复习笔记

操作系统

1.多线程操作相关

- 创建线程

```
//创建线程
int pthread_creat(pthread_t* thread, const thread_attr_t* attr, void *
(*start_rout)(void *), void* arg);
```

创建线程，thread指向内存会被设置为线程ID

attr为线程属性，通过pthread_attr_init/destory(pthread_attr_t* attr)来创建/销毁线程属性(malloc动态分配的话，destory会释放)。可以通过pthread_attr_getdetachstate来设置线程的分离状态，也可以通过pthread_detached直接设置为分离状态。

start_rtn为线程的起始地址，参数为void*类型的arg

pthread类函数失败返回错误码，并不设置errno

- 等待线程结束

```
//等待线程结束
int pthread_join(pthread_t thread, void **retval);
```

调用线程将一直阻塞

thread指定的线程退出/被取消

rval_ptr包含返回码，被取消时被设为 PTHREAD_CANCELED，不感兴趣可以设为NULL

可以调用pthread_detached让操作系统进行资源回收

- 获取线程号

```
pthread_t pthread_self(void);
```

返回线程ID，进程中唯一，但不同进程可能会有相同的，但是getpid返回的是内核中唯一的。

- 并发和并行
  - 并发：宏观上两个程序同时运行，单核CPU上也可以并发的，交织的运行多个程序，提高效率
  - 并行：多核CPU上，两个程序分别运行在不同的核上，提高计算机的效率
- 线程和进程优劣势对比?
  - 进程是资源分配的单位，线程是执行的基本单位
  - 进程切换代价大，线程切换代价小
  - 多个线程共享进程的资源

所以，进程是资源分配的最小单位，而线程是CPU调度的最小单位。多线程之间共享同一个进程的地址空间，所以通信简单，但是同步复杂，线程创建、销毁和切换简单，速度快，占用内存少，但线程间会互相影响，一个意外终止，会导致整个进程终止，可靠性更弱。
多进程间各自独立，不会互相影响，程序可靠性强，但是创建、销毁和切换复杂，速度更慢，占用内存更多，通信更加复杂，但是同步简单。

- 进程切换通常由两部分
  - 切换页目录和使用新的地址空间
  - 切换内核栈和硬件上下文

  第一步线程是不需要做的

- 死锁
  产生死锁的四个必要条件
  - 互斥：一个资源每次只能一个进程使用
  - 占有且等待：一个进程因请求资源而阻塞，对已获得的资源不进行释放
  - 不可强行占有：进程不可强行剥夺已被获取资源
  - 循环等待条件：进程之间形成头尾相接的循环等待资源的关系

  处理死锁的基本方法
  - **死锁预防**：通过设置某些限制条件，去破坏死锁的四个条件中的一个或几个条件，来预防发生死锁。但由于所施加的限制条件往往太严格，因而导致系统资源利用率和系统吞吐量降低
  - **死锁避免**：允许前三个必要条件，但通过明智的选择，确保永远不会到达死锁点，因此死锁避免比死锁预防允许更多的并发
  - **死锁检测**：不须实现采取任何限制性措施，而是允许系统在运行过程发生死锁，但可通过系统设置的检测机构及时检测出死锁的发生，并精确地确定于死锁相关的进程和资源，然后采取适当的措施，从系统中将已发生的死锁清除掉
  - **死锁解除**：与死锁检测相配套的一种措施。当检测到系统中已发生死锁，需将进程从死锁状态中解脱出来。常用方法：撤销或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程。死锁检测盒解除有可能使系统获得较好的资源利用率和吞吐量，但在**实现上难度也最大**

  银行家算法

2.多线程同步相关

- mutex

```
//创建mutex
int pthread_mutex_init(pthread_mutex_t*  mutex, const
pthread_mutexattr_t*  attr);
//销毁mutex
int pthread_mutex_destroy(pthread_mutex_t* mutex);
//上锁
int pthread_mutex_lock(pthread_mutex_t* mutex);
//尝试上锁，失败，返回EBUSY
int pthread_mutex_trylock(pthread_mutex_t* mutex);
```

```
//解锁
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

互斥体用来在进入临界区前加锁，其他加锁的都会被阻塞知道锁被释放。用这种方式保护临界区。

一般是通过栈上的MutexLockGuard的RAII类进行自动的加锁和解锁，更不容易发生错误。

- 信号量

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量。而可以取多个正整数的信号量被称为通用信号量。

linux sem信号量具有原子性，用于解决共享资源的同步问题。

```
//创建信号量
int sem_init(sem_t *sem, int pshared, unsigned int value);
//等待，尝试等待
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
//信号量+1
int sem_post(sem_t *sem);
//获取当前信号量的值，保存在sval，如果多个线程阻塞在wait上，返回等待的个数
int sem_getvalue(sem_t *sem, int *sval);
```

sem指向信号对象，并给他初值value sem_wait可以用来阻塞当前线程，直到信号量的值大于0，解除阻塞，然后把信号量-1，继续运行接下来的程序。 pshared为0表示用于多线程的同步，>0表示可以共享，用于多进程同步

- 条件变量

条件变量也可以用于多线程的同步，需要由互斥量保护

```
//初始化信号量
int pthread_cond_init(pthread_cond_t *restrict cond, const
pthread_condattr_t *restrict attr);
//反初始化
int pthread_cond_destroy(pthread_cond_t *cond);
//等待条件为真
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t
*restrict mutex);
//等待到abstime，就不在等待
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex, const struct timespec *restrict
abstime);
//唤醒一个，或者全部
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

需要注意wait的时候需要传递一个**锁住的互斥锁**，函数会自动的把线程放到等待条件的线程列表，并解锁，返回时，会再次加锁。

在判断条件时，需要用while，而不能用if，防止虚假的唤醒。

- 读写锁 读写锁是为了解决读的时候能并发，但是例如mutex却是一棒子打死，让读也不能并发的问题。

```
//创建读写锁
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const
pthread_rwlockattr_t *restrict attr);
//销毁读写锁
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
//加锁解锁
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

- 自旋锁 自旋锁采用忙等的方式进行加锁，因为不会进行调度的特点，当锁被短时间持有，可以考虑

```
//创建
int pthread_spin_init (pthread_spinlock_t *_lock, int pshared);
//销毁
int pthread_spin_destroy (__pthread_spinlock_t *__lock);
//解锁，加锁
int pthread_spin_trylock (__pthread_spinlock_t *__lock);
int pthread_spin_unlock (__pthread_spinlock_t *__lock);
int pthread_spin_lock (__pthread_spinlock_t *__lock);
```

pshared用来设置是否支持进程共享还是只能被初始化的进程内的线程访问。

3.守护进程

要了解守护进程，首先要理解会话以及进程组的概念

```
//获取group id，组长的进程ID为组ID
pid_t getpgrp(void);
//加入现有进程组，或者创建新的进程组
int setpgid(pid_t pid,pid_t pgid);
```

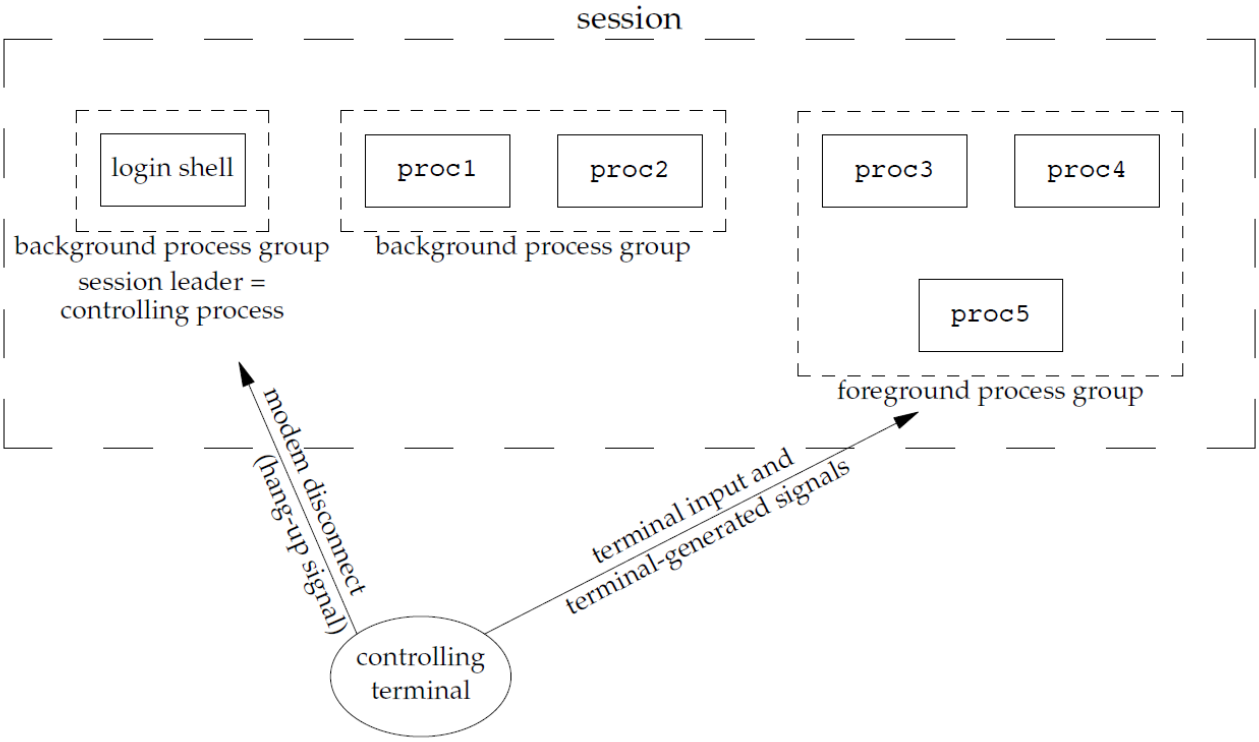
每个进程除了有一进程ID之外，还属于一个进程组，进程组是一个或多个进程的集合，每个进程组有一个唯一的进程组ID。

