# 操作系统

## liunx常见命令

### 线上查询及帮助命令

| man | 查看命令帮助，命令的词典，更复杂的还有info，但不常用。 |
| --- | --- |
| help | 查看Linux内置命令的帮助，比如cd命令。 |

### 文件和目录操作命令(18个)

| rename | 用于重命名文件。 |
| --- | --- |
| rm | 全拼remove，其功能是删除一个或多个文件或目录。 |
| rmdir | 全拼remove empty directories，功能是删除空目录。 |
| touch | 创建新的空文件，改变已有文件的时间戳属性。 |
| tree | 功能是以树形结构显示目录下的内容。 |
| basename | 显示文件名或目录名。 |
| dirname | 显示文件或目录路径。 |
| chattr | 改变文件的扩展属性。 |
| lsattr | 查看文件扩展属性。 |
| file | 显示文件的类型。 |
| md5sum | 计算和校验文件的MD5值。 |

### 查看文件及内容处理命令

| tac | tac是cat的反向拼写，因此命令的功能为反向显示文件内容。 |
| --- | --- |
| more | 分页显示文件内容。 |
| less | 分页显示文件内容，more命令的相反用法。 |
| head | 显示文件内容的头部。 |
| tail | 显示文件内容的尾部。 |
| cut | 将文件的每一行按指定分隔符分割并输出。 |
| split | 分割文件为不同的小片段。 |
| paste | 按行合并文件内容。 |
| sort | 对文件的文本内容排序。 |
| uniq | 去除重复行。oldboy |
| wc | 统计文件的行数、单词数或字节数。 |
| iconv | 转换文件的编码格式。 |
| dos2unix | 将DOS格式文件转换成UNIX格式。 |
| diff | 全拼difference，比较文件的差异，常用于文本文件。 |
| vimdiff | 命令行可视化文件比较工具，常用于文本文件。 |
| rev | 反向输出文件内容。 |
| grep/egrep | 过滤字符串，三剑客老三。 |
| join | 按两个文件的相同字段合并。 |
| tr | 替换或删除字符。 |
| vi/vim | 命令行文本编辑器。 |

### 信息显示命令

| uname | 显示操作系统相关信息的命令。 |
| --- | --- |
| hostname | 显示或者设置当前系统的主机名。 |
| dmesg | 显示开机信息，用于诊断系统故障。 |
| uptime | 显示系统运行时间及负载。 |
| stat | 显示文件或文件系统的状态。 |
| du | 计算磁盘空间使用情况。 |
| df | 报告文件系统磁盘空间的使用情况。 |
| top | 实时显示系统资源使用情况。 |
| free | 查看系统内存。 |
| date | 显示与设置系统时间。 |
| cal | 查看日历等时间信息。 |

### 搜索文件命令

| which | 查找二进制命令，按环境变量PATH路径查找。 |
| --- | --- |
| find | 从磁盘遍历查找文件或目录。 |
| whereis | 查找二进制命令，按环境变量PATH路径查找。 |
| locate | 从数据库 (/var/lib/mlocate/mlocate.db) 查找命令，使用updatedb更新库。 |

### 用户管理命令

| useradd | 添加用户。 |
| --- | --- |
| usermod | 修改系统已经存在的用户属性。 |
| userdel | 删除用户。 |
| groupadd | 添加用户组。 |
| passwd | 修改用户密码。 |
| chage | 修改用户密码有效期限。 |
| id | 查看用户的uid,gid及归属的用户组。 |
| su | 切换用户身份。 |
| visudo | 编辑/etc/sudoers文件的专属命令。 |

### 网络操作命令.

| telnet | 使用TELNET协议远程登录。 |
| --- | --- |
| ssh | 使用SSH加密协议远程登录。 |
| scp | 全拼secure copy，用于不同主机之间复制文件。 |
| wget | 命令行下载文件。 |
| ping | 测试主机之间网络的连通性。 |
| route | 显示和设置linux系统的路由表。 |
| ifconfig | 查看、配置、启用或禁用网络接口的命令。 |
| ifup | 启动网卡。 |
| ifdown | 关闭网卡。 |
| netstat | 查看网络状态。 |
| ss | 查看网络状态。 |

| nmap | 网络扫描命令。 |
| --- | --- |
| lsof | 全名list open files，也就是列举系统中已经被打开的文件。 |
| mail | 发送和接收邮件。 |
| mutt | 邮件管理命令。 |
| nslookup | 交互式查询互联网DNS服务器的命令。 |
| dig | 查找DNS解析过程。 |
| host | 查询DNS的命令。 |
| traceroute | 追踪数据传输路由状况。 |
| tcpdump | 命令行的抓包工具。 |

### 磁盘与文件系统的命令

| mount | 挂载文件系统。 |
| --- | --- |
| umount | 卸载文件系统。 |
| fsck | 检查并修复Linux文件系统。 |
| dd | 转换或复制文件。 |
| dumpe2fs | 导出ext2/ext3/ext4文件系统信息。 |
| dump | ext2/3/4文件系统备份工具。 |
| fdisk | 磁盘分区命令，适用于2TB以下磁盘分区。 |
| parted | 磁盘分区命令，没有磁盘大小限制，常用于2TB以下磁盘分区。 |
| mkfs | 格式化创建Linux文件系统。 |
| partprobe | 更新内核的硬盘分区表信息。 |
| e2fsck | 检查ext2/ext3/ext4类型文件系统。 |
| mkswap | 创建Linux交换分区。 |
| swapon | 启用交换分区。 |
| swapoff | 关闭交换分区。 |
| sync | 将内存缓冲区内的数据写入磁盘。 |
| resize2fs | 调整ext2/ext3/ext4文件系统大小。 |

### 系统权限及用户授权相关命令

| chmod | 改变文件或目录权限。 |
| --- | --- |
| chown | 改变文件或目录的属主和属组。 |
| chgrp | 更改文件用户组。 |
| umask | 显示或设置权限掩码 |

### 系统管理与性能监视命令

| chkconfig | 管理Linux系统开机启动项。 |
| --- | --- |
| vmstat | 虚拟内存统计。 |
| mpstat | 显示各个可用CPU的状态统计。 |
| iostat | 统计系统IO。 |
| sar | 全面地获取系统的CPU、运行队列、磁盘 I/O、分页（交换区）、内存、 CPU中断和网络等性能数据。 |
| ipcs | 用于报告Linux中进程间通信设施的状态，显示的信息包括消息列表、共享内存和信号量的信息。 |
| ipcrm | 用来删除一个或更多的消息队列、信号量集或者共享内存标识。 |
| strace | 用于诊断、调试Linux用户空间跟踪器。我们用它来监控用户空间进程和内核的交互，比如系统调用、信号传递、进程状态变更等。 |
| ltrace | 命令会跟踪进程的库函数调用,它会显现出哪个库函数被调用。 |

### 关机/重启/注销和查看系统信息的命令

| shutdown | 关机。 |
| --- | --- |
| halt | 关机。 |
| poweroff | 关闭电源。 |
| logout | 退出当前登录的Shell。 |
| exit | 退出当前登录的Shell。 |
| Ctrl+d | 退出当前登录的Shell的快捷键。 |

### 进程管理相关命令

| bg | 将一个在后台暂停的命令，变成继续执行 （在后台执行）。 |
| --- | --- |
| fg | 将后台中的命令调至前台继续运行。 |
| jobs | 查看当前有多少在后台运行的命令。 |
| kill | 终止进程。 |
| killall | 通过进程名终止进程。 |
| pkill | 通过进程名终止进程。 |
| crontab | 定时任务命令。 |
| ps | 显示进程的快照。 |
| pstree | 树形显示进程。 |
| nice/renice | 调整程序运行的优先级。 |
| nohup | 忽略挂起信号运行指定的命令。 |
| pgrep | 查找匹配条件的进程。 |
| runlevel | 查看系统当前运行级别。 |
| init | 切换运行级别。 |
| service | 启动、停止、重新启动和关闭系统服务，还可以显示所有系统服务的当前状态。 |

## 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

##### 线程间通信

目的：用于线程同步，所以线程没有像进程一样的数据交换的通信机制

方式：同⼀进程的不同线程共享同⼀份内存区域，只需要将数据复制到共享（全局或堆）变量中即可。

## 协程

#### 协程和线程的区别

* 协程虽然被称为轻量级线程，但在单线程内，协程并不能并发执行，只能是一个协程结束或yield后，再执行另一个协程，而线程则是可以真正并发执行的。
* 每个协程在创建时都会指定一个入口函数，这点可以类比线程。协程的本质就是函数和函数运行状态的组合 。

#### 协程和函数的区别

* 函数一旦被调用，只能从头开始执行，直到函数执行结束退出
* 而协程则可以执行到一半就退出（称为yield），但此时协程并未真正结束，只是暂时让出CPU执行权，在后面适当的时机协程可以重新恢复运行（称为resume），在这段时间里其他的协程可以获得CPU并运行，所以协程也称为轻量级线程。

#### 协程上下文

* 协程能够半路yield、再重新resume的关键是协程存储了函数在yield时间点的执行状态，这个状态称为协程上下文。
* 协程上下文包含了函数在当前执行状态下的全部CPU寄存器的值，这些寄存器值记录了函数栈帧、代码的执行位置等信息，如果将这些寄存器的值重新设置给CPU，就相当于重新恢复了函数的运行。在Linux系统里这个上下文用ucontext\_t结构体来表示，通getcontext()来获取。

因为单线程下协程并不是并发执行，而是顺序执行的，所以不要在协程里使用线程级别的锁来做协程同步，比如pthread\_mutex\_t。如果一个协程在持有锁之后让出执行，那么同线程的其他任何协程一旦尝试再次持有这个锁，整个线程就锁死了，这和单线程环境下，连续两次对同一个锁进行加锁导致的死锁道理完全一样。

同样是单线程环境下，协程的yield和resume一定是同步进行的，一个协程的yield，必然对应另一个协程的resume，因为线程不可能没有执行主体。并且，协程的yield和resume是完全由应用程序来控制的。与线程不同，线程创建之后，线程的运行和调度也是由操作系统自动完成的，但协程创建后，协程的运行和调度都要由应用程序来完成，就和调用函数一样，所以协程也被称为用户态线程。

## 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：

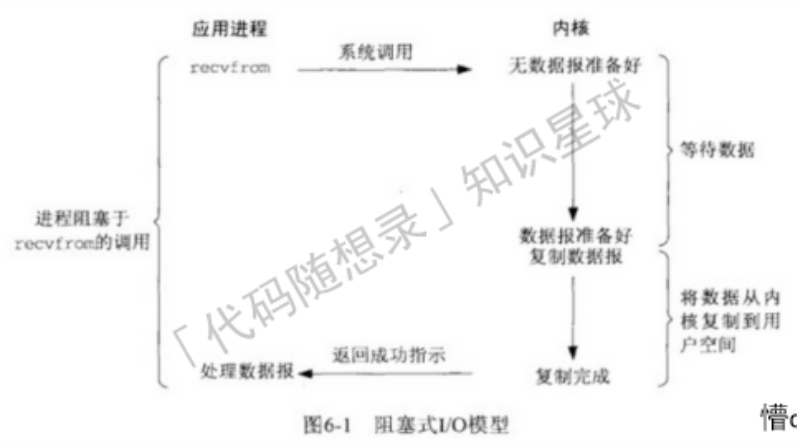
## IO复用

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。

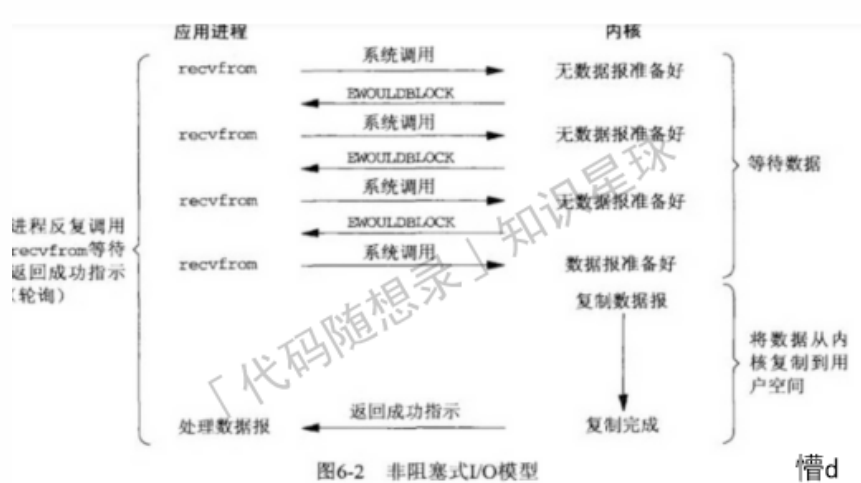
IO操作一般分为两步：1、等待数据准备好。2、从内核向进程复制数据。例如：等待数据从网络中到达后复制到内核的缓冲区；然后将数据从缓冲区输入到应用进程。

#### 五种IO模型

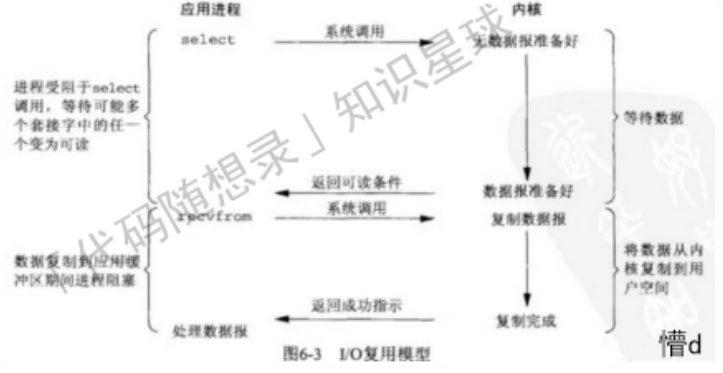
1. 阻塞式IO：进程或者线程等待某个条件，如果条件不满足就一直等下去。条件满足就进行下一步操作。应用进程会阻塞在系统调用recvfrom。

* 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
* 
* image-20230821135615392

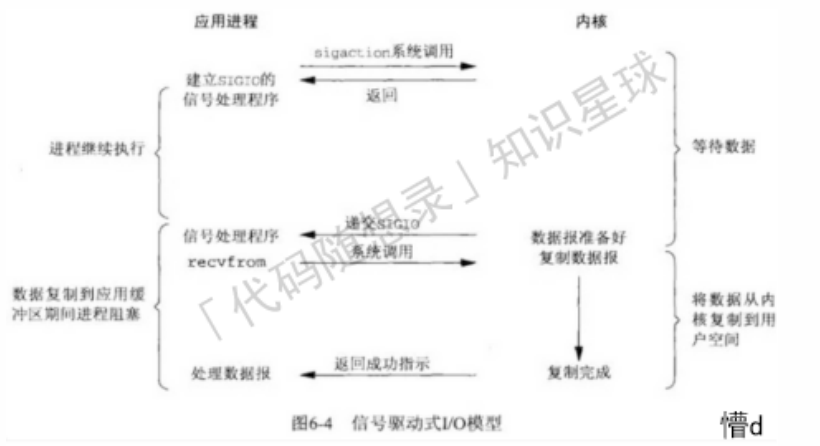
1. 非阻塞式IO：应用进程和内核交互，数据未准备好的时候，不会阻塞等待唤醒，而是不停的轮询recvfrom，如果数据准备好，就复制数据到进程空间。

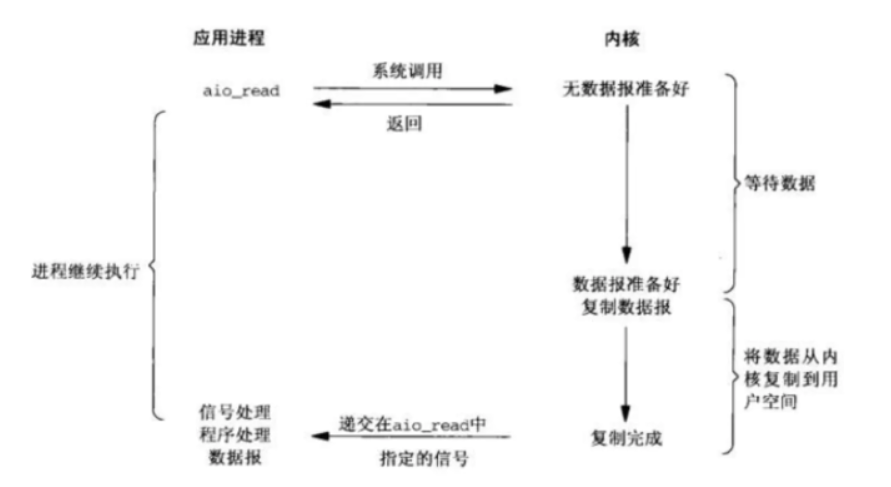
* 缺点：轮询操作是系统调用，不停轮询会耗费大量的cpu时间。

1. 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。

1. 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；

1. 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。
2. 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

