# 操作系统

## 进程

**1、进程的概念：**

是静态可执行文件加载到内存中，cpu执行内存中的指令。资源调度的基本单位

**2、进程的管理：**

os维护进程表，表项PCB，PCB包含进程描述信息（进程/用户标识符），控制信息（状态、优先级）、资源分配清单（内存、文件列表、IO设备）、CPU信息；通过链表组织起来。

**3、进程的状态：**

就绪、运行、阻塞、创建、结束、就绪挂起、阻塞挂起

**4、和线程对比：**

虚拟地址切换较慢，线程切换不涉及，进程切换导致TLB（快表失效），命中率降低，线程不会导致失效

**5、进程的种类：**

- 守护进程：web服务器
    - 程序在后台运行，`fork()`一个新的子进程
    - 调用`setsid()`创建一个新的对话期。新的子进程成为新的会话组长和进程组长，并摆脱父进程的影响。
    - 禁止进程重新打开控制终端。通过`fork()`再次创建新的子进程，使调用的fork进程退出
    - 关闭文件描述符
    - 将当前目录更新位根目录
    - 使用unmask将屏蔽字清零
    - 处理SIGCHLD信号。比如将信号SIGCHLD设为SIG_IGN，这样子进程结束不会产生僵尸进程
- 僵尸进程
    - 子进程退出后，父进程还在运行，子进程就成了僵尸进程。目的是为了维护子进程信息，方便主进程通过wait或者waitpid获取。但是僵尸进程会占用内核资源。、所以通过执行非阻塞可以提高程序效率，或者通过设置SIG_IGN（忽略信号）表示内核对子进程结束不关心，由内核回收，主进程就不用管了，正常结束就行了。

**6、多进程问题：**

代码段，堆栈端，数据段。代码段多个进程共享

父子进程除了pid都一样

父子进程共享全部数据，但是子进程写数据会采用写时复制，不是对同一块数据操作

调用execv()可以加载新的代码段，与父进程独立开

**7、进程调度：**

- FCFS：非抢占式，利长作业
- 最短优先：非抢占式。利于短作业，长作业会“饿”’
- 最短剩余时间：
- 时间片：
- 优先级调度：
- 多级队列：时间片+优先级
- 最短进程优先：

**8、进程通信**

- 同一主机
  - 管道：
    - 无名pipe

      半双工、先入先出、无格式、只存在内存、读数据是一次性操作、没有名字之存在于亲缘进程间，存在阻塞。pipe函数、读端和写端、
    - 有名fifo

      有名字非亲缘进程也可以访问、有具体文件、mkfifo fifo创建
  - 信号：
    - 软中断
    - 异步通信
    - 用户和内核空间进程交互
    - 硬件异常或者软件异常，调用系统函数，按下终端键（crtl+C）、kill命令
    - 编号，名称，事件，执行动作
  - 消息队列：
  - 共享存储映射：
    - 最快，不需要数据拷贝，直接读写内存
    - 使得磁盘文件和存储空间的缓冲区映射
    - 不适用read和write，使用指针完成IO操作。
    - mmap、munmap函数
    - 零拷贝
- 不同主机
  - socket

# 线程

轻量级进程，也有PCB、创建线程使用的底层函数和进程一样都是clone

最小执行单位

clone复制对方地址空间就是进程，共享对方地址空间就是线程

linux内核不区分进程和线程，只在用户面区分，所以线程函数都是库函数，不是系统调用

**三种线程：**

- 用户线程：用户空间的线程
  - 不由OS调度，一旦阻塞，此进程下的用户线程都无法运行
  - 用户线程一旦执行，不会被别的用户线程打断。因为OS不会参与用户线程调度
- 内核线程：
  - os调度
  - 内核线程阻塞，不会影响别的线程
  - 占用内核资源、开销较大
- 轻量级LWP
  - 内核支持的用户线程，向普通进程一样调度，类似进程中的执行线程
  - 实际的用户线程是运行再LWP之上的

**资源：**

线程共享的资源：文件描述符表、每种信号处理方式、工作目录、用户组iD和组id

非共享资源：线程id、处理线程和栈指针、栈空间、erro变量、信号屏蔽字、调度优先级

**优缺点**

优点：提高并发性、开销小、数据通信，共享数据方便

缺点：库函数，不稳定、调试困难、对信号支持不好

创建快、切换快、终止快、通信快

多线程的好处：开销小、IO密集型、并发

# 协程

# 互斥同步

**互斥锁**

两种状态：枷锁和解锁

**死锁：**

- 必要条件：互斥、占有和等待、不可抢占、环路
- 处理：
    - 鸵鸟：当死锁发生概率低并且影响小
    - 检测和恢复：
        - 检测算法：有向图是否有环
        - 恢复：抢占、回滚、杀死进程
    - 预防
        - 破坏互斥条件、破坏占有等待、破坏不可抢占、破坏环路等待
    - 避免：
        - 安全状态、银行家算法

**读写锁：**

允许多读，不允许多写。

**条件变量：**

用来等待的而不是上锁的。

- mutex在消费者之间也会竞争，使用条件变量，只有在生产者完成生产消费者才会竞争

信号量：控制对公共资源的访问、PV原语，P是-1V是+；

**管程**

# 存储系统

**层次结构：**

- 寄存器：最快，半个cpu时钟周期完成读写。
- cache：SRAM（静态随机存储器）、分三层
- 内存：DRAM（动态随机存储器），更便宜，电容定时刷新，速度在200左右时钟周期
- 外存：固体硬盘>机械硬盘（物理读取）

**页面置换算法：**

- 最佳页面置换OPT：无法实现，作为衡量标准
- 先进先出：
- 最近最久未使用LRU
- 时钟页面置换：环形链表，遍历时为1则值为0，为0则置换出去
- 最不常用：需要统计页面访问次数，额外开销

**分段：**

将表分段，一个段构成一个独立的地址空间，长度可以不同，并且可以动态增长

分页用于实现虚拟空间，获得更大的地址空间；分段是为了程序的独立有利于共分享和保护

- 纯分段：有利于几个进程间共享数据、每个段独立增长，不会影响其他段（保护）
- 分段+分页：先分段，段上分页：既共享个保护又又分页的虚拟内存功能
- 比较
    - 分页对程序员透明，分段需要程序员显示操作
    - 维度：分页一维、分段二维
    - 大小：分页可改、分段不行

**虚拟内存（分页）：**

应用程序以为的连续内存，实际上是多个物理页

通过硬件异常、硬件地址翻译、主存、磁盘和内核软件共同完成

看起来足够大、独立所以可以简化程序连接装在和内存分配过程、隔离对物理内存的访问权限更安全

- 加速分页
    - TLB加速分页：TLB将虚拟地址和物理地址直接映射。硬件进行匹配，如果匹配到就不用访问页表，没匹配到就查询页表并更新到TLB
    - 软件TLB管理
- 多级页表、倒排页表
- 高速缓存：
- 内存保护：通过页表中页表条目的一些标志位实现对虚拟页的访问控制权限
- 内存管理：