```
//新建一个会话
pid_t setsid(void);
```

```
//获取会话首进程的进程组ID
pid_t getsid(pid_t pid);
```

会话是一个或多个进程组的集合，如果pid为0，返回调用进程首进程的组ID。

当进程不是经常组长时，setsid才会新建一个会话，并且该会话没有控制终端(即使之前有，也会切断)。



关系如图， 一个会话可以有一个控制终端，建立会话的被称为首进程，可以有一个或多个后台进程组合一个前台进程组，中断键被发给前台进程组所有进程

守护进程不受终端影响，默默的在后台服务于系统/某个用户，并且不与终端进行交互。 创建一个守护进程的流程：

- umask(0)，这是为了和继承来的屏蔽字区分开。
- 调用fork，然后令父进程exit(使终端认为指令执行完毕，同时满足设置session不是组长的要求)
- 调用setsid创建新的session(这样没有控制终端)
- 将当前目录更改为根目录(也是为了分离以前关系)
- 关闭不需要的描述符
- **可选**，有些会打开/dev/null使其拥有0 1 2描述符，防止交互

创建守护进程的核心就在于要一个不是进程组组长的终端调用setsid，并且，要将守护进程和以前的进程彻底分离，其他特性和基本的进程没有什么区别。

4.进程、线程状态查看

- ps指令查看进程(快照) 首先介绍一下Linux上的进程状态：
    - R(Running)：可执行状态，包括就绪态和运行态，也就是可执行队列。
    - S(Interruptible/Sleep)：可以被中断的睡眠态，在等待某件事发生
    - D(Uniterruptible)：不可中断的睡眠态，不响应各种异步信号等。这个状态可以用来避免内核的某些流程被打断，比如再做IO操作时

- T(Traced/Stopped)：表示暂停或者跟踪状态，可以通过发送SIGSTOP(不可捕获)使进程暂停，发送SIGCONT可以恢复进程
  跟踪是指进程被跟踪他的进程暂停，如gdb调试时，停在断点处，可以通过ptrace系统调用进行操作
- X(Exit)：退出状态，即将被销毁，如果是分离状态，则不会持有task_struc
- Z(Zombie)：退出状态，进程称为僵尸进程，等待回收资源

可以利用ps命令查看系统当前运行的进程(快照)：

- a,-A,-e显示所有进程
- -w显示加宽，以显示较多信息
- -au显示终端下用户的所有程序
- -aux列出所有的正在内存中的程序
- -u 显示某用户的进程信息

可以利用ps和grep配合，查找特定的进程

- top(性能分析) top类似win下的资源管理器，可以看到各种资源进程的占用情况

  这里参考一个网上的例子，便于理解各个参数：

```
$top
    top - 09:14:56 up 264 days, 20:56,  1 user,  load average: 0.02, 0.04, 0.00
    Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
    Cpu(s):  0.0%us,  0.2%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.2%st
    Mem:    377672k total,   322332k used,    55340k free,    32592k buffers
    Swap:   397308k total,    67192k used,   330116k free,    71900k cached
    PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0  2856  656  388 S  0.0  0.2   0:49.40 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   7:15.20 ksoftirqd/0
    4 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
```

  - 第一行是时间，日期，在线用户，以及1分钟，5分钟，15分钟的CPU负载
  - 第二行是进程各个状态的统计信息
  - 第三行是统计CPU的总体信息
  - us(user)：用户态占比
  - sy(system)：内核态占比
  - ni(nice)：改变过优先级的占比
  - id(idle)：空闲时间占比
  - wa(wait)：等待时间占比
  - hi(hardware)：硬件中断占比
  - si(software)：软件占比

  按1可以看每个CPU的占比

  - 第四行内存信息，total物理内存总量，used是使用的物理内存，free是空闲的，buffers用作内核缓存的物理量
  - 第五行是交换空间(虚拟内存)的总量，空闲，以及缓冲交换区总量
  - 最后是进程信息，一次是PID，用户，优先级，nice值，虚拟内存，物理内存，共享内存，CPU率和屋里内存率，最后是占用CPU的总时间和启动命令。

  这里有个很有意思的事情，Linux有Nice值和优先级两个来进行调度，那么他么是怎么进行配合的呢?

实时优先级是0 ~ MAX_RT_PRO-1，nice值是MAX_RT_PRO ~ MAX_RT_PRO + nice_max，也就是nice值的程序比实时优先级的更低(数越小，优先级越高)。

- strace(跟踪系统调用)

  可以用strace来跟踪进程执行的系统调用和所接受的信号。

```
$strace cat /dev/null
execve("/bin/cat", ["cat", "/dev/null"], [/* 22 vars */]) = 0
brk(0)                                  = 0xab1000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f29379a7000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
...
```

  每一行会输出一个系统调用，左边是系统调用函数，右边是返回值

  strace用法很多，这里只举几个：

  - -f：跟踪fork后的子进程
  - -F：跟踪vfork后的子进程
  - -o：输出到文件
  - -T：显示每次耗费时间
  - -tt：每次输出加上时间信息
  - -e trace = *：可以调整只输出某部分相关的系统调用
  - -p pid：跟踪PID

- pstack(跟踪进程栈) 可以跟踪每个进程的栈变化，这样可以帮助我们知道程序的调用顺序，方便debug。

- 协程
  协程是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

  协程的一大特点是不由内核调度，而是由程序进程调度的，因此调度的代价比较小

  关于 goroutine，可以看Go并发原理

  核心就是 M:N 式调度，以及 MPG 模型。

  关于为什么要用 P，见goroutine调度过程中P到底扮演什么角色?

  - 避免频繁的访问全局队列，争用锁
  - 可以让 mcache 等资源，绑定给 P，减少 M 造成的资源浪费。

  为了保证均衡的调度，上下文P会定期的检查全局的goroutine 队列中的goroutine，以便自己在消费掉自身Goroutine队列的时候有事可做。假如全局goroutine队列中的goroutine也没了呢？就从其他运行的中的P的runqueue里偷(直接偷一半)。

5.内存和IO

- free(查看可用内存)

```
/opt/app/tdev1$free
             total       used       free     shared    buffers     cached
Mem:       8175320    6159248    2016072          0     310208    5243680
-/+ buffers/cache:     605360    7569960
Swap:      6881272      16196    6865076
```

  - 第一行表示内存总量，使用和空闲。然后是共享内存，缓冲大小(还没被写入磁盘的)，缓存大小(从磁盘读取备用的内容)
  - 第二行是从应用程序来看的，第一行是从OS角度来看的，从OS来说，buffers和cached都是被使用的，而对应用程序来说那两者都是可以使用的
  - 最后一行是交换分区的信息，不再赘述
- iostat(监视IO子系统) 可以方便的查看CPU/网卡等设备的活动情况，负载信息。

```
/root$iostat
Linux 2.6.32-279.el6.x86_64 (colin)   07/16/2014      _x86_64_        (4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
10.81    0.00   14.11    0.18    0.00   74.90

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               1.95         1.48        70.88    9145160  437100644
dm-0              3.08         0.55        24.34    3392770  150087080
dm-1              5.83         0.93        46.49    5714522  286724168
dm-2              0.01         0.00         0.05      23930     289288
```

  输出如上，重点关注两个：
  - iowait：用于等待输入、输出完成时间的百分比
  - idle：空闲时间百分比

    iowait可以帮助我们判断瓶颈是否在IO(等待占比是否过长)，idle可以帮助我们判断CPU是否存在瓶颈(空闲时间长时间很小)
- ls ls可以查看当前目录下的内容，-l输出详尽信息。
- du 用来查看目录或文件所占用的磁盘大小，常用du -sh
  - -h以人类可读方式输出
  - -a显示目录和子目录文件占用
  - -s不显示子目录和文件
  - -c统计几个选项
- df 用来查看文件系统整体的使用情况。

6.文件基本

- 文件属性

```
int stat(const char * path, struct stat * buf);
int fstat(int fildes, struct stat *buf);
int lstat(const char * path, struct stat *restrict buf);
int fstatat(int fd, const char *path, struct stat *buf, int flag);
```

  带l当文件是索引文件时，返回的是符号链接(指向另一个链接的项)的有关信息，f开头的，返回的是fd对应的信息。

  stat 结构如下：

```
struct stat {

    mode_t     st_mode;       //文件对应的模式，文件，目录等
    ino_t      st_ino;       //inode节点号
    dev_t      st_dev;        //设备号码
    dev_t      st_rdev;       //特殊设备号码
    nlink_t    st_nlink;      //文件的连接数
    uid_t      st_uid;        //文件所有者
    gid_t      st_gid;        //文件所有者对应的组
    off_t      st_size;       //普通文件，对应的文件字节数
    time_t     st_atime;      //文件最后被访问的时间
    time_t     st_mtime;      //文件内容最后被修改的时间
    time_t     st_ctime;      //文件状态改变时间
    blksize_t st_blksize;    //文件内容对应的块大小
    blkcnt_t   st_blocks;     //伟建内容对应的块数量
  };
```

- 文件权限
  文件权限有9位，在stat中的st_mode字段，分别是用户、组、其他读写。

```
//更改权限
int chmod(const char* pathname, mode_t mode);
//fd版本
int fchmod(int fd, mode_t mode);
```

  通过宏我们可以对每一位进行设置，权限分别为用户、用户组、其他组。

- 三个时间信息 stat中有三个时间信息，分别为st_atime,st_mtime,st_ctime，都是timespec结构的。

  分别为：

  - st_atime：读取文件或执行文件时更改的，任何对文件的访问都会更改此值
  - st_mtime：写入文件时随文件内容更改
  - st_ctime：写入、更改所有者或链接设置时，随iNode更改而更改。只要stat出来的内容改变，ctime改变，也就是修改属性的时候改变

  具体来说，有三种情况：

  - 读文件，st_atime改变
  - 修改文件，三个都改变
  - 修改文件属性，st_ctime改变，其余不变

- inode 因为存储文件的时候还需要顺带记录文件权限等信息，这也就是inode，stat调用读取的就是inode的信息。Linux允许多个文件名指向同一个inode，只有inode计数为0的时候，才会删除文件。 调用link可以创建一个指向现有文件的连接。

- 用户ID
  Linux有用户ID和组ID。在对文件进行操作的时候：* 首先检查用户ID是否为0(root)

- 检查用户ID是否设置了对应位
- 如果进程的进程组ID或附属组ID(Linux允许属于多个组)，则允许访问
- 其他位被设置，则允许访问

```
//更改用户ID和组ID
chown(const char* pathname, uid_t owner, gid_t group);
```

通过chown我们可以对文件拥有者，如果设为-1，则不变。

- lsof(list open files) 列出所有打开的文件，一般以root用户运行。

7.软连接

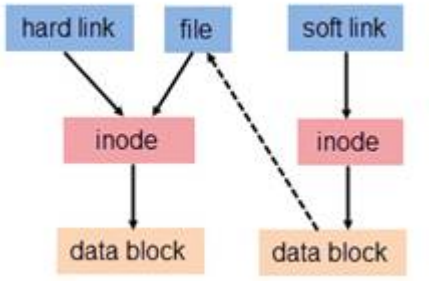
硬链接：建立inode和文件名之间的连接关系，`ln source_file_name new_file_name`

通过连接名->inode->block->sector

但是目录不能创建硬链接，并且不能跨分区创建

使用硬链接可以实现文件备份的功能，并且节省磁盘空间

软连接：又称符号链接，也就是快捷方式,`ln -s target_name soft_name`



软连接和硬链接都有自己的inode，硬链接是一个inode对应多个文件名，软连接有自己独立的inode，但是数据内容比较特殊

8.进程间通信

- 管道(pipe)

1. 半双工
2. 只能用于有亲缘关系进程间的通信，通常是fork一个子进程，然后利用管道在二者之间通信。

