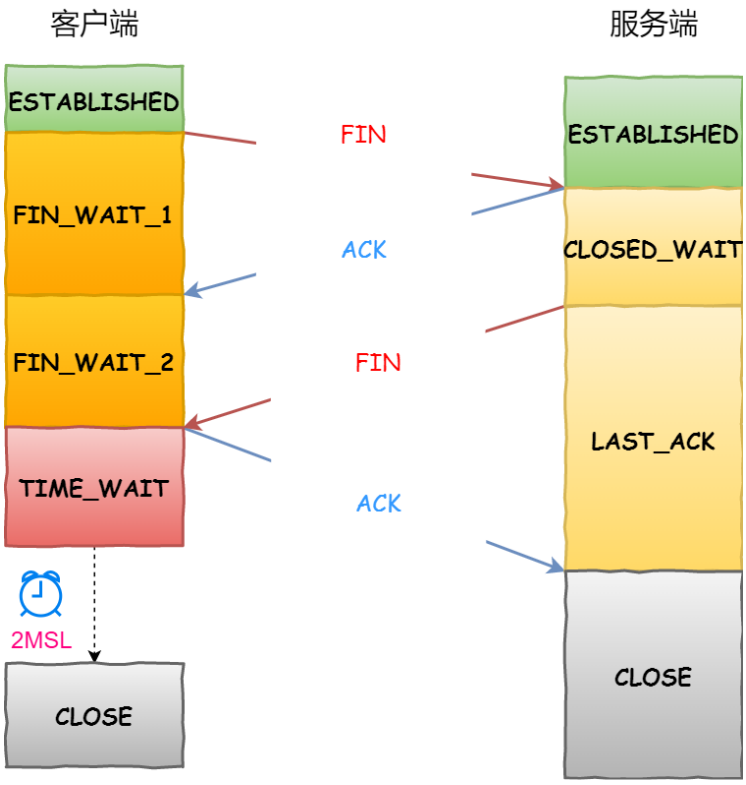


        - SYN队列：表示收到客户端的SYN报文，还未建立连接；Accept队列：表示已经3次握手完成，完成建立连接
        - 1. 服务器收到客户端的SYN报文后，将客户端信息加入到SYN队列
        - 2. 服务器发送响应，等待客户端的响应
        - 3. 服务器收到响应后，将客户端从SYN队列移除，放入到Accept队列
        - 4. 应用通过调用accept() socket接口，从Accept队列中取出连接
    - 如果发生异常：
        - 应用程序处理过慢：那么从Accept队列移除的速度很慢，那么Accept队列容易满并溢出
        - 收到SYN攻击：那么SYN队列易溢出，但是Accept队列为空
    - ——使用tcp_syncookies，可以解决：
        - 当SYN队列满了，服务器再收到SYN报文，不再加入SYN队列，而是计算出一个cookies，放在响应报文的序列号中发送给客户端
        - 服务器再次收到客户端的响应报文后，检查该ACK包的合法性，如果合法就直接放在accept队列中，表示已经正确建立连接