# 文件系统

**磁盘调度算法：**

- FCFS
- SSTF：寻道时间最短，造成饥饿
- SCAN：类似电梯

**中断处理：**

# IO复用

IO操作就是针对内存而言，在运行代码的过程中，内存对文件的读写操作。可以分为网络IO和磁盘IO。

IO操作一般分为两步：1、等待数据准备好。2、从内核向进程复制数据。例如：等待数据从网络中到达后复制到内核的缓冲区；然后将数据从缓冲区输入到应用进程。

## 五种IO模型

1. 阻塞式IO：进程或者线程等待某个条件，如果条件不满足就一直等下去。条件满足就进行下一步操作。应用进程会阻塞在系统调用recvfrom。

   优点：设备文件不可操作时，可以进入休眠状态，将cpu资源让出去；当可以操作时就唤醒进程；

   缺点：耗费时间，适合并发低，时效性低的情况

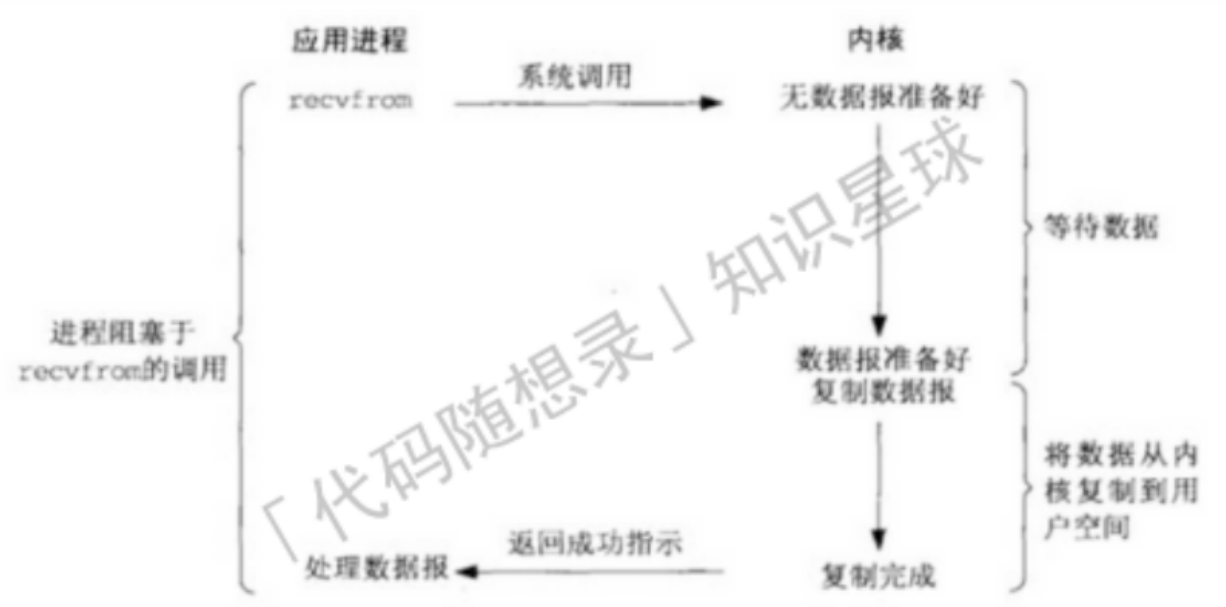




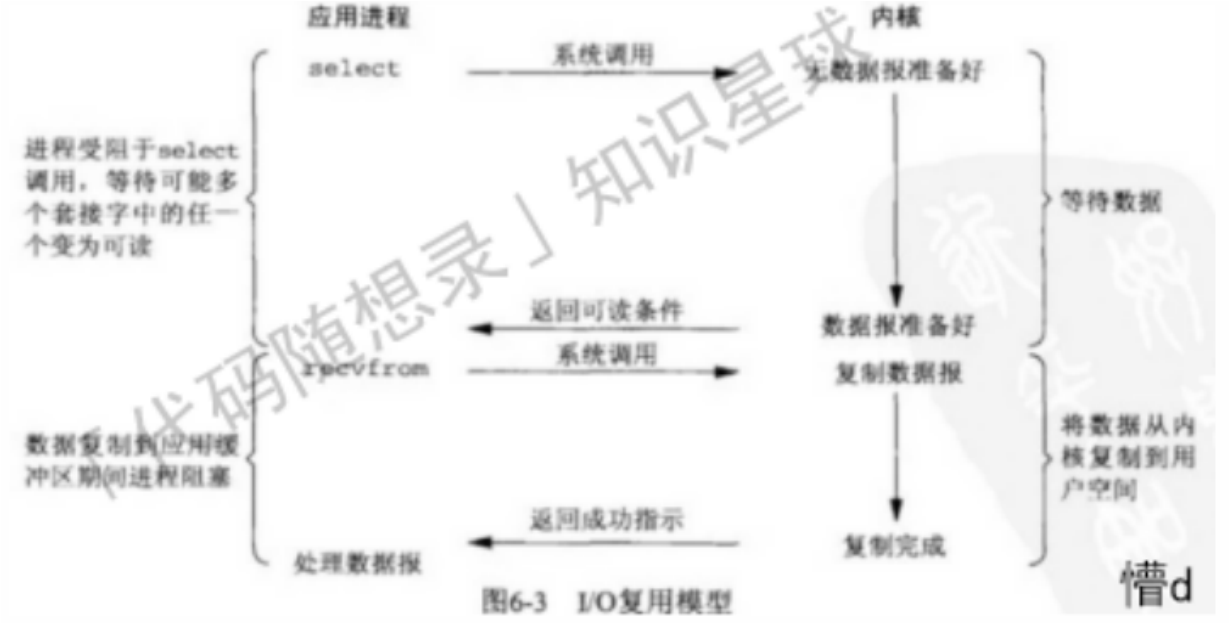
图6-1 阻塞式I/O模型

2. 非阻塞式IO：应用进程和内核交互，数据未准备好的时候，不会阻塞等待唤醒，而是不停的轮询recvfrom，如果数据准备好，就复制数据到进程空间。

缺点：轮询操作是系统调用，不停轮询会耗费大量的cpu时间。

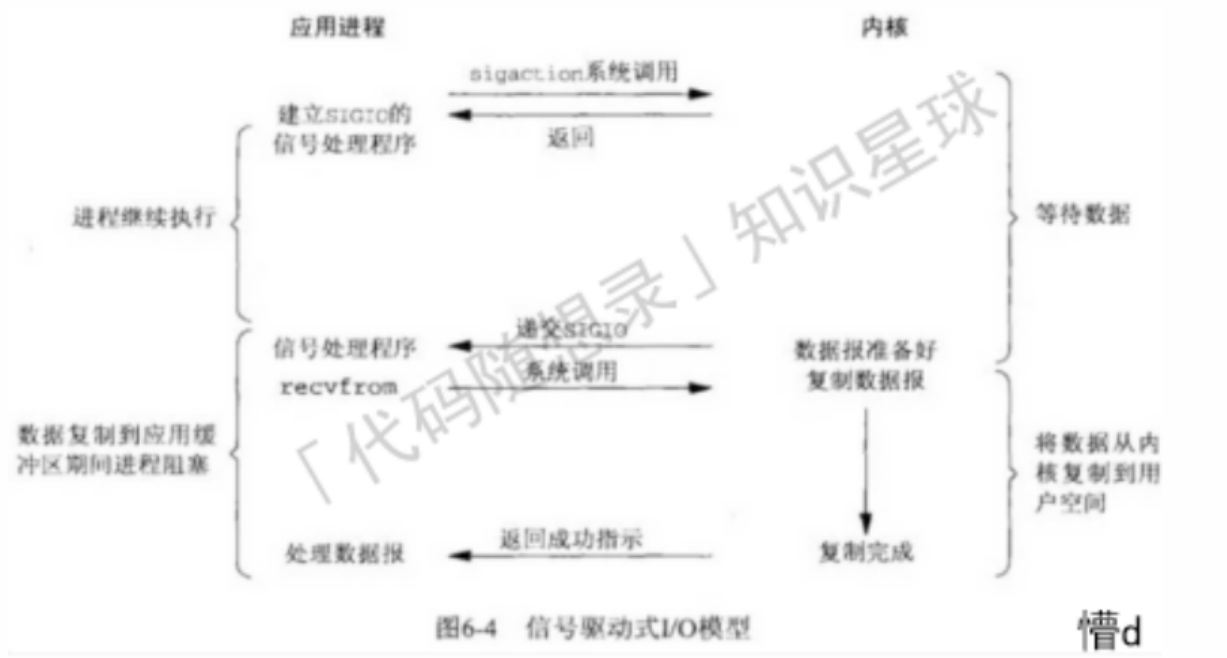

3. IO复用：通过调用select或者poll，阻塞在着三个系统调用中的一个，而不是阻塞在recvform系统调用上。当select监视的文件描述符fd返回可读时，再调用recvfrom将数据读到进程空间。
    - 如果所有监听的fd都为准备好，就阻塞
    - 任意一个fd准备好，select调用返回
    - 用户进程通过recvfrom进行数据拷贝

    优点：不会一直轮询，释放了cpu资源；可以同时监听多个描述符。

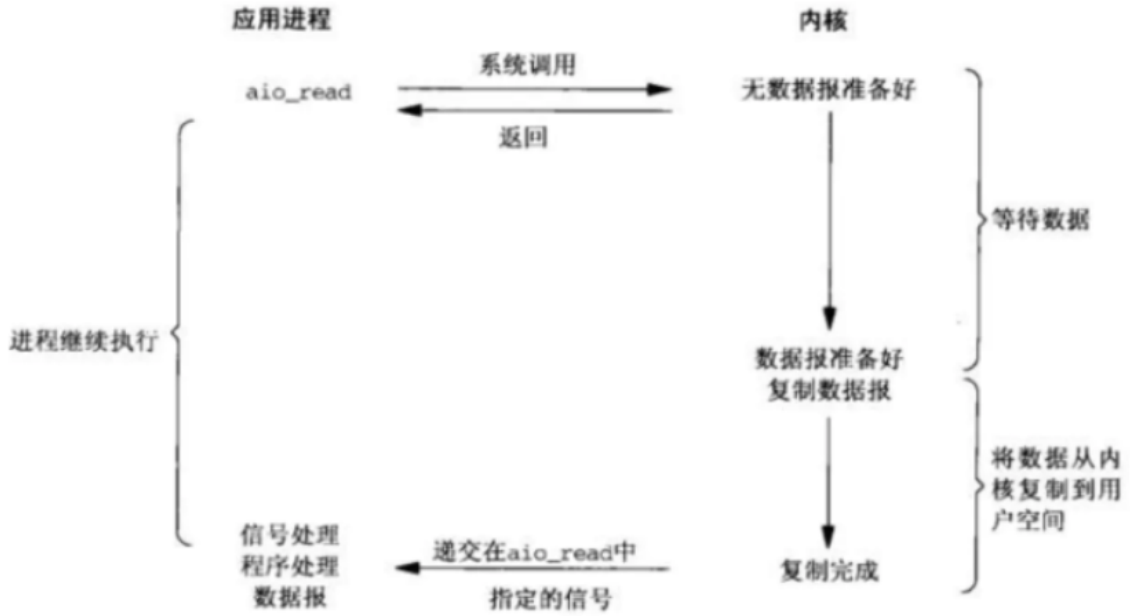



4. 信号驱动IO：开启套接字信号驱动功能，通过sigaction系统调用安装一个信号处理函数，该函数立即返回，不阻塞；数据报准备好后，内核为该进程产生一个sigio信号交给进程；然后信号处理函数调用recvfrom读取数据。

优点：等待数据等待期间不会阻塞，进程可以继续执行等待信号到来；




5. 异步IO：用户进程通知内核启动某个操作，并由内核完成后通知用户进程。也叫事件驱动IO。信号驱动IO是内核通知可以执行IO操作了，异步IO是内核通知IO操作何时完成。




6. 关于**epoll**是异步还是同步：原理上是同步，但是在实际使用中，实现了和异步一样的效率，和异步是一样的。同步和异步最大的区别就在于这两个阶段是否有一个或者全部阻塞。因为虽然使用**epoll**的程序也会阻塞在**epoll**处，但是在返回可读条件后，进程调用**read**即**recvfrom**的时候不会阻塞在复制数据处，因为**mmap**技术，用户空间和内核空间实现了共享，所以调用**read**后可以直接返回不会阻塞。 但是，epoll是事件驱动机制的，从这个角度上来说，epoll也是异步的。

## poll/epoll/select

select:两次遍历+两次拷贝

- 将已连接的socket放在一个文件描述符集合中，调用select函数将文件描述符拷贝到内核，内核检查是否有网络事件发生
- 遍历，有事件就将改socket置为读/写，然后再拷贝回用户空间
- 用户再遍历处理刚刚标记的socket

poll：动态数组，以链表的形式来组织，相比于select，没有文件描述符个数的限制，当然也会收到系统文件描述符的限制

epoll：红黑树

- 内核里面使用红黑树跟踪进程待检测的文件描述符
- 调用epoll_ctl()，将需要监控的socket加入到内核的红黑树中，每次只需要传入一个待检测的sockey，减少了内核和用户空间大量的拷贝和内存分配
- 使用事件驱动机制，内核维护了一个链表来记录就绪事件。当某个socket有时间发生时，通过回调函数，内核会将其加入到这个就绪事件列表中
- 当用户调用epoll_wait的时候，只会返回有事件发生的文件描述符的个数，不需要向另外两个函数轮询
- 两种触发模式
    - ET：当被监控的Scoket描述符上有事件发生，服务器只从epoll_waitr苏醒一次，因此程序需要一次将数据读完，读到EGAIN。只触发一次
    - LT：当被监控的Scoket描述符上有事件发生，服务器不断从epoll_waitr苏醒，直到缓冲区数据读完。
    - ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比**LT**模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。
    - 使用ET的例子:nginx
      使用LT的例子:redis

select和epoll区别：

- 时间复杂度：select和poll采用轮询方式检查就绪事件，复杂度**O**（**n**）epoll采用回调方式检测就绪时间，只返回有时间发生的文件描述符的个数，复杂度**O(1)**
- 工作模式：select工作在**LT**模式，**epoll**可以工作在**ET**模式
- 操作系统：epoll是linux特有的，select是os都有的
- 描述符数量：select单个进程监视的文件描述符有限，64位是2048个；epoll没有最大并发连接的限制，远大于2048
- 消息传递：select需要将消息传递到用户空间，需要拷贝；epoll通过共享内存实现。

# 网络

## OSI七层模型参考

1、物理层：主要定义物理设备标准，如网线的接又类型、光纤的接又类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由 1、0 转化为电流强弱来进行传输，到达目的地后再转化为 1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。

2、数据链路层：建立逻辑连接、进行硬件地址寻址、差错校验等功能。定义了如何让格式化数据以帧为单位进行传输，以及如何控制对物理介质的访问。将比特组合成字节进而组合成帧，用 MAC 地址访问介质。