```
int pipe(int fd[2]);
```

当管道建立成功，会创建两个文件描述符，fd[0]为读打开，fd[1]为写打开。

一般是fork前创建pipe，然后写端关闭读，读端关闭写。其实可以互相不关闭，但是即使不关闭，两个进程也不敢互相收发，因为不知道什么时候读取的是自己发出去的数据了。具体见won't close pipe

- 命名管道(FIFO)

    1. 与pipe不同，不相关的进程也能通过FIFO交换数据
    2. FIFO有路径名相关联，确实存在于文件系统中

```
int mkfifo(const char* path, mode_t)
//因为类似文件，我们用open来打开
int open(const char *pathname, int flags);
```

    FIFO类似使用文件在进程中通信，但是和pipe类似，数据读出时，FIFO会清楚数据。

    并发写时，长度不能超过PIPE_BUF字

    FIFO主要用来解决pipe不能用于没有亲缘关系的进程通信的问题 **FIFO的打开规则：**

    如果当前打开操作是为读而打开FIFO时，若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。

    如果当前打开操作是为写而打开FIFO时，如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作设置了阻塞标志）；或者，返回ENXIO错误（当前打开操作没有设置阻塞标志）。

- XSI IPC相似之处(消息队列，信号量，)

    - 多个IPC汇聚方式：

        - 指定键IPC_PRIVATE，创建一个新的IPC结构，将返回的标识符放在某处，供客户进程使用
        - 在公共头文件中指定一个都认可的键
        - 认同一个路径名和项目ID(0-255)，调用`key_t ftok(const char* path, int id);`创建一个key_t类型的键

    - 缺点

        - 在系统范围内有效，没有引用计数，有可能会一直存在于系统之中
        - 在文件系统中没有名字不能用各种文件的操作
        - 没有fd，多路复用难以应用在此

- 消息队列
  消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

    1. 面向记录，消息具有特定的格式和优先级
    2. 独立利于发送和接收进程
    3. 可以随机查询，不一定先入先出，可以按类型取消息

```
#include <sys/msg.h>
// 创建或打开消息队列：成功返回队列ID，失败返回-1
int msgget(key_t key, int flag);
// 添加消息：成功返回0，失败返回-1
int msgsnd(int msqid, const void *ptr, size_t size, int flag);
```

```
// 读取消息：成功返回消息数据的长度，失败返回-1
int msgrcv(int msqid, void *ptr, size_t size, long type,int flag);
// 控制消息队列：成功返回0，失败返回-1
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

消息发送指针ptr指向要求是以long int开头的消息结构，用来确定成员的消息类型，msgrcv中的ptr类似

linux用MSGMAX和MSGMNB来限制消息的长度和消息队列的长度

- 信号量(semaphore)
  最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量。而可以取多个正整数的信号量被称为通用信号量。 Linux 下的信号量函数都是在通用的信号量数组上进行操作，而不是在一个单一的二值信号量上进行操作。

```
#include <sys/sem.h>
// 创建或获取一个信号量组：若成功返回信号量集ID，失败返回-1
int semget(key_t key, int num_sems, int sem_flags);
// 对信号量组进行操作，改变信号量的值：成功返回0，失败返回-1
int semop(int semid, struct sembuf semoparray[], size_t numops);
// 控制信号量的相关信息
int semctl(int semid, int sem_num, int cmd, ...);
```

当semget创建新的信号量集合时，必须指定集合中信号量的个数（即num_sems），通常为1； 如果是引用一个现有的集合，则将num_sems指定为 0 。 在semop函数中，sembuf结构的定义如下：

```
struct sembuf
{
    short sem_num; // 信号量组中对应的序号，0～sem_nums-1
    short sem_op;  // 信号量值在一次操作中的改变量
    short sem_flg; // IPC_NOWAIT不阻塞，返回EAGAIN，SEM_UNDO
}
```

- 共享内存
  共享内存允许两个或多个进程共享一个给定的存储区，因为数据不要在各个进程复制，所以是最快的一种IPC。

  1. 速度快
  2. 允许多个进行操作，因此需要同步措施

```
#include <sys/shm.h>
// 创建或获取一个共享内存：成功返回共享内存ID，失败返回-1
int shmget(key_t key, size_t size, int flag);
// 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回-1,一般
addr为0，由操作系统进行选取
void *shmat(int shm_id, const void *addr, int flag);
```

```
// 断开与共享内存的连接：成功返回0，失败返回-1
int shmdt(void *addr);
// 控制共享内存的相关信息：成功返回0，失败返回-1
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

当调用共享内存结束后，调用shmdt断开连接，但是这并不在系统中删除共享内存，只有当以IPC_RMID命令的调用shmdt才会删除

- 信号
  信号用于通知进程发生了某种情况，进程可以忽略，按默认方式处理或者捕获该信号，或设置自己的处理函数。 常见的信号：
  - SIGINT(程序终止，在用户输入INTR字符时发出的)
  - SIGKILL(立即结束程序运行，不能被捕获)
  - SIGSTOP(停止进程的运行，不能被捕获)
  - SIGTERM(程序结束信号，可以被捕获，kill默认产生这个)
  - SIGSEGV(试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据)
  - SIGCHLD(子进程结束时, 父进程会收到这个信号)
  - SIGALRM(时钟定时信号, 计算的是实际的时间或时钟时间. alarm函数使用该信号)
  - SIGPIPE(对于已经关闭的TCP链接，发送一次，客户会收到RST响应，再发一次，系统会返回SIGPIPE，默认的操作是停止运行，可以设置SIG_IGN忽视)
    对于服务器，如果比较繁忙，没能按时发出数据，有可能会发送超过一个包到已经关闭的链接，因此要对这个信号信息处理

```
//设置信号处理函数，handler可以为SIG_IGN忽略，SIG_DFL系统默认，以及函数指针
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);]
//发送信号到进程组或进程
int kill(pid_t pid, int signo);
//发送信号给自己
int raise(int signo);
```

当kill的pid>0时，发送给pid，==0时，发送给同一进程的所有进程组，<0时发送给发送给abs(pid)进程组，==-1，发送给有权限发送信号的所有进程

还可以通过sigaction进行安装

```
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

其中：

- signum：要操作的信号
- act：对信号新的处理方式
- oldact：原来的方式
  sigaction的结构如下：

```
struct sigaction {
    void (*sa_handler)(int); //信号处理函数
    void (*sa_sigaction)(int, siginfo_t *, void *); //替代的信号处理函
数，flags为SA_SIGINFO启用这个
    sigset_t sa_mask; //信号处理时需要屏蔽的信号
    int sa_flags; //一些标志
}
```

如果是可靠信号，那么屏蔽之后，如果触发N次，恢复后会执行N次信号处理，如果是不可靠信号，那么最多只会注册一次。

- 套接字 socket也可以用于进程间通信，具体见网络部分

9.虚拟内存相关

- 虚拟内存用途
    - 把主存看做是磁盘的缓存，在主存中只保留活动区域，并且根据需要在主存和磁盘交换数据，高效的使用主存
    - 为每个进程提供了一致的地址空间
    - 保护了每个进程的地址空间不被其他破坏
- 页表
  为了实现虚拟内存，我们需要一种方式，能够知道虚拟页是否在物理页中，不命中的话，还需要实现置换算法。

  这些功能通过操作系统、MMU(内存管理单元)以及内存中的页表实现的。页表将虚拟页映射为内存页，每次MMU需要翻译时会读取页表，操作系统负责维护页表

  页表就是由页表条目(PTE，除了物理页号外，还会有一些权限标志)构成的数组

  页表也有着自己的一个缓存，在 MMU 中，叫做TLB。

- 多级页表 假设是32位，4字节的PTE，页面大小为常见的4K，那么一个进程就要有4MB的页表常驻在内存中(即使大部分内容都是无效的)

  为了解决这个问题，将PTE进行分级，这样下级页表中无效的表项在上一级就可以指出，有效的减少了页表占用的内存

- 缺页中断
  malloc和mmap等内存分配函数实际上只是建立了虚拟页表，并没有分配对应的物理内存，而是当CPU需要访问时，产生缺页中断再进行页面调度

  缺页中断会通过调度算法来决定将哪些页面换出，作为替换。(LRU, LFU)

  缺页中断是软中断的一种，和普通的中断一样，需要
    - 保护现场
    - 分析中断原因
    - 转入缺页中断处理程序

    - 恢复CPU
      但是，和平常不一样的是，缺页中断返回的是产生中断的指令，而不是下一条指令。

- fork和vfork
  fork会创建一个和当前进程几乎一样的进程，只是子进程返回的是0。在过去，fork会将父进程的资源都复制给子进程，因此效率很低。因为fork之后通常就exec导入新的程序了，复制毫无意义。

  后来就引入了写时复制，子进程拷贝父进程的页表原样副本，并且标记位写时复制，这样代价就比复制所有资源小得多。

  vfork实际上是没有写时复制的一个中间产物，为了避免无谓的复制，vfork会休眠父进程直到子进程终止或者运行一个新的可执行文件，在过程中，父子进程共享地址空间和页表。通过这种方式避免了拷贝。

- 结构体对齐

    - 对齐的原因
      主要是因为32位的Intel处理器通过总线访问，一次读取32个bit，如果不对齐的话，可能会造成两次总线读写，浪费时间。

    - 对齐原则

        - 结构体变量的起始地址能够被其最宽的成员大小整除
        - 结构体每个成员相对于起始地址的偏移能够被其自身大小整除，如果不能则在前一个成员后面补充字节
        - 结构体总体大小能够被最宽的成员的大小整除，如不能则在后面补充字节
        - 取上述规则和指定大小的最小值，一般为4