- **TCP连接断开**

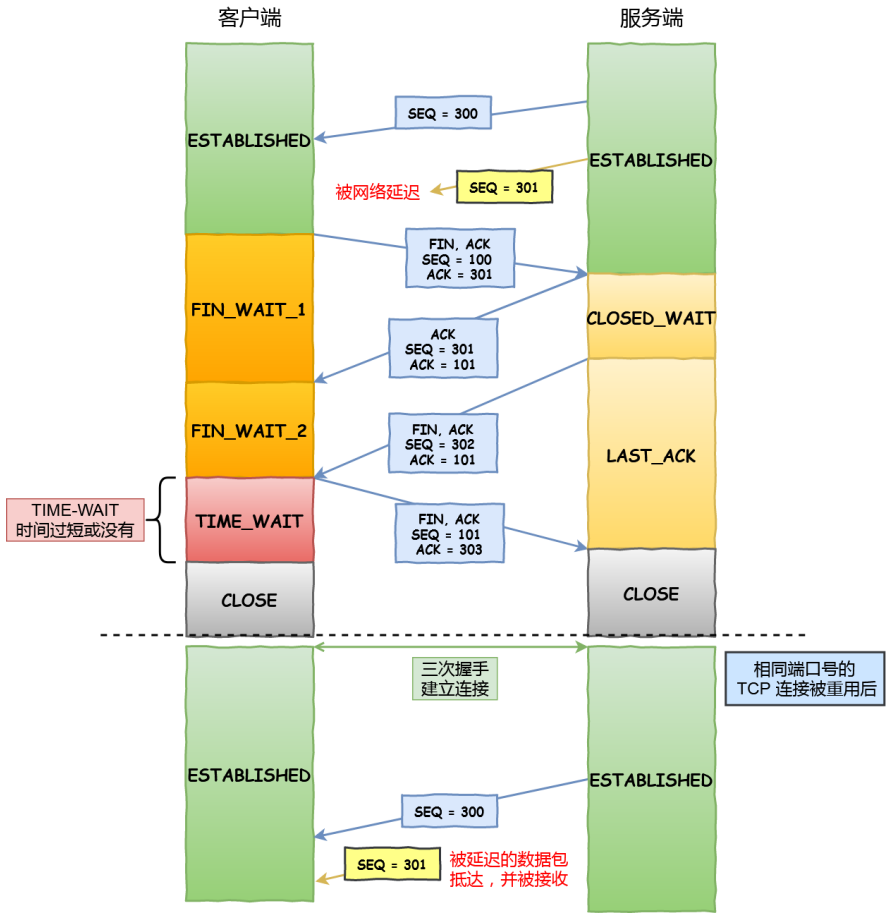
  双发都可以主动发起断开请求，断开后相关的资源会被释放

  - **TCP四次挥手**

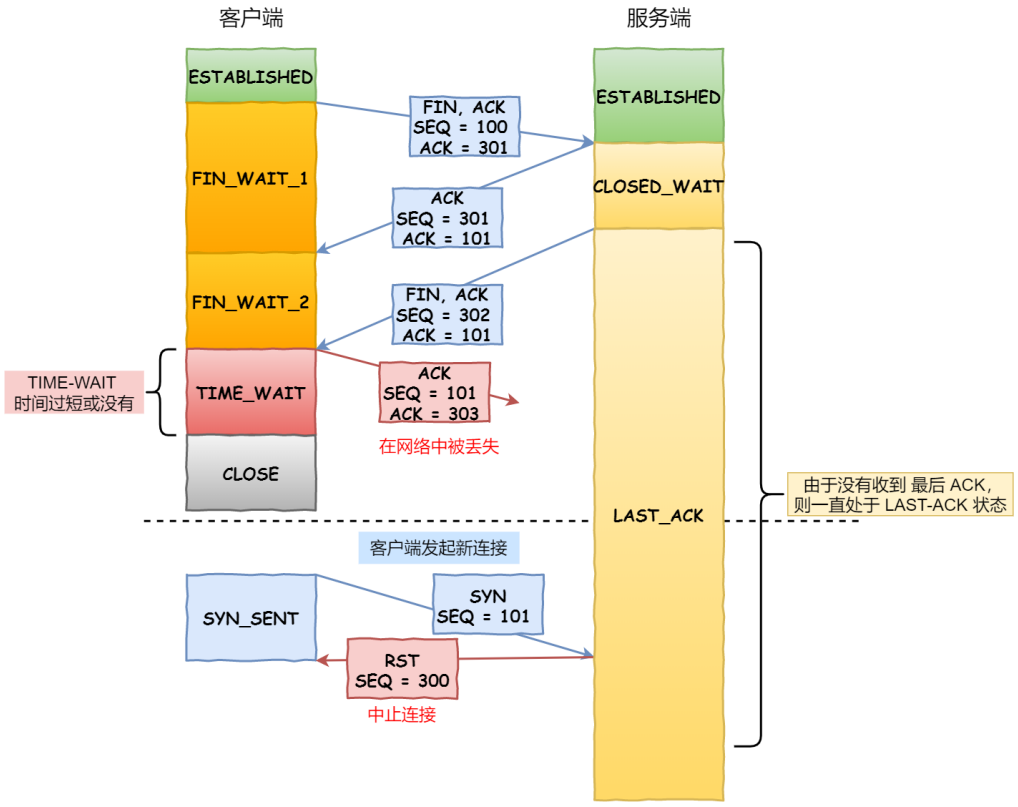


    - 1. 客户端准备关闭连接，会发送一个FIN报文：
        - 首部FIN = 1

    - 客户端**进入FIN_WAIT_1状态**
  - 2. 服务端收到该报文后，向客户端发送一个ACK报文，服务端进入**CLOSED_WAIT状态**
  - 3. 客户端收到ACK报文后，进入**FIN_WAIT_2状态**（暂时不发送报文了）
  - 4. 服务端在CLOSED_WIAT状态，会处理剩余的数据，当处理完成后，会像客户端发送FIN报文，服务端进入**LAST_ACK状态**
  - 5. 客户端收到FIN报文后，会发送ACK报文，客户端进入**TIME_WAIT状态**
  - 6. 服务器收到ACK报文后，进入**CLOSE状态**
  - 7. 客户端不会马上关闭，而是进入等待状态，过2MSL之后，就关闭连接
  - 总结：共有2对FIN-ACK报文，只不过方向是相反的
  - ps：需要注意：主动关闭连接的，才有TIME_WAIT状态
- **why是四次**
  - 比3次多一次的原因：在服务端收到客户端的FIN报文，回复ACK报文时，仅仅发送ACK表示正确收到，而不会附带服务端的FIN报文，因为服务端通常还有需要处理的数据，所以需要4次握手
- **why需要TIME_WAIT状态——Why需要等待**
  - 发起断开请求的一方，会存在TIME_WAIT状态
  - 主要原因：