3、网络层：进行逻辑地址寻址，在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

4、传输层：定义了一些传输数据的协议和端又号（ WWW 端又 80 等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与 TCP 特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如 QQ 聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。

5、会话层：通过传输层（端又号：传输端又与接收端又）建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求。

6、表示层：数据的表示、安全、压缩。主要是进行对接收的数据进行解释、加密与解密、压缩与解压缩等（也就是把计算机能够识别的东西转换成人能够能识别的东西（如图片、声音等）。

7、应用层：网络服务与最终用户的一个接又。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

# 应用层

### HTTP

### HTTP是什么，有什么特点

超文本传输协议（HTTP）是万维网的基础，用于通过超文本链接加载网页。HTTP是应用层协议的一种，旨在在联网设备之间传输信息，并在网络协议堆栈的其他层之上运行。HTTP是基于TCP / IP的通信协议，默认端口是TCP 80，但也可以使用其他端口。

- **HTTP** 是无连接的： HTTP客户端，即浏览器发出请求后，客户端等待响应。服务器处理该请求并发送回响应，然后客户端断开连接。客户端和服务器仅在当次请求中互相了解，至于上一次是否有连接或者连接的信息是无从得知的。

- **HTTP**是独立于媒体的： 这意味着，只要客户端和服务器都知道如何处理数据内容，任何类型的数据都可以通过HTTP发送。客户端和服务器都需要使用适当的 MIME 类型 指定内容类型。
- **HTTP**是无状态的： 如上所述，HTTP 是无连接的，这是 HTTP 是无状态协议的直接结果。服务器和客户端仅在当前请求期间彼此知道，之后他们俩彼此忘记。由于协议的这种性质，客户端和浏览器都无法在整个网页的不同请求之间保留信息。

## HTTP请求过程

1. 首先，我们在浏览器地址栏中，输入要查找页面的URL，按下Enter
2. 浏览器依次在 浏览器缓存 -->>系统缓存 -->>路由器缓存中去寻找匹配的URL，若有，就会直接在屏幕中显示出页面内容。若没有，则跳到第三步操作
3. 发送HTTP请求前，浏览器需要先进行域名解析(即DNS解析)，以获取相应的IP地址;（浏览器DNS缓存、路由器缓存、DNS缓存）
4. 获取到IP地址之后，浏览器向服务器发起TCP连接，与浏览器建立TCP三次握手
5. 握手成功之后，浏览器就会向服务器发送HTTP请求，来请求服务器端的数据包
6. 服务器处理从浏览器端收到的请求，接着将数据返回给浏览器
7. 浏览器收到HTTP响应
8. 查询状态，状态成功则进行下一步，不成功则弹出相应指示
9. 再读取页面内容、进行浏览器渲染、解析HTML源码;（生成DOM树、解析CCS样式、处理JS交互，客户端和服务器交互）进行展示
10. 关闭TCP连接（四次挥手）

## HTTP报文格式

### 请求报文

第一行是包含了请求方法、URL、协议版本;

接下来的多行都是请求首部 Header，每个首部都有一个首部名称，以及对应的值。

一个空行用来分隔首部和内容主体 Body

最后是请求的内容主体

```
GET http://www.example.com/ HTTP/1.1
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,
application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Cache-Control: max-age=0
Host: www.example.com
If-Modified-Since: Thu, 17 Oct 2019 07:18:26 GMT
If-None-Match: "3147526947+gzip"
Proxy-Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 xxx

param1=1&param2=2
```

**响应报文**

第一行包含协议版本、状态码以及描述，最常见的是 200 OK 表示请求成功了

接下来多行也是首部内容

一个空行分隔首部和内容主体

最后是响应的内容主体

```
HTTP/1.1 200 OK
Age: 529651
Cache-Control: max-age=604800
Connection: keep-alive
Content-Encoding: gzip
Content-Length: 648
Content-Type: text/html; charset=UTF-8
Date: Mon, 02 Nov 2020 17:53:39 GMT
Etag: "3147526947+ident+gzip"
Expires: Mon, 09 Nov 2020 17:53:39 GMT
Keep-Alive: timeout=4
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Proxy-Connection: keep-alive
Server: ECS (sjc/16DF)
Vary: Accept-Encoding
X-Cache: HIT

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
 // 省略...
</body>
</html>
```

## HTTP状态码

| | | |
|---|---|---|
| 1×× | 提示信息，表示目前是协议处理的中间状态，还需要后续的操作； | |
| 2×× | 成功，报文已经收到并被正确处理； | 200、204、206 |
| 3×× | 重定向，资源位置发生变动，需要客户端重新发送请求； | 301、302、304 |
| 4×× | 客户端错误，请求报文有误，服务器无法处理； | 400、403、404 |
| 5×× | 服务器错误，服务器在处理请求时内部发生了错误。 | 500、501、502、503 |

200：客户端请求成功

206：partial content 服务器已经正确处理部分GET请求，实现断点续传或同时分片下载，该请求必须包含Range请求头来指示客户端期望得到的范围

301（永久重定向）：该资源已被永久移动到新位置，将来任何对该资源的访问都要使用本响应返回的若干个URL之一

302（临时重定向）：请求的资源现在临时从不同的URI中获得

304：如果客户端发送一个待条件的GET请求并且该请求以经被允许，而文档内容未被改变，则返回304,该响应不包含包体（即可直接使用缓存）

400：请求报文语法有误，服务器无法识别

401：请求需要认证

403：请求的对应资源禁止被访问

404：服务器无法找到对应资源

500：服务器内部错误

503：服务器正忙

## HTTP请求方法

1. GET：申请获取资源，不对服务器产生影响
2. POST：客户端向服务器提交数据。会影响服务器，服务器可能动态创建新的资源或更新原有资源
3. HEAD：类似GET，仅要求服务器返回头部信息
4. PUT：上传某个资源
5. DELETE：删除某个资源
6. TRACE：用于测试。要求目标服务器返回原始的HTTP请求内容
7. CONNECT：用于代理服务器
8. OPTION：查询服务器对特定URL支持的请求方法

### GET和POST的区别：

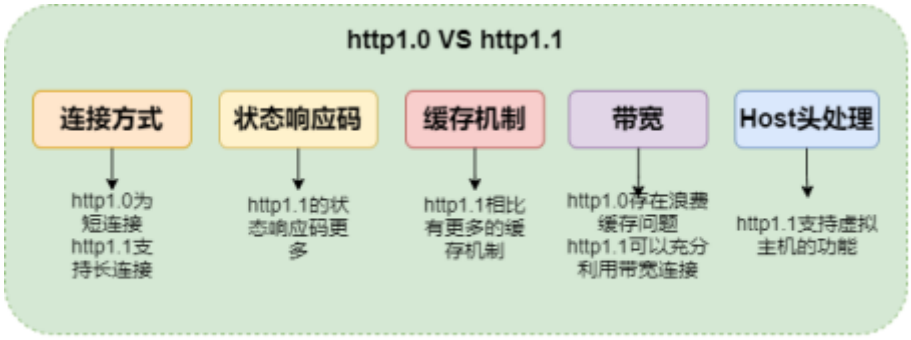
**1、GET**：

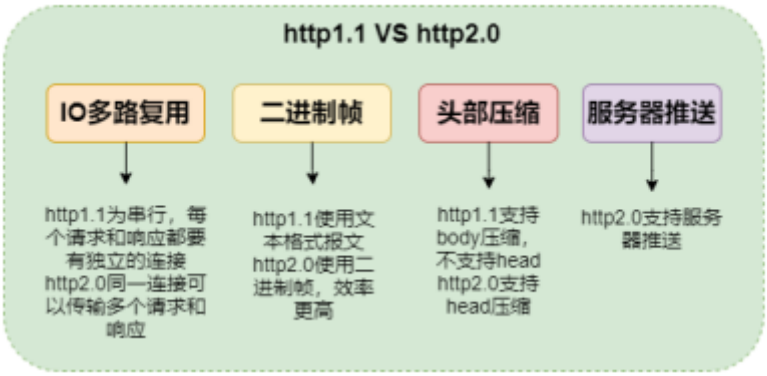
- 申请获取资源，不对服务器产生影响
- 请求行中请求方法为get，URL不为空。GET的URL一般都有长度限制，但需注意HTTP协议中并未规定GET请求的长度。这个长度限制主要是由浏览器和Web服务器所决定的，并且各个浏览器对长度的限制也各不相同
- GET方法只产生一个TCP数据包，浏览器会把请求头和请求数据一并发送出去，服务器响应200 ok（返回数据）。
- 幂等（多次调用请求会产生相同的结果）

**2、POST**：

- 客户端向服务器提交数据，会影响服务器，服务器可能动态创建新的资源或更新原有资源（表单）
- 请求行中请求方法为POST，URL为空，它的请求参数是位于请求数据中，其请求信息没有长度限制
- POST会产生两个TCP数据包，浏览器会先将请求头发送给服务器，待服务器响应100 continue，浏览器再发送请求数据，服务器响应200 ok（返回数据）
- 非幂等（多次调用请求会产生相同的结果）

## HTTP各版本比较




### HTTP1.1版本新特性

**1**、默认持久连接

只要客户端任意一端没有明确提出断开TCP连接，就一直保持连接，可以发送多次HTTP请求

HTTP1.0默认使用短连接，而HTTP1.1默认使用长链接

**2**、管线化

客户端可以同时发送多个HTTP请求，不用等待响应

**3**、断点续传

利用HTTP消息头使用分块传输编码，将实体主体进行分块传输

### HTTP2.0版本新特性

**1、传输格式变化，采用了新的二进制格式**

HTTP1.X的解析都是基于文本，文本的表现形式多样，不利于健壮性考虑

HTTP2.0采用二进制，只认0/1组合，实现更加快的方法，健壮性更加完善

**2、多路复用、并发传输**

HTTP1.1 基于请求-响应模型。同一个连接种，HTTP完成一个事务（请求与响应），才能处理下一个事务。即：再发出请求等待响应的过程种是没办法做其他事情的，会造成【队头阻塞】问题。

HTTP2通过Stream这个设计（多个Stream复用一条TCP连接，达到并发的效果），解决了【队头阻塞】的问题，提高了HTTP传输的吞吐量。

- 1 个 TCP 连接包含一个或者多个 Stream，Stream 是 HTTP/2 并发的关键技术；
- Stream 里可以包含 1 个或多个 Message，Message 对应 HTTP/1 中的请求或响应，由 HTTP 头部和包体构成；
- Message 里包含一条或者多个 Frame，Frame 是 HTTP/2 最小单位，以二进制压缩格式存放 HTTP/1 中的内容（头部和包体）；

在HTTP2连接上，不同Stream的帧可以乱序发送（因此可以并发不同的Stream），接收端可以通过Stream ID 有序组装HTTP消息。

HTTP/2 通过 Stream 实现的并发，比 HTTP/1.1 通过 TCP 连接实现并发要牛逼的多，**因为当 HTTP/2 实现 100 个并发 Stream 时，只需要建立一次 TCP 连接，而 HTTP/1.1 需要建立 100 个 TCP 连接，每个 TCP 连接都要经过 TCP 握手、慢启动以及 TLS 握手过程，这些都是很耗时的。**