- 页面置换算法

    - FIFO(先进先出)：最近刚访问的，将来访问的概率大，用队列即可实现
    - LFU(最不经常访问算法)：数据过去被访问过多次，将来被访问的频率也很高(感觉不太符合局部性的原理？很久之前多次访问的，岂不是很难被置换出去了)，用引用计数可以实现
    - LRU(最近最少使用)：最近被访问过，将来被访问的几率高。用栈可以实现，每次命中移动到栈底，每次替换栈顶。对热点数据命中率高，但是大量偶发访问时，内存中存放大量冷数据
    - LRU-K(最久未使用K次淘汰)：避免偶发访问造成的缓存污染问题，当访问次数达到K次时，才访问缓存

- 动态内存分配器

    - 隐式空闲链表：在链表块头部添加标志，表明是否是空闲，这种方式，每次查找合适的空闲块需要O(n)的时间
    - 显式空闲链表：将空闲块单独开辟一个链表，查找时间为O(empty)
    - 分离空闲链表：更近一步，将空闲链表进行分类，加速寻找空闲链表的时间，具体有如下做法
        - 简单分离存储：将大小类内的大小固定，也就是同一大小类内含有相等大小的块，不分割也不合并，实现简单，但是内部碎片大。
        - 分离适配：找到第一个适配的，进行分割，分割后插入适当的空闲链表中
        - 伙伴系统：以2^m次方开始，二分的递归分割这个块(因为是二分，所以叫伙伴)，直到分配到满足我们要求的块。合并的时候，很容易递归的和自己的伙伴合并，直到遇到已经分配的伙伴。伙伴系统快速搜索和快速合并，但是显然有内部碎片问题。

- 内存映射

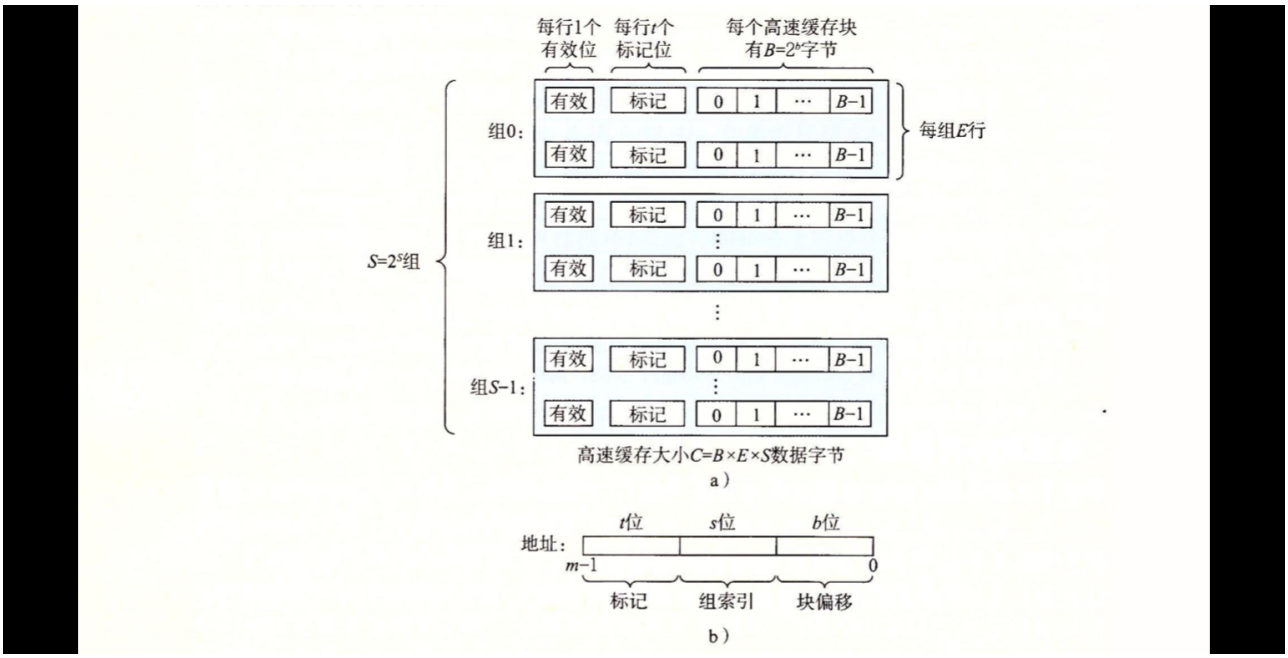
```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
off_t offset);
int munmap(void *addr, size_t length);
```

  - addr：需要映射的内存起始地址，通常为NULL，让系统自动选定，映射成功会返回该地址
  - length：文件多长进行映射
  - prot：保护方式，可执行、读、写、存取等
  - flags：MAP_SHARED会进行写回，MAP_PRIVATE写入操作会产生一个映射文件的复制，不会写回给原始内容，MAP_ANONYMOUS匿名映射，MAP_DENYWRITE只允许映射方式写入，MAP_LOCKED不会被置换出去
  - fd：映射到内存的文件的描述符
  - offset：偏移量

内存映射可以减少拷贝次数，因为通过read等需要将磁盘文件读到页缓存，再从页缓存拷贝到用户的buffer中，而映射可以直接通过指针操作mmap映射的内存，效率更高。

10.页缓存、页回写、内存映射

- 存储器缓存结构 存储器的中心思想是，把第 k 层的，当做第 k+1 层的缓存。



高速缓存结构如上图，将地址分为了 t,s,b 三个部分，s 确定所在的缓存组，t 标记着缓存存储着哪个字节。 **缓存很有意思的没有用高位，而是用了中间位：** 如果采用高位，那么连续的内存块，就会映射到相同的高速缓存块。那么如果程序有着良好的局部性(一般而言都有)，连续访问数组的时候，缓存最多只会保存一小部分的数组内容，而如果用中间段，那么在开头几次不命中时，会缓存数组中的大部分内容。

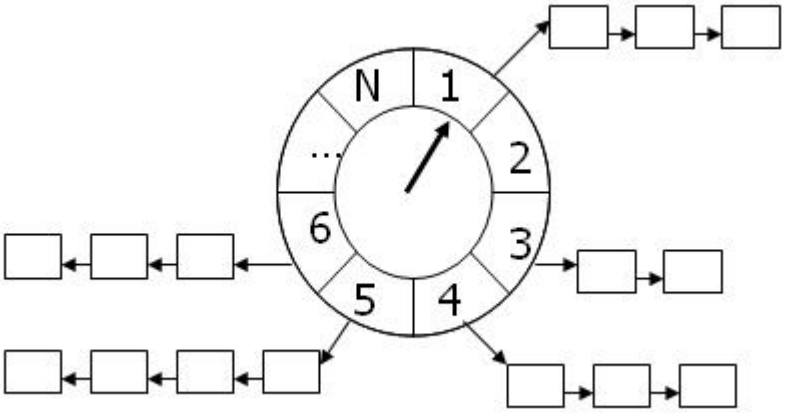
- 为什么要有页缓存
  页缓存的主要原因是因为磁盘和和内存访问速度有着巨大的差别，为了加快从磁盘读取文件的速率而设计的。具体来说，当要读取磁盘文件的时候，先在page cache中查找，命中的话就不需要再从磁盘

读取了。
对于写操作，有三种策略，不缓存写操作/更新缓存，写入磁盘(写透缓存)/通过标记脏页，定时回写

- 操作系统怎么设计的
  Linux采用通过两个双向链表实现LRU-2进行缓存回收策略，通过radix tree(压缩版的prefix tree，如果trie-tree的子节点是父节点的唯一孩子，那么就会进行合并)用页索引来管理已经缓存了的页面

11.定时器

- 时间轮



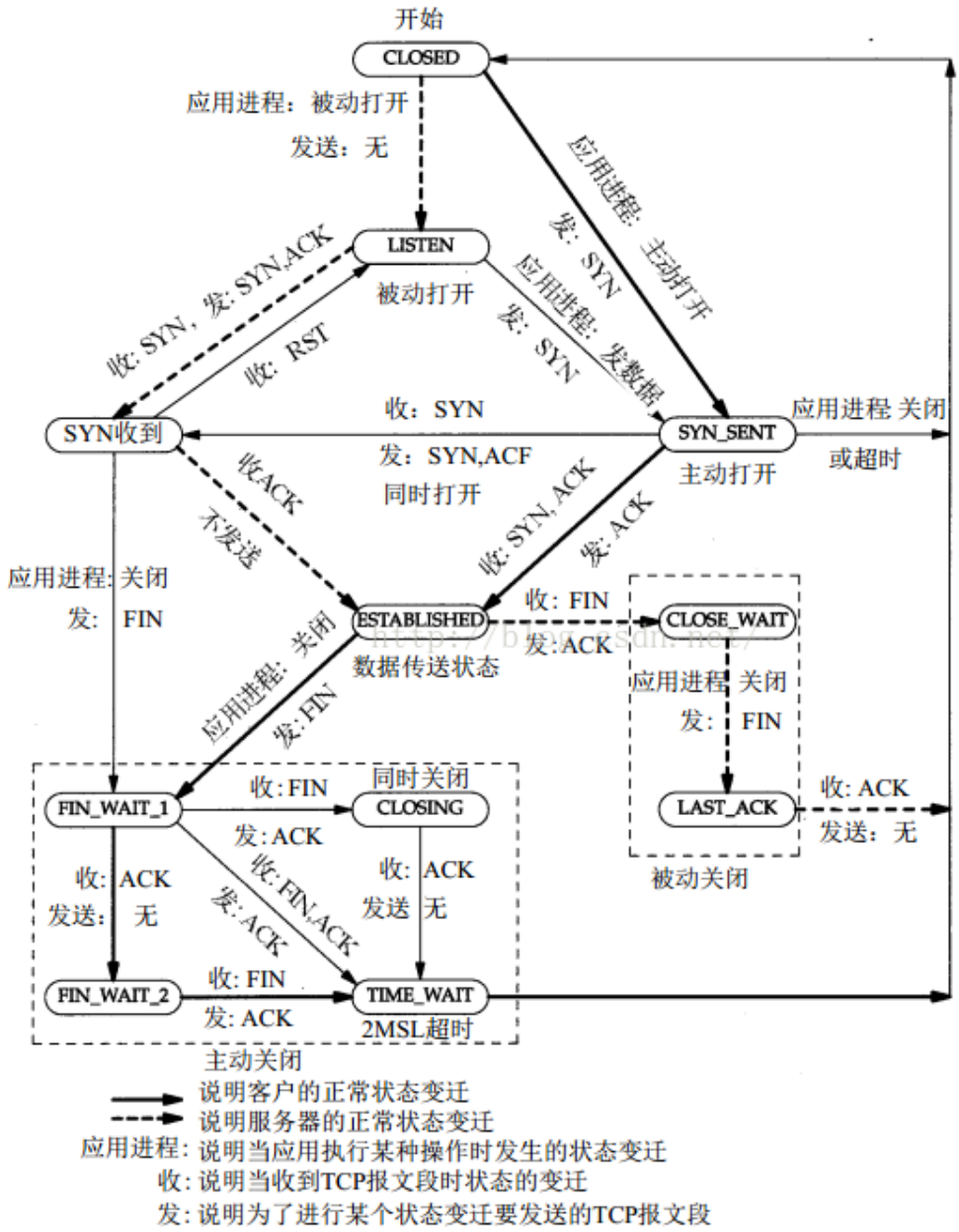
在早期只支持低精度的时钟，系统通过定时器中断更新时间轮(多级时间轮，图示只有一级)