    - 1. 防止具有相同四元组的旧数据包被收到



    - 主要考虑到特殊情况：如果在established状态下发送的一个数据包因为网络包延迟没有即时到，而在这个过程中：TCP关闭了，且又让相同端口的TCP连接了，如果没有TIME_WAIT时间/时间不够长，那么该包还是有可能被下一个TCP连接所接收，那么会出现数据错乱
    - ——所以，设置了TIME_WAIT这个2MSL超时等待时间，能够保证该时间之后，**数据包能到的都到了，不能到的也都被丢弃了**，网络中不存在旧的数据包了，之后数据包都是新连接产生的
- 2. 保证另一端能够被正确关闭，即主动方最后一个ACK能确定被被动方收到，并且正确关闭



    - 如果TIME_WAIT过短或者不存在，那么请求方发出最后的ACK就关闭了，那么如果ACK在传输过程丢失，那么即使被动方重发也到达不了请求方，而被动方就一直处于LAST_ACK状态，那么如果被动方收到其他主机的SYN请求，就会发出RST，连接建立被终止
    - ——所以，一定要让TIME_WAIT等待足够长的时间：
        - ACK正确收到，那么等待2MSL就关闭了
        - ACK未正确收到，那么被动方会超时重传FIN报文，请求方也有能力来接收FIN并且再次发送确认

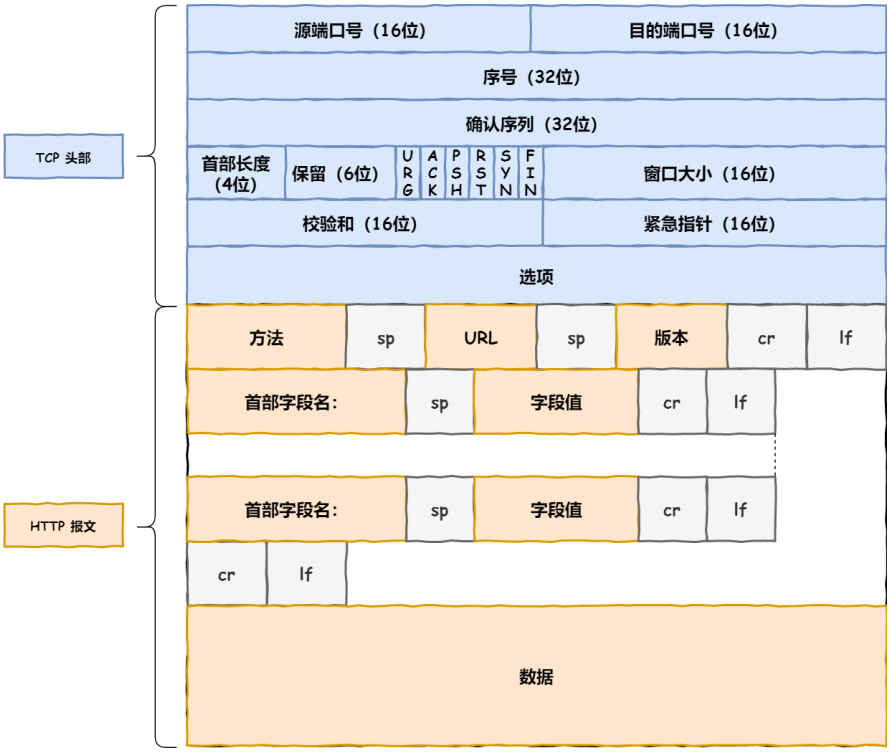
- **why TIME_WAIT等待的时间是2MSL——Why等待这么长**
    - MSL：报文最大生存时间，是报文在网络上生存的最大时间，超过这个时间还未接收的报文将会被丢弃
    - TTL：是IP头部的一个字段，表示IP数据报可以经过的最大路由数，每经过一个路由器就会-1，当TTL=0时，该数据报将会被丢弃，同时会发送ICMP报文给发送方