##### 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状态码

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

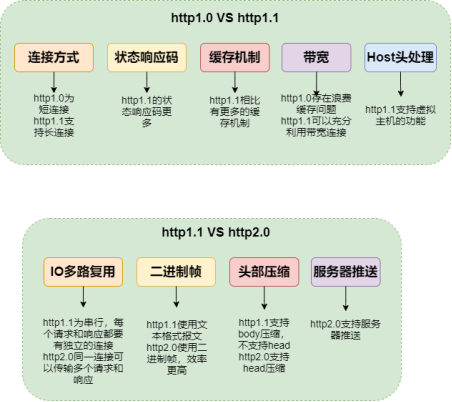


image-20230825114913395

##### **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。



image-20230825115643480

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。

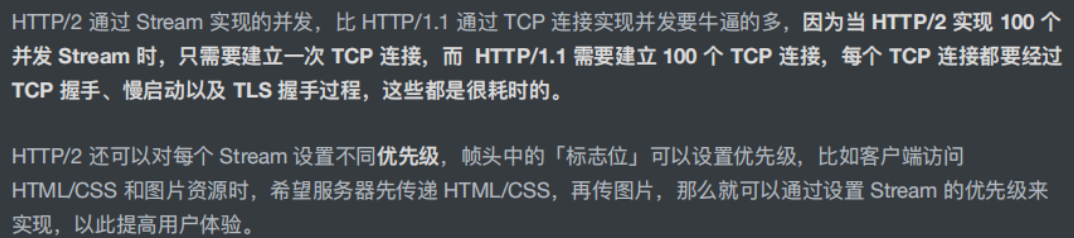


image-20230825115706599

###### **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

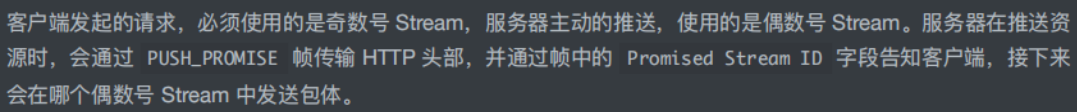


image-20230825115744687

###### 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 视⻆看，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。

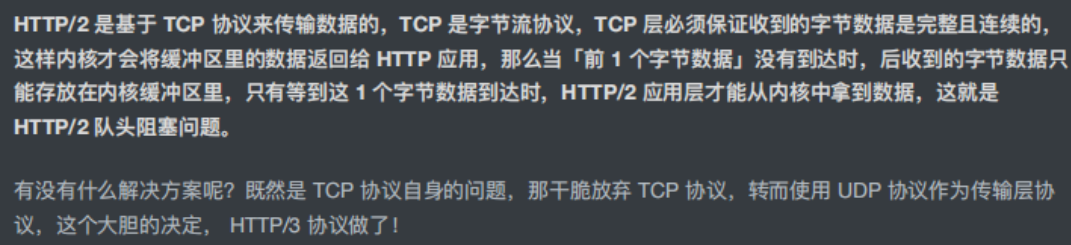


image-20230825120115176

##### 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

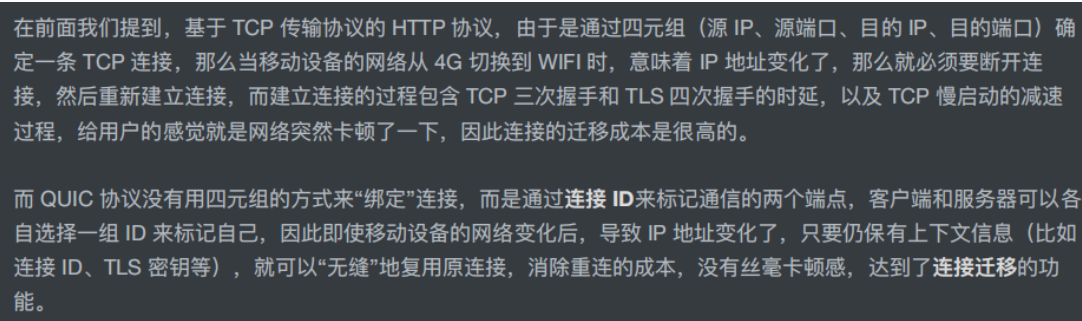


image-20230825141438753

#### 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

**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



image-20230825101007780

**3**、绕过三次握⼿

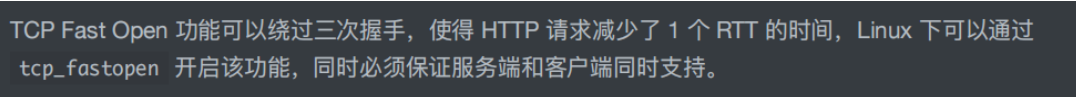


image-20230825101033531

##### 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。

##### 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。

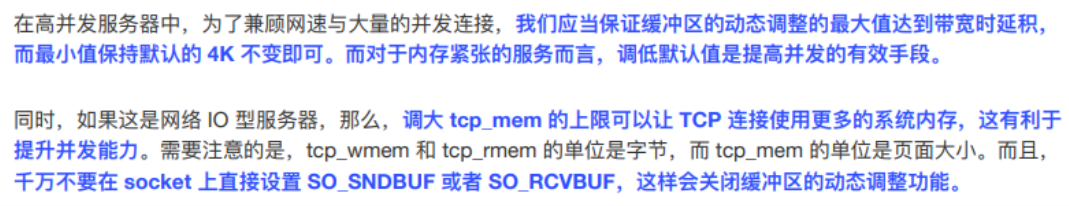


image-20230825095842700

#### 抓包分析

##### **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

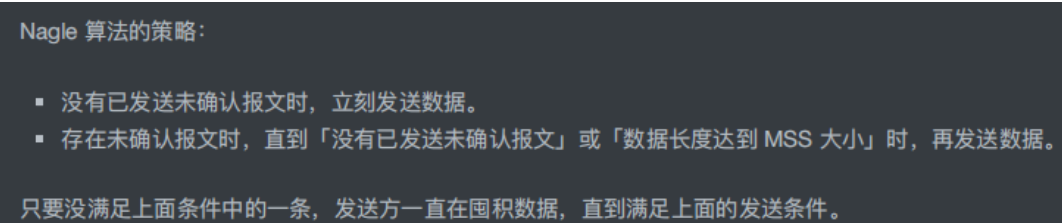


image-20230825101416267

* 延迟确认

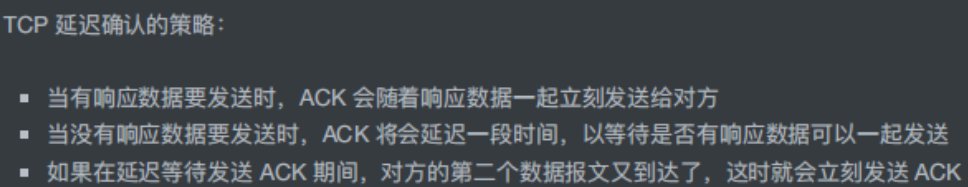


image-20230825101437348

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、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](https://zhuanlan.zhihu.com/p/513307151)

**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](https://link.zhihu.com/?target=http%3A//libc.so/) 链接，[libc.so](https://link.zhihu.com/?target=http%3A//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

# C++基本功

## 一、基础

### 1、面向对象三大特性

#### 访问权限

* public、protected、private 三个关键字来控制成员变量和成员函数的访问权限.
* 在类的内部（定义类的代码内部），⽆论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。
* 在类的外部（定义类的代码之外），通过对象只能访问 public 属性的成员，不能访问private、protected 属性的成员。
* 共有继承、私有和保护继承，私有成员不能被“派⽣类”访问，基类中的共有和保护成员能被“派⽣类”访问
* 对于共有继承，只有基类中的共有成员能被“派⽣类对象”访问，保护和私有成员不能被“派⽣类对象”访问。对于私有和保护继承，基类中的所有成员不能被“派⽣类对象”访问。

#### 继承

让某种类型对象获得另⼀个类型对象的属性和⽅法，它可以使⽤现有类的所有功能，并在⽆需重新编写原来的类的情况下对这些功能进⾏扩展。继承的过程，就是从⼀般到特殊的过程

常见的继承有三种⽅式：

**1**、实现继承：指使⽤基类的属性和⽅法⽽⽆需额外编码的能⼒

**2**、接口继承：指仅使⽤属性和⽅法的名称、但是⼦类必须提供实现的能⼒

**3**、可视继承：指⼦窗体（类）使⽤基窗体（类）的外观和实现代码的能⼒

#### 封装

数据和代码捆绑在⼀起，避免外界⼲扰和不确定性访问; 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，对不可信的进⾏信息隐藏。在⼀个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种⽅式，对象对内部数据提供了不同级别的保护，以防⽌程序中⽆关的部分意外的改变或错误的使⽤了对象的私有部分。

#### 多态

同⼀事物表现出不同事物的能⼒，即向不同对象发送同⼀消息，不同的对象在接收时会产⽣不同的⾏为（重载实现编译时多态，虚函数实现运⾏时多态）。多态性是允许你将⽗对象设置成为和⼀个或更多的他的⼦对象相等的技术，赋值之后，⽗对象就可以根据当前赋值给它的⼦对象的特性以不同的⽅式运作;**简单⼀句话：允许将⼦类类型的指针赋值给⽗类类型的指针。**

##### 分类：

1. 重写**override**：⼦类重新定义⽗类的虚函数的做法
2. 重载**overload**：允许存在多个同名函数，⽽这些函数的参数表不同（或许参数个数不同，或许参数类型

* 不同，或许两者都不同）

##### **虚函数**

当⼀个类中包含虚函数时，编译器会为该类⽣成⼀个虚函数表，保存该类中虚函数的地址，同样，派⽣类继承基类，派⽣类中⾃然⼀定有虚函数，所以编译器也会为派⽣类⽣成⾃⼰的虚函数表。当我们定义⼀个派⽣类对象时，编译器检测该类型有虚函数，所以为这个派⽣类对象⽣成⼀个虚函数指针，指向该类型的虚函数表，这个虚函数指针的初始化是在构造函数中完成的。

后续如果有⼀个基类类型的指针，指向派⽣类，那么当调⽤虚函数时，就会根据所指真正对象的虚函数表指针去寻找虚函数的地址，也就可以调⽤派⽣类的虚函数表中的虚函数以此实现多态。

**对于派⽣类来说，编译器建⽴虚函数表的过程其实⼀共是三个步骤**：

* 拷⻉基类的虚函数表，如果是多继承，就拷⻉每个有虚函数基类的虚函数表
* 还有⼀个基类的虚函数表和派⽣类⾃身的虚函数表共⽤了⼀个虚函数表，也称为某个基类为派⽣类的主基类
* 查看派⽣类中是否有重写基类中的虚函数， 如果有，就替换成已经重写的虚函数地址；查看派⽣类是否有⾃身的虚函数，如果有，就追加⾃身的虚函数到⾃身的虚函数表中。

###### 析构函数⼀般写成虚函数的原因

为了降低内存泄漏的可能性。举例来说就是，⼀个基类的指针指向⼀个派⽣类的对象，在使⽤完毕准备销毁时，如果基类的析构函数没有定义成虚函数，那 么编译器根据指针类型就会认为当前对象的类型是基类，调⽤基类的析构函数 （该对象的析构函数的函数地址早就被绑定为基类的析构函数），仅执⾏基类的析构，派⽣类的⾃身内容将⽆法被析构，造成内存泄漏。如果基类的析构函数定义成虚函数，那么编译器就可以根据实际对象，执⾏派⽣类的析构函数，再执⾏基类的析构函数，成功释放内存。

###### 构造函数为什么⼀般不定义为虚函数

* 虚函数调⽤只需要知道函数接⼝，⽽不需要知道对象的具体类型。但是，我们要创建⼀个对象的话，是需要知道对象的完整信息的。特别是，需要知道要创建对象的确切类型，因此，构造函数不应该被定义成虚函数；
* ⽽且从⽬前编译器实现虚函数进⾏多态的⽅式来看，虚函数的调⽤是通过实例化之后对象的虚函数表指针来找到虚函数的地址进⾏调⽤的，如果说构造函数是虚的，那么虚函数表指针则是不存在的，⽆法找到对应的虚函数表来调⽤虚函数，那么这个调⽤实际上也是违反了先实例化后调⽤的准则

### 2、指针与引用

* 指针和引⽤都是⼀种内存地址的概念，区别呢，指针是⼀个实体，引⽤只是⼀个别名。
* 指针存放某个对象的地址，其本⾝就是变量（命了名的对象），本⾝就有地址，所以可以有指向指针的指针；可变，包括其所指向的地址的改变和其指向的地址中所存放的数据的改变。在编译的时候，则是将“指针变量名-指针变量的地址”添加到符号表中，所以说，指针包含的内容是可以改变的，允许拷⻉和赋值，有 const 和⾮ const 区别，甚⾄可以为空，
* 引⽤就是变量的别名，从⼀⽽终，不可变，必须初始化。在添加到符号表的时候，是将”引⽤变量名-引⽤对象的地址”添加到符号表中，符号表⼀经完成不能改变，所以引⽤必须⽽且只能在定义时被绑定到⼀块内存上，后续不能更改，也不能为空，也没有 const 和⾮ const 区别。
* 作为参数时也不同，传指针的实质是传值，传递的值是指针的地址；传引⽤的实质是传地址，传递的是变量的地址。
* 不存在指向空值的引⽤，但是存在指向空值的指针

### 3、关键字

#### **const**

1. **修饰基本类型数据类型：**修饰符 const 可以⽤在类型说明符前，也可以⽤在类型说明符后，其结果是⼀样的。在使⽤这些常量的时候，只要不改变这些常量的值即可。
2. **const** **修饰指针变量和引⽤变量**：如果 const 位于指针符号的左侧，则 const 就是⽤来修饰指针所指向的变量，即指针指向为常量；如果 const 位于⼩星星的右侧，则 const 就是修饰指针本身，即指针本身是常量。
3. **const** **函数参数中**：作为参数的 const 修饰符：调⽤函数的时候，⽤相应的变量初始化const 常量，则在函数体中，按照 const 所修饰的部分进⾏常量化，保护了原对象的属性。
4. 参数 const 通常⽤于参数为指针或引⽤的情况; 作为函数返回值的 const 修饰符：声明了返回值后，const 按照”修饰原则”进⾏修饰，起到相应的保护作⽤
5. **const 成员函数**：const 成员函数的主要⽬的是防⽌成员函数修改对象的内容。要注意，const关键字和 static 关键字对于成员函数来说是不能同时使⽤的，因为 static 关键字修饰静态成员函数不含有 this 指针，即不能实例化，const 成员函数⼜必须具体到某⼀个函数。

#### **define** **和** **const**

* **对于** **define** **来说，**宏定义实际上是在预编译阶段进⾏处理，没有类型，也就没有类型检查，仅仅做的是遇到宏定义进⾏字符串的展开，遇到多少次就展开多少次，⽽且这个简单的展开过程中，很容易出现边界效应，达不到预期的效果。因为 define 宏定义仅仅是展开，因此运⾏时系统并不为宏定义分配内存，但是从汇编 的⻆度来讲，define 却以⽴即数的⽅式保留了多份数据的拷⻉。
* **对于** **const** **来说，**const 是在编译期间进⾏处理的，const 有类型，也有类型检查，程序运⾏时系统会为 const 常量分配内存，⽽且从汇编的⻆度讲，const 常量在出现的地⽅保留的是真正数据的内存地址，只保留了⼀份数据的拷⻉，省去了不必要的内存空间。⽽且，有时编译器不会为普通的 const 常量分配内存，⽽是直接将 const 常量添加到符号表中，省去了读取和写⼊内存的操作，效率更⾼。

##### define 和别名 typedef 的区别

* 执⾏时间不同，typedef 在编译阶段有效，typedef 有类型检查的功能；#define 是宏定义，发⽣在预处理阶段，不进⾏类型检查；
* 功能差异，typedef ⽤来定义类型的别名，定义与平台⽆关的数据类型，与 struct 的结合使⽤等。
* define 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。
* 作⽤域不同，#define 没有作⽤域的限制，只要是之前预定义过的宏，在以后的程序中都可以使⽤。⽽ typedef 有⾃⼰的作⽤域。

##### define 与 inline 的区别

* #define是关键字，inline是函数；
* 宏定义在预处理阶段进⾏⽂本替换，inline 函数在编译阶段进⾏替换；
* inline 函数有类型检查，相⽐宏定义⽐较安全；

#### static

作用：控制变量的存储方式和可见性

1. **修饰局部变量**：⼀般情况下，对于局部变量在程序中是存放在栈区的，并且局部的⽣命周期在包含语句块执⾏结束时便结束了。但是如果⽤ static 关键字修饰的话，该变量便会存放在静态数据区，其⽣命周期会⼀直延续到整个程序执⾏结束。**静态⾮常量数据成员，其只能在类外定义和初始化，在类内仅是声明而已。**
2. **修饰全局变量：**⽤ static 对全局变量进⾏修饰改变了其作⽤域范围，由原来的整个⼯程可⻅变成了本⽂件可⻅。
3. **修饰函数：**⽤ static 修饰函数，情况和修饰全局变量类似，也是改变了函数的作⽤域。
4. **修饰类**：如果 C++ 中对类中的某个函数⽤ static 修饰，则表示该函数属于⼀个类⽽不是属于此类的任何特定对象；如果对类中的某个变量进⾏ static 修饰，则表示该变量以及所有的对象所有，存储空间中只存在⼀个副本，可以通过；类和对象去调⽤。
5. **类成员/类函数声明 static：**

**其他：**

* static 类对象必须要在类外进⾏初始化，static 修饰的变量先于对象存在，所以 static 修饰的变量要在类外初始化；
* 由于 static 修饰的类成员属于类，不属于对象，因此 static 类成员函数是没有 this 指针，this 指针是指向本对象的指针，正因为没有 this 指针，所以 static 类成员函数不能访问⾮static 的类成员，只能访问 static修饰的类成员
* static 成员函数不能被 virtual 修饰，static 成员不属于任何对象或实例，所以加上 virtual没有任何实际意义；静态成员函数没有 this 指针，虚函数的实现是为每⼀个对象分配⼀个vptr 指针，⽽ vptr 是通过 this 指针调⽤的，所以不能为 virtual；虚函数的调⽤关系，this->vptr->ctable->virtual function。

#### **new / delete** **，**malloc / free

* 都可以⽤来在堆上分配和回收空间。new /delete 是操作符，malloc/free 是库函数。
* **执⾏** **new** **实际上执⾏两个过程**：1.分配未初始化的内存空间（malloc）；2.使⽤对象的构造函数对空间进⾏初始化；返回空间的⾸地址。如果在第⼀步分配空间中出现问题，则抛出std::bad\_alloc 异常，或被某个设定的异常处理函数捕获处理；如果在第⼆步构造对象时出现异常，则⾃动调⽤ delete 释放内存
* **执⾏** **delete** **实际上也有两个过程**：1. 使⽤析构函数对对象进⾏析构；2.回收内存空间（free）。
* new 得到的是经过初始化的空间，⽽ malloc 得到的是未初始化的空间。所以 new 是 new ⼀个类型，⽽ malloc 则是malloc ⼀个字节⻓度的空间。delete 和 free 同理，delete 不仅释放空间还析构对象，delete ⼀个类型，free ⼀个字节⻓度的空间。
* **为什么有了** **malloc**／free **还需要** **new**／delete？因为对于⾮内部数据类型⽽⾔，光⽤ malloc／free ⽆法满⾜动态对象的要求。对象在创建的同时需要⾃动执⾏构造函数，对象在消亡以前要⾃动执⾏析构函数。由于 mallo／free 是库函数⽽不是运算符，不在编译器控制权限之内，不能够把执⾏的构造函数和析构函数的任务强加于 malloc／free，所以有了 new／delete 操作符。