- 红黑树
  在后来支持了高精度的时钟，内核该用了红黑树的方式进行定时器管理，采用了基于事件触发的方式，将下次触发时间设置为红黑树中最早到期的timer的时间。

- 红黑树和时间轮对比，为什么高精度弃用了时间轮
  因为在多级时间轮的操作中，虽然大部分时间为O(1)的复杂度，但是当进位发生时，有可能坏至O(N)的定时器迁移时间，这种不稳定极大的影响了定时器的精度。
  而红黑树方式采用时间触发方式，能够实现较为精确的定时
  时间轮介绍
  时间轮原理 选用红黑树的原因

网络

1.TCP状态机



TCP状态图如图所示。

三次握手：客户发送SYN，服务回复SYN-ACK，客户回复ACK。
通过三次握手，客户和服务互相验证了各自的收发没有问题。

四次挥手：客户发起FIN，进入FIN_WAIT_1，服务器收到后，回复ACK，进入CLOSE_WAIT，此时进入半关闭状态，客户流向服务端的那端已经被关闭。接着客户收到ACK后进入，FIN_WAIT_2状态，收到FIN后，回复ACK，进入TIME_WAIT状态，等待2MSL后关闭。
通过四次挥手，数据流的两个方向均被关闭。

- **TIME_WAIT状态**：
  在主动发起关闭的一方，收到两个FIN后，有一个TIME_WAIT状态，停留的时间为2MSL(最长分节生命期)。

  TIME_WAIT主要解决两个问题：

  - 可靠实现TCP全双工连接的终止
  - 允许老的分节在网络中消逝

对于第一个来说，服务器在发送FIN，试图把发端也关闭后，会进入LAST_ACK等待最后一个ACK，此时TIME_WAIT状态会在服务器重传FIN时，重新回复ACK，帮助服务器关闭。

此外TIME_WAIT能够保证旧的IP数据报已经在网络中消失，新建立的连接不会收到过去的报文。

- **LAST_ACK**： 这个状态，服务器等待最后一个ACK后就可以关闭，如果没有收到，会进行重传，此时有几种情况：
    - C端处于TIME_WAIT，那么会回复ACK
    - 已经CLOSED了，那么会认为连接错误，发送RST，S收到后CLOSED
    - C端已经不再了，S会不断重传，直到重传超时，进入CLOSED
- TIME_WAIT状态的问题 TIME_WAIT 出现在主动断开的一方，如 NGINX 做反向代理时，就会有着大并发的情况，导致 TIME_WAIT 状态大量出现，占用端口。可以通过 SO_REUSEADDR 设置，来使得客户端复用 TIME_WAIT 状态下的端口。SO_REUSEADDR 允许启动一个监听服务器并捆绑其众所周知端口，即使以前建立的将此端口用做他们的本地端口的连接仍存在

2.tcpdump(抓包)

tcpdump可以截获网络数据包，并对其进行分析。

- -i：指定网卡
- host：指定特定的主机和本机之间的通信包
- src host：指定特定来源
- dst host：指定特定的目标地址
- port：指定端口
- tcp/udp：指定传输层
- -c：抓包数量
- -s 0：抓完整数据包
- -w：保存到本地(带缓冲)

3.网络状态和防火墙(netstat/ifconifg/iptables)

- netstat用来查询网络相关的信息
    - -a：查看所有端口(监听以及未监听)
    - -t：TCP端口
    - -l：所有监听服务状态
    - -u：UDP端口
    - -p：显示相关连接的程序名
    - -c：每隔固定时间执行
- ifconifg获取和修改网络接口配置
  ifconifg [网络设备] [参数]
  直接调用ifconifg可以看到所以网卡的MAC地址，MTU，广播地址等
    - up/down：启动、关闭设备
    - arp：设置网卡是否支持arp
    - -a：显示所有接口信息
    - add/del：给指定网卡配置/删除IPv6地址

- mtu：设置最大传输单元
- netmask：子网掩码
- address：设置IPv4地址

4.socket API(重点)

- 连接建立/销毁相关
  连接建立/销毁即三次握手过程和四次挥手，主要有几个函数：

  - socket：建立套接字描述符sockfd
  - bind：绑定地址到套接字
  - listen：设置为监听套接字(默认为主动)
  - accept：从就绪队列中获取，并且返回连接的sockfd
  - connect：用于客户端和服务器建立连接
  - close：将套接字描述符引用减一，为0时会关闭套接字

  接下来按顺序介绍每个函数：

  - socket

    ```
    //返回：若成功则为非负描述符sockfd，若出错则为-1
    int socket (int family, int type, int protocol);
    ```

    调用socket可以创建一个套接字
    family常用的为AF_INET(IPv4)和AF_INET6(IPv6)
    type常用的有SOCK_STREAM(配合TCP)和SOCK_DGRAM(配合UDP)
    protocol选择传输层协议，有IPPROTO_TCP和IPPROTO_UDP

    ```
    socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK | SOCK_CLOEXEC, 0);
    ```

    可以直接通过socket的type设置非阻塞和exec关闭，也可以创建之后通过fcntl设置

  - bind

    ```
    //绑定本地地址到套接字
    int bind(int socket, const struct sockaddr *address, socklen_t
    address_len);
    ```

    主要是sockaddr结构，这是通用套接字格式，一般我们操作sockaddr_in：

    ```
    struct sockaddr_in{
     uint8_t sin_len; //带符号8位整数地址结构长度
     sa_family_t sin_family; //协议族，IPv4为AF_INET
     in_port_t sin_port; //端口号
     struct in_addr sin_addr; //32位IPv4网络字节序地址
     char sin_zero[8];       //填充对齐位，未使用
    };
    ```

- listen

```
int listen(int sockfd, int backlog)
```

backlog指定了套接字排队的最大连接个数。

- accept

```
int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addr
len);
```

服务器从已完成连接队列中取出一个连接，并且返回一个全新的描述符，cliaddr是客户的地址信息。

- connect

```
//返回：若成功则为0，若出错则为-1
int connect(int sockfd, const struct sockaddr *servaddr, socklen_
t addrlen);
```

对servaddr发起连接，如果没有收到SYN响应，会进行重传，如果响应时收到RST，说明服务器没有在该端口监听，errno会设置为ECONNREFUSED，如果客户发出SYN不被接收，errno会被设为EHOSTUNREACH。

上述描述的是TCP调用connect的场景，实际上，UDP也是可以调用的，此时只是起到记录对端IP地址和端口号的作用。

不调用connect时，情况为建立连接，发送报文，断开连接，建立连接，再发送。。。调用之后变为建立连接，发送，发送。。。，断开，提高了效率，此外，UDP可以调用多次connect。

- close

```
int close(int sockfd);
```

将套接字引用-1，如果为0，关闭套接字，>0，则不会。

- shutdown

```
int shutdown(int sockfd, int howto);
```

可以不顾close引用数量的限制，并且可以关闭读端或者写端

- 输入/输出(五组)

- read、write(unistd.h)

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

将buf里的文件写入fd中，count为需要读取或者写入的字节数，返回值为成功操作的字节数。

因为Linux中所有设备都能看成文件，所以也可以用来操作sockfd

- recv、send(sys/socket.h)

```
ssize_t recv(int sockfd, void* buff, size_t nbytes, int flags);
ssize_t send(int sockfd, const void* buff, size_t nbytes, int
flags);
```

前三个意义和上节相同，当flags取0时，等价于上节的两个。但是可以通过设置flags设置为非阻塞等操作

如果连接终止，会返回0，否则会返回SOCKET_ERROR错误

- recvfrom、sendto(UDP)

```
ssize_t recvfrom(int sockfd, void* buff, size_t nbytes, int
flags, struct sockaddr* from, socklent_t* addr_len);
ssize_t sendto(int sockfd, void* buff, size_t nbytes, int flags,
struct sockaddr* to, socklen_t addrlen);
```