HTTP/2 还可以对每个 Stream 设置不同**优先级**，帧头中的「标志位」可以设置优先级，比如客户端访问 HTML/CSS 和图片资源时，希望服务器先传递 HTML/CSS，再传图片，那么就可以通过设置 Stream 的优先级来实现，以此提高用户体验。

**3、header压缩： 静态表、动态表、Huffman编码共同完成**

在HTTP1.X中，header带有大量信息，而且每次都要重复发送

HTTP2.0通过encoder减少header大小，通讯双方会各自缓存一份header字段表

既可以避免重复**header**传输，又减小了需要传输的大小

**4、服务端推送**

把客户端所需要的资源伴随着index.html一起发送到客户端，省去了客户端重复请求的步骤因为没有发起请求，建立连接等操作，所以静态资源通过服务器推送，可以极大的提升速度

客户端发起的请求，必须使用的是奇数号 Stream，服务器主动的推送，使用的是偶数号 Stream。服务器在推送资源时，会通过 `PUSH_PROMISE` 帧传输 HTTP 头部，并通过帧中的 `Promised Stream ID` 字段告知客户端，接下来会在哪个偶数号 Stream 中发送包体。

**5、缺点：**

- 队头阻塞

TCP 是字节流协议，TCP 层必须保证收到的字节数据是完整且有序的，如果序列号较低的 TCP 段在网络传输中丢失了，即使序列号较高的 TCP 段已经被接收了，应用层也无法从内核中读取到这部分数据，从 HTTP 视角看，就是请求被阻塞了。

- **TCP**和**TLS**握手延迟

  发出HTTP请求时，需要经过TCP三次握手和TLS四次握手，共计3RTT的时延才能发出请求数据。

- 网络迁移需要重新连接

  一个TCP连接由【源IP地址，源端又，目标IP地址，目标端又】确定。若IP地址或端又发生变换，这需要重新进行连接。这不利于移动设备切换网络的场景。要解决该问题，就要修改传输层协议。在HTTP3中传输层协议修改为了UDP。

**HTTP/2 是基于 TCP 协议来传输数据的，TCP 是字节流协议，TCP 层必须保证收到的字节数据是完整且连续的，这样内核才会将缓冲区里的数据返回给 HTTP 应用，那么当「前 1 个字节数据」没有到达时，后收到的字节数据只能存放在内核缓冲区里，只有等到这 1 个字节数据到达时，HTTP/2 应用层才能从内核中拿到数据，这就是 HTTP/2 队头阻塞问题。**

**有没有什么解决方案呢？既然是 TCP 协议自身的问题，那干脆放弃 TCP 协议，转而使用 UDP 协议作为传输层协议，这个大胆的决定， HTTP/3 协议做了！**

**HTTP3.0版本**

**1、*QUIC***

UDP是一个简单的、不可靠的传输协议，而且UDP包之间是无序的，也没有依赖关系。UDP也不需要连接。HTTP3基于UDP协议在 应用层 实现了QUIC协议，它有类似TCP的连接管理、拥塞窗口、流量控制的网络特性，相当于将不可靠的UDP协议变成可靠的了，无需担心数据包丢包的问题。

**2、连接建立：**

对于 HTTP/1 和 HTTP/2 协议，TCP 和 TLS 是分层的，分别属于内核实现的传输层、openssl 库实现的表示层，因此它们难以合并在一起，需要分批次来握手，先 TCP 握手，再 TLS 握手。

HTTP/3 在传输数据前虽然需要 QUIC 协议握手，这个握手过程只需要 1 RTT，握手的目的是为确认双方的「连接ID」，连接迁移就是基于连接 ID 实现的。但是 HTTP/3 的 QUIC 协议并不是与 TLS 分层，而是QUIC 内部包含了 TLS，它在自己的帧会携带 TLS 里的“记录”，再加上 QUIC 使用的是 TLS1.3，因此仅需 1 个 RTT 就可以「同时」完成建立连接与密钥协商，甚至在第二次连接的时候，应用数据包可以和 QUIC 握手信息（连接信息 + TLS 信息）一起发送，达到 0-RTT 的效果。

### 3、连接迁移

在前面我们提到，基于 TCP 传输协议的 HTTP 协议，由于是通过四元组（源 IP、源端口、目的 IP、目的端口）确定一条 TCP 连接，那么当移动设备的网络从 4G 切换到 WIFI 时，意味着 IP 地址变化了，那么就必须要断开连接，然后重新建立连接，而建立连接的过程包含 TCP 三次握手和 TLS 四次握手的时延，以及 TCP 慢启动的减速过程，给用户的感觉就是网络突然卡顿了一下，因此连接的迁移成本是很高的。

而 QUIC 协议没有用四元组的方式来“绑定”连接，而是通过**连接 ID**来标记通信的两个端点，客户端和服务器可以各自选择一组 ID 来标记自己，因此即使移动设备的网络变化后，导致 IP 地址变化了，只要仍保有上下文信息（比如连接 ID、TLS 密钥等），就可以“无缝”地复用原连接，消除重连的成本，没有丝毫卡顿感，达到了**连接迁移**的功能。

## Cookie和Session区别

### 一、Cookie

**1**、作用

1. 在第一次登录服务器之后，返回一些数据（cookie）给浏览器
2. 浏览器将数据保存在本地
3. 两次发送请求时，自动把上一次请求存储的cookie发送给服务器
4. 服务器通过该数据判断用户

**2**、特点：可存储的数据量有限，一般不会超过4KB，Cookie的保存形式分为会话Cookie和持久Cookie

### 二、Session

session的作用与cookie类似，都是为了存储用户相关的信息

**1**、区别

1. cookie存储在本地浏览器的数据
2. session存储在服务器的数据
3. Cookie存储数据的大小有限制，而Session一般无限制
4. Cookie对用户信息的生命周期的控制方式为累计方式，而Session使用间隔方式

**2**、优势：数据存储在服务器更加的安全

**3**、缺陷：会占用服务器资源

## 三、SSO

单点登录（英语：**Single sign-on**，缩写为 **SSO**），在一个多系统的环境中，用户只需要登录一次，就可以同时登陆访问其他互相信任的系统。

**SSO**的优点：

- 降低访问第三方网站风险（用户密码不存储或外部管理）；
- 从不同的用户名和密码的组合减少密码疲劳；
- 减少花费的时间重新输入密码相同的身份；
- 降低IT成本适当降低一些IT帮助台调用有关密码；
- SSO集中的所有其他应用程序和系统，用于身份验证服务器的身份验证，并与技术相结合是为了确保用户不必主动输入凭据一次以上。

## HTTP如何禁用缓存？如何确认缓存？

HTTP/1.1 通过 Cache-Control 首部字段来控制缓存。

（**1**）禁止进行缓存

no-store 指令规定不能对请求或响应的任何一部分进行缓存。

（**2**）强制确认缓存

no-cache 指令规定缓存服务器需要先向源服务器验证缓存资源的有效性，只有当缓存资源有效时才能使用该缓存对客户端的请求进行响应。

Cache-Control: no-store [禁止进行缓存]/no-cache [强制确认缓存]

## HTTP长连接

**1**、重用连接的机制

**2**、使用方法

在请求头中加入**Connecttion**：keep-alive，通知对应在该请求响应完成之后不要关闭，下一次继续用

问：HTTP能不能一次连接多次请求，不等后端返回?

答：可以，HTTP本质是使用socket连接，写入TCP缓冲是可以连接多次的

**HTTP**无状态：

即使第一次和服务器连接并且登录成功之后，第二次请求服务器仍然不知道当前请求的是哪个用户

# HTTPS

## 特点

1. 信息加密：交互信息无法被窃取
2. 校验机制：无法篡改通信内容，篡改了就不能正常显示
3. 身份证书：证明报文的完整

## 优点

1. 在数据传输过程中，使用秘钥加密，安全性更高
2. 可认证用户和服务器，确保数据发送到正确的用户和服务器

## 缺点

1. 握手阶段延时较高：在会话前还需进行SSL握手
2. 部署成本高：需要购买CA证书；需要加解密计算，占用CPU资源，需要服务器配置或数目高

## 加密方式

**HTTPS**采用对称加密和**非**对称加密结合的**[**混合加密**]方**式。

通信建立前：采用非对称加密的方式交换[会话密钥]，后续不再使用非对称加密;

通信过程中：全部使用对称加密的[会话密钥]方式，加密明文数据。

**1**、对称加密：只使用一个密钥，运算速度快，密钥必须保密，无法做到安全的密钥交换;

**2**、非对称加密：使用两个密钥，公钥可以任意分发而私钥保密，解决密钥交换问题，但速度慢。

**3**、混合加密：实现信息的机密性，解决窃听风险;

## 验证流程（SSL / TLS 握手详细过程）

1. **"client hello"**消息：客户端通过发送"client hello"消息向服务器发起握手请求，该消息包含了客户端所支持的 TLS 版本和密码组合以供服务器进行选择，还有一个"client random"随机字符串。
2. **"server hello"**消息：服务器发送"server hello"消息对客户端进行回应，该消息包含了数字证书，服务器选择的密码组合和"server random"随机字符串。
3. 验证：客户端对服务器发来的证书进行验证，确保对方的合法身份，验证过程可以细化为以下几个步骤：
    1. 检查数字签名
    2. 验证证书链 (这个概念下面会进行说明)
    3. 检查证书的有效期
    4. 检查证书的撤回状态 (撤回代表证书已失效)

4. **"premaster secret"**字符串：客户端向服务器发送另一个随机字符串"premaster secret (预主密钥)"，这个字符串是经过服务器的公钥加密过的，只有对应的私钥才能解密。
5. 使用私钥：服务器使用私钥解密"premaster secret"。
6. 生成共享密钥：客户端和服务器均使用 client random，server random 和 premaster secret，并通过相同的算法生成相同的共享密钥 **KEY**。
7. 客户端就绪：客户端发送经过共享密钥 **KEY**加密过的"finished"信号。
8. 服务器就绪：服务器发送经过共享密钥 **KEY**加密过的"finished"信号。
9. 达成安全通信：握手完成，双方使用对称加密进行安全通信。

**数字证书**

**1**、摘要算法用来实现完整性

**2**、数字签名是私钥对摘要的加密，实现了身份认证和不可抵赖性

**3**、公钥的分发需要使用数字证书，必须由CA的信任链来认证，否则就是不可信的

**4**、证书体系中，**Root CA**最大，也叫根证书。相当于是自己证明自己的，必须相信，如果不相信，这整个信任链就走不下去了。上网的时候只要有证书，就可以顺着一直找到根证书来确定是否可信;

**5**、如果遇到**CA**伪造，就会加这个CA加入CRL证书吊销列表中，终止信任。

**数字证书的获取过程**

1. 根据服务器公钥、证书颁发者、证书用途、过期时间等经哈希计算得到值H1;
2. 将H1由CA私钥加密生成签名数字证书;
3. 客户端收到数字证书后经同样哈希计算得到值H2;
4. 经嵌入在浏览器或者操作系统的CA公钥解密数字证书，对比H1、H2相等验证服务器真实性，取出服务器公钥。