#### **volatile和**extern\*\*\*\*

**volatile** **三个特性：**

* 易变性：在汇编层⾯反映出来，就是两条语句，下⼀条语句不会直接使⽤上⼀条语句对应的volatile 变量的寄存器内容，⽽是重新从内存中读取。
* 不可优化性：volatile 告诉编译器，不要对我这个变量进⾏各种激进的优化，甚⾄将变量直接消除，保证程序员写在代码中的指令，⼀定会被执⾏。
* 顺序性：能够保证 volatile 变量之间的顺序性，编译器不会进⾏乱序优化。

**extern：**

在 C 语⾔中，修饰符 extern ⽤在变ᰁ或者函数的声明前，⽤来说明 “此变量/函数是在别处定义的，要在此处引⽤”。注意 extern 声明的位置对其作⽤域也有关系，如果是在 main 函数中进⾏声明的，则只能在main 函数中调⽤，在其它函数中不能调⽤。其实要调⽤其它⽂件中的函数和变量，只需把该⽂件⽤ #include 包含进来即可，为啥要⽤ extern？因为⽤ extern 会加速程序的编译过程，这样能节省时间。

在 C++ 中 extern 还有另外⼀种作⽤，⽤于指示 C 或者 C＋＋函数的调⽤规范。⽐如在 C＋＋ 中调⽤ C 库函数，就需要在 C＋＋ 程序中⽤ extern “C” 声明要引⽤的函数。这是给链接器⽤的，告诉链接器在链接的时候⽤C 函数规范来链接。主要原因是 C＋＋ 和 C 程序编译完成后在⽬标代码中命名规则不同，⽤此来解决名字匹配的问题。

#### auto /decltype

跳到C++11新特性类型推导

#### **final** **和** **override**

final 来禁⽌虚函数被重写/禁⽌类被继承， override 来显示地重写虚函数。

#### **default** **和** **delete**

编译器会为类⾃动⽣成⼀些⽅法，⽐如构造和析构函数。现在可以显式地指定和禁⽌这些⾃动⾏为了。

### **3、fork，wait，exec**函数

⽗进程产⽣⼦进程使⽤ fork 拷⻉出来⼀个⽗进程的副本，此时只拷⻉了⽗进程的⻚表，两个进程都读同⼀块内存。

当有进程写的时候使⽤写实拷⻉机制分配内存，exec 函数可以加载⼀个 elf ⽂件去替换⽗进程，从此⽗进程和⼦进程就可以运⾏不同的程序了。

fork 从⽗进程返回⼦进程的 pid，从⼦进程返回 0，调⽤了 wait 的⽗进程将会发⽣阻塞，直到有⼦进程状态改变，执⾏成功返回 0，错误返回 -1。

exec 执⾏成功则⼦进程从新的程序开始运⾏，⽆返回值，执⾏失败返回 -1。

### 4、c++和c的区别

1. 基本语句：C++ 有新增的**语法和关键字**，语法的区别有头⽂件的不同和命名空间的不同，C++ 允许我们

* ⾃⼰定义⾃⼰的空间，C 中不可以。关键字⽅⾯⽐如 C++ 与 C 动态管理内存的⽅式不同，C++ 中在malloc 和 free 的基础上增加了 new 和 delete，⽽且 C++ 中在指针的基础上增加了引⽤的概念，关键字例如 C++中还增加了 auto，explicit 体现显示和隐式转换上的概念要求，还有 dynamic\_cast 增加类型安全⽅⾯的内容。

1. **函数⽅⾯：**C++ **中有重载和虚函数的概念**：C++ ⽀持函数重载⽽ C 不⽀持，是因为 C++ 函数的名字修饰与 C 不同，C++ 函数名字的修饰会将参数加在后⾯，例如，int func(int,double)经过名字修饰之后会变成\_func\_int\_double，⽽ C 中则会变成 \_func，所以 C++ 中会⽀持不同参数调⽤不同函数。
2. **虚函数概念，⽤以实现多态**
3. **类⽅⾯：**C **的** **struct** **和** **C++** **的类也有很⼤不同**：C++ 中的 struct 不仅可以有成员变量还可以成员函数，⽽且对于 struct 增加了权限访问的概念，struct 的默认成员访问权限和默认继承权限都是 public，C++ 中除了 struct 还有 class 表示类，struct 和 class 还有⼀点不同在于 class 的默认成员访问权限和默认继承权限都是 private。
4. **C++** **中增加了模板还᯿⽤代码，提供了更加强⼤的** **STL** **标准库**

### 5、c++和java的区别

* **指针：**Java 语⾔让程序员没法找到指针来直接访问内存，没有指针的概念，并有内存的⾃动管理功能，从⽽有效的防⽌了 C++ 语⾔中的指针操作失误的影响。但并⾮ Java 中没有指针，Java 虚拟机内部中还是⽤了指针，保证了 Java 程序的安全。
* **多重继承：**C++ ⽀持多重继承但 Java 不⽀持，但⽀持⼀个类继承多个接⼝，实现 C++ 中多重继承的功能，⼜避免了 C++ 的多᯿继承带来的不便。.
* **⾃动内存管理：**Java 程序中所有对象都是⽤ new 操作符建⽴在内存堆栈上，Java ⾃动进⾏⽆⽤内存回收操作，不需要程序员进⾏⼿动删除。⽽ C++ 中必须由程序员释放内存资源，增加了程序设计者的负担。Java 中当⼀个对象不再被⽤到时， ⽆⽤内存回收器将给他们加上标签。Java ⾥⽆⽤内存回收程序是以线程⽅式在后台运⾏的，利⽤空闲时间⼯作来删除。
* Java 不⽀持操作符᯿载。操作符᯿载被认为是 C++ 的突出特性。
* Java 不⽀持预处理功能。C++ 在编译过程中都有⼀个预编译阶段，Java 没有预处理器，但它提供了 import 与 C++ 预处理器具有类似功能。
* **类型转换：**C++ 中有数据类型隐含转换的机制，Java 中需要限时强制类型转换。
* **字符串：**C++中字符串是以 Null 终⽌符代表字符串的结束，⽽ Java 的字符串 是⽤类对象（string 和 stringBuffer）来实现的。

### 6、智能指针

智能指针其作⽤是管理⼀个指针，避免程序员申请的空间在函数结束时忘记释放，造成内存泄漏这种情况滴发⽣。然后使⽤智能指针可以很⼤程度上的避免这个问题，因为智能指针就是⼀个类，当超出了类的作⽤域是，类会⾃动调⽤析构函数，析构函数会⾃动释放资源。所以智能指针的作⽤原理就是在函数结束时⾃动释放内存空间，不需要⼿动释放内存空间。

#### **1、auto\_ptr**（c++11抛弃）

采⽤所有权模式。

#### 2、**unique\_ptr**（替换 auto\_ptr ）

unique\_ptr 实现独占式拥有或严格拥有概念，保证同⼀时间内只有⼀个智能指针可以指向该对象。它对于避免资源泄露特别有⽤。采⽤所有权模式

#### 3、**shared\_ptr**（共享型，强引⽤）

shared\_ptr 实现共享式拥有概念，多个智能指针可以指向相同对象，该对象和其相关资源会“最后⼀个引⽤被销毁”时候释放。

可以通过成员函数 use\_count() 来查看资源的所有者个数，除了可以通过 new 来构造，还以通过传入auto\_ptr, unique\_ptr,weak\_ptr 来构造。当我们调⽤ release() 时，当前指针会释放资源所有权，计数减⼀。当计数等于 0 时，资源会被释放。

shared\_ptr 是为了解决 auto\_ptr 在对象所有权上的局限性 (auto\_ptr 是独占的)，在使⽤引⽤计数的机制上提供了可以共享所有权的智能指针。

#### 4、**weak\_ptr**（弱引⽤）

**weak\_ptr 是⼀种不控制对象⽣命周期的智能指针，它指向⼀个 shared\_ptr 管理的对象。**进⾏该对象的内存管理的是那个强引⽤的 shared\_ptr。 weak\_ptr 只是提供了对管理对象的⼀个访问⼿段。weak\_ptr 设计的⽬的是为配合 shared\_ptr ⽽引⼊的⼀种智能指针来协助 shared\_ptr ⼯作，它只可以从⼀个 shared\_ptr 或另⼀个 weak\_ptr 对象构造,，**它的构造和析构不会引起引⽤记数的增加或减少。**

weak\_ptr 是⽤来解决 shared\_ptr 相互引⽤时的死锁问题，如果说两个 shared\_ptr 相互引⽤，那么这两个指针的引⽤计数永远不可能下降为0，也就是资源永远不会释放。它是对对象的⼀种弱引⽤，不会增加对象的引⽤计数，和 shared\_ptr 之间可以相互转化，shared\_ptr 可以直接赋值给它，它可以通过调⽤ lock 函数来获得shared\_ptr。

举例：当两个智能指针都是 shared\_ptr 类型的时候，析构时两个资源引⽤计数会减⼀，但是两者引⽤计数还是为 1，导致跳出函数时资源没有被释放（的析构函数没有被调⽤），解决办法：把其中⼀个改为weak\_ptr就可以。

### 7、所有构造函数

#### 1、**⽆参数构造函数：**

即默认构造函数，如果没有明确写出⽆参数构造函数，编译器会⾃动⽣成默认的⽆参数构造函数，函数为空，什么也不做

#### 2、**⼀般构造函数：**

也称重载构造函数，⼀般构造函数可以有各种参数形式，⼀个类可以有多个⼀般构造函数，前提是参数的个数或者类型不同，创建对象时根据传⼊参数不同调⽤不同的构造函数

#### **3、拷⻉构造函数：**

拷⻉构造函数的函数参数为对象本身的引⽤，⽤于根据⼀个已存在的对象复制出⼀个新的该类的对象，⼀般在函数中会将已存在的对象的数据成员的值⼀⼀复制到新创建的对象中。如果没有显示的写拷⻉构造函数，则系统会默认创建⼀个拷⻉构造函数，但当类中有指针成员时，最好不要使⽤编译器提供的默认的拷⻉构造函数，最好⾃⼰定义并且在函数中执⾏深拷⻉。

##### **为什么是引⽤传递，不能是值传递？**

为了防⽌递归调⽤。当⼀个对象需要以值⽅式进⾏传递时，编译器会⽣成代码调⽤它的拷⻉构造函数⽣成⼀个副本，如果类 A 的拷⻉构造函数的参数不是引⽤传递，⽽是采⽤值传递，那么就⼜需要为了创建传递给拷⻉构造函数的参数的临时对象，⽽⼜⼀次调⽤类 A 的拷⻉构造函数，这就是⼀个⽆限递归。

##### **构造函数析构函数不可以抛出异常**

* C++ 只会析构已经完成的对象，对象只有在其构造函数执⾏完毕才算是完全构造妥当。在构造函数中发⽣异常，控制权转出构造函数之外。因此，在对象 b 的构造函数中发⽣异常，对象b的析构函数不会被调⽤。因此会造成内存泄漏。
* **优化：**⽤ auto\_ptr 对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发⽣资源泄漏的危机，不再需要在析构函数中⼿动释放资源；
* 如果控制权基于异常的因素离开析构函数，⽽此时正有另⼀个异常处于作⽤状态，C++ 会调⽤ terminate 函数让程序结束；
* 如果异常从析构函数抛出，⽽且没有在当地进⾏捕捉，那个析构函数便是执⾏不全的。如果析构函数执⾏不全，就是没有完成他应该执⾏的每⼀件事情

#### 4、**赋值运算符的重载**：

这个类似拷⻉构造函数，将＝右边的本类对象的值复制给＝左边的对象，它不属于构造函数，＝左右两边的对象必需已经被创建。如果没有显示的写赋值运算符的重载，系统也会⽣成默认的赋值运算符，做⼀些基本的拷⻉⼯作。

### 8、强制转换

C++ 的四种强制转换包括：**static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast**

1. static\_cast：明确指出类型转换，⼀般建议将隐式转换都替换成显示转换，因为没有动态类型检查，上⾏转换（派⽣类->基类）安全，下⾏转换（基类->派⽣类） 不安全，所以主要执⾏⾮多态的转换操作；
2. dynamic\_cast：专⻔⽤于派⽣类之间的转换，type-id 必须是类指针，类引⽤或 void\*，对于下⾏转换是安全的，当类型不⼀致时，转换过来的是空指针，⽽static\_cast，当类型不⼀致时，转换过来的事错误意义的指针，可能造成⾮法访问等问题。
3. const\_cast：专⻔⽤于 const 属性的转换，去除 const 性质，或增加 const 性质， 是四个转换符中唯⼀⼀个可以操作常量的转换符。
4. reinterpret\_cast：不到万不得已，不要使⽤这个转换符，⾼危操作。使⽤特点： 从底层对数据进⾏重新解释，依赖具体的平台，可移植性差； 可以将整形转 换为指针，也可以把指针转换为数组；可以在指针和引⽤之间进⾏肆⽆忌惮的转换。

### 9、野指针和悬空指针的区别

野指针(wild pointer)：就是没有被初始化过的指针。⽤ gcc -Wall 编译, 会出现 used uninitialized 警告。

悬空指针：是指针最初指向的内存已经被释放了的⼀种指针。

⽆论是野指针还是悬空指针，都是**指向⽆效内存区域**的指针。 访问”不安全可控”(invalid)的内存区域将导致”Undefined Behavior”。

### 10、函数指针

**定义：**函数指针是指向函数的指针变量。函数指针本身⾸先是⼀个指针变量，该指针变量指向⼀个具体的函数。这正如⽤指针变量可指向整型变量、字符型、数组⼀样，这⾥是指向函数。

在编译时，每⼀个函数都有⼀个⼊⼝地址，该⼊⼝地址就是函数指针所指向的地址。有了指向函数的指针变量后，可⽤该指针变量调⽤函数，就如同⽤指针变量可引⽤其他类型变量⼀样，在这些概念上是⼤体⼀致的。

**⽤途**：调⽤函数和做函数的参数，⽐如回调函数

### 11、堆和栈的区别

#### 栈：

* 由编译器进⾏管理，在需要时由编译器⾃动分配空间，在不需要时候⾃动回收空间，⼀般保存的是局部变量和函数参数等。
* 连续的内存空间，在函数调⽤的时候，⾸先⼊栈的主函数的下⼀条可执⾏指令的地址，然后是函数的各个参数。
* ⼤多数编译器中，参数是从右向左⼊栈（原因在于采⽤这种顺序，是为了让程序员在使⽤C/C++的“函数参数⻓度可变”这个特性时更⽅便。如果是从左向右压栈，第⼀个参数（即描述可变参数表各变量类型的那个参数）将被放在栈底，由于可变参的函数第⼀步就需要解析可变参数表的各参数类型，即第⼀步就需要得到上述参数，因此，将它放在栈底是很不⽅便的。）本次函数调⽤结束时，局部变量先出栈，然后是参数，最后是栈顶指针最开始存放的地址，程序由该点继续运⾏，不会产⽣碎⽚。
* 栈是⾼地址向低地址扩展，栈低⾼地址，空间较⼩。

#### 堆：

* 由程序员管理，需要⼿动 new malloc delete free 进⾏分配和回收，如果不进⾏回收的话，会造成内存泄漏的问题。
* 不连续的空间，实际上系统中有⼀个空闲链表，当有程序申请的时候，系统遍历空闲链表找到第⼀个⼤于等于申请⼤⼩的空间分配给程序，⼀般在分配程序的时候，也会空间头部写⼊内存⼤⼩，⽅便 delete 回收空间⼤⼩。当然如果有剩余的，也会将剩余的插⼊到空闲链表中，这也是产⽣内存碎⽚的原因。
* 堆是低地址向⾼地址扩展，空间交⼤，较为灵活。

### 12、**深拷⻉和浅拷⻉的区别**

当出现类的等号赋值时，会调⽤拷⻉函数，在未定义显示拷⻉构造函数的情况下， 系统会调⽤默认的拷⻉函数－即浅拷⻉，它能够完成成员的⼀⼀复制。当数据成员中没有指针时，浅拷⻉是可⾏的。

但当数据成员中有指针时，如果采⽤简单的浅拷⻉，则两类中的两个指针指向同⼀个地址，当对象快要结束时，会调⽤两次析构函数，⽽导致指野指针的问题。

所以，这时必需采⽤深拷⻉。深拷⻉与浅拷⻉之间的区别就在于深拷⻉会在堆内存中另外申请**空间来存储数据，从⽽也就解决来野指针的问题**。简⽽⾔之，当数据成员中有指针时，必需要⽤深拷⻉更加安全。

### 13、**内存对齐**

**结构体内存对⻬⽅式和为什么要进⾏内存对⻬？**

结构体中**内存对⻬的规则**：

* 对于结构体中的各个成员，第⼀个成员位于偏移为 0 的位置，以后的每个数据成员的偏移量必须是 min(#pragma pack() 制定的数，数据成员本身⻓度) 的倍数。
* 在所有的数据成员完成各⾃对⻬之后，结构体或联合体本身也要进⾏对⻬，整体⻓度是min(#pragma pack()制定的数，⻓度最⻓的数据成员的⻓度) 的倍数。

**内存对⻬的作⽤：**

* **经过内存对⻬之后，CPU 的内存访问速度大大提升。**因为 CPU 把内存当成是⼀块⼀块的，块的⼤⼩可以是 2，4，8，16 个字节，因此 CPU 在读取内存的时候是⼀块⼀块进⾏读取的，块的⼤⼩称为内存读取粒度。如果数据是从 1 字节开始的，就⾸先要将前 4 个字节读取到寄存器，并再次读取 4-7 个字节数据进⼊寄存器，接着把 0 字节，5，6，7 字节的数据剔除，最后合并 1，2，3，4字节的数据进⼊寄存器，所以说，当内存没有对⻬时，寄存器进⾏了很多额外的操作，⼤⼤降低了 CPU 的性能。
* **有利于平台移植：**有的 CPU 遇到未进⾏内存对⻬的处理直接拒绝处理，某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常

### 14、内存泄漏

内存泄漏简单的说就是申请了⼀块内存空间，使⽤完毕后没有释放掉。 它的⼀般表现⽅式是程序运⾏时间越⻓，占⽤内存越多，最终⽤尽全部内存，整个系统崩溃。由程序申请的⼀块内存，且没有任何⼀个指针指向它，那么这块内存就泄漏了。