前三个参数同上，后两个填入的是对端的地址和结构长度

他们可以用于TCP，但是通常不这么做，因为TCP建立连接创建了新的套接字，带有了信息，没必要

- readv、writev

```
ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

利用iovec进行分块读写，iovec里用base和len来表示每块的范围

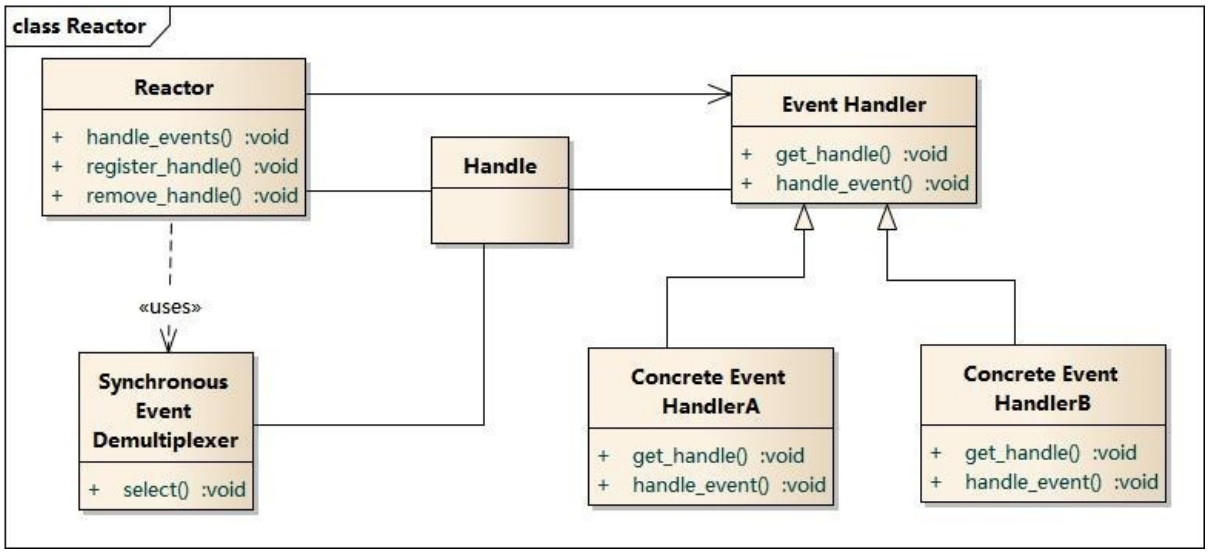
- 典型调用顺序 TCP服务端一般调用顺序:socket--->bind--->listen--->accept--->process user input;

  TCP客户端调用顺序：socket --->connect ---->process user input;

  UDP服务端一般调用顺序:socket--->bind--->recvfrom

  UDP客户端调用顺序：socket --->sendto;

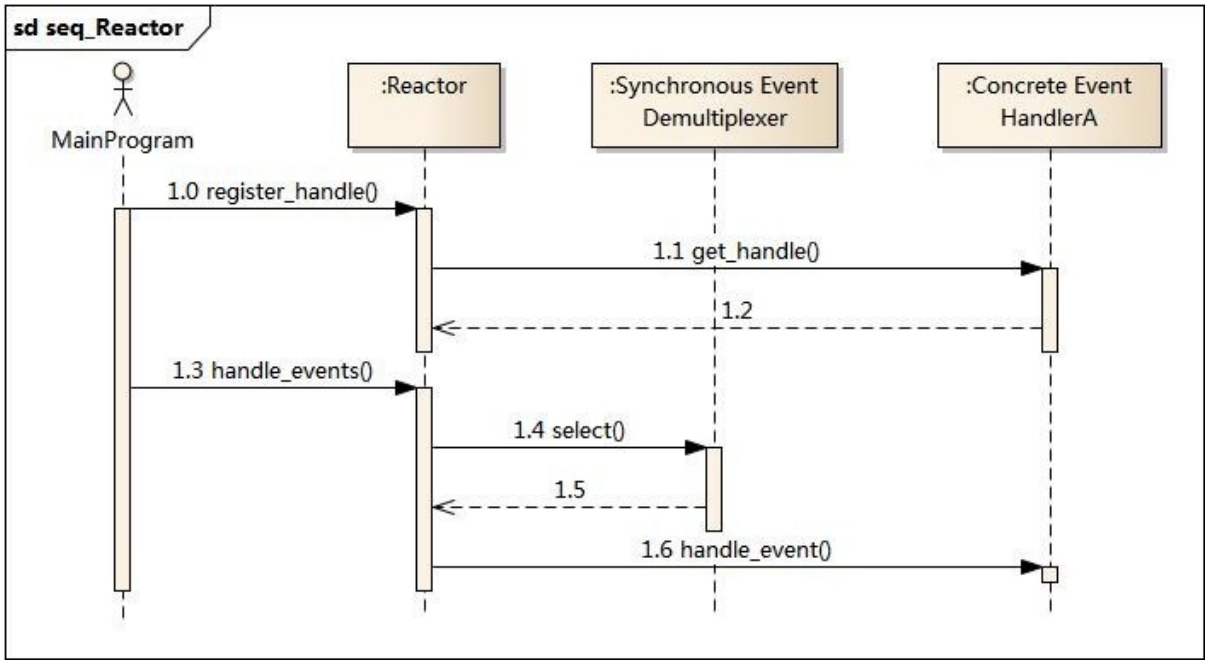
- 网络模型
  - Reactor模式



Reactor包含以下结构：

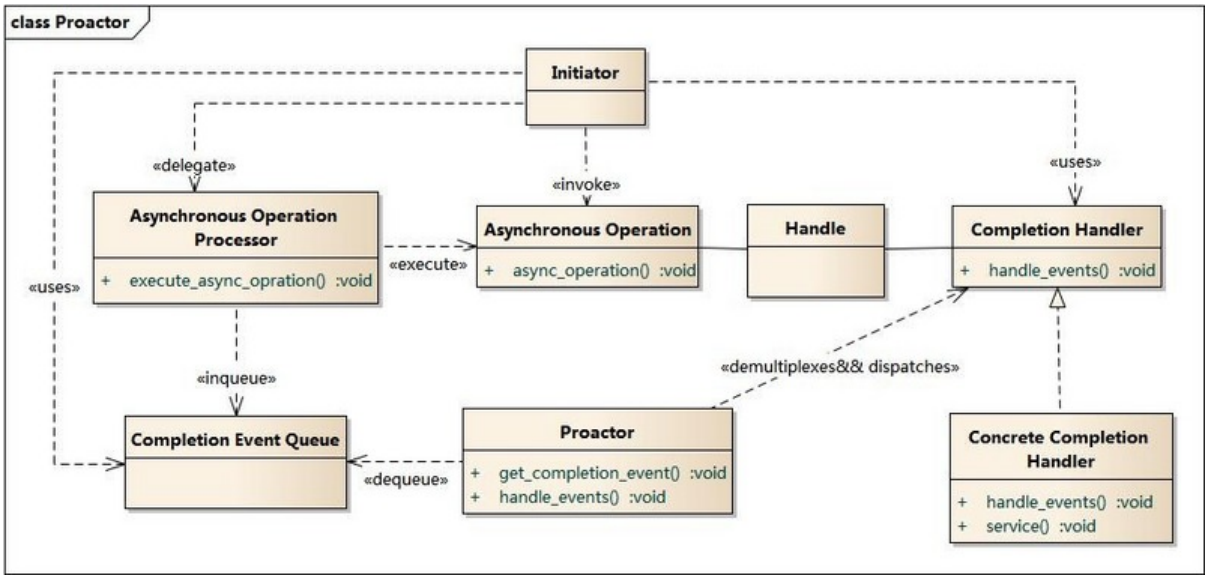
- Reactor：反应器，需要有以下功能
  - 注册和删除关注的事件
  - 事件循环
  - 事件到来时，能够执行对应的回调函数
- 多路复用器：操作系统提供的时间多路分解器(select/poll/epoll)
- Handle：句柄，用来标识socket连接或者打开的文件
- Event Handler：事件处理器

业务流程和时序如图：



- 应用启动，注册事件
- 进入事件循环
- 事件到来，执行回调

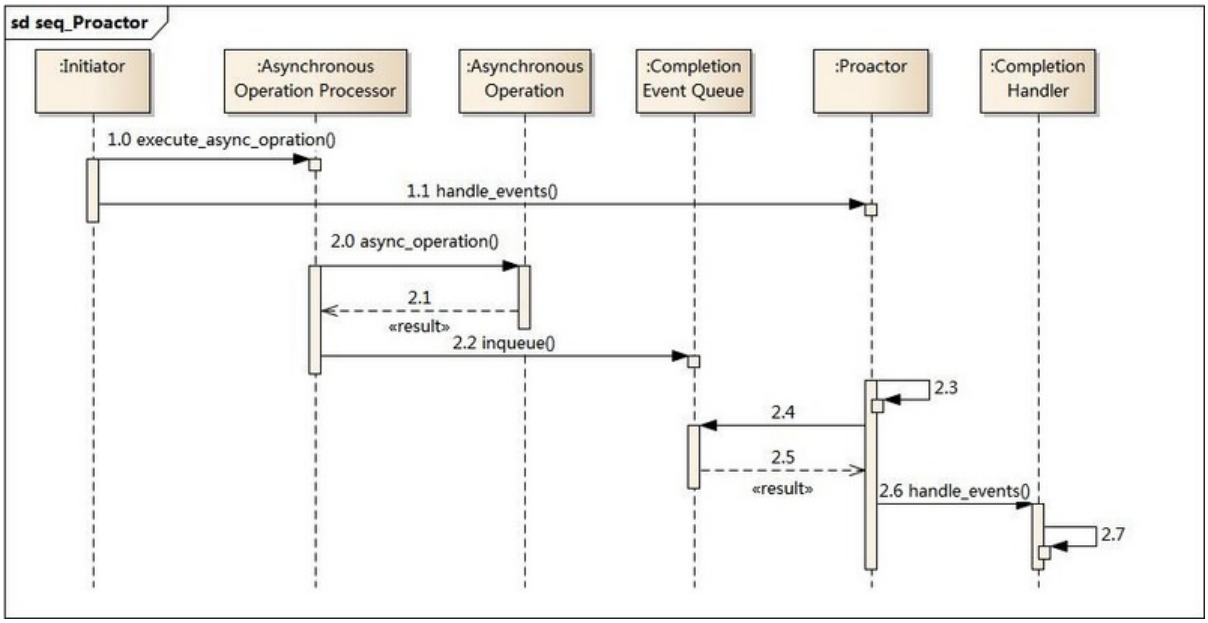
- Proactor模式



Proactor包含以下结构

- Proactor：主动器，为程序提供事件循环，从完成事件中取出异步结构，分发给后序处理的逻辑
- Asynchronous Operation Processor：异步操作处理器，负责执行异步操作，由操作系统实现
- Asynchronous Operation：异步操作
- Handle：句柄
- Completion Handler：完成事件接口，一般是回调函数
- Concrete Completion Handler：完成事件处理逻辑

业务流程和时序图：



- 调用异步处理接口，调用后其他操作可以并发进行
- 启动Proactor，进入事件循环，等待完成事件
- 异步操作将完成结构放入完成事件队列
- 主动器从完成事件队列取出事件，分发回调

5.IO复用

- select

```
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set
*errorfds, struct timeval *timeout)
```

select是最初的复用函数，参数分别为

  - 最大要测试描述符+1
  - 读、写、错误描述符集合
  - 等待时间(可以实现us定时器)，NULL为阻塞到有事件，0为不等待的轮询

有几个函数可以用来设置fd_set，FD_ZERO，FD_SET，FD_CLR，FD_ISSET，并且每次需要重新SET

返回时我们需要遍历每个集合来找到响应的事件

因为select返回会将没有响应的置0，我们感兴趣的事件可能被置位，所以每次需要重新设置

- poll

```
int poll(struct pollfd * fdarray, unsigned long nfds, int timeout);
```

poll的参数含义分别为

  - 等待的集合，pollfd结构为：

```
struct pollfd{
     int fd; //描述符
     short events; //兴趣事件
     short revents; //等待事件
};
```

  - 关心描述符的个数
  - 等待事件，毫秒

和select相比，poll把关注和回复分开了，因此不再需要每次重新设置

- epoll