## HTTPS和HTTP的区别

HTTPS= HTTP+SSL（安全套接字）+TLS（安全传输层协议）

1、加密：

- **HTTP**： 以明文的方式在网络中传输数据
- HTTPS 解决HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。

2、端口号：

- HTTP 的端口号是 80，HTTPS 的端口号是 443。

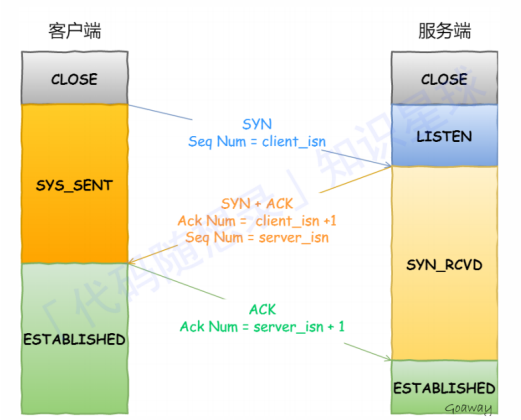
3、可信：

- HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的。

# 传输层

## TCP

### 三次握手




**1**、第一个**SYN**报文：

客户端随机初始化序列号client_isn，放进TCP首部序列号段，然后把SYN置1。把SYN报文发送给服务端，表示发起连接，之后客户端处于SYN-SENT状态。

**2**、第二个报文**SYN+ACK**报文：

服务端收到客户端的SYN报文，把自己的序号server_isn放进TCP首部序列号段，确认应答号填入client_isn + 1，把SYN和ACK置1。把SYN+ACK报文发送给客户端，然后进入SYN-RCVD状态。

**3**、第三个报文**ACK**：

客户端收到服务端报文后，还要向服务端回应最后一个应答报文。首先该应答报文 TCP 首部 ACK 标志位置为 1，其次「确认应答号」字段填入 server_isn + 1，最后把报文发送给服务端，这次报文可以携带客户到服务器的数据，之后客户端处于 ESTABLISHED 状态。服务器收到客户端的应答报文后，也进入 ESTABLISHED 状态。

### 为什么需要三次握手?

1、三次握手才可以阻止重复历史连接的初始化(主因)

当旧的SYN报文先到达服务端，服务端回一个ACK+SYN报文；客户端收到后可以根据自身的上下文，判断这是一个历史连接（序列号过期或超时），那么客户端就会发送 RST报文给服务端，表示中止这一次连接。两次握手在收到服务端的响应后开始发生数据，不能判断当前连接是否是历史连接。

三次握手可以在客户端准备发送第三次报文时，客户端因有足够的上下文来判断当前连接是否是历史连接。

2、三次握手才可以同步双方的初始序列号

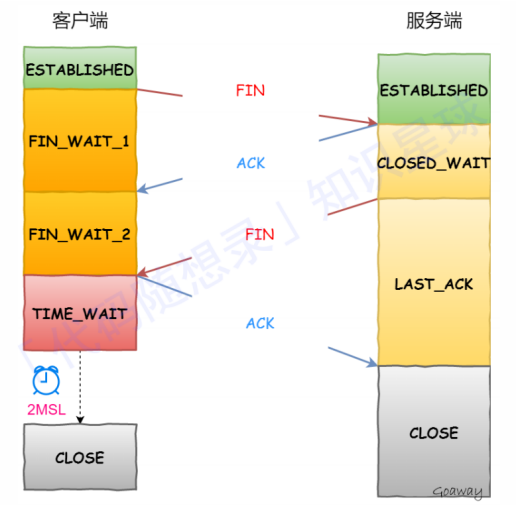
TCP 协议的通信双方， 都必须维护一个「序列号」， 序列号是可靠传输的一个关键因素。接收端可以去除重复数据。接收端可以按照序列号顺序接收。标识发送的数据包，哪些已经被收到。

两次握手只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。

3、三次握手才可以避免资源浪费

- 两次握手会造成消息滞留情况下，服务器重复接受无用的连接请求 SYN 报文，而造成重复分配资源。
- 只有两次握手时，如果客户端的SYN请求连接在网络中阻塞，客户端没有收到服务端的ACK报文，会重新发送SYN。
- 由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 ACK 确认信号，所以每收到一个SYN 就只能先主动建立一个连接。

## 四次挥手



### 1、断开过程

1、假设客户端打算关闭连接，发送一个TCP首部FIN被置1的FIN报文给服务端。

2、服务端收到以后，向客户端发送ACK应答报文。

3、等待服务端处理完数据后，向客户端发送FIN报文。

4、 客户端接收到FIN报文后回一个ACK应答报文。

5、 服务器收到ACK报文后，进入close状态，服务器完成连接关闭。

6、客户端在经过 2MSL 一段时间后，自动进入close状态，客户端也完成连接的关闭。

### 2、为什么挥手需要四次?

关闭连接时，客户端发送FIN报文，表示其不再发送数据，但还可以接收数据。

客户端收到FIN报文，先回一个ACK应答报文，服务端可能还要数据需要处理和发送，等到其不再发送数据时，才发送FIN报文给客户端表示同意关闭连接。

从上面过程可知：

1、服务端通常需要等待完成数据的发送和处理，所以服务端的**ACK**和**FIN**一般都会分开发送，从而比三次握手导致多了一次。

2、延迟确认：即接收方收到包后，如果暂时没有内容回复给发送方，则延迟一段时间再确认，假如在这个时间范围内刚好有数据需要传输，则和确认包一起回复。这种也被称为数据捎带。延迟确认只是减轻网络负担，未必可以提升网络性能，有些情况下反而会影响性能。

### 3、为什么 TIME_WAIT 是 2MSL?

1. MSL是 Maximum Segment Lifetime，报文最大生存时间，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。
2. 等待MSL两倍：网络中可能存在发送方的数据包，当这些发送方的数据包被接收方处理后又会向对方发送响应，所以一来一回需要等待 2 倍的时间。
3. 2MSL 的时间是从客户端接收到 FIN 后发送 ACK 开始计时的。如果在 TIME-WAIT 时间内，因为客户端的 ACK没有传输到服务端，客户端口接收到了服务端重发的 FIN 报文，那么 2MSL 时间将重新计时。

### 4、为什么需要 TIME_WAIT 状态?

主动发起关闭连接的一方，才会有 TIME-WAIT 状态。需要 TIME-WAIT 状态，主要是两个原因：

1. 防止具有相同「四元组」的「旧」数据包被收到
2. 保证「被动关闭连接」的一方能被正确的关闭，即保证最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭。

有相同端又的 TCP 连接被复用后，被延迟的相同四元组的数据包抵达了客户端，那么客户端是有可能正常接收这个过期的报文，这就会产生数据错乱等严重的问题。 经过 2MSL 这个时间，足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。最后的ACK如果丢失，客户端直接进入close，服务端一直在等待ACK状态。当客户端发起建立连接的SYN请求，服务端会发送RST报文回应，连接建立会关闭。

如果 TIME-WAIT 等待足够长的情况就会遇到两种情况：

1. 服务端正常收到四次挥手的最后一个 ACK 报文，则服务端正常关闭连接。
2. 服务端没有收到四次挥手的最后一个 ACK 报文时，则会重发 FIN 关闭连接报文并等待新的 ACK 报文。

**5、TIME_WAIT 过多有什么危害?**

1. 内存资源占用；
2. 对端又资源的占用，一个 TCP 连接至少消耗一个本地端口；

如果发起连接一方的 TIME_WAIT 状态过多，占满了所有端口资源，则会导致无法创建新连接。

## 重传机制

**1、超时重传：**

设定一个计时器，当超过指定的时间后，没有收到对方的确认ACK应答报文，就会重发该数据。

超时重传的两种情况如下：

- 数据包丢失
- 确认应答丢失

**2、快速重传：**

以数据驱动重传：当收到三个相同的ACK报文时，在定时器过期之前，重传丢失的报文段。

- SACK：选择性确认，选择重传哪些报文
- D_SACK：
    - 使用SACK告诉发送方哪些是重复发送的
    - 告诉发送方是发出去的包丢了还是接收方回应的ACK包丢了

## 流量控制（滑动窗口，速度匹配）

### 1、窗口的作用

- 无需等待应答，继续发送数据。
- 只要收到较大的确认ACK，就认为之前的都受到了，累计确认或者累计应答

### 2、窗口大小

接收方的窗口大小决定的,告诉发送方自己还有多少缓冲区可以使用。然后发送方维护一个发送窗口。

## 拥塞控制（拥塞窗口）

拥塞控制通过拥塞窗又来防止过多的数据注入网络，使得网络中的路由器或者链路过载。

拥塞窗口cwnd是发送方维护的一个状态变量，根据网络拥塞程度而变化。

发送窗口的值是swnd = min(cwnd, rwnd)，也就是拥塞窗口和接收窗口中的最小值。

网络中没有出现拥塞，cwnd增大，出现拥塞，cwnd减小。

其实只要发送方没有在规定时间内接收到 ACK 应答报文，也就是发生了超时重传，就会认为网络出现了拥塞。

拥塞控制的常见算法：

1. 慢启动：阈值之前指数增长
2. 拥塞避免：阈值之后线性增长
3. 快速恢复：
    1. 超时重传：阈值减半+慢启动
    2. 快速重传：窗口阈值减半进入快速恢复
        1. 拥塞窗又 cwnd = ssthresh + 3 （ 3 的意思是确认有 3 个数据包被收到了）；
        2. 重传丢失的数据包；
        3. 如果再收到重复的 ACK，那么 cwnd 增加 1；
        4. 如果收到新数据的 ACK 后，把 cwnd 设置为第一步中的 ssthresh 的值，原因是该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态；

## 内核参数优化

### 三次握手性能

| 优化三次握手的策略 | |
|---|---|
| 策略 | TCP 内核参数 |
| 调整 SYN 报文的重传次数 | tcp_syn_retries |
| 调整 SYN 半连接队列长度 | tcp_max_syn_backlog、somaxconn、backlog |
| 调整 SYN+ACK 报文的重传次数 | tcp_synack_retries |
| 调整 accpet 队列长度 | min(backlog, somaxconn) |
| 绕过三次握手 | tcp_fastopen |

1、客户端的优化

当客户端发起SYN包时，可以通过tcp__synb__retries控制其重传的次数。

2、服务端的优化

当服务端 SYN 半连接队列溢出后，会导致后续连接被丢弃，可以通过 `netstat -s` 观察半连接队列溢出的情况，如果 SYN 半连接队列溢出情况比较严重，可以通过 `tcp_max_syn_backlog、somaxconn、backlog` 参数来调整 SYN 半连接队列的大小。

服务端回复 SYN+ACK 的重传次数由 `tcp_synack_retries` 参数控制。如果遭受 SYN 攻击，应把 `tcp_syncookies` 参数设置为 1，表示仅在 SYN 队列满后开启 syncookie 功能，可以保证正常的连接成功建立。