**如何检测内存泄漏：**

* ⾸先可以通过观察猜测是否可能发⽣内存泄漏，Linux 中使⽤ swap 命令观察还有多少可⽤的交换空间，在⼀两分钟内键⼊该命令三到四次，看看可⽤的交换区是否在减少。
* 还可以使⽤ 其他⼀些 /usr/bin/stat ⼯具如 netstat、vmstat 等。如发现波段有内存被分配且从不释放，⼀个可能的解释就是有个进程出现了内存泄漏。
* 当然也有⽤于内存调试，内存泄漏检测以及性能分析的软件开发⼯具 valgrind 这样的⼯具来进⾏内存泄漏的检测。

### 15、程序处理过程

**预处理，编译，汇编，链接程序的区别**

##### **预处理阶段**

写好的⾼级语⾔的程序⽂本⽐如 hello.c，预处理器根据 #开头的命令，修改原始的程序，如#include<stdio.h> 将把系统中的头⽂件插⼊到程序⽂本中，通常是以 .i 结尾的⽂件。

##### 编译

编译器将 hello.i ⽂件翻译成⽂本⽂件 *hello.s*，这个是汇编语⾔程序。⾼级语⾔是源程序。所以注意概念之间的区别。汇编语⾔程序是⼲嘛的？每条语句都以标准的⽂本格式确切描述⼀条低级机器语⾔指令。不同的⾼级语⾔翻译的汇编语⾔相同。

##### 汇编

汇编器将 hello.s 翻译成机器语⾔指令。把这些指令打包成可᯿定位⽬标程序，即.o⽂件。hello.o是⼀个⼆进制⽂件，它的字节码是机器语⾔指令，不再是字符。前⾯两个阶段都还有字符。

##### 链接

⽐如 hello 程序调⽤ printf 程序，它是每个 C 编译器都会提供的标准库 C 的函数。这个函数存在于⼀个名叫 printf.o 的单独编译好的⽬标⽂件中，这个⽂件将以某种⽅式合并到 hello.o 中。链接器就负责这种合并。得到的是可执⾏⽬标⽂件。

### 16、静态编译和动态编译

**静态编译，**编译器在编译可执⾏⽂件时，把需要⽤到的对应动态链接库中的部分提取出来，连接到可执⾏⽂件中去，使可执⾏⽂件在运⾏时不需要依赖于动态链接库；

**动态编译，**可执⾏⽂件需要附带⼀个动态链接库，在执⾏时，需要调⽤其对应动态链接库的命令。所以其优点⼀⽅⾯是缩⼩了执⾏⽂件本身的体积，另⼀⽅⾯是加快了编译速度，节省了系统资源。缺点是哪怕是很简单的程序，只⽤到了链接库的⼀两条命令，也需要附带⼀个相对庞⼤的链接库；⼆是如果其他计算机上没有安装对应的运⾏库，则⽤动态编译的可执⾏⽂件就不能运⾏。

静态链接库与动态链接库都是共享代码的⽅式，如果采⽤静态链接库，则⽆论你愿不愿意，lib中的指令都全部被直接包含在最终⽣成的 EXE ⽂件中了。但是若使⽤ DLL，该 DLL 不必被包含在最终 EXE ⽂件中，EXE ⽂件执⾏时可以“动态”地引⽤和卸载这个与 EXE 独⽴的 DLL ⽂件。

## 二、STL

### 六大组件

容器，算法，迭代器，仿函数，配接器和配置器。彼此可以组合套⽤。容器通过配置器取得数据存储空间，算法通过迭代器存取容器内容，仿函数可以协助算法完成不同的策略变化，配接器可以应⽤于容器、 仿函数和迭代器。

**容器：**各种数据结构，如 vector，list，deque，set，map，⽤来存放数据， 从实现的⻆度来讲是⼀种类模板。

**算法：**各种常⽤的算法，如 sort（插⼊，快排，堆排序），search（⼆分查找）， 从实现的⻆度来讲是⼀种⽅法模板。

**迭代器：**从实现的⻆度来看，迭代器是⼀种将 operator\*,operator->,operator++, operator–等指针相关操作赋予重载的类模板，所有的 STL 容器都有⾃⼰的迭代器。

**仿函数：**从实现的⻆度看，仿函数是⼀种重载了 operator()的类或者类模板。 可以帮助算法实现不同的策略。

**配接器：**⼀种⽤来修饰容器或者仿函数或迭代器接⼝的东⻄。STL提供的queue 和 stack，虽然看似容器，但其实只能算是⼀种容器配接器，因为它们的底部完全借助deque，所有操作都由底层的deque供应。

**配置器：**负责空间配置与管理，从实现的⻆度讲，配置器是⼀个实现了动态空间配置、空间管理，空间释放的类模板。⼀般的分配器的std:alloctor都含有两个函数allocate与deallocte，这两个函数分别调⽤operator new()与delete()，这两个函数的底层又分别是malloc()and free();但是每次malloc会带来格外开销（因为每次malloc⼀个元素都要带有附加信息）

### 容器

#### vector

是动态空间，随着元素的加⼊，它的内部机制会⾃⾏扩充空间以容纳新元素。vector 维护的是⼀个连续的线性空间，⽽且普通指针就可以满⾜要求作为 vector 的迭代器（RandomAccessIterator）。

vector 的数据结构中其实就是三个迭 代器构成的，⼀个指向⽬前使⽤空间头的 iterator，⼀个指向⽬前使⽤空间尾的iterator，⼀个指向⽬前可⽤空间尾的 iterator。

当有新的元素插⼊时，如果⽬前容器够⽤则直接插⼊，如果容器不够，则容器扩充⾄两倍，如果两倍容器不⾜， 就扩张⾄⾜够⼤的容器。扩充的过程并不是直接在原有空间后⾯追加容器，⽽是重新申请⼀块连续空间，将原有的数据拷⻉到新空间中，再释放原有空间，完成⼀次扩充。需要注意的是，每次扩充是重新开辟的空间，所以扩充后，原有的迭代器将会失效。

##### 1、底层实现

**Vector**在堆中分配了⼀段连续的内存空间来存放元素

三个迭代器：

1. **first** ： 指向的是vector中对象的起始字节位置
2. **last** ： 指向当前最后⼀个元素的末尾字节
3. **end** ： 指向整个vector容器所占⽤内存空间的末尾字节

##### 2、扩容过程

如果集合已满，在新增数据的时候，就要分配⼀块更⼤的内存，将原来的数据复制过来，释放之前的内存，在插⼊新增的元素 。所以对vector的任何操作，⼀旦引起空间重新配置，指向原vector的所有迭代器就都失效了。

##### **3**、**resize()**和**reserve()**

resize()：改变当前容器内含有元素的数量(size())，⽽不是容器的容量

reserve()：改变当前容器的最⼤容量（capacity）

#### list

STL list 是⼀个双向链表，普通指针已经不能满⾜ list 迭代器的需求，因为 list 的存储空间是不连续的。list 的迭代器必需具备前移和后退功能，所以 list 提供的是BidirectionalIterator。list 的数据结构中只要⼀个指向 node 节点的指针就可以了。与 vector 相⽐，list 的好处就是每次插⼊或删除⼀个元素，就配置或释放⼀个空间，⽽且原有的迭代器也不会失效。

**vector**和**list**的区别

1. vector底层实现是数组；list是双向链表
2. vector是顺序内存,⽀持随机访问，list不⾏
3. vector在中间节点进⾏插⼊删除会导致内存拷贝，list不会
4. vector⼀次性分配好内存，不够时才进⾏翻倍扩容；list每次插⼊新节点都会进⾏内存申请
5. vector随机访问性能好，插⼊删除性能差；list随机访问性能差，插⼊删除性能好

#### deque

**1、概述**

deque是⼀个双端开⼜的连续线性空间，其内部为分段连续的空间组成，随时可以增加⼀段新的空间并链接。⽀持快速随机访问，由于deque需要处理内部跳转，因此速度上没有vector快。

**2、deque中控器：**

deque是由⼀段⼀段的定量连续空间构成。⼀旦有必要在其头端或者尾端增加新的空间，便配置⼀段定量连续空间，串接在整个deque的头端或者尾端。deque采⽤⼀块**map**作为主控，其中的每个元素都是指针，指向另⼀⽚连续线性空间，称之为缓存区，这个区才是⽤来储存数据的。

好处：避免“vector的重新配置，复制，释放”的轮回，维护连整体连续的假象，并提供随机访问的接⼜；

坏处：其迭代器变得很复杂

3、**deque**数据结构**:**

deque除了维护⼀个map指针以外，还维护了start与finish迭代器分别指向第⼀缓冲区的第⼀个元素，和最后⼀个缓冲区的最后⼀个元素的下⼀个元素，同时它还必须记住当前map的⼤⼩。

#### *stack && queue*

栈与队列被称之为duque的配接器，其底层是以deque为底部架构。通过deque执⾏具体操作

#### *heap && priority\_queue*

**heap**（堆）：

建⽴在完全⼆叉树上，分为两种，⼤根堆，⼩根堆,其在STL中做priority\_queue的助⼿，即，以任何顺序将元素推⼊容器中，然后取出时⼀定是从优先权最⾼的元素开始取，完全⼆叉树具有这样的性质，适合做priority\_queue的底层

**priority\_queue:**

优先队列，也是配接器。其内的元素不是按照被推⼊的顺序排列，⽽是⾃动取元素的权值排列，确省情况下利⽤⼀个max-heap完成，后者是以vector—表现的完全⼆叉树。

#### *map && set*

共同点：都是C++的关联容器,只是通过它提供的接⼜对⾥⾯的元素进⾏访问，底层都是采⽤红⿊树实现。

不同点：

set：⽤来判断某⼀个元素是不是在⼀个组⾥⾯。

map：映射，相当于字典，把⼀个值映射成另⼀个值，可以创建字典。

##### 细节

**1**、为什么要成倍的扩容⽽不是⼀次增加⼀个固定⼤⼩的容量呢？

采⽤成倍⽅式扩容，可以保证常数的时间复杂度，⽽增加指定⼤⼩的容量只能达到O(n)的时间复杂度。

**2**、为什么是以两倍的⽅式扩容⽽不是三倍四倍，或者其他⽅式呢

考虑可能产⽣的堆空间浪费，所以增长倍数不能太⼤，⼀般是1.5或2；GCC是2；VS是1.5，k =2 每次扩展的新尺⼨必然刚好⼤于之前分配的总和，之前分配的内存空间不可能被使⽤，这样对于缓存并不友好，采⽤1.5倍的增长⽅式可以更好的实现对内存的重复利⽤。

#### *map && unordered\_map*

map中元素是⼀些key-value对，关键字起索引作⽤，值表⽰和索引相关的数据。

**底层实现：**

* map底层是基于红⿊树实现的，因此map内部元素排列是有序的。
* ⽽unordered\_map底层则是基于哈希表实现的，因此其元素的排列顺序是杂乱⽆序的。

**map**：

优点：有序性，这是map结构最⼤的优点，其元素的有序性在很多应⽤中都会简化很多的操作。 map的查找、删除、增加等⼀系列操作时间复杂度稳定，都为O(logn )。

缺点：查找、删除、增加等操作平均时间复杂度较慢，与n相关。

**unordered\_map**：

优点：查找、删除、添加的速度快，时间复杂度为常数级O(1）。

缺点：因为unordered\_map内部基于哈希表，以（key,value）对的形式存储，因此空间占⽤率⾼。 unordered\_map的查找、删除、添加的时间复杂度不稳定，平均为O(1)，取决于哈希函数。极端情况下可能为O(n)。

**细节：**

1、为什么insert之后，以前保存的iterator不会失效？

因为 map 和 set 存储的是结点，不需要内存拷⻉和内存移动。但是像 vector 在插⼊数据时如果内存不够会重新开辟⼀块内存。map 和 set 的 iterator 指向的是节点的指针，vector 指向的是内存的某个位置。

2、为何map和set的插⼊删除效率⽐其他序列容器⾼？

因为 map 和 set 底部使⽤红⿊树实现，插⼊和删除的时间复杂度是 O(logn)，⽽向 vector 这样的序列容器插⼊和删除的时间复杂度是 O(N)

#### *pair*容器

保存两个数据成员，⽤来⽣成特定类型的模板。 pair在底层被定义为⼀个struct，其所有成员默认是public，两个成员分别是first和second

### 迭代器

iterator ：**⼀种能够顺序访问容器中每个元素的⽅法，使⽤该**⽅法不能暴露容器内部的表达⽅式。⽽类型萃取技术就是为了要解决和 **iterator** **有关的问题**的。

#### 萃取机

萃取之前的版本有缺陷：不⽀持原⽣指针。⽽通过萃取机的封装，**可以通过类模**板的特化来⽀持原⽣指针的版本！如此⼀来，⽆论是智能指针，还是原⽣指针，iterator\_traits::value\_type 都能起作⽤。

#### 型别

**五种：**

1. value\_type ：迭代器所指对象的类型，原⽣指针也是⼀种迭代器，对于原⽣指针 int\*，int 即为指针所指对象的类型，也就是所谓的 value\_type 。
   * **内嵌型别，用于返回值的类型。参数值类型可以使用模板的参数推导机制**。
   * **并不是所有的迭代器都是class type ，原⽣指针也是⼀种迭代器，由于原⽣指针不是class type ，所以没法为它定义内嵌型别。这个时候就需要用到模板偏特化**
2. difference\_type ： ⽤来表示两个迭代器之间的距离，对于原⽣指针，STL 以 C++ 内建的 ptrdiff\_t 作为原⽣指针的 difference\_type。
3. reference\_type ： 是指迭代器所指对象的类型的引⽤，reference\_type ⼀般⽤在迭代器的 \* 运算符᯿载上，如果 value\_type 是 T，那么对应的 reference\_type 就是 T&；如果value\_type 是 const T，那么对应的reference\_type 就是 const T&。
4. pointer\_type ： 就是相应的指针类型，对于指针来说，最常⽤的功能就是 operator\* 和operator-> 两个运算符。
5. iterator\_category ： 的作⽤是标识迭代器的移动特性和可以对迭代器执⾏的操作，iterator\_category 上，可将迭代器分为 Input Iterator、Output Iterator、ForwardIterator、Bidirectional Iterator、Random Access Iterator 五类，这样分可以尽可能地提⾼效率。

iterator class 不包含任何成员变量，只有类型的定义，因此不会增加额外的负担。这个类主要是⽤来继**承的，在实现具体的迭代器时，可以继承上⾯的类，这样⼦就不会漏掉上⾯的5** 个型别了。

**对应的迭代器萃取机：**

#### 类别

迭代器型别 iterator\_category 对应的迭代器类别，这个类别会限制迭代器的操作和移动特性。

**除了原⽣指针以外**，迭代器被分为五类：

* Input Iterator ： 此迭代器不允许修改所指的对象，是只读的。⽀持 ==、!=、++、*、-> 等操作。*
* *Output Iterator ：允许算法在这种迭代器所形成的区间上进⾏只写操作。⽀持 ++、*等操作。
* Forward Iterator ：允许算法在这种迭代器所形成的区间上进⾏读写操作，但只能单向移动，每次只能移动⼀步。⽀持 Input Iterator 和 Output Iterator 的所有操作。
* Bidirectional Iterator ：允许算法在这种迭代器所形成的区间上进⾏读写操作，可双向移动，每次只能移动⼀步。⽀持 Forward Iterator 的所有操作，并另外⽀持 – 操作。
* Random Access Iterator ：包含指针的所有操作，可进⾏随机访问，随意移动指定的步数。⽀持前⾯四种 Iterator 的所有操作，并另外⽀持 [n] 操作符等操作

## 三、泛型编程

对于c++来说，泛型编程就是模板编程。模板分为类模板与函数模板，特化分为特例化（全特化）和部分特例化（偏特化）。对模板特例化是因为对特定类型，可以利⽤某些特定知识来提⾼效率，⽽不是使⽤通⽤模板。

#### 函数模板：

1. 模板和特例化版本应该声明在同⼀头⽂件，所有同名模板的声明应放在前⾯，接着是特例化版本。
2. ⼀个模板被称为全特化的条件：1.必须有⼀个主模板类 2.模板类型被全部明确化。
3. 函数模板，**只有全特化**，偏特化的功能可以通过函数的重载完成。

#### 类模板

对主版本模板类、全特化类、偏特化类的调⽤优先级从⾼到低进⾏排序是：全特化类>偏特化类>主版本模板类。

## 四、C++11新特性

### 1、空指针nullptr

nullptr 出现的⽬的是为了替代 NULL。专⻔⽤来区分空指针、0。nullptr 的类型为nullptr\_t，能够隐式 的转换为任何指针或成员指针的类型，也能和他们进⾏相等或者不等的⽐较。

C++ 中重载载特性会发⽣混乱：

### 2、Lambda表达式

#### 概述

Lambda 表达式实际上就是提供了⼀个类似匿名函数的特性，⽽匿名函数则是在需要⼀个函数，但是⼜不想费⼒去命名⼀个函数的情况下去使⽤的。利⽤ lambda 表达式可以编写内嵌的匿名函数，⽤以替换独⽴函数或者函数对象，并且使代码更可读。从本质上来讲， lambda 表达式只是⼀种语法糖，因为所有其能完成的⼯作都可以⽤其它稍微复杂的代码来实现。**从⼴义上说， lamdba 表达式产⽣的是函数对象。**

#### 组成

lambda 表达式⼀般都是从⽅括号[]开始，然后结束于花括号{}，花括号⾥⾯就像定义函数那样，包含了 lamdba 表达式体。如果需要参数，那么就要像函数那样，放在圆括号⾥⾯，如果有返回值，返回类型要放在->后⾯，即拖尾返回类型，当然你也可以忽略返回类型，**lambda**会帮你⾃动推断出返回类型：

#### 闭包

最前边的 **[]** 是 lambda 表达式的闭包功能。每当你定义⼀个 lambda 表达式后，编译器会⾃动⽣成⼀个匿名类（这个类当然重载了（）运算符），称为闭包类型（closure type）。那么在运⾏时，这个 lambda 表达式就会返回⼀个匿名的闭包实例，其实⼀个**右值**。所以，我们上⾯的 lambda 表达式的结果就是⼀个个闭包实例。

闭包的⼀个强⼤之处是其可以通过传值或者引⽤的⽅式捕捉其封装作⽤域内的变量，前⾯的⽅括号就是⽤来定义捕捉模式以及变量。

### 3、右值引用

#### 什么是右值引用？为什么会出现右值引用？什么是移动语义？什么是完美转发？

右值引用是 C++11 引入的一种新引用类型，通过 && 符号表示。它允许我们引用临时对象（右值），并且可以通过移动语义来实现高效的资源管理。

右值引用的引入是为了解决两个问题：

1. **临时对象（右值）的高效传递**：传统的引用（左值引用）只能绑定到左值，而不能绑定到右值（临时对象）。右值引用的引入允许我们通过绑定到右值，来将临时对象作为参数传递给函数，避免不必要的拷贝和内存分配，提高效率。
2. **完美转发：**传统的引用（左值引用）在传递参数时，会失去参数的类型信息。而右值引用可以保留参数的类型信息，从而在进行参数转发时，能够实现更加灵活、准确的转发，避免多次拷贝和重复代码。