```
int epoll_create(int size); //创建一个epoll的句柄，size用来告诉内核这个监听的
数目一共有多大
int epoll_wait(int epfd, struct epoll_event * events, int maxevents,
int timeout);
```

epoll通过一个文件描述符管理多个描述符，返回epoll的描述符

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

epoll_ctl用来更改epoll里的event集合

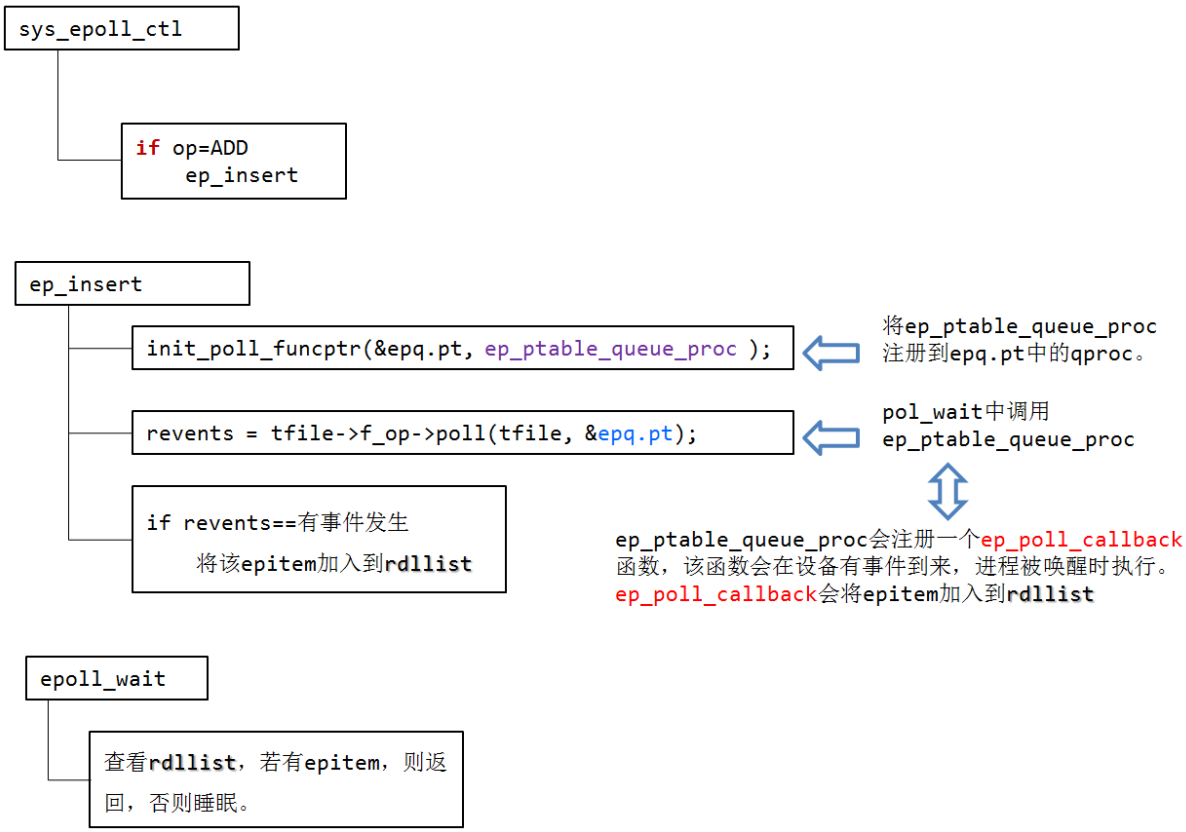
- epoll描述符
- op表示操作，可以是EPOLL_CTL_ADD，EPOLL_CTL_DEL和EPOLL_CTL_MOD
- fd是要监听的fd
- event是要监听的事件，同时可以设置ET和LT