- MSL 和 TTL 的区别：MSL的单位是时间，TTL的单位是跳数，所以MSL的时间应该要大于TTL减到0的时间，确保IP数据报已经完全消亡
- 2MSL的原因：对于一些发送方发送的数据包，需要接受方接收后发回响应，让发送方接收知道已经数据包已经被正确接收，这两个来回最大就是2MSL
- eg：4次挥手中的最后一个ACK，如果被动方没有接收到，那么会重新发送FIN，来去就是2MSL
- ps：注意：2MSL是从请求方发送ACK开始计时的，如果超时之后重发FIN报文，那么请求方会重新发送ACK，那么计时又重新开始了。Linux的2MSL是60s

- **如何优化TIME_WAIT**
  - TIME_WAIT不是越长越好，如果越长：
    - 如果请求方是服务端，那么TIME_WAIT在服务端出现
    - 1. 内存资源占用——已经准备断开连接了，却长期不释放TCP连接，那么内存资源也被占用无法得到释放
      - 针对请求方是服务端：端口消耗不大，但是**连接是由线程负责的**，如果存在很多TIME_WAIT的TCP连接，那么线程池消耗太大，系统资源被占满
    - 2. 端口资源占用，一个TCP连接消耗一个本地端口——容易占满端口，导致无法建立新的连接
      - 针对请求方是客户端：端口占用过多，而端口是有限的，无法建立新的连接
  - 优化：
    - 1. 打开net.ipv4.tcp_tw_reuse 和 net.ipv4.tcp_timestamps
    - 2. net.ipv4.tcp_max_tw_buckets
    - 3. 使用SO_LINGER，强制使用RST关闭
- **连接后，客户端出现故障的处理**
  - TCP有一个**保活机制**
  - 保活机制的概念：
    - 设定一个时间段，在该时间段内，如果没有任何连接相关的活动，那么TCP保活机制开始工作：每隔一个时间段，发送一个**探测报文**，如果连续几个探测报文都没有得到响应，那么就认为该TCP连接发生故障，系统内核将错误信息通知给上层应用
    - Linux内核有对应的配置设置了保活时间、保活的探测次数、保活探测的时间间隔，默认是：
      - net.ipv4.tcp_keepalive_time=7200 ——7200s;
        net.ipv4.tcp_keepalive_intvl=75——75s时间间隔;
        net.ipv4.tcp_keepalive_probes=9——9次无响应，则认为是不可达的
    - ——所以，2h11min15s可以确认一个TCP连接已经死亡
  - 所以激活保活机制后会出现如下情况

    - 1. 如果端程序正常：在发送探测报文时，是能够得到响应，那么TCP的保活时间会被重置：
    - 2. 如果是端程序崩溃后重启：探测报文能够得到响应，但是没有该TCP连接的有效信息，会产生一个RST报文，该中断终止（重启之后，所有信息，包括连接已经全部重置了）
    - 3. 如果端程序奔溃，或者报文不可达等特殊情况：探测报文没有得到响应，超过探测的次数后，就会认为该TCP连接已经终止

- **TCP报文的生成**
    - TCP协议中有两个端口：
        - 1. 浏览器监听的端口：随机生成的
        - 2. Web服务器监听的端口：固定的，HTTP默认是80；HTTPS默认是443
    - TCP报文：TCP头部 + TCP数据（HTTP的头部 + 数据）——如果不需要分片（如果需要对TCP数据分片，那么只有第一个TCP的数据中包含HTTP头部，其余TCP数据报不包含头部）



- **Socket编程（暂时不了解）**