**移动语义：** 是基于右值引用的特性，通过移动构造函数和移动赋值运算符实现对资源的高效转移。传统的拷贝构造函数和拷贝赋值运算符是将资源复制到新的对象中，**而移动构造函数和移动赋值运算符则是将资源从一个对象转移到另一个对象中，避免不必要的内存分配和释放，提高运行效率**。

**完美转发：** 是指在函数传递参数时，以**原样转发参数**，既能传递左值，也能传递右值，并保留参数的类型信息。通过引入右值引用和模板类型推断，实现了函数参数转发时的完美匹配，并避免了多次拷贝和重复代码的问题。

综上所述，右值引用和移动语义的引入主要是为了提高程序的效率和资源管理的灵活性。而完美转发则是在函数参数传递中，实现了更精确、高效的参数转发。它们是C++11引入的一系列特性，使C++语言更加现代化、高效和易用。

#### 示例

##### 右值引用示例

1. 右值引用和移动语义的示例代码：

#include <iostream>  
#include <string>  
  
// 定义一个具有资源的类，使用动态分配的内存作为资源  
class Resource {  
public:  
 Resource(const std::string& data) : data\_(new std::string(data)) {  
 std::cout << "Resource acquired." << std::endl;  
 }  
  
 // 移动构造函数  
 Resource(Resource&& other) noexcept : data\_(other.data\_) {  
 other.data\_ = nullptr;  
 std::cout << "Resource moved." << std::endl;  
 }  
  
 // 移动赋值运算符  
 Resource& operator=(Resource&& other) noexcept {  
 if (this != &other) {  
 delete data\_;  
 data\_ = other.data\_;  
 other.data\_ = nullptr;  
 std::cout << "Resource moved." << std::endl;  
 }  
 return \*this;  
 }  
  
 ~Resource() {  
 delete data\_;  
 std::cout << "Resource released." << std::endl;  
 }  
  
private:  
 std::string\* data\_;  
};  
  
// 使用右值引用作为参数的函数  
void ProcessResource(Resource&& resource) {  
 // 对资源进行处理...  
}  
  
int main() {  
 Resource a("Hello");  
 Resource b(std::move(a)); // 使用 std::move 将 a 转为右值引用  
  
 Resource c("World");  
 c = std::move(b); // 使用 std::move 进行移动赋值  
  
 ProcessResource(std::move(c)); // 将 c 作为右值传递给函数  
  
 return 0;  
}

输出结果：

Resource acquired.  
Resource moved.  
Resource acquired.  
Resource moved.  
Resource moved.  
Resource released.  
Resource released.  
Resource released.

在示例代码中，我们定义了一个 Resource 类，它具有一个动态分配的内存作为资源。通过定义移动构造函数和移动赋值运算符，我们可以实现资源的高效转移。

在主函数中，我们创建了三个 Resource 对象 a、b 和 c。通过 std::move 将 a 转换为右值引用，并使用移动构造函数创建了 b。然后通过 std::move 将 b 赋值给 c，使用了移动赋值运算符。

最后，我们将 c 作为右值传递给 ProcessResource 函数，演示了通过右值引用传递参数的用法。

##### 完美转发示例

1. 完美转发的示例代码：

#include <iostream>  
#include <vector>  
  
// 函数模板，使用完美转发将参数传递给其他函数  
template <typename T>  
void ForwardFunction(T&& arg) {  
 OtherFunction(std::forward<T>(arg));  
}  
  
// 其他函数，用于演示参数转发  
void OtherFunction(std::vector<int>&& vec) {  
 std::cout << "Received rvalue reference to vector!" << std::endl;  
 // 对参数进行处理...  
}  
  
void OtherFunction(std::vector<int>& vec) {  
 std::cout << "Received lvalue reference to vector!" << std::endl;  
 // 对参数进行处理...  
}  
  
int main() {  
 std::vector<int> data{1, 2, 3, 4, 5};  
  
 ForwardFunction(data); // 传递左值  
 ForwardFunction(std::vector<int>{1, 2, 3, 4, 5}); // 传递右值  
  
 return 0;  
}

输出结果：

Received lvalue reference to vector!  
Received rvalue reference to vector!

在示例代码中，我们定义了一个函数模板 ForwardFunction，使用了完美转发将参数传递给 OtherFunction 函数。OtherFunction 函数重载了左值引用和右值引用两个版本。

在主函数中，创建了一个名为 data 的 vector 对象，并通过 ForwardFunction 分别将 data 和一个临时创建的 vector 对象作为参数进行传递。因为 ForwardFunction 使用了完美转发，所以参数会根据原始参数的左值/右值属性被准确地转发给 OtherFunction。

最终输出结果中，显示了对左值引用和右值引用两个版本的 OtherFunction 函数的调用。这展示了完美转发的能力，使得参数可以按照原样传递给正确的函数重载版本。

### 4、**类型推导**

#### auto /decltype

C++ 提供了 auto 和 decltype 来静态推导类型，在我们知道类型没有问题但⼜不想完整地写出类型的时候， 便可以使⽤静态类型推导。

decltype ⽤于获取⼀个表达式的类型，⽽不对表达式进⾏求值（类似于sizeof ）。 decltyp(e) 规则如下：

* 若 e 为⼀个⽆括号的变量、函数参数、类成员，则返回类型为该变量/参数/类成员在源程序中的声明类型；
* 否则的话，根据表达式的值分类（value categories），设设 T 为 e 的类型：的类型：
  + 若 e 是⼀个左值（lvalue，即“可寻址值”），返回 T& ；
  + 若 e 是⼀个临终值（xvalue），则返回值为 T&& ；
  + 若 e 是⼀个纯右值（prvalue），则返回值为 T 。

### 5、**正则表达式**

### 6、哈希表

C++ 的 map , multimap , set , multiset 使⽤红⿊树实现， 插⼊和查询都是 O(lgn) 的复杂度，但 C++11 为这四种模板类提供了（底层哈希实现）以达到 O(1) 的复杂度：

### 7、统一的初始化列表

# 算法&数据结构

## 算法

### 排序算法

#### 冒泡排序：

冒泡优化：定义一个isSorted标志位，如果有一轮没有交换表示有序，置isSorted为True，break

#### 选择排序：

#### 插入排序：（适用于数组基本有序）

#### 快速排序：

#### 归并排序

#include<iostream>  
#include <thread>  
#include <vector>  
#include <functional>  
using namespace std;  
  
void mergeCount(int a[],int L,int mid,int R) {  
 int \*tmp = new int[L+mid+R];  
 int i=L;  
 int j=mid+1;  
 int k=0;  
 while( i<=mid && j<=R ) {  
 if(a[i] < a[j])  
 tmp[k++] = a[i++];  
 else  
 tmp[k++] = a[j++];  
 }  
 while( i<=mid )  
 tmp[k++] = a[i++];  
 while( j<=R)  
 tmp[k++] = a[j++];  
 for(int p=0; p<k; ++p)  
 a[L+p] = tmp[p];  
 delete tmp;  
}  
  
void mergeSort(int a[],int L,int R) {  
 if( L>=R ) { return; }  
 int mid = (R + L)/2;  
 mergeSort(a,L,mid);  
 mergeSort(a,mid+1,R);  
 mergeCount(a,L,mid,R);  
 // thread t1(std::bind(mergeSort,std::ref(a),L,mid));  
 // thread t2(std::bind(mergeSort,std::ref(a),mid+1,R));  
 // t1.join();  
 // t2.join();  
 // mergeCount(a,L,mid,R);  
   
}  
  
int main() {  
 int a[] = {0,34,66,2,5,95,4,46,27};  
 mergeSort(a, 0, sizeof(a)/sizeof(int));  
 for(int i=0; i<=8; ++i) {  
 std::cout<<a[i]<<" "; // print => 0 2 4 5 27 34 46 66 95  
 }  
 std::cout<<endl;  
 return 0;  
}

## 数据结构

### 树

#### 平衡⼆叉树、二叉搜索树

**⼆叉搜索树：**⼆叉搜索树可以**提供对数时间的元素插⼊和访问**。

* **节点的放置规则**是：任何节点的键值⼀定⼤于其左⼦树的每⼀个节点的键值，并⼩于其右⼦树中的每⼀个节点的键值。因此⼀直向左⾛可以取得最⼩值，⼀直向右⾛可以得到最⼤值。
* **插⼊：**从根节点开始，遇键值较⼤则向左，遇键值较⼩则向右，直到尾端，即插⼊点。删除：如果删除点只有⼀个⼦节点，则直接将其⼦节点连⾄⽗节点。如果删除点有两个⼦节点，以右⼦树中的最⼩值代替要删除的位置。

**平衡⼆叉树：**其实对于树的平衡与否没有⼀个绝对的标准，“平衡”的⼤致意 思是：没有任何⼀个节点过深，不同的平衡条件会造就出不同的效率表现。以及不同的实现复杂度。有数种特殊结构例如 AVL-tree, RB-tree, AA-tree，均可以实现平衡⼆叉树。

* AVL-tree ：⾼度平衡的平衡⼆叉树（严格的平衡⼆叉树）AVL-tree 是要求任何节点的左右⼦树⾼度相差最多为 1 的平衡⼆叉树。 当插⼊新的节点破坏平衡性的时候，从下往上找到第⼀不平衡点，需要进⾏单旋转，或者双旋转进⾏调整。

#### 红黑树

红⿊树的定义：

* 性质1：每个节点要么是⿊⾊，要么是红⾊。
* 性质2：根节点是⿊⾊。
* 性质3：每个叶⼦节点（NIL）是⿊⾊。
* 性质4：每个红⾊结点的两个⼦结点⼀定都是⿊⾊。
* **性质**5：任意⼀结点到每个叶⼦结点的路径都包含数ᰁ相同的⿊结点。

# helens项目

### 项目介绍

这个项目是在学习网络编程的时候开发的一个高性能服务器框架，服务器的网络模型是基于协程开发的，IO处理基于epoll使用了IO多路复用技术，可以处理多个客户端的http请求。项目工作分为两部分，一部分是服务器框架的基本模块的封装，比如日志模块，配置模块、网络模块、socket模块、线程模块等一些基本系统的搭建。另一部分是为了提高服务器性能做的一些优化，比如协程模块，协程调度模块、io协程调度模块。最后此项目也是成功部署再华为云服务器上，并且基于此框架实现了一个聊天室的项目，聊天室的项目参与不多，基本是负责和开发此聊天室项目的同学进行联调，对服务器框架进行一些完善，包括添加了长连接模式，添加了数据库的接口实现等。

最后使用apache的ab压测工具在相同的机器环境下对HTTP的访问做了压力测试，对页面访问和nginx对比，性能可能要差%5左右，和libevent相比性能基本持平。ab -n

在开发此项目期间参考了一些开源框架的实现？（这块再想一下）

sylar

### **项目中的难点**

该服务器框架主要分为两个部分：第一部分关于服务器基本系统的搭建，难点在于技术的理解和选型，以及一些开源的框架调整后应用到项目中。第二部分的难点主要集中在提高服务器性能方面，包括找到服务器的性能瓶颈，如何突破？（这块在想一下）：协程调试上、并发上，展现异步性能，一核有时候会报错

## 项目中遇到的困难？如何解决的？

1、一方面是对不同技术的理解和选型，难以选出最合适的技术框架。这部分的话基本上是阅读和对比各个开源项目的代码和技术文档，然后也去搜索一些技术对比的论文、博客去看，如果没有相关资料，尝试联系一些提供开源代码的大佬们。

2、编程期间遇到的困难，在代码编写过程中总会出现各种各样的bug。这也是我为什么做日志模块的一个很大的驱动力。一般通过日志定位bug，然后推断bug的原因尝试修复，如果自己看不出来可能就到网上看一下此类问题的解决办法，或者到一些知名论坛stackoverflow、csdn或者找同学等。

## 针对项目做了哪些优化？

1、程序本身

* 减少程序等待IO的事件：非阻塞+IO复用
* 涉及协程相关的优化（这块再想一下，仔细展开）
  + 线程池
* 锁相关
  + 优化锁的使用，尽量减少临界区（线程池的时候）
* hook模块
  + 通过hook模块，可以使得一些不具备异步功能的API，展现出异步的性能，比如mysql

2、系统参数调优（这块再搜集一下）

* 最大文件描述符（用户级和系统级）
* tcp连接的参数（半连接/连接队列的长度、TCP sync cookies）

## 项目用到了哪些设计模式？（这块要收集一下）

单例：封装了单例工具类

观察者模式：

## 面向对象特性再项目中的体现

封装，继承，多态

封装：在项⽬中将各个模块使⽤类进⾏封装，⽐如连接httpconnection类来封装，日志就⽤ log 类来封装，将类的属性私有化，⽐如请求的解析状态，并且对外的接⼝设置为公有，⽐如连接的重置，不对外暴露⾃身的私有⽅法，⽐如读写的回调函数等。还有⼀个就是，项⽬中每个模块都使⽤了各⾃的命名空间进行封装，避免了命名冲突或者名字污染。**还有就是每个类都会定义一个share\_ptr的变量名，这样方便用智能指针进行管理，防止不必要的内存泄漏**

继承：主要是对工具类的继承（单例模式），项目中多个地方到 noncopyable 和enable\_shared\_from\_this，保证了代码的复用性。还有比较重要的一个继承就是IOmanger类对协程调度器的继承，既使用了协程调度器的一部功能，又添加了epoll的功能，将两个模块更融洽的结合在一起。

多态：主要是用了静态多态。比如日志系统中对流运算符的重载，以及各种函数模板的泛型编程。

## 项目某些重点模块的细节

#### 日志模块

* **目的：**
* 用于格式化输出程序日志，方便从日志中定位程序运行过程中出现的问题。这里的日志除了日志内容本身之外，还应该包括文件名/行号，时间戳，线程/协程号，模块名称，日志级别等额外信息，甚至在打印致命的日志时，还应该附加程序的栈回溯信息，以便于分析和排查问题。
* **支持的内容**
  + 支持流式日志风格写日志和格式化风格写日志
  + 流式日志使用：HELENS\_LOG\_INFO(g\_logger) << "this is a log"; 格式化日志使用：HELENS\_LOG\_FMT\_INFO(g\_logger, "%s", "this is a log");
  + 支持日志格式自定义、日志级别、多日志分离等等功能
  + 支持时间,线程id,线程名称,日志级别,日志名称,文件名,行号等内容的自由配置
* **具备的功能**
  1. 区分不同的级别，比如常的DEBUG/INFO/WARN/ERROR等级别。日志模块可以通过指定级别实现只输出某个级别以上的日志，这样可以灵活开关一些不重要的日志输出，比如程序在调试阶段可以设置一个较低的级别，以便看到更多的调度日志信息，程序发布之后可以设置一个较高的级别，以减少日志输出，提高性能。
  2. 区分不同的输出地。不同的日志可以输出到不同的位置，比如可以输出到标准输出，输出到文件，输出到syslog，输出到网络上的日志服务器等，甚至同一条日志可以同时输出到多个输出地。
  3. 区分不同的类别。日志可以分类并命名，一个程序的各个模块可以使用不同的名称来输出日志，这样可以很方便地判断出当前日志是哪个程序模块输出的。
  4. 日志格式可灵活配置。可以按需指定每条日志是否包含文件名/行号、时间戳、线程/协程号、日志级别、启动时间等内容。
  5. 可通过配置文件的方式配置以上功能。
* **参考**
* C++日志框架log4cpp
  1. Category对应日志类别，使用不同的名称来区别日志器，使用 log4cpp::Category::getInstance() 获取指定名称的日志器实例，如果两个日志器名称相同，那么对应同一个日志器实例。
  2. 使用Category的的setPriority()方法设置日志器的日志级别，日志级别使用log4cpp::Priority枚举值来表示，一共有FATAL/ALERT/CRIT/ERROR/WARN/NOTICE/INFO/DEBUG/NOTSET几个等级。
  3. 使用Appender来表示日志输出地，Appender可以细分为OstreamAppender和FileAppender等不同类型。一个Category可以有多个Appender，这样一条日志就可以输出到多个地方，通过addAppender()方法为Category新增Appender。
  4. 使用PatternLayout来表示日志的格式，格式通过模板字符串来指定，比如%d表示时间（后面可用{}指定具体的时间格式），%R表示UTC秒数，%c表示日志器名称，%p表示日志级别，%m表示日志消息等。
  5. PatternLayout和Appender绑定，Priority和Category绑定，一条日志经过Category判断级别通过后由Appender输出，Appender输出的格式由PatternLayout指定。
* **类设计**
* class LogFormatter;  
  class LogAppender;  
  class Logger;  
  class LogEvent;  
  class LogEventWrap;  
  class LogManager;
  1. LogFormatter: 日志格式器，与log4cpp的PatternLayout对应，用于格式化一个日志事件。该类构建时可以指定pattern，表示如何进行格式化。提供format方法，用于将日志事件格式化成字符串。
  + 用户并不希望每次输出日志时都将全部进行输出，而是希望可以自由地选择要输出的。并且，用户还可能需要在每条日志里增加一些指定的字符，比如在文件名和行号之间加上一个冒号的情况。为了实现这项功能，LogFormatter使用了一个模板字符串来指定格式化的方式
  + 模板字符串由普通字符和转义字符构成，转义字符以%开头，比如%m，%p等。除了转义字符，剩下的全部都是普通字符，包括空格。
  + LogFormatter根据模板字符串来格式化日志事件。首先，在构造LogFormatter对象时会指定一串模板字符，LogFormatter会首先解析该模板字符串，将其中的转义字符和普通字符解析出来。然后，在格式化日志事件时，根据模板字符串，将其中的转义字符替换成日志事件的具体内容，普通字符保持不变。当前实现支持以下转义字符：
  + %m 消息``%p 日志级别``%c 日志器名称``%d 日期时间，后面可跟一对括号指定时间格式，比如%d{%Y-%m-%d %H:%M:%S}，这里的格式字符与C语言``strftime``一致``%r 该日志器创建后的累计运行毫秒数``%f 文件名``%l 行号``%t 线程id``%F 协程id``%N 线程名称``%% 百分号``%T 制表符``%n 换行
  1. LogAppender: 日志输出器，用于将一个日志事件输出到对应的输出地。该类内部包含一个LogFormatter成员和一个log方法，日志事件先经过LogFormatter格式化后再输出到对应的输出地。从这个类可以派生出不同的Appender类型，比如StdoutLogAppender和FileLogAppender，分别表示输出到终端和文件。
  2. Logger: 日志器，负责进行日志输出。一个Logger包含多个LogAppender和一个日志级别，提供log方法，传入日志事件，判断该日志事件的级别高于日志器本身的级别之后调用LogAppender将日志进行输出，否则该日志被抛弃。
  3. LogEvent: 日志事件，用于记录日志现场，比如该日志的级别，文件名/行号，日志消息，线程/协程号，所属日志器名称等。
  + 日志事件，用于记录日志现场，具体包括以下内容：
    - 日志内容
    - 日志器名称
    - 日志级别
    - 文件名，对应\_\_FILE\_\_宏
    - 行号，对应\_\_LINE\_\_宏
    - 程序运行时间，通过sylar::GetElapsedMS()获取
    - 线程ID
    - 协程ID
    - UTC时间戳，对应time(0)
    - 线程名称
  + **日志事件的构造需要通过宏来简化，否则，每次生成一个日志事件时都要对上面这些内容进行赋值，够麻烦的。**
  1. LogEventWrap: 日志事件包装类，其实就是将日志事件和日志器包装到一起，因为一条日志只会在一个日志器上进行输出。将日志事件和日志器包装到一起后，方便通过宏定义来简化日志模块的使用。另外，LogEventWrap还负责在构建时指定日志事件和日志器，在析构时调用日志器的log方法将日志事件进行输出。
  2. LogManager: 日志器管理类，单例模式，用于统一管理所有的日志器，提供日志器的创建与获取方法。LogManager自带一个root Logger，用于为日志模块提供一个初始可用的日志器。