```
 int epoll_wait(int epfd, struct epoll_event * events, int maxevents,
int timeout);
```

进行事件等待，events返回等待成功的时间，maxevents告诉events的个数，timeout等待时间精度为ms。

LT和ET：LT只要可行，就会重复提醒，ET只有当状态变为可行时，会提醒。
ET只支持非阻塞IO
因为二者的特点，LT可以不取完所有数据，但是ET必须操作到不能再次操作(errno为EAGAIN)，也因此，ET的触发次数更少

- epoll为什么高效
  - select/poll每次需要传递所有/要监控的fd给系统调用(这意味着每次都要将fd列表拷贝到内核态，很低效)，而epoll_wait时，不需要这步，因为之前已经通过epoll_ctl进行更改感兴趣的fd了。
  - epoll以红黑树保存fd在内核的cache里，因此有着高性能的插入、查找等操作
  - epoll通过设置回调函数，使就绪事件自动加入到就绪list中，因此每次只需要拷贝少许的就绪事件到用户态即可，而select和poll每次返回还得进行O(n)的遍历才能得到激活的事件。
- epoll 详解 epoll本质1 epoll本质2 epoll本质3 流程总结如下：
  - 创建epoll对象后，可以用epoll_ctl添加或删除所要监听的socket。以添加socket为例，如下图，如果通过epoll_ctl添加sock1、sock2和sock3的监视，内核会将eventpoll添加到这三个socket的等待队列中。
  - 当socket收到数据后，中断程序会给eventpoll的“就绪列表”添加socket引用。如下图展示的是sock2和sock3收到数据后，中断程序让rdlist引用这两个socket。
  - 假设计算机中正在运行进程A和进程B，在某时刻进程A运行到了epoll_wait语句。如下图所示，内核会将进程A放入eventpoll的等待队列中，阻塞进程。
  - 当socket接收到数据，中断程序一方面修改rdlist，另一方面唤醒eventpoll等待队列中的进程，进程A再次进入运行状态（如下图）。 epoll整体流程如图



5.HTTP和HTTPS的区别(优缺点，不同)

HTTP 有以下安全性问题：

- 通信使用明文，内容可能会被窃听；
- 不验证通信方的身份，因此有可能遭遇伪装；
- 无法证明报文的完整性，所以有可能已遭篡改。

HTTPs 并不是新协议，而是 HTTP 先和 SSL（Secure Socket Layer）通信，再由 SSL 和 TCP 通信。通过使用 SSL，HTTPs 提供了加密、认证和完整性保护。 认证 通过使用 证书 来对通信方进行认证。证书中有公开密钥数据，如果可以验证公开密钥的确属于通信方的，那么就可以确定通信方是可靠的。

数字证书认证机构（CA，Certificate Authority）颁发的公开密钥证书，可以通过 CA 对其进行验证。 进行 HTTPs 通信时，服务器会把证书发送给客户端，客户端取得其中的公开密钥之后，就可以开始加密过程。

HTTPS在三次握手后，要进行SSL进行协商加密使用的秘钥，且HTTP用80，HTTPS用的 是443端口。

使用HTTPS需要SSL，延时更高，并且需要购买CA证书，且占用CPU资源更高，因此应该有选择的在需要加密的时候才选用HTTPS协议。

https://www.cnblogs.com/sunsky303/p/10628894.html

6.HTTP返回码

2XX 成功
200 OK
204 No Content：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
206 Partial Content

3XX 重定向
301 Moved Permanently：永久性重定向
302 Found：临时性重定向
303 See Other
注：虽然 HTTP 协议规定 301、302 状态下重定向时不允许把 POST 方法改成 GET 方法，但是大多数浏览器都会把 301、302 和 303 状态下的重定向把 POST 方法改成 GET 方法。
304 Not Modified：如果请求报文首部包含一些条件，例如：If-Match，If-ModifiedSince，If-None-Match，If-Range，If-Unmodified-Since，但是不满足条件，则服务器会返回 304 状态码。
307 Temporary Redirect：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法。

4XX 客户端错误
400 Bad Request：请求报文中存在语法错误
401 Unauthorized：该状态码表示发送的请求需要有通过 HTTP 认证（BASIC 认证、DIGEST 认证）的认证信息。如果之前已进行过一次请求，则表示用户认证失败。

7.浏览器输入URL发生的事，用到了哪些层

浏览器中输入URL，首先浏览器要将URL解析为IP地址，解析域名就要用到DNS协议，首先主机会查询DNS的缓存，如果没有就给本地DNS发送查询请求。DNS查询分为两种方式，一种是递归查询，一种是迭代查询。如果是迭代查询，本地的DNS服务器，向根域名服务器发送查询请求，根域名服务器告知该域名的一级域名服务器，然后本地服务器给该一级域名服务器发送查询请求，然后依次类推直到查询到该域名的IP地址。DNS服务器是基于UDP的，因此会用到UDP协议。

得到IP地址后，浏览器就要与服务器建立一个http连接。因此要用到http协议，http协议报文格式上面已经提到。http生成一个get请求报文，将该报文传给TCP层处理。如果采用https还会先对http数据进行加密。TCP层如果有需要先将HTTP数据包分片，分片依据路径MTU和MSS。TCP的数据包然后会发送给IP层，用到IP协议。IP层通过路由选路，一跳一跳发送到目的地址。当然在一个网段内的寻址是通过以太网协议实现(也可以是其他物理层协议，比如PPP，SLIP)，以太网协议需要直到目的IP地址的物理地址，有需要ARP协议。

8.GET和POST的区别

GET：获取资源
POST：传输实体主体
POST 主要目的不是获取资源，而是传输实体主体数据。 GET 和 POST 的请求都能使用额外的参数，但是 GET 的参数是以查询字符串出现在 URL中，而 POST 的参数存储在实体主体部分。

GET 的传参方式相比于 POST 安全性较差，因为 GET 传的参数在 URL 是可见的，可能会泄露私密信息。并且 GET 只支持 ASCII 字符，如果参数为中文则可能会出现乱码，而 POST 支持标准字符集。(**个人认为都不是很安全，要安全不该用 https 吗，看到个相对合理的答案：get 请求的url是在服务器上有日志记录，在浏览器也能查到历史记录，但是post请求的参数都在body里面，浏览器历史记录不到**)

- http 的几个方法，括号内为对应的 SQL 命令
    - GET(SELECT)：从服务器取出资源
    - POST(CREATE):在服务器新建资源
    - PUT(UPDATE):在服务器更新资源
    - DELETE(DELETE):从服务器删除资源

- restful(表象层状态转变) api
    - 每一个URI代表一种资源(我理解这个是核心，把http 理解为对资源的增删改查)；
    - 客户端和服务器之间，传递这种资源的某种表现层；
    - 客户端通过四个HTTP动词（get、post、put、delete），对服务器端资源进行操作，实现"表现层状态转化"。 对于操作的类型，不用参数表示，而是采用 http 命令来表示不同的操作类型。
- http 传参
    - url 传参：url 的一般格式为 `scheme://[userinfo@]host/path[?query][#fragement]`，各种方法都支持，但常用于 GET 请求。 其中 query 可以用来传参，
    - 利用 html 表单：content-type 决定了在表单中数据是什么形式的，默认为 application/x-www-form-urlencoded(以键值对的形式发送)，还可以为 multipart/form-data(每个键值对都有自己的内容类型及配置)，还可以传 json 啊什么的，比较灵活。

9.TCP

这部分在计网总结中有着很详细的介绍，刚看完不就，这里不再赘述了

脚本工具

1.cat(concatenate files and print on the standard output)

cat用来连接文件或标准输入输出，常用来显示内容，或者拼起来显示，常和重定位配合

cat[选项][文件]

主要三个用法：

- cat filename显示一整个文件
- cat > filenme创建新文件
- cat file1 file2 >file合并两个文件

参数如下：

- -A：显示所有
- -b：对非空行显示行号
- -E：每行结束显示$
- -n：输出行号
- -s：一行空白替换两行以上的

2. ls(list directory contents)

列出当前目录下的文件

常用参数如下：

- -l：除了文件名之外，还将文件的权限、所有者、文件大小等信息详细列出来。
- -a：列出目录下的所有文件，包括以 . 开头的隐含文件
- -h：human readable

3.head/tail

打印文件开头/结尾的内容。

常用参数：

- -n: 指定打印的行数，默认为 10 行
- -f(only tail): 追踪文件的末尾

4.less

不可编辑版的 vim

常用参数：

- -N: 显示行号
- -s: 连续空行为一行
- x[数字]: 将 tab 规定为指定数字空格

5.grep

- -e: 使用正则搜索, 注意:{}等一些特殊符号需要进行转义. 即使用{}
- -i: 不区分大小写
- -v: 查找不包含指定内容的行
- -w: 按单词搜索
- -c: 统计匹配到的次数
- -n: 显示行号
- -r: 逐层遍历目录查找
- -A: 显示匹配行及前面多少行, 如: -A3, 则表示显示匹配行及前3行
- -B: 显示匹配行及后面多少行, 如: -B3, 则表示显示匹配行及后3行
- -C: 显示匹配行前后多少行, 如: -C3, 则表示显示批量行前后3行
- -E: 扩展的正则表达式. 使用该参数时不用加转义符, 如可以直接使用[a-z]{5}
- --color: 匹配到的内容高亮显示
- --include: 指定匹配的文件类型
- --exclude: 过滤不需要匹配的文件类型

6.wc(word count)

统计行数、单词以及字节数。

- -l: 输出行号

7.chmod

更改文件权限。

常用参数：

- +x/+r/+w: 增加执行/读/写权限，-也可以
- chmod 数字 文件名: 修改权限为数字，排序为owner group others，常用 644(rw-r-r)，755(rwx-rx-rx) 以及 777(rwx-rwx-rwx)

8.ps(Process Status)

默认显示当前用户，有控制终端的进程

- -aux: 显示所有进程，包括其他用户的

9. &/ jobs / fg / bg

- &：在后台运行进程，终端关闭/ssh关闭，命令会被终止
- jobs：显示出当前终端下启动的命令，-l 显示 pid
- fg %1: 把 1 号 job 放到前台，并开始运行
- bg %1: 继续被挂起的进程，默认是最后一个

9.kill

向进程发送信号，莫问发送 SIGTERM 信号给进程(该信号默认为终止进程，但是可以被捕获，所以有可能失败)。可以指定 SIGKILL(终止进程，不能被捕获)，强制杀死进程、 killall + 名字，会删掉所有有这个名字的进程。 ctrl+c -> SIGINT 进程终止 ctrl+z -> SIGTSTP 停止信号，前台进程放入后台并挂起

编程语言

基础

- static作用是什么？在C和C++中有何区别?
    - static可以修饰局部变量（静态局部变量）、全局变量（静态全局变量）和函数，被修饰的变量存储位置在静态区。对于静态局部变量，相对于一般局部变量其生命周期长，直到程序运行结束而非函数调用结束，且只在第一次被调用时定义；对于静态全局变量，相对于全局变量其可见范围被缩小，只能在本文件中可见；修饰函数时作用和修饰全局变量相同，都是为了限定访问域。
    - 在c++中，static还可以修饰类的成员(成员函数或者成员变量)，成员变量和函数都不属于任何一个对象，所有实例共有。利用这种特性，可以在所有类实例中进行通信
- 指针和引用区别?
    - 引用只是别名，不占用具体存储空间，只有声明没有定义；指针时具体变量，需要占用存储空间。
    - 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
    - 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。
- 宏定义和内联函数(inline)区别?
    - 在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。
    - 内联函数本身是函数，强调函数特性，具有重载等功能。
    - 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了。
- restrict关键字?
    - restrict是c99标准引入的，它只可以用于限定和约束指针，并表明指针是访问一个数据对象的唯一且初始的方式. 即它告诉编译器，所有修改该指针所指向内存中内容的操作都必须通过该指针

来修改, 而不能通过其它途径(其它变量或指针)来修改;这样做的好处是,能帮助编译器进行更好的优化代码,生成更有效率的汇编代码。
    - 现在程序员用restrict修饰一个指针，意思就是“只要这个指针活着，我保证这个指针独享这片内存，没有‘别人’可以修改这个指针指向的这片内存，所有修改都得通过这个指针来”。由于这个指针的生命周期是已知的，编译器可以放心大胆地把这片内存中前若干字节用寄存器cache起来。

- .cpp和.h
  没有头文件其实也能够进行工作，但是每次需要重复的进行声明。引入头文件，可以使模块被使用时，include头文件(会在include文件中进行扩展)，这样便避免了重复的进行声明。

  头文件中应该放： * 变量或者函数的声明(放置定义的话，如果有多个文件include，那么会重复定义) * 全局const对象，因为默认的const对象是没有extern声明的，只在包含的文件内部有效(相当于每个文件都有一个自己的const对象，而且都一样)。 * static对象类似const，也一样默认不带extern * inline函数，因为inline在遇到的地方会进行内联展开，因此，编译器反而需要inline函数的完整定义，才能做展开 * class的定义，因为创建类对象时，也要知道类的完全定义，一般将定义放在头文件，实现放在.cpp中，或者都写在定义中写实现，这种情况默认是内联的

- c++的四种cast(static_cast,dynamic_cast,const_cast,reinterpret_cast)
  首先说一下，和旧式的强制转换(T(exp) or (T)exp)，更推荐采用新式的转换，因为更容易辨认，并且对转换的功能进行了分类，更不容易出错

    - const_cast
      通常被用来移除对象的常量性，也是唯一有这个能力的cast运算符，甚至 static 都不行
    - dynamic_cast(run-time type identification)
        - 多态类型转换(上下均可)
        - 执行运行时的类型检查
        - 只能操作指针或引用，指针失败返回nullptr，引用失败跑出bad_cast异常
        - 子类转父类，成功，父类转子类
            - p指向子类，转换成功
            - 否则，转换失败
        - **很重要的是**，只能用于基类至少有一个虚函数才能使用，因为RTTI必须要有虚函数表的支持。
    - static_cast(编译阶段)
        - 强迫的隐式转换
        - 不执行运行时的类型检查(没有dynamic_cast安全)
        - 可以在整个类的层次中移动指针，向上或向下转换(不安全)
    - reinterpret_cast
        - 低级转换，用于位的简单重新解释
        - 具体取决于编译器，难以移植

- 智能指针

    - unique_ptr
        - 独占智能指针指向的资源
        - 不允许复制，只能移动(因为要独占资源)
    - shared_ptr
        - 带有一个引用计数，记录资源被引用的次数，这样当最后一个对象不再持有资源时，可以释放资源

        - 同样支持移动语意
        - 常用成员
            - use_count：引用个数
            - unique：应用个数是否为1
            - swap：交换shared_ptr
            - reset：当前变量放弃资源
            - get：返回内部指针
    - weak_ptr
        - shared_ptr的一种扩充，不管理指向的对象，但是可以对对象有效性进行监控
        - 可以用来解决shared_ptr环形引用的问题
        - expired()：测试是否失效
        - lock()：返回shared_ptr
          在项目中，有过timer的回调持有Connection以及Connecttion通过Http类持有timer的问题，最后就是通过weak_ptr解决的

    对于智能指针，建议优先选用make_unique和make_shared来创建智能指针，可以避免new之后没有被赋给智能指针就被析构的风险，不容易造成内存泄露，并且，make系列只需要一次内存分配，效率更高 make_shared优点

- 在继承体系中，为什么析构函数必须是虚函数？为什么默认的不是虚函数？构造函数呢？ 析构函数是虚函数可以保证我们用基类指针操纵子类对象时，能够调用子类的析构函数，正确的释放资源

  默认的不是是因为实现多态需要虚函数表和虚表指针，没有继承关系，或者不准备使用多态的话，没必要引入这个额外的开销

  对于构造函数，因为虚函数的调用依赖vptr(`类似于*this->vptr[1](para)`，然而vptr要在构造函数中初始化，所以构造函数不能为虚函数

- 重载和重写 重载是两个函数名相同，但是参数列表不同(个数，类型)，在同一作用域中。const当形参为值传递是不能重载的，为引用/指针时，可以重载。

  重写是子类重写父类的虚函数

- 字符串操作

    - strcpy `char* strcpy(char* dest, const char* src);` 用来将src拷贝到dest，直到遇见'\0'，因为没有指定长度，所以拷贝可能越界，不安全

      `char *strncpy(char *strDest,const char *strSource,size_t count);` 将src的count个字符拷贝到dest中，如果count小于source，则后面的不会进行拷贝，如果超过，将填入NULL，是strcpy的安全版本

    - strlen计算字符串长度，返回到'\0'的字符个数

- 构造一个在main之前执行的函数

    - 利用函数属性__attribute((constructor))，可以在main函数之前执行
    - 全局变量类的构造在main之前
    - 利用全局lambda表达式

- extern "C"
  由于c++支持重载，所以c++生成的符号会根据参数、命名空间来确定函数的签名，和c生成的方式不一样。

  所以，在要被c++调用的内容中，增加extern "C"来告诉编译器这部分是c的接口

  同时，为了使内容也能够被c本身调用，可以通过__cplusplus宏来判断是否是c++编译器，具体如下：

```
#include<stdio.h>
#ifdef __cplusplus
extern "C"{
#endif
void testCfun();
#ifdef __cplusplus
}
#endif
```

- new/delete和malloc/free的区别
  - new/delete基本用法

```
int *pi=new int;
int *pi=new int();
int *pi=new int(1024);
delete pi;
pi = nullptr