服务端收到客户端返回的 ACK，会把连接移入 accpet 队列，等待进行调用 accpet() 函数取出连接。

可以通过 `ss -lnt` 查看服务端进程的 accept 队列长度，如果 accept 队列溢出，系统默认丢弃 ACK，如果可以把 `tcp_abort_on_overflow` 设置为 1，表示用 RST 通知客户端连接建立失败。

如果 accpet 队列溢出严重，可以通过 listen 函数的 `backlog` 参数和 `somaxconn` 系统参数提高队列大小，accept 队列长度取决于 min(backlog, somaxconn)。

3、绕过三次握手

TCP Fast Open 功能可以绕过三次握手，使得 HTTP 请求减少了 1 个 RTT 的时间，Linux 下可以通过 `tcp_fastopen` 开启该功能，同时必须保证服务端和客户端同时支持。

### 四次挥手性能

关闭连接的方式通常有两种，分别是 RST 报文关闭和 FIN 报文关闭。如果进程异常退出了，内核就会发送 RST 报文来关闭，它可以不走四次挥手流程，是一个暴力关闭连接的方式。

安全关闭连接的方式必须通过四次挥手，它由进程调用 **close** 和 **shutdown** 函数发起 FIN 报文（shutdown 参数须传入 SHUT_WR 或者 SHUT_RDWR 才会发送 FIN）。

调用了 **close** 函数意味着完全断开连接，完全断开不仅指无法接收数据，而且也不能发送数据。了一种优雅关闭连接的 **shutdown** 函数，它可以控制只关闭 一个方向的连接。

- **FIN_WAIT1**状态的优化：降低 tcp_orphan_retries 的值。当重传次数超过 tcp_orphan_retries 时，连接就会直接关闭掉（即：新增的孤儿连接将不再走四次挥手，而是直接发送 RST 复位报文强制关闭）
- **FIN_WAIT2**状态的优化：对于孤儿连接（调用 close 关闭的连接），如果在 60 秒后还没有收到 FIN 报文，连接就会直接关闭。
- **TIME_WAIT**状态的优化：Linux 提供了 tcp_max_tw_buckets 参数，当 TIME_WAIT 的连接数量超过该参数时，新关闭的连接就不再经历TIME_WAIT 而直接关闭。

| 优化四次挥手的策略 | |
|---|---|
| **策略** | **TCP 内核参数** |
| 调整 FIN 报文重传次数 | tcp_orphan_retries |
| 调整 FIN_WAIT2 状态的时间<br>(只适用 close 函数关闭的连接) | tcp_fin_timeout |
| 调整孤儿连接的上限个数<br>(只适用 close 函数关闭的连接) | tcp_max_orphans |
| 调整 time_wait 状态的上限个数 | tcp_max_tw_buckets |
| 复用 time_wait 状态的连接<br>(只适用客户端) | tcp_tw_reuse 、tcp_timestamps |

**数据传输性能**

TCP 连接是由内核维护的，内核会为每个连接建立内存缓冲区：

如果连接的内存配置过小，就无法充分使用网络带宽，TCP 传输效率就会降低；

如果连接的内存配置过大，很容易把服务器资源耗尽，这样就会导致新连接无法建立；

因此，我们必须理解 Linux 下 TCP 内存的用途，才能正确地配置内存大小。

**在高并发服务器中，为了兼顾网速与大量的并发连接，我们应当保证缓冲区的动态调整的最大值达到带宽时延积，而最小值保持默认的 4K 不变即可。而对于内存紧张的服务而言，调低默认值是提高并发的有效手段。**

**同时，如果这是网络 IO 型服务器，那么，调大 tcp_mem 的上限可以让 TCP 连接使用更多的系统内存，这有利于提升并发能力。**需要注意的是，tcp_wmem 和 tcp_rmem 的单位是字节，而 tcp_mem 的单位是页面大小。而且，**千万不要在 socket 上直接设置 SO_SNDBUF 或者 SO_RCVBUF，这样会关闭缓冲区的动态调整功能。**

| 数据传输的优化策略 | |
|---|---|
| **策略** | **TCP 内核参数** |
| 扩大窗口大小 | tcp_window_scaling |
| 调整发送缓冲区范围 | tcp_wmem |
| 调整接收缓冲区范围 | tcp_rmem |
| 打开接收缓冲区动态调节 | tcp_moderate_rcvbuf |
| 调整内存范围 | tcp_mem |

## 抓包分析

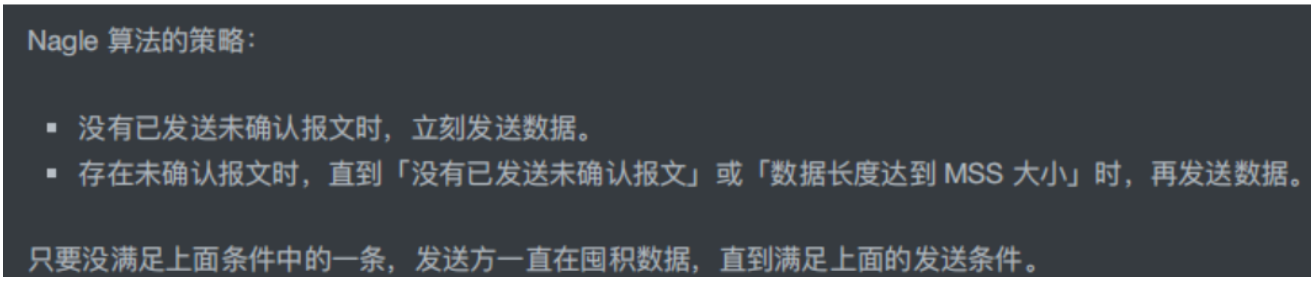
| tcpdump 使用 —— 选项类 | | |
|---|---|---|
| 选项 | 示例 | 说明 |
| -i | tcpdump -i eth0 | 指定网络接口，默认是 0 浩接口（如 eth0），any 表示所有接口 |
| -nn | tcpdump -nn | 不解析 IP 地址和端口号的名称 |
| -c | tcpdump -c 5 | 限制要抓取的网络包的个数 |
| -w | tcpdump -w file.pcap | 保持到文件中，文件名通常以 .pcap 为后缀 |

| tcpdump 使用 —— 过滤表达式类 | | |
|---|---|---|
| 选项 | 示例 | 说明 |
| host、src host、dst host | tcpdump -nn host 192.168.1.100 | 主机过滤 |
| port、src port、dst port | tcpdump -nn port 80 | 端口过滤 |
| ip、ip6、arp、tcp、udp、icmp | tcpdump -nn tcp | 协议过滤 |
| and、or、not | tcpdump -nn host 192.168.1.100 and port 80 | 逻辑表达式 |
| tcp[tcoflages] | tcpdump -nn "tcp[tcpflags] & tcp-syn != 0" | 特定状态的 TCP 包 |

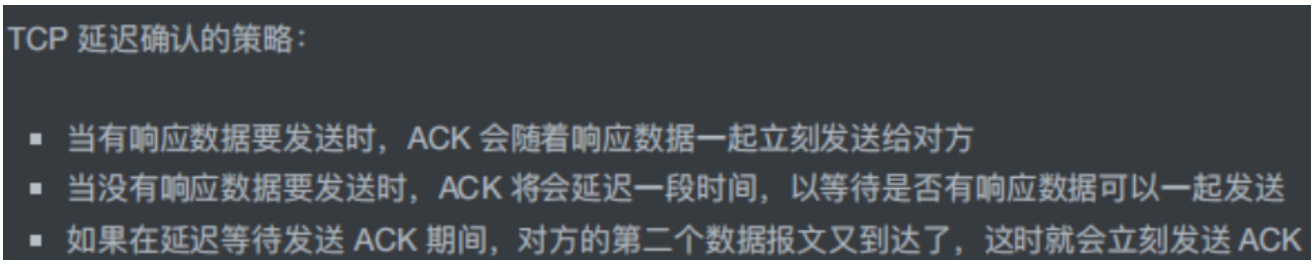
### TCP延迟确认与Nagle算法

当TCP报文承载的数据非常小的时候，整个网络的效率很低。就好像快递员开着大货车送一个小包裹一样浪费。所以就出现了常见的两种策略，来减少小报文的传输。

- Nagle算法




- 延迟确认




当延迟确认和Nagle混合使用时，会出现新的问题（导致时耗增长），要解决这个问题，只有两个办法：

1. 发送方关闭Nagle算法
2. 接收方关闭TCP延迟确认

# TCP与UDP

## 区别

1、连接：TCP是面向连接的，在传输前需要三次握手建立连接，UDP不需要连接，即刻传输数据。

2、服务形式：TCP只能一对一，点对点服务，UDP支持一对一、一对多、多对多通信。

3、可靠性：TCP保证数据可靠交付，拥有确认应答和重传机制，无重复、不丢失、按序到达;UDP尽可能交付，不保证可靠性。

4、连接控制机制 TCP拥有流量控制、拥塞控制，保证传输安全性等，UDP在网络拥堵情况下不会降低发送速率。

5、首部大小：TCP首部长度不适用选项字段是20字节，使用选项字段长度增加(可变)，UDP首部固定8字节。

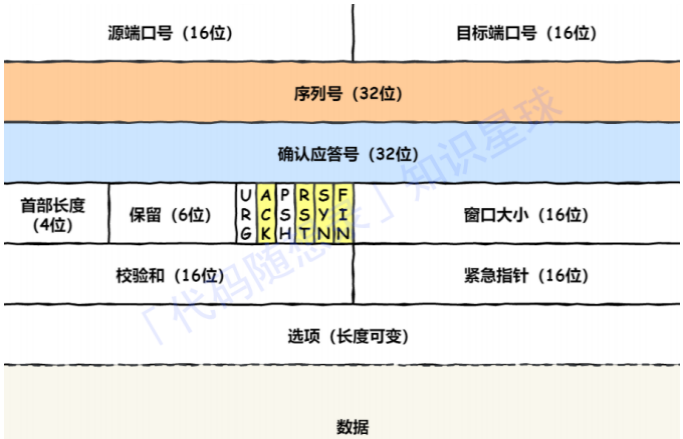
6、传输方式：TCP基于字节流，没有边界，但是保证传输顺序和可靠性;UDP继承了IP层特性，基于数据包，有边界可能出现乱序和丢包。

7、分片方式：TCP数据大于MSS时会在TCP层将数据进行分片传输，到达目的地后同样在传输层进行合并，如果有某个片丢失则只需要重传丢失的分片即可;UDP数据大于MTU时会在IP层分片，同样也在目的IP层合并，如果某个IP分片丢失，则需要将所有分片都进行重传，开销大。

## 头部

- UDP


- TCP




窗口**大小**：接收窗口，告诉对方本端TCP缓冲区还有多少空间可以接收数据，用来做流量控制。

标志字段：

- ACK：用于指示确认应答号值是否有效，置1表示包含一个对已成功接收报文段的确认；
- RST：用于重置一个已经混乱的连接，或拒绝一个无效的数据段或者连接请求；
- SYN：用于连接建立过程，请求建立一个连接；
- FIN：用于断开连接，表示发送方没有数据要传输了。

检验和：接收方使用检验和来检查该报文段(头部+数据)中是否出现差错（CRC算法），同 UDP。

选项：

选项的第一个字段**kind**说明选项的类型，有的TCP选项没有后面两个字段，仅包含1字节的kind字段。

第二个字段**length**（如果有的话）指定该选项的总长度。该长度包括kind字段和length字段占据的2字节。

第三个字段**info**（如果有的话）是选项的具体信息。

- **kind=0**，选项表结束（**EOP**）选项：一个报文段仅用一次。放在末尾用于填充，用途是说明：首部已经没有更多的消息，应用数据在下一个32位字开始处。
- **kind=1**，空操作（**NOP**）选项：没有特殊含义，一般用于将TCP选项的总长度填充为4字节的整数倍。
- **kind=2**，最大报文段长度（**MSS**）选项：**TCP**连接初始化时，通信双方使用该选项来协商最大报文段长度。TCP模块通常将MSS设置为（MTU-40）字节（减掉的这40字节包括20字节的TCP头部和20字节的IP头部）。这样携带TCP报文段的IP数据报的长度就不会超过MTU（假设TCP头部和IP头部都不包含选项字段，并且这也是一般情况），从而避免本机发生IP分片。对以太网而言，MSS值是1460（1500-40）字节。
- **kind=3**，窗口扩大因子选项：TCP连接初始化时，通信双方使用该选项来协商接收窗又的扩大因子。在TCP的头部中，接收窗又大小是用16位表示的，故最大为65535字节，但实际上TCP模块允许的接收窗又大小远不止这个数（为了提高TCP通信的吞吐量）。窗口扩大因子解决了这个问题。
- **kind=4**，选择性确认（**Selective Acknowledgment**，**SACK**）选项：SACK 技术使 TCP 只重新发送丢失的 TCP 报文段，而不用发送所有未被确认的 TCP 报文段。选择性确认选项用在连接初始化时，表示是否支持 SACK 技术。
- **kind=5**，**SACK**实际工作的选项：该选项的参数告诉发送方本端已经收到并缓存的不连续的数据块，从而让发送端可以据此检查并重发丢失的数据块。

  每个块边沿（edge of block）参数包含一个4字节的序号。其中块左边沿表示不连续块的第一个数据的序号，而块右边沿则表示不连续块的最后一个数据的序号的下一个序号。这样一对参数（块左边沿和块右边沿）之间的数据是没有收到的。因为一个块信息占用8字节，所以 TCP 头部选项中实际上最多可以包含4个这样的不连续数据块（考虑选项类型和长度占用的2字节）。
- **kind=8**，时间戳选项：该选项提供了较为准确的计算通信双方之间的回路时间（Round Trip Time，RTT）的方法，为TCP流量控制提供信息。

### SYN攻击

**1**、原理

攻击者伪造不同IP地址的SYN报文请求连接，服务端收到连接请求后分配资源，回复ACK+SYN包，但是由于IP地址是伪造的，无法收到回应，久而久之造成服务端半连接队列被占满，无法正常工作。

**2**、避免方式

（1）修改半连接队列大小：使服务端能够容纳更多半连接。此外还可以修改服务端超时重传次数，使服务端尽早丢弃无用连接

（2）正常服务端行为是收到客户端SYN报文后，将其加入到内核半连接队列，接着发送ACK+SYN报文给客户端，当收到客户端ACK报文后把连接从半连接队列移动到accept队列。

当半连接队列满时，启动syn cookie,后续连接不进入半连接队列，而是计算一个cookie值，作为请求报文序列号发送给客户端，如果服务端收到客户端确认报文，会检查ack包合法性，如果合法直接加入到accept队列。

### TCP保活机制

**1**、概念：在一个定义的时间段内TCP连接无任何活动时，会启动TCP保活机制，每隔一定时间间隔发送一个探测报文，等待响应。

**2**、机制

1. 对端正常响应，重置保活时间;
2. 对端程序崩溃，响应一个RTS报文，将TCP连接重置;
3. 保活报文不可达，等待达到保活探测次数后关闭连接。

# 网络层（数据平面）：

# 网络层（控制平面）：

**概念:**

控制源主机到目的主机之间如何沿着端到端路径转发数据报、控制网络层组件和服务器的配置管理

**两类：**

- 每路由控制
- 集中控制（SDN控制）
    - SDN控制应用程序
    - SDN控制器

**路由选择算法：**

- 链路状态LS：OSPF
- 距离向量DV：BGP

**协议**

- ICMP：互联网控制报文协议
- SNMP：简单网络控制协议（应用层UDP）

# 链路层

**服务：**

成帧、链路访问控制MAC、差错检测纠正、可靠交付

网络适配器

**差错检测：**

- 奇偶校验：检测不能纠正
- 循环冗余校验
- 检验和（一般运输层使用）

**多路访问协议**

- 信道划分：时分多路复用、频分多路复用、码分多址（每个节点不同的编码。可以同时发送数据进行编码）
- 随机接入：ALOHA（时隙<p重传>、非时隙<>）、载波侦听CSMA（以太网,CSAM（说话之前先听）和CSMA/CD（如果同时说话就停止说话））
- 轮流协议：轮询、令牌

**交换局域网**

- 链路层寻址和ARP

**数据中心：**

# web页面请求历程

1、准备DHCP、UDP、IP、以太网

1. 电脑发送DHCP请求报文
    1. 生成DHCP请求报文，端口号67（目的）和68。该UDP报文段广播目的（255.255.255.255）和源（0.0.0.0）的IP数据报
    2. DHCP请求报文放入以太网帧中，目的MAC地址（全F），广播到DHCP服务器。源MAC就是本机MAC地址
    3. 该帧经过交换机广播出去
    4. 和交换机连接的路由器接受到广播帧，该帧中包含了DHCP报文，从以太网帧中抽出Ip数据包，解析ip数据报的载荷udp报文段，DHCP请求报文从UDP报文中抽出来。DHCP服务器就有了DHCP报文
2. DHCP服务器响应
    1. DHCP服务器使用CIDP分配一个IP地址。生成包含这个IP地址、DNS服务器的IP地址、默认网关的路由器IP、子网掩码的DHCP ACK应答报文。该DHCP报文放入UDP报文中，IP数据包中、以太帧中，目的MAC即电脑的MAC地址
    2. DHCP应答报文经过路由器发送给交换机，然后发送给便携机
    3. 便携机收到DHCP ACK，然后一步步解析获取出其中分配给本机的IP地址、DNS服务器地址，并且在IP转发表中安装默认网关的地址。

2、准备DNS、ARP

当输入URL后，为了交互必须知道URL的ip地址，所以必须经过DNS查询

1. 生成一个DNS查询报文，将URL放入DNS的报文段中，然后放在53号目的端口的UDP报文段中，该UDP报文段放入具有DNS服务器的目的IP的IP报文中。
2. 将该ip数据报文放入以太网帧中。经过第一步已经知道网关ip地址，但是不知道MAC地址，于是电脑使用ARP协议
3. 生成一个ARP查询报文，广播该ARP报文（全F）的以太网帧，发送
4. 网关路由器收到ARP查询报文的帧，准备一个ARP应答报文，将IP地址和MAC地址打包发送回去
5. 电脑接收ARP应答报文，解析出网关MAC地址。然后就可以将DNS查询报文经过网关路由器发送出去了。

3、准备域内路由选择到DNS服务器

1. DNS查询报文经过域内协议（OSPF、RIP）生成的转发表转发出去，经过BGP域间转发
2. DNS服务器收到DNS查询报文后，在DNS数据库中找到对应的IP地址。经过迭代和轮询的方式访问各级DNS服务器。将映射的IP地址打包成DNS回答报文传回去。
3. 电脑收到谷歌服务器的IP地址。

4、web客户和服务器交互：TCP和HTTP

1. 电脑有了目的IP地址，可以生成TCP套接字,该套接字发送HTTP get报文，当生成TCP套接字的时候执行三次握手。生成一个具有80端口的TCPSYN报文段，将该数据报文发送到百度IP地址。
2. 百度服务器从数据报中抽出TCPSYN报文，生成一个TCP SYNACK报文段，发送出去。

3. 操作系统收到TCPSYNACK数据报后，进入连接状态，发送带有载荷的数据报，捎带会用一个三次握手最后一次。
4. 服务器生成HTTP响应报文发送给电脑
5. 电脑浏览器收到响应报文，抽取出HTML页面，然后由HTML解释器渲染出画面

# 设计模式

## 单例

一个类一个实例，并提供一个全局访问点;避免全局使用的类，频繁创建和销毁，耗费系统资源。

**实现：**

- 构造函数私有化
- 静态方法访问点
- 私有静态变量
- 加锁互斥判唯一

**6种实现方式(重点1-5)**

**1、懒汉线程不安全**

优点：延迟了实例化，不调用就不会实例，节省资源

缺点：线程不安全。多线程同时进入判断中

```
class Single{
    private:
        Single(){};
        static Single  instance;
    public:
        static Single getIns(){
            if(instance == nullptr){
                instance = new Single();
            }
            return instance;
    }
}
```

**2、懒汉安全**

优点：线程安全，在函数上加了锁

缺点：即使已经实例化了，进入函数还是都要加锁，进入该方法会堵塞，等待时间长

```
class Single{
    private:
        Single(){};
        static Single ins;
```

```
        static mutex mtx;
    public:
        static Single getIns(){
            lock_guard<mutex> lock(mtx);
            if(instance == nullptr){
                instance = new Single();
            }
            return instance;
        }
}
```

### 3、双重锁检查

线程安全，并且不会因为获取锁阻塞。

```
class Single{
    private:
        Single(){}
        static Single ins;
        static mutex mtx;
    public:
        static Single getIns(){
            if(ins == nullptr){
                lock_groud<mutex> lock(mtx);
                if(ins == nullptr){
                    ins = new Single();
                }
            }
            return ins;
        }
}
```

当使用了锁来保护对 `instance` 的访问时，在单线程的情况下，不需要将其声明为 `volatile`。只有在没有使用锁或其他同步机制的情况下，当存在多个线程同时访问 `instance` 时，才需要将其声明为 `volatile` 来确保内存的可见性和一致性。

### 4、饿汉式线程安全

直接实例化好，所以天然没有线程安全问题；

不延迟实例化会浪费系统资源。

```
class Single{
    private:
        Single(){}
        static Single ins = new Single();
    public:
        Single getIns(){
            return ins;
        }
}
```

### 5、静态内部类实现(线程安全)

当single被加载的时候，静态内部类并没有被记载进内存。当调用getIns的时候彩才会被加载进内存，并初始化实例。

```
class Single{
    private:
        Single(){}
        static class SingleHold{
            private static final Single ins = new Single();
        }
    public:
        static Single getIns(){
            return SingleHold.ins;
        }
}
```

### 6、枚举类实现(java C++感觉不太行)

默认枚举实例的创建就是线程安全的，且在任何情况下都是单例。

优点:

写法简单，线程安全，天然防止反射和反序列化调用。

```
public enum Singleton {
 INSTANCE;
 //添加自己需要的操作
 public void doSomeThing() {}
}
```

### 应用场景

频繁创建和销毁的对象、线程池等控制资源，方便资源之间的通信

1. 日志应用
2. 配置对象的读取
3. 数据库连接池
4. 多线程池
5. 网站计数器

# 工厂模式

创建型设计模式、在创建对象时，不会对客户端暴露对象的创建逻辑，而是通过使用共同的接又来创建对象。其用来封装和管理类的创建，本质是对获取对象过程的抽象。
工厂模式分成简单**工厂方**法和抽象**工厂**

优点：

解耦：将对象的创建和使用进行分离

可复**用**：对于创建过程比较复杂且在很多地方都使用到的对象，通过工厂模式可以提高对象创建的代码的复用性。

降低成本：由于复杂对象通过工厂进行统一管理，所以只需要修改工厂内部的对象创建过程即可维护对象，从而达到降低成本的目的。

### 简单工厂模式

在简单工厂模式中，可以根据实际的参数不同返回不同的实例。同时在简单工厂模式中会定义一个类负责创建其他类的实例，被创建的实例也通常具有共同的父类。

虽然实现了对象的创建和使用的分离，但是不够灵活，工厂类集合了所有产品的创建逻辑，职责过重，同时新增一个产品就需要在原工厂类内部添加一个分支，违反了开闭原则。并且若是有多个判断条件共同决定创建对象，则后期修改会越来越复杂。

### 抽象工厂模式：多加了一层抽象类

工厂方法模式中，将简单工厂中的工厂类变为一个抽象接又。负责给出不同工厂应该实现的方法，自身不再负责创建各种产品，而是将具体的创建操作交给实现该接又的子工厂类来做。

通过多态的形式解决了简单工厂模式过多的分支问题。虽然在新增产品时不仅要新增一个产品类还要实现与之对应的子工厂，但是相较于简单工厂模式更符合开闭原则。

# 观察者模式

行为型模式、一对多的依赖关系、让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

优点：解除耦合，让耦合的双方都依赖于抽象，从而使得各自的变换都不会影响另一边的变换

缺点：调试复杂，而且在Java中消息的通知一般是顺序执行，那么一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般会采用异步实现。

# 装饰器模式

装饰模式把每个要装饰的功能放在单独的类中，并让这个类包装它所要装饰的对象。因此，当需要执行特殊行为时，客户代码就可以在运行时根据需要有选择地、按顺序地使用装饰功能包装对象了。

### 代理模式

为其他对象提供一种代理以控制对这个对象的访问。

#### 应用

- 远程代理：一个对象在不同的地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实。
- 虚拟代理：根据需要创建开销很大的对象。通过它来存放实例化需要很长时间的真实对象，这样就可以达到性能的最优化。比如说你打开一个很大的HTML网页时，里面可能有很多的文字和图片，但你还是可以很快打开它，此时你所看到的是所有的文字，但图片却是一张一张地下载后才能看到。那些未打开的图片框，就是通过虚拟代理来替代了真实的图片，此时代理存储了真实图片的路径和尺寸。
- 安全代理：用来控制真实对象访问时的权限。
- 智能指引：是指当调用真实的对象时，代理处理另外一些事。如计算真实对象的引用次数，这样当该对象没有引用时，可以自动释放它；或当第一次引用一个持久对象时，将它装入内存；或在访问一个实际对象前，检查是否已经锁定它，以确保其他对象不能改变它。它们都是通过代理在访问一个对象时附加一些内务处理。

# 计算机系统

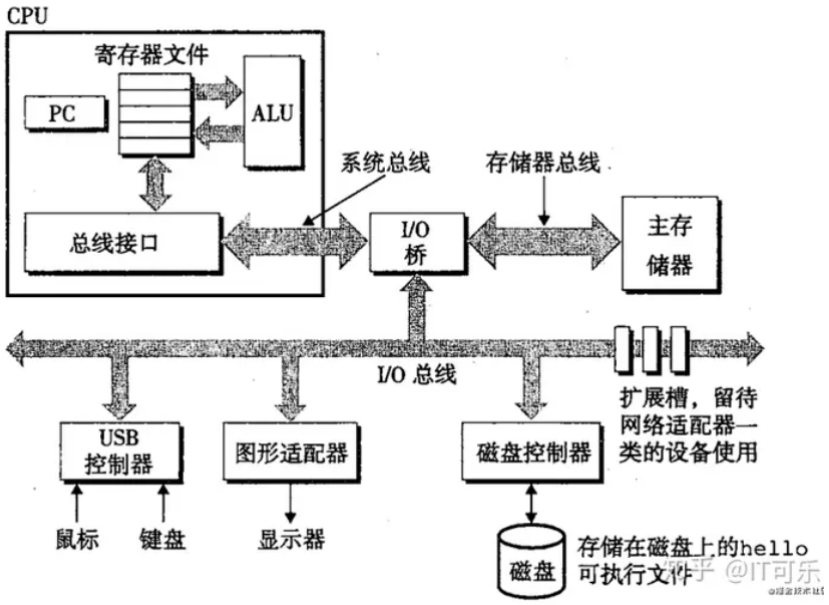
#### 字符编码

固定长度编码

可变长度编码（UTF-8）

#### 冯诺依曼模型

五大部件：CPU（控制单元、寄存器、逻辑运算单元）、内存、总线（数据、控制、内存）、输入、输出



**CPU**

32位CPU一次可以处理4字节

通用寄存器：放运算数据、程序计数器（PC计数器、存放下一条要执行的指令）、指令寄存器（存放PC执行的指令，PC去之后放入指令寄存器中）

**总线：CPU和其他部件的通信**

地址总线：CPU要操作的内存地址

数据总线：读写内存数据

控制总线：收发信号、中断、设备复位

**程序执行过程**

流水线：4级

取指、译指、执行、数据回写

取指：cpu控制单元通过地址总线发送PC中的地址，内存中找到待执行指令、然后传输到指令寄存器、更新PC

译指：分析指令，计算型交给逻辑单元、存储型交给控制单元

执行：根据指令执行

回写：会写到寄存器或者内存

**编译系统**

预处理-----》编译-----》汇编------》链接

gcc -E------》gcc -S----》gcc -c---》

预处理：.c--->gcc -E---->.i 展开宏、头文件、替换条件编译、删除注释

编译：.i----->gcc -S----->.s 检查语法规范。时间最久，系统资源最多

汇编：.s----->gcc -c----->.o汇编指令翻译成机器指令

链接：a.out 数据段合并，地址回填

静态库在编译时候链接，嵌入到可执行程序中，动态库在运行时连接。

**运行一个hello程序**

简单版本

1、输入“./hello”，shell程序会将字符读入寄存器，处理器将hello字符传放入内存中

2、按下空格键，完成命令的输入，然后执行一系列的指令来加载可执行文件，将hello中的数据和代码从磁盘复制到内存。数据就是hello。复制过程利用DMA技术，不经处理器从磁盘直达内存。

3、处理器开始执行main函数中的代码。

4、cpu将hello从内存复制到寄存器文件，然后从寄存器复制到显示设备、显示

复杂版本：

第一步、进程

1. 在 Shell 中输入 hello 程序的路径
2. Shell 判断用户输入的是否为内置命令，如果不是，就认为它是一个可执行目标文件
3. Shell 构造 argv 和 envp
4. Shell 使用 fork() 创建子进程，调用 execve() 函数在新创建的子进程的上下文中加载并运行 hello 程序。将 hello 中的 .text 节、.data 节、.bss 节等内容加载到当前进程的虚拟地址空间
5. execve() 函数调用加载器，跳转到程序的入口点，开始执行 __start 函数，我们的 hello 程序便正式开始执行了
6. 运行在用户模式，运行过程中，内核不断切换上下文，使运行过程被切分成时间片，与其他进程交替占用执行，实现进程的调度。如果在运行过程中收到信号等，那么就会进入内核模式，运行信号处理程序，之后再返回用户模式。

第二步、存储

1. **fork**创建子进程，为 hello 程序的运行创建上下文，并分配一个与父进程不同的PID。通过 fork 创建的子进程拥有父进程相同的区域结构、页表等的一份副本，同时子进程也可以访问任何父进程已经打开的文件。当 fork 在新进程中返回时，新进程现在的虚拟内存刚好和调用 fork 时存在的虚拟内存相同，当这两个进程中的任一个后来进行写操作时，写时复制机制就会创建新页面，因此，也就为每个进程保持了私有地址空间。
2. **execve()** 函数调用驻留在内核区域的启动加载器代码，在当前进程中加载并运行包含在可执行目标文件 hello 中的程序，用 hello 程序有效地替代了当前程序。加载并运行 hello 需要以下几个步骤：
    1. 删除已存在的用户区域，删除当前进程虚拟地址的用户部分中的已存在的区域结构。
    2. 映射私有区域，为新程序的代码、数据、bss 和栈区域创建新的区域结构，所有这些新的区域都是私有的、写时复制的。代码和数据区域被映射为 hello 文件中的 .text 和 .data 区，bss 区域是请求二进制零的，映射到匿名文件，其大小包含在 hello 中，栈和堆地址也是请求二进制零的，初始长度为零。
    3. 映射共享区域， hello 程序与共享对象 libc.so 链接，libc.so 是动态链接到这个程序中的，然后再映射到用户虚拟地址空间中的共享区域内。
    4. 设置程序计数器（PC），execv() 做的最后一件事情就是设置当前进程上下文的程序计数器，使之指向代码区域的入口点。

第三步、IO管理

所有的 I/O 设备（例如网络、磁盘和终端）都被模型化为文件，而所有的输入和输出都被当作对相应文件的读和写来执行。这使得所有的输入和输出都能以一种统一且一致的方式来执行

1. 随后 write 函数将参数放入寄存器，然后用 int 21h 调用 sys_call 。sys_call 将字符串中的字节从寄存器中通过总线复制到显卡的显存中，显存中存储的是字符的 ASCII 码。
2. 字符显示驱动子程序通过 ASCII 码在字模库中找到点阵信息，并将点阵信息存储到 vram 中。

3. 显示芯片会按照一定的刷新频率逐行读取 vram，并通过信号线向液晶显示器传输每一个点（RGB分量）。
4. 最后，hello 程序的输出：hello 就显示在了屏幕上。、

总结：

- 源程序——hello.c
- 预处理器——hello.i
- 编译器——hello.s
- 汇编器——hello.o
- 链接器——hello
- Shell 创建子进程，真正成为系统中的个体
- 加载器映射虚拟内存，分配空间
- CPU 的逻辑控制流将硬件与操作系统联系起来
- 虚拟地址来进行虚拟内存的管理
- malloc 的高效管理
- 信号与异常约束它的行为，让它总是走在康庄大道之上
- Unix I/O 打开它与程序使用者交流的窗口
- 当 hello 垂垂老矣，运行完最后一行代码，___libc_start_main 将控制转移给内核，Shell 回收子进程，内核删除与它相关的所有数据结构，它在这个世界的所有痕迹至此被抹去。

# 数据库

## Mysql

## Redis