#### 配置模块

采用约定由于配置的思想。定义即可使用。不需要单独去解析。支持变更通知功能。使用YAML文件做为配置内容。支持级别格式的数据类型，支持STL容器(vector,list,set,map等等),支持自定义类型的支持（需要实现序列化和反序列化方法)使用方式如下：

static helens::ConfigVar<int>::ptr g\_tcp\_connect\_timeout =  
 helens::Config::Lookup("tcp.connect.timeout", 5000, "tcp connect timeout");

定义了一个tcp连接超时参数，可以直接使用 g\_tcp\_connect\_timeout->getValue() 获取参数的值，当配置修改重新加载，该值自动更新 上述配置格式如下：

tcp:  
 connect:  
 timeout: 10000

#### 线程模块

线程模块，封装了pthread里面的一些常用功能，Thread,Semaphore,Mutex,RWMutex,Spinlock等对象，可以方便开发中对线程日常使用 为什么不适用c++11里面的thread 本框架是使用C++11开发，不使用thread，是因为thread其实也是基于pthread实现的。并且C++11里面没有提供读写互斥量，RWMutex，Spinlock等，在高并发场景，这些对象是经常需要用到的。所以选择了自己封装pthread

#### 协程模块

协程：用户态的线程，相当于线程中的线程，更轻量级。后续配置socket hook，可以把复杂的异步调用，封装成同步操作。降低业务逻辑的编写复杂度。 目前该协程是基于ucontext\_t来实现的，后续将支持采用boost.context里面的fcontext\_t的方式实现

#### 协程调度模块

协程调度器，管理协程的调度，内部实现为一个线程池，支持协程在多线程中切换，也可以指定协程在固定的线程中执行。是一个N-M的协程调度模型，N个线程，M个协程。重复利用每一个线程。

#### io协程调度模块

继承与协程调度器，封装了epoll（Linux），并支持定时器功能（使用epoll实现定时器，精度毫秒级）,支持Socket读写时间的添加，删除，取消功能。支持一次性定时器，循环定时器，条件定时器等功能

#### hook模块

hook系统底层和socket相关的API，socket io相关的API，以及sleep系列的API。hook的开启控制是线程粒度的。可以自由选择。通过hook模块，可以使一些不具异步功能的API，展现出异步的性能。如（mysql）

#### Socket模块

封装了Socket类，提供所有socket API功能，统一封装了地址类，将IPv4，IPv6，Unix地址统一起来。并且提供域名，IP解析功能。

#### ByteArray序列化模块

ByteArray二进制序列化模块，提供对二进制数据的常用操作。读写入基础类型int8\_t,int16\_t,int32\_t,int64\_t等，支持Varint,std::string的读写支持,支持字节序转化,支持序列化到文件，以及从文件反序列化等功能

#### tcpServert模块

基于Socket类，封装了一个通用的TcpServer的服务器类，提供简单的API，使用便捷，可以快速绑定一个或多个地址，启动服务，监听端口，accept连接，处理socket连接等功能。具体业务功能更的服务器实现，只需要继承该类就可以快速实现

#### http模块

采用Ragel（有限状态机，性能媲美汇编），实现了HTTP/1.1的简单协议实现和uri的解析。基于SocketStream实现了HttpConnection(HTTP的客户端)和HttpSession(HTTP服务器端的链接）。基于TcpServer实现了HttpServer。提供了完整的HTTP的客户端API请求功能，HTTP基础API服务器功能

# 中兴项目

## 项目背景

探针项目是公司统一的数据源平台，支持多种网络制式，多种业务分析，多种数据源采集。探针项目是vmax大数据产品，核心网软采，以及其他需要采集核心网数据源的产品提供数据源支撑，开发了解密，视频，重复信令去重，语音mos等功能；

技术上，需要熟悉全网通讯协议，对通讯网络中的原始码流进行采集解码，流量过大，对性能要求过高，涉及网元，接口，协议，项目内部集成了dpi技术；

市场方面，主要集中在通讯网络采集，涵盖2345g，各种制式以及无线侧数据全网采集，和大数据产品vmax配套提供感知，网优，运维，网络检测，安全等领域解决方案；另外也可以作为通讯网络独立数据源产品对接第三方应用。

### 1、新需求的开发：平台代码新增协程模块

#### 需求目的

最终想要实现一个类似于PhotonLibOS协程库的东西，后期和DPDK结合，简化DPDK应用程序的开发，并且增加更多的功能，比如文件IO。**但是现在项目中的DPDK只使用了简单的数据链路层高性能包转发的功能**，为了之后将DPDK支持TCP/IP协议栈，必须进行的两个工作，第一个是用户态TCP/IP协议栈，**第二个是高性能的IO处理过程。**

#### 用户态TCP/IP协议栈：

用户态TCP/IP协议栈：关于开源的一些尝试：一是移植现有的协议栈如FreeBSD，另一种就是完全重写。这些基于DPDK的开源项目的测试数据通常能够获得比Linux内核的原生socket更好的性能。关于DPDK用户态TCP/IP协议栈的开发不是我在进行，我测试协程代码库的时候使用了腾讯开源的F-Stack（基于DPDK并且集成了FreeBSD），集成了 FreeBSD做了一些裁剪，提供了一组POSIX API，如socket、epoll、kqueue等，提供用户态的网络协议栈。看了F-Stack 示例代码，它目前的核心代码仍然是 while 循环的模式，需要提供一个loop，然后注册到大循环内部执行，网卡的收发包代码被安排在了loop的上下文附近。**虽然说已经提供了事件接口（这是协程化的必备条件），但由于没有调度器，所以整体代码风格跟之前的DPDK仍然差不多。**

#### IO多路复用（见操作系统IO复用）

在IO多路复用中实现业务逻辑时，随着事件的等待和就绪，需要频繁的保存和恢复现场，适合使用协程。

在IO多路复用这里，事件循环依然存在，依然要一个处理就绪的fd，但是处理内容不是具体业务而是面向协程的调度。

* 如果是用于监听端口的fd就绪了，就建立链接创建一个新的fd，交给一个协程来负责，协程执行入口就指向业务处理函数入口，业务处理过程中，需要等待时就注册IO事件，然后让出，执行权回到切换协程前的地方继续执行。
* 如果是其他等待IO事件的fd就绪了，只需要恢复关联的协程即可。

IO多路复用这一层的事件循环就和具体业务解耦了，可以重新包装一下read等函数，在其中实现IO事件注册和主动让出，这样在业务逻辑层就可以使用这些包装函数按照常规业务逻辑实现函数。

#### 实现（待补充）

**所谓创建协程**：其实就是把一个函数包装成一个协程对象，然后再用协程的方式把这个函数跑起来；

**所谓协程调度：**，其实就是创建一批的协程对象，然后再创建一个调度协程，通过调度协程把这些协程对象一个一个消化掉（协程可以在被调度时继续向调度器添加新的调度任务）；

**所谓IO协程调度：**，其实就是在调度协程时，如果发现这个协程在等待IO就绪，那就先让这个协程让出执行权，等对应的IO就绪后再重新恢复这个协程的运行；

##### 协程封装？

基于ucontext\_t实现非对称协程

协程模块基于ucontext\_t实现，在学习之前，必须对ucontext\_t和ucontext\_t的操作函数非常熟悉。关于ucontext\_t的定义和相关的接口如下：

// 上下文结构体定义  
// 这个结构体是平台相关的，因为不同平台的寄存器不一样  
// 下面列出的是所有平台都至少会包含的4个成员  
typedef` `struct` `ucontext\_t {  
 ``// 当前上下文结束后，下一个激活的上下文对象的指针，只在当前上下文是由makecontext创建时有效  
 ``struct` `ucontext\_t \*uc\_link;  
 ``// 当前上下文的信号屏蔽掩码  
 ``sigset\_t uc\_sigmask;  
 ``// 当前上下文使用的栈内存空间，只在当前上下文是由makecontext创建时有效  
 ``stack\_t uc\_stack;  
 ``// 平台相关的上下文具体内容，包含寄存器的值  
 ``mcontext\_t uc\_mcontext;  
 ``...  
} ucontext\_t;  
  
// 获取当前的上下文  
int` `getcontext(ucontext\_t \*ucp);  
// 恢复ucp指向的上下文，这个函数不会返回，而是会跳转到ucp上下文对应的函数中执行，相当于变相调用了函数  
int` `setcontext(``const` `ucontext\_t \*ucp);  
// 修改由getcontext获取到的上下文指针ucp，将其与一个函数func进行绑定，支持指定func运行时的参数，  
// 在调用makecontext之前，必须手动给ucp分配一段内存空间，存储在ucp->uc\_stack中，这段内存空间将作为func函数运行时的栈空间，  
// 同时也可以指定ucp->uc\_link，表示函数运行结束后恢复uc\_link指向的上下文，  
// 如果不赋值uc\_link，那func函数结束时必须调用setcontext或swapcontext以重新指定一个有效的上下文，否则程序就跑飞了  
// makecontext执行完后，ucp就与函数func绑定了，调用setcontext或swapcontext激活ucp时，func就会被运行  
void` `makecontext(ucontext\_t \*ucp, ``void` `(\*func)(), ``int` `argc, ...);  
// 恢复ucp指向的上下文，同时将当前的上下文存储到oucp中，  
// 和setcontext一样，swapcontext也不会返回，而是会跳转到ucp上下文对应的函数中执行，相当于调用了函数  
// swapcontext是sylar非对称协程实现的关键，线程主协程和子协程用这个接口进行上下文切换  
int` `swapcontext(ucontext\_t \*oucp, ``const` `ucontext\_t \*ucp);

**使用非对称协程模型：**，也就是子协程只能和线程主协程切换，而不能和另一个子协程切换，并且在程序结束时，一定要再切回主协程，以保证程序能正常结束，像下面这样：

* 实现借助了线程局部变量
  + 借助了线程局部变量的功能来实现协程模块。线程局部变量与全局变量类似，不同之处在于声明的线程局部变量在每个线程都独有一份，而全局变量是全部线程共享一份。
  + 用线程局部变量（C++11 thread\_local变量）来保存协程上下文对象，这点很好理解，因为协程是在线程里运行的，不同线程的协程相互不影响，每个线程都要独自处理当前线程的协程切换问题。
  + 对于每个线程的协程上下文，设计了两个线程局部变量来存储上下文信息（对应源码的t\_fiber和t\_thread\_fiber），也就是说，一个线程在任何时候最多只能知道两个协程的上下文。
  + 如果将线程的局部变量设置成一个类似链表的数据结构，那理论上应该也可以实现**对称协程**，也就是子协程可以直接和子协程切换，但代码复杂度上肯定会增加不少，因为要考虑多线程和公平调度的问题。
  + **优缺点：**非对称协程代码实现简单，并且在后面实现协程调度时可以做到公平调度，缺点是子协程只能和线程主协程切换，意味着子协程无法创建并运行新的子协程，并且在后面实现协程调度时，完成一次子协程调度需要额外多切换一次上下文。
* 协程状态
* READY，代表就绪态，RUNNING，代表正在运行，TERM，代表运行结束、INIT状态，HOLD状态，和EXCEPT状态。
* INIT状态是协程对象刚创建时的状态，这个状态可以直接归到READY状态里，sylar的HOLD状态和READY状态与协程调度有关，READY状态的协程会被调度器自动重新调度，而HOLD状态的协程需要显式地再次将协程加入调度，这两个状态也可以归到READY状态里，反正都表示可执行状态。sylar还给协程设计了一个EXCEPT状态，表示协程入口函数执行时出现异常的状态。
* 协程原语
* 对于非对称协程来说，协程除了创建语句外，只有两种操作，一种是resume，表示恢复协程运行，一种是yield，表示让出执行。协程的结束没有专门的操作，协程函数运行结束时协程即结束，协程结束时会自动调用一次yield以返回主协程
* 协程类实现
* 通过Fiber类来表示，这个类包含以下
  + 成员变量：
* /// 协程id``uint64\_t m\_id = 0;``/// 协程栈大小``uint32\_t m\_stacksize = 0;``/// 协程状态``State m\_state = READY;``/// 协程上下文``ucontext\_t m\_ctx;``/// 协程栈地址``void` `\*m\_stack = nullptr;``/// 协程入口函数``std::function<``void``()> m\_cb;
  + 全局变量和线程局部变量。
* Fiber的源码定义了两个全局静态变量，用于生成协程id和统计当前的协程数，如下：
* /// 全局静态变量，用于生成协程id``static` `std::atomic<uint64\_t> s\_fiber\_id{0};``/// 全局静态变量，用于统计当前的协程数``static` `std::atomic<uint64\_t> s\_fiber\_count{0};
* 然后是线程局部变量，对于每个线程，sylar设计了以下两个线程局部变量用于保存协程上下文信息：
* /// 线程局部变量，当前线程正在运行的协程``static` `thread\_local Fiber \*t\_fiber = nullptr;``/// 线程局部变量，当前线程的主协程，切换到这个协程，就相当于切换到了主线程中运行，智能指针形式``static` `thread\_local Fiber::ptr t\_thread\_fiber = nullptr;
* 这两个线程局部变量保存的协程上下文对协程的实现至关重要，它们的用途如下：
* t\_fiber：保存当前正在运行的协程指针，必须时刻指向当前正在运行的协程对象。协程模块初始化时，t\_fiber指向线程主协程对象。
* t\_thread\_fiber：保存线程主协程指针，智能指针形式。协程模块初始化时，t\_thread\_fiber指向线程主协程对象。当子协程resume时，通过swapcontext将主协程的上下文保存到t\_thread\_fiber的ucontext\_t成员中，同时激活子协程的ucontext\_t上下文。当子协程yield时，从t\_thread\_fiber中取得主协程的上下文并恢复运行。
  + 协程类Fiber的成员方法。
* 首先是协程的构建函数。Fiber类提供了两个构造函数，带参数的构造函数用于构造子协程，初始化子协程的ucontext\_t上下文和栈空间，要求必须传入协程的入口函数，以及可选的协程栈大小。不带参的构造函数用于初始化当前线程的协程功能，构造线程主协程对象，以及对t\_fiber和t\_thread\_fiber进行赋值。这个构造函数被定义成私有方法，不允许在类外部调用，只能通过GetThis()方法，在返回当前正在运行的协程时，如果发现当前线程的主协程未被初始化，那就用不带参的构造函数初始化线程主协程。因为GetThis()兼具初始化主协程的功能，在使用协程之前必须显式调用一次GetThis()
  + 协程原语实现resume和yield
  + 
  + image-20230830140848433
  + 其他实现细节
    - 关于协程id。sylar通过全局静态变量s\_fiber\_id的自增来生成协程id，每创建一个新协程，s\_fiber\_id自增1，并作为新协程的id（实际是先取值，再自增1）。
    - 关于线程主协程的构建。线程主协程代表线程入口函数或是main函数所在的协程，这两种函数都不是以协程的手段创建的，所以它们只有ucontext\_t上下文，但没有入口函数，也没有分配栈空间。
    - 关于协程切换。子协程的resume操作一定是在主协程里执行的，主协程的resume操作一定是在子协程里执行的，这点完美和swapcontext匹配，参考上面协程原语的实现。
    - 关于智能指针的引用计数，由于t\_fiber和t\_thread\_fiber一个是原始指针一个是智能指针，混用时要注意智能指针的引用计数问题，不恰当的混用可能导致协程对象已经运行结束，但未析构问题。关于协程对象的智能指针引用计数跟踪可参考test\_fiber.cc。

##### 协程调度器设计？

实现了一个N-M的协程调度器，N个线程运行M个协程，协程可以在线程之间进行切换，也可以绑定到指定线程运行。

实现协程调度之后，可以解决协程模块中子协程不能运行另一个子协程的缺陷，子协程可以通过向调度器添加调度任务的方式来运行另一个子协程。

**整体设计：**

调度器内部维护一个任务队列和一个调度线程池。开始调度后，线程池从任务队列里按顺序取任务执行。调度线程可以包含caller线程。当全部任务都执行完了，线程池停止调度，等新的任务进来。添加新任务后，通知线程池有新的任务进来了，线程池重新开始运行调度。停止调度时，各调度线程退出，调度器停止工作。

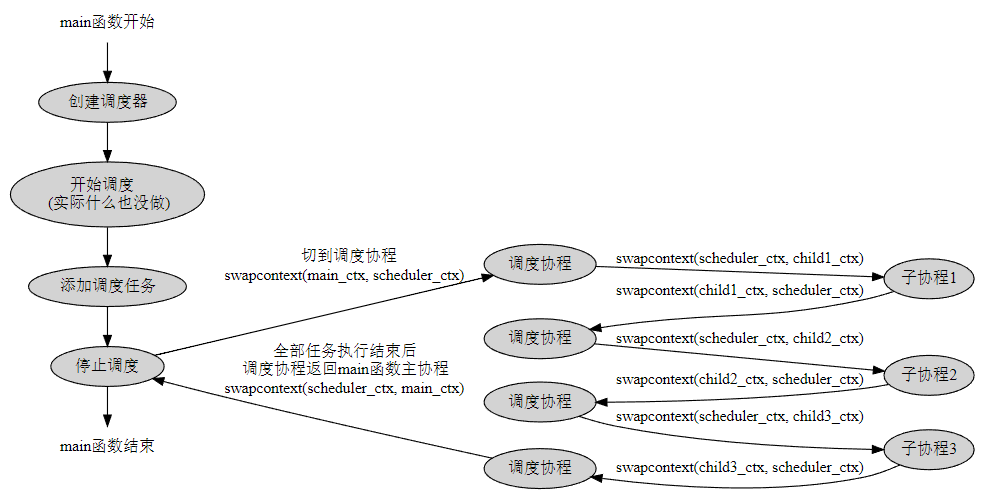


image-20230830141616569

##### io协程调度设计？

继承自协程调度器，封装了epoll，支持为socket fd注册读写事件回调函数。

IO协程调度还解决了调度器在idle状态下忙等待导致CPU占用率高的问题。IO协程调度器使用一对管道fd来tickle调度协程，当调度器空闲时，idle协程通过epoll\_wait阻塞在管道的读描述符上，等管道的可读事件。添加新任务时，tickle方法写管道，idle协程检测到管道可读后退出，调度器执行调度。

在前面的协程调度模块中，调度器对协程的调度是无条件执行的，在调度器已经启动调度的情况下，任务一旦添加成功，就会排队等待调度器执行。调度器不支持删除调度任务，并且调度器在正常退出之前一定会执行完全部的调度任务，所以在某种程度上可以认为，把一个协程添加到调度器的任务队列，就相当于调用了协程的resume方法。

IO协程调度支持协程调度的全部功能，因为IO协程调度器是直接继承协程调度器实现的。除了协程调度，IO协程调度还增加了IO事件调度的功能，这个功能是针对描述符（一般是套接字描述符）的。IO协程调度支持为描述符注册可读和可写事件的回调函数，当描述符可读或可写时，执行对应的回调函数。（这里可以直接把回调函数等效成协程，所以这个功能被称为IO协程调度）

IO事件调度功能对服务器开发至关重要，因为服务器通常需要处理大量来自客户端的socket fd，使用IO事件调度可以将开发者从判断socket fd是否可读或可写的工作中解放出来，使得程序员只需要关心socket fd的IO操作。后续的socket api hook模块也依赖IO协程调度。

很多的库都可以实现类似的工作，比如libevent，libuv，libev等，这些库被称为异步事件库或异步IO库，从网上可以很容易地找到大把的资料介绍这类库。有的库不仅可以处理socket fd事件，还可以处理定时器事件和信号事件。

这些事件库的实现原理基本类似，都是先将套接字设置成非阻塞状态，然后将套接字与回调函数绑定，接下来进入一个基于IO多路复用的事件循环，等待事件发生，然后调用对应的回调函数。这里可以参考一个基于epoll实现的简单事件库：[3.2 epoll的反应堆模式实现 · libevent深入浅出 · 看云](https://www.kancloud.cn/aceld/libevent_aceld/1858559)，IO调度和这种写法类似。

##### 验证

我将写的协程调度器与F-Stack的while循环集成到一起，创建协程调度器，协程调度器使用epoll作为调度器的主事件引擎，可以注册关注的fd，不管是文件fd还是socket，当I/O事件完成时，对应的协程会被唤醒，并执行后续操作。

#### 测试结果：

使用产品商用版本的测试网卡（100G的）进行测试，服务端单线程，包大小512字节，ping流量类型，主要是针对啊吞吐量进行一个测试，大约可以实现的5gb/s的流量。当时没有和PhotonLibOS库做对比，因为当时没有使用过这个库，开发完成之后可能还需要和用户态TCP/IP协议栈模块完成之后在联调测试。

#### 最大的困难

1、可能还是在于协程的实现以及和后面的DPDK进行联调方面，因为之前没有接触过DPDK，所以学习使用DPDK联调测试这块比较困难。

2、集中在协程调度的实现上，如何更加高效的实现协程之间的一个调度。

#### 后续

后续有时间的话可能1、在关键路径加入汇编代码进一步提高性能 。

#### DPDK

**DPDK**程序在每个CPU核心上运行的函数都是一个大的 while 循环。在循环中，可以添加网卡收发包的代码，以及对应的业务逻辑，整体上的架构是一个异步回调的事件模型。由于是 polling，需要让一次循环尽快结束以便开启下一轮，因此代码中一般不能出现长时间的阻塞调用，如sleep。业务逻辑一般只关注网络，比如网关、防火墙这种典型应用，涉及到文件I/O的功能，可能还需要通过跨线程通信的方式，转交给专门的I/O线程去执行。

内核旁路的主要思想是Linux只用来处理控制流；所有数据流都在用户空间中处理。因此，内核旁路可以避免内核数据包复制、线程调度、系统调用和中断带来的性能瓶颈。（DPDK就是一种内核旁路技术）

### 2、招标用户面落盘文件的统计工具的开发：

使用python进行的一个统计工具开发，在项目经过数据采集和事件回填之后，生成大量的话单，在话单入库之前，需要对生成的话单进行统计，包括按照话单用户号段、隧道、网络协议进行筛选，并且计算上下行流量，还有各种回填字段的回填率。

项目数据很大，解压后大约600G左右，由很多个小txt文件组成，每个文件大小在几kb大小不等，每个文件里包含若干行数据，且每个文件的数据长度不一致，甚至会遇到空文件。不了解数据具体情况前，进行统计的时候时候，至少需要遍历一遍所有文件，如果使用pandas的read\_csv方法去循环遍历读取，速度非常慢。

##### 实现技术功能：

1. 读取ini配置文件的类，这样可以方便工具使用人员的更加灵活的进行配置路径、进程数、筛选条件以及统计内容等。
2. 设计了多进程处理文件，以便充分利用服务器多CPU的性能。

* 多进程方面，可以将所以的文件路径放到一个list中，利用多进程分别处理其中每个文件。
  1. 获取所有文件的路径，去重；
  2. 将文件按照数量平均分配到各个组中；
  3. open方法取代pandas的read\_csv方法读取文件，每组内多个文件用多线程处理，同时每个样本组用多进程处理；

##### 技术难点：

* 统计速度，效率
  + 利用多cpu，使用python的multipricessing 库
  + python中的多线程无法利用多核优势，如果想要充分的使用多核CPU的资源，在python中大部分情况需要使用多进程。python中提供了 multipricessing 模块多进程-Pool进程池模块，该模块用于开启进程，并在子进程中执行我们定制的任务，比如函数。可以提供指定数量的进程供用户调用，当有新的请求提交到pool中时，如果池还没有满，那么就会创建一个新的进程用来执行该请求；但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来执行它。
  + class multiprocessing.pool.Pool([processes[, initializer[, initargs[, maxtasksperchild[, context]]]]])
  + # 导入multiprocessing模块  
    import multiprocessing  
    # 导入multiprocessing模块下的Process类  
    from multiprocessing import Process
  + 多进程之间的通信，然后汇总
    1. 进程之间不共享全局变量,当一个进程对全局变量中的数据进行修改，对于其他进程而言，不会造成任何影响，可以理解为每个进程拿的都是最初的全局变量，或者可以理解为全局变量就是所谓的资源，当创建一个进程，则系统会直接给这个进程复制一个全局变量，针对于这个全局变量而言，再进程之间都是相互独立存在的，之间没有任何的关系。
    2. 所以需要统计的全局数据只能通过进程间通信的方式进行处理。多个子进程间的通信就要采用Queue，比如有以下的需求，一个子进程向队列中写数据，另外一个进程从队列中取数据。但是queue使用了1个线程互斥锁(pthread.Lock())，以及3个条件标量(pthread.condition()),来保证了线程安全。这样处理虽然使用了多进程，并且实现了进程间全局变量的通信，但是会导致使用该全局变量的进程对这个变量加锁，其他进程只能等待，实际运行中速度还是会很慢
    3. **最后解决：**干脆放弃进程间通信，每个进程统计自己的数据，最后将所有进程的统计结果进行汇总，虽然最后汇总增加了一下工作量，但是针对要处理的几百G的数据来说，这点处理时间可以忽略不记。实现将5分钟处理150G的数据。

### 3、项目数据采集：

协议分析模块：使用dpdk接管网卡，将码流读取出来，然后进行实时分析，并生成规则的话单。其中要进行码流的解码，还有一些加密数据需要解密等等

事件回填模块：因为协议分析模块生成的话单是用户面的数据，不包括控制面的数据，比如号段、地域、小区等，所以需要该模块将控制面的数据回填到话单中。

每一个模块中间使用一个中转模块xdr，方便使用抓包工具在数据传输过程中进行数据抓取，然后分析数据是否正确。

### 4、安全探针预研

模块预研，创新压力，需要新增安全模块，最终决定将入侵检测的安全模块加入到项目中。调研了几个开源的入侵检测模块，最终决定使用snort，将其集成到项目中。

##### 原因如下：

1. 具有实时数据流量分析和记录IP网络数据包的能力，能够进行协议分析，对网络数据包内容进行搜索/匹配。
2. 它能够检测各种不同的攻击方式，对攻击进行实时报警。
3. Snort是开源的入侵检测系统，它扩展性和可移植性很好，可以很容易的集成到其他系统中。

snort拥有三大基本功能：嗅探器、数据包记录器和入侵检测。

* 嗅探器模式仅从网络上读取数据包并作为连续不断的流显示在终端上，常用命令snort -dev，公司的探针项目可以支持这部分的工作，并且更加高效。
* 数据包记录器模式是把数据包记录到硬盘上，常用命令snort -b，项目中不需要。
* 网络入侵检测模式是最复杂的，而且是可配置的。我们可以让Snort分析网络数据流以匹配用户定义的一些规则，并根据检测结果采取一定的动作。最终决定将这部分代码模块集成到项目中，在生成话单时，如果检测到码流有风险，就在话单中设置标志位进行判断。

最终实现的功能，将入侵模块集成到了项目数据采集模块之后，进行分析，取消了snort原本的报警模式，取而代之的是将报警类型在话单字段中进行体现。简单设置了一些安全规则，包括告警IP的连接等，然后话单字段设置了一个1位的字段，进行判断是否有安全问题。

后续可能增加字段位数来判断是何种告警以及是否需要进行处理和屏蔽，如何进行屏蔽等、进一步降低对性能的影响。

##### 难点：

1、调研合适的开源组件。需要实时性好、易扩展、支持动态扩展检测规则。

2、集成过程。这部分是在同事的合作下共同完成的，因为代码需要基于自己平台库和函数进行开发，有很多函数不熟悉，需要其他同事的帮助。

## NFC项目

1、NFC运行原理，**NFC驱动开发**，钱包升级、刷卡异常率、

2、学习相关协议模型

3、集成到穿戴设备中

## TP-LINK

#### 1、jenkins & docker

##### jenkinsfile

**使用Jenkinsfile构建Pipeline**，Pipeline是用户定义的CI/CD模型，比如编译、测试和发布。由Groovy语言实现。Jenkinsfile放在项目根目录，随项目一起受源代码管理软件控制。

Jenkinsfile使用两种语法进行编写，分别是**声明式**和**脚本式**：声明式是Jenkins流水线更友好的特性，使编写和读取流水线**代码更容易设计**；脚本式则提供**更丰富的语法特性**。

* **agent{}**
* 用于制定构建所用的节点，定义好流水线在某节点的运行。
  + 在pipeline{}中必须定义agent{}，作为所有stage{}的默认代理。
  + 在单个stage{}中可选定义agent{}只作用于该阶段。
* **parameters{}：**用于构建参数
* **environment{}：**用于**定义环境变量**，它们会保存为Groovy变量和Shell环境变量
* 其中，在最前面定义的环境变量的生命周期是全局的；在stage里定义的环境变量是局部的。Jenkins有一些**内置变量**也可以通过env字典读取（env也可以读取用户自己定义的环境变量）。
* 常见的内置变量有：
  + **BUILD\_ID**：Jenkins构建需要，比如：12，在Jenkins 1.597之后的版本里等同于BUILD\_NUMBER
  + **BUILD\_NUMBER**：Jenkins构建序号
  + **BUILD\_TAG**：比如jenkins-{BUILD\_NUMBER}
  + **BUILD\_URL**：Jenkins某次构建的链接
  + **NODE\_NAME**：当前构建使用的机器
* **stages{}**

​ pipeline{} 流水线的主要内容写在 stages{} 中，其中可以定义一个或多个 stage{} ，表示执行的各个阶段，Jenkins 会按先后顺序执行各个 stage{} ，并显示执行进度。

1. 每个 stage{} 的名称不能重复。
2. 每个 stage{} 可以包含一些复杂的选项，比如：
   * stages{} // 嵌套
   * steps{} // 具体执行什么
   * when{} // 满足条件时才会执行
   * environment{} // 局部的环境变量
   * input{} // 暂停某个阶段的执行，等待用户输入参数
   * script{} // 执行脚本

* **post{}**

用于当构建状态满足某些条件时，才执行的操作。pipeline出现语法错误时，Jenkins会直接报错，而不会执行post部分。**发邮件**

1. 可用范围：pipeline{}、stage{}
2. 可用条件：success, failure, unstable, aborted, unsuccessful, always, cleanup, changed

##### docker

###### 一些问题

* **docker是怎么工作的?**
* 实际上docker使用了常见的CS架构，也就是client-server模式，docker client负责处理用户输入的各种命令，比如docker build、docker run，真正工作的其实是server，也就是docker demon，值得注意的是，docker client和docker demon可以运行在同一台机器上。
* Docker是一个Client-Server结构的系统，Docker守护进程运行在主机上， 然后通过Socket连接从客户端访问，守护进程从客户端接受命令并管理运行在主机上的容器。守护进程和客户端可以运行在同一台机器上。
* **docker的组成包含哪几大部分？**
* 一个完整的docker有以下几个部分组成： 1、docker client，客户端，为用户提供一系列可执行命令，用户用这些命令实现跟 docker daemon 交互； 2、docker daemon，守护进程，一般在宿主主机后台运行，等待接收来自客户端的请求消息； 3、docker image，镜像，镜像run之后就生成为docker容器； 4、docker container，容器，一个系统级别的服务，拥有自己的ip和系统目录结构；运行容器前需要本地存在对应的镜像，如果本地不存在该镜像则就去镜像仓库下载。
* **docker技术的三大核心概念是什么？**
  + 镜像：
  + 镜像是一种轻量级、可执行的独立软件包，它包含运行某个软件所需的所有内容，我们把应用程序和配置依赖打包好形成一个可交付的运行环境(包括代码、运行时需要的库、环境变量和配置文件等)，这个打包好的运行环境就是image镜像文件。
  + 容器：
  + 容器是基于镜像创建的，是镜像运行起来之后的一个实例，容器才是真正运行业务程序的地方。如果把镜像比作程序里面的类，那么容器就是对象。
  + 镜像仓库：
  + 存放镜像的地方，研发工程师打包好镜像之后需要把镜像上传到镜像仓库中去，然后就可以运行有仓库权限的人拉取镜像来运行容器了。
* **基本的Docker使用流程**
  + 一切都从Dockerfile开始。Dockerfile是镜像的源代码。
  + 创建Dockerfile后，您可以构建它以创建容器的镜像。镜像只是“源代码”的“编译版本”，即Dockerfile。
  + 获得容器的镜像后，应使用注册表重新分发容器。注册表就像一个git存储库 - 你可以推送和拉取镜像。
  + 接下来，您可以使用该镜像来运行容器。在许多方面，正在运行的容器与虚拟机（但没有管理程序）非常相似。
* **Docker 安全么？**
* Docker 利用了 Linux 内核中很多安全特性来保证不同容器之间的隔离，并且通过签名机制来对镜像进行验证。大量生产环境的部署证明，Docker 虽然隔离性无法与虚拟机相比，但仍然具有极高的安全性。
* **Docker 与 虚拟机 有何不同？**
* 1、传统虚拟机是需要安装整个操作系统的，然后再在上面安装业务应用，启动应用，通常需要几分钟去启动应用，而docker是直接使用镜像来运行业务容器的，其容器启动属于秒级别； 2、Docker需要的资源更少，Docker在操作系统级别进行虚拟化，Docker容器和内核交互，几乎没有性能损耗，而虚拟机运行着整个操作系统，占用物理机的资源就比较多; 3、Docker更轻量，Docker的架构可以共用一个内核与共享应用程序库，所占内存极小;同样的硬件环境，Docker运行的镜像数远多于虚拟机数量，对系统的利用率非常高; 4、与虚拟机相比，Docker隔离性更弱，Docker属于进程之间的隔离，虚拟机可实现系统级别隔离;
* 5、虚拟化创建是分钟级别的，Docker容器创建是秒级别的，Docker的快速迭代性，决定了无论是开发、测试、部署都可以节省大量时间;
* **Dockerfile 中最常见的指令是什么？**
  + FROM：指定基础镜像
  + LABEL：功能是为镜像指定标签
  + RUN：运行指定的命令
  + CMD：容器启动时要运行的命令
* **centos镜像几个G，但是docker centos镜像才几百兆，这是为什么？**
  + 一个完整的Linux操作系统包含Linux内核和rootfs根文件系统，即我们熟悉的/dev、/proc/、/bin等目录。平时看到的centOS除了rootfs，还会选装很多软件，服务，图形桌面等，所以centOS镜像有好几个G。
  + 而对于容器镜像而言，所有容器都是共享宿主机的Linux 内核的，而对于docker镜像而言，docker镜像只需要提供一个很小的rootfs根文件系统即可，只需要包含即我们熟悉的/dev、/proc/、/bin等目录，这是最基本的命令，工具，程序库即可，所以，docker镜像才会这么小。

###### docker信息常用命令和日志

* 查看docker信息

# 查看docker版本  
docker version  
# 显示docker系统的信息  
docker info  
# 日志信息  
docker logs  
# 故障检查  
service docker status  
# 启动关闭docker  
sudo service docker start|stop

* 查看容器日志

docker logs -f <容器名orID>

* docker daemon 日志位置

也称之为 引擎日志 根据系统不同各不相同

\* CoreOS - journalctl -u docker.service  
\* Ubuntu(16.04) - journalctl -u docker.service  
\* Ubuntu(14.04) - /var/log/upstart/docker.log  
\* Boot2Docker - /var/log/docker.log  
\* Debian GNU/Linux 8 - journalctl -u docker.service  
\* Debian GNU/Linux 7 - /var/log/daemon.log  
\* CentOS 7/RHEL 7 - journalctl -u docker.service  
\* CentOS - /var/log/daemon.log | grep docker  
\* Fedora - journalctl -u docker.service  
\* OpenSuSE - journalctl -u docker.service  
\* Red Hat Enterprise Linux Server - /var/log/messages | grep docker  
\* OSX - ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/log/d‌ocker.log

* journalctl 日志工具使用

# 最后行数的日志  
journalctl -n  
# 详细信息  
journalctl -f  
# 本次启动后的所有日志  
journalctl -b  
# 查看启动记录  
journalctl --list-boots  
# 查看某次运行过程中的日志  
sudo journalctl -b [启动顺序号，或者启动hash]  
# 查看记录中指定单元 docker.service 的日志  
journalctl -u docker.service

* 显示可以匹配日志级别

journalctl -F PRIORITY

比如显示 docker 内核日志中 info 类的数据

# 只显示错误、冲突和重要告警信息  
journalctl -u docker.service -p err..alert  
# -p 等效于 PRIORITY 也可以用数字来表示  
journalctl -u docker.service PRIORITY=6  
# 还可以加入 MESSAGE、MESSAGE\_ID、\_PID、\_UID、\_HOSTNAME、\_SYSTEMD\_UNIT 等条件  
# 查询字段 `man 7 systemd.journal-fields`  
# 例如查询 info 信息，PID 为 2331  
journalctl -u docker.service PRIORITY=6 \_PID=2331

###### 容器常用命令

docker容器可以理解为在沙盒中运行的进程

这个沙盒包含了该进程运行所必须的资源，包括文件系统、系统类库、shell 环境等等。但这个沙盒默认是不会运行任何程序的。你需要在沙盒中运行一个进程来启动某一个容器。这个进程是该容器的唯一进程，所以当该进程结束的时候，容器也会完全的停止。

* 查看容器信息

# 查看当前运行的容器  
docker ps  
# 查看全部容器  
docker ps -a  
# 查看全部容器的id和信息  
docker ps -a -q  
# 查看全部容器占用的空间  
docker ps -as  
# 查看一个正在运行容器进程，支持 ps 命令参数  
docker top  
# 查看容器的示例id  
sudo docker inspect -f '{{.Id}}' [id]  
# 检查镜像或者容器的参数，默认返回 JSON 格式  
docker inspect  
# 返回 ubuntu:14.04 镜像的 docker 版本  
docker inspect --format '{{.DockerVersion}}' ubuntu:14.04  
docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' ubuntu:14.04

# 保存对容器的修改  
docker commit  
# 保存某个容器成为一个镜像  
docker commit -a "user" -m "commit info" [CONTAINER] [imageName]:[imageTag]  
# 推送一个容器到中心仓库  
docker login --username=[userName] --password=[pwd] [registryURL]  
## 建议登录后查看 docker info  
docker tag [imageID] [remoteURL]:[imageTag]  
docker push [remoteURL]:[imageTag]  
# 拉取提交的容器  
docker pull [remoteURL]:[imageTag]  
# 对比容器的改动  
docker diff  
# 附加到一个运行的容器上  
docker attach

* **创建删除容器**

# 创建一个容器命名为 test 使用镜像daocloud.io/library/ubuntu  
docker create -it --name test daocloud.io/library/ubuntu  
# 创建并启动一个容器 名为 test 使用镜像daocloud.io/library/ubuntu  
docker run --name test daocloud.io/library/ubuntu  
# 删除一个容器  
docker rm [容器id]  
# 删除所有容器  
docker rm `docker ps -a -q`  
# 根据Dockerfile 构建  
docker build -t [image\_name] [Dockerfile\_path]

* docker容器随系统自启

docker run --restart=always

1. no – 默认值，如果容器挂掉不自动重启
2. on-failure – 当容器以非 0 码退出时重启容器,同时可接受一个可选的最大重启次数参数 (e.g. on-failure:10).
3. always – 不管退出码是多少都要重启

**容器资源限制参数**

# 限制内存最大使用  
-m 1024m --memory-swap=1024m  
# 限制容器使用CPU  
--cpuset-cpus="0,1"

* 把一个正在运行的容器保存为镜像

docker commit <CONTAIN-ID> <IMAGE-NAME>

* 启动停止容器等操作

docker start|stop|restart [id]  
# 暂停|恢复 某一容器的所有进程  
docker pause|unpause [id]  
# 杀死一个或多个指定容器进程  
docker kill -s KILL [id]  
# 停止全部运行的容器  
docker stop `docker ps -q`  
# 杀掉全部运行的容器  
docker kill -s KILL `docker ps -q`

* 交互式进入容器

sudo docker exec -it {{containerName or containerID}} bash  
sudo docker exec -i {{containerName or containerID}} bash  
sudo docker exec -t {{containerName or containerID}} bash  
sudo docker exec -d {{containerName or containerID}} bash

1. 只用 -i 参数，由于没有分配伪终端，看起来像pipe执行一样。但是执行结果、命令返回值都可以正确获取
2. 只用 -t 参数，则可以看到一个 console 窗口，但是执行命令会发现由于没有获得stdin的输出，无法看到命令执行情况
3. 使用 -it 时，则和我们平常操作 console 界面类似，而且也不会像attach方式因为退出，导致整个容器退出
4. 使用 -d 参数，在后台执行一个进程。如果一个命令需要长时间进程，会很快返回

* Docker attach

Docker attach可以attach到一个已经运行的容器的stdin，然后进行命令执行的动作

docker attach {{containerName or containerID}}

需要注意的是，如果从这个stdin中exit，会导致容器的停止

* 查看容器的root用户密码

docker logs <容器名orID> 2>&1 | grep '^User: ' | tail -n1

因为Docker容器启动时的root用户的密码是随机分配的。所以，通过这种方式就可以得到容器的root用户的密码

* 容器于宿主拷贝文件

docker cp [OPTIONS] CONTAINER:SRC\_PATH DEST\_PATH|-  
docker cp [OPTIONS] SRC\_PATH|- CONTAINER:DEST\_PATH  
# 本地文件上传到对应容器的目录  
docker cp local.sh [CONTAINERid]:[TagPath]

* 运行一个新容器，同时为它命名、端口映射、文件夹映射

docker run --name redmine -p 9003:80 -p 9023:22 -d -v /var/redmine/files:/redmine/files -v /var/redmine/mysql:/var/lib/mysql sameersbn/redmine

* 一个容器连接到另一个容器

docker run -i -t --name sonar -d -link mmysql:db tpires/sonar-server sonar

* 导入导出容器

# 支持远程文件 .tar, .tar.gz, .tgz, .bzip, .tar.xz, .txz  
docker import  
# 导出  
docker export [id] >~/Downloads/ubuntu\_nexus.tar

###### 镜像操作常用

docker login

docker search  
# 搜索处收藏数不小于 3 ，并且能够自动化构建的 django 镜像，并且完整显示镜像描述  
docker search -s 3 --automated --no-trunc django  
docker pull  
# 拉取ubuntu最新的镜像  
docker pull ubuntu:latest  
# 服务器拉取个人动态，可选择时间区间  
docker events  
# 拉取个人从 2015/07/20 到 2015/08/08 的个人动态  
docker events --since="20150720" --until="20150808"

* 镜像同步操作

# 标记本地镜像，将其归入某一仓库  
docker tag  
# 将 ID 为 5db5f84x1261 的容器标记为 mine/lnmp:0.2 镜像  
docker tag 5db5f84x1261 mine/lnmp:0.2  
# 将镜像推送至远程仓库，默认为 Docker Hub  
docker push

* 本地镜像

# 列出本地所有镜像  
docker images  
# 本地镜像名为 ubuntu 的所有镜像  
docker images ubuntu  
# 查看指定镜像的创建历史  
docker history [id]  
# 本地移除一个或多个指定的镜像  
docker rmi  
# 移除本地全部镜像  
docker rmi `docker images -a -q`  
# 指定镜像保存成 tar 归档文件， docker load 的逆操作  
docker save  
# 将镜像 ubuntu:14.04 保存为 ubuntu14.04.tar 文件  
docker save -o ubuntu14.04.tar ubuntu:14.04  
# 从 tar 镜像归档中载入镜像， docker save 的逆操作  
docker load  
# 上面命令的意思是将 ubuntu14.04.tar 文件载入镜像中  
docker load -i ubuntu14.04.tar  
docker load < /home/save.tar  
# 构建自己的镜像  
docker build -t <镜像名> <Dockerfile路径>  
docker build -t xx/gitlab .

###### docker run

1. 后台运行(-d)、并暴露端口

docker run -d -p 127.0.0.1:33301:22 centos6-ssh

1. run 命令详解

-a, --attach=[] Attach to STDIN, STDOUT or STDERR 指定标准输入输出内容类型，可选 STDIN/STDOUT/STDERR 三项  
 --add-host=[] Add a custom host-to-IP mapping (host:ip)  
 --blkio-weight=0 Block IO (relative weight), between 10 and 1000  
 -c, --cpu-shares=0 CPU shares (relative weight)  
 --cap-add=[] Add Linux capabilities  
 --cap-drop=[] Drop Linux capabilities  
 --cgroup-parent= Optional parent cgroup for the container  
 --cidfile= Write the container ID to the file  
 --cpu-period=0 Limit CPU CFS (Completely Fair Scheduler) period  
 --cpu-quota=0 Limit the CPU CFS quota  
 --cpuset-cpus= CPUs in which to allow execution (0-3, 0,1) 绑定容器到指定CPU运行  
 --cpuset-mems= MEMs in which to allow execution (0-3, 0,1) 绑定容器到指定MEM运行  
 -d, --detach=false Run container in background and print container ID 后台运行容器，并返回容器ID  
 --device=[] Add a host device to the container  
 --dns=[] Set custom DNS servers 指定容器使用的DNS服务器，默认和宿主一致  
 --dns-search=[] Set custom DNS search domains 指定容器DNS搜索域名，默认和宿主一致  
 -e, --env=[] Set environment variables 设置环境变量  
 --entrypoint= Overwrite the default ENTRYPOINT of the image  
 --env-file=[] Read in a file of environment variables 从指定文件读入环境变量  
 --expose=[] Expose a port or a range of ports  
 -h, --hostname= Container host name 指定容器的hostname  
 --help=false Print usage  
 -i, --interactive=false Keep STDIN open even if not attached 以交互模式运行容器，通常与 -t 同时使用  
 --ipc= IPC namespace to use  
 -l, --label=[] Set meta data on a container  
 --label-file=[] Read in a line delimited file of labels  
 --link=[] Add link to another container  
 --log-driver= Logging driver for container  
 --log-opt=[] Log driver options  
 --lxc-conf=[] Add custom lxc options  
 -m, --memory= Memory limit  
 --mac-address= Container MAC address (e.g. 92:d0:c6:0a:29:33)  
 --memory-swap= Total memory (memory + swap), '-1' to disable swap  
 --name= Assign a name to the container 为容器指定一个名称  
 --net=bridge Set the Network mode for the container 指定容器的网络连接类型，支持 bridge/host/none/container:<name|id> 四种类型  
 --oom-kill-disable=false Disable OOM Killer  
 -P, --publish-all=false Publish all exposed ports to random ports  
 -p, --publish=[] Publish a container's port(s) to the host  
 --pid= PID namespace to use  
 --privileged=false Give extended privileges to this container  
 --read-only=false Mount the container's root filesystem as read only  
 --restart=no Restart policy to apply when a container exits  
 --rm=false Automatically remove the container when it exits  
 --security-opt=[] Security Options  
 --sig-proxy=true Proxy received signals to the process  
 -t, --tty=false Allocate a pseudo-TTY 为容器重新分配一个伪输入终端，通常与 -i 同时使用  
 -u, --user= Username or UID (format: <name|uid>[:<group|gid>])  
 --ulimit=[] Ulimit options  
 --uts= UTS namespace to use  
 -v, --volume=[] Bind mount a volume  
 --volumes-from=[] Mount volumes from the specified container(s)  
 -w, --workdir= Working directory inside the container

##### **1.1 实现代码速度上报以及精细化度量的Devops任务上报。**

###### **精细化度量**

* **CI测试失败率**
* 测量 CI 管道因测试失败而失败的频率。测试是一个安全网，因此失败并没有什么问题。尽管如此，开发人员应该在提交代码之前在他们的机器上运行测试。如果失败率太高，则可能表明开发人员发现很难在本地运行测试。
* **CI 成功率**
* CI 成功率是 CI 成功运行的次数除以运行总数。成功率低表明 CI/CD 过程很脆弱，需要更多维护，或者开发人员过于频繁地合并未经测试的代码。
* **CI 每天运行**
* 这是每天 CI 管道执行的数量。希望保持这个数字较高——每个活跃开发人员至少运行 4 或 5 次——因为这意味着开发人员信任并依赖 CI/CD 流程。 当每天运行的 CI 减少时，可能是由于 CI/CD 系统缓慢或难以使用造成的。
* **CI 平均恢复时间 (MTTR)**
* 当构建不起作用时，我们无法测试、发布或部署。在这种情况下，每个人都应该停止正在做的事情，专注于恢复构建。平均恢复时间衡量团队修复损坏的 CI 构建平均需要多长时间。在衡量这个指标时，我们通常只关心主分支。
* 代码覆盖率
* 代码覆盖率是测试套件覆盖的代码的百分比。这有点有争议，因为众所周知，这是一个经常被滥用的指标。例如，要求 100% 的覆盖率并不能提高质量——相反，它会导致对琐碎代码进行不必要的测试。
* **缺陷逃逸率**
* 测量 CI/CD 过程未检测到的错误数量。高值意味着测试不充分。在这种情况下，我们应该检查覆盖率值，然后重新评估测试套件的结构。我们的测试套件中可能需要更多类型的测试。

###### 代码速度上报

提交-》编译-》单元测试-》冒烟测试-》集成测试-》回归测试

———-c0——–c1——–c2——c3——c4

###### 自动化流程

* 提交：工程师将代码在本地测试后，提交到版本控制系统，如 Git代码仓库中。
* 构建：持续整合系统（如Jenkins CI），在检测到版本控制系统更新时，便自动从Git代码仓库里拉取最新的代码，进行编译、构建。
* 单元测试：Jenkins完成编译构建后，会自动执行指定的单元测试代码。
* 部署到测试环境：在完成单元测试后，Jenkins可以将应用程序部署到与生产环境相近的测试环境中进行测试。
* 预生产环境测试：在预生产测试环境里，可以进行一些最后的自动化测试，例如使用Appium自动化测试工具进行测试，以及与实际情况类似的一些测试可由开发人员或客户人员手动进行测试。
* 部署到生产环境：通过所有测试后，便可以使用灰度更新将最新的版本部署到实际生产环境里。

##### **1.2 完成项目的jenkins配合docker容器化编译的工作**

###### 需求背景

当时所有的编译环境均是在本机上，每一种编译环境都需要占据一台机器，不仅浪费了资源，而且不易管理，没有容错，一旦某台机器发生瘫痪，会导致编译环境恢复难度增加。于是将所有的编译环境集成到docker中，使用一个个docker镜像来保存每一个编译环境，然后使用一台服务器来与运行docker，将所有的编译环境对应的镜像与运行成容器，然后在jenkins中将docker配置进去，修改jenkins file将编译环境集成到docker中，这样所有的编译环境都集中到一台机器上，而且只要机器上有docker就可以快速恢复。

###### 难点

1. docker镜像的制作
   1. 之前没有使用过docker，所以使用起来有一些困难，包括制作docker镜像，使用docker容器等等。后面熟悉docker基本操作之后，就很快了。
      1. 将编译环境配置成tar包，然后将tar包打包成docker镜像，使用export和import命令
      2. 拉取一个新的docker的编译镜像，然后在这个镜像的容器中配置编译环境，最后将该容器到宝成一个新的编译环境镜像，然后push到制品库中。之后只需要拉去该镜像就可以使用该编译环境了。
2. jenkins的文件配置
   1. docker环境和jenkins的融合
   2. 多线程执行dockers容器，因为有时候一副代码需要在几种环境上跑，而这些环境都部署在一台机器的docker容器中，会导致编译速度降低。

###### 流程

首先通过Docker容器保存好完整的开发环境，在Jenkins任务中调用容器进行工程的编译。

1. 如何在Jenkins中操作Docker？
   * 当前通过容器方式启动Jenkins，为了在容器内部操作Docker，需要在启动命令时配置如下参数，将宿主机docker挂载到容器中

* $ docker run -it -v "/var/run/docker.sock:/var/run/docker.sock" -v "/usr/bin/docker:/usr/bin/docker" --privileged zjykzj/ubuntu:18.04 bash
  + docker-compose.yml配置Jenkins如下:主要是将映射的端口号和映射的路径
  + version: "3.7"  
    services:   
     jenkins:  
     labels:  
     AUTHOR: "zhujian <zjzstu@github.com>"  
     container\_name: jenkins  
     user: jenkins  
     image: jenkins/jenkins:latest  
     volumes:   
     - "jenkins\_home:/var/jenkins\_home"  
     - "/var/run/docker.sock:/var/run/docker.sock"  
     - "/usr/bin/docker:/usr/bin/docker"  
     ports:   
     - "7070:8080"  
     - "50000:50000"  
     restart: always  
     tty: true  
     stdin\_open: true  
     privileged: true  
    volumes:   
     jenkins\_home:  
     external: true

1. 如何在任务中使用Docker容器？
   * 首先需要创建pipeline任务，然后在Jenkinsfile文件中配置docker容器。

* pipeline {  
   agent {  
   docker { image 'node:7-alpine' }  
   }  
   stages {  
   stage('Test') {  
   steps {  
   sh 'node --version'  
   }  
   }  
   }  
  }

1. 如何在任务中使用自建Docker容器？

#### 2、过滤指定字段

使用C++编写相关工具类对某段特性数据进行识别，然后过滤，是一个类似于**tcpdump**的工具，可以过滤指定的字段

#### 3、使用wireshark以及内部抓包工具

对涉及到的相关通信协议字段进行抓取，解读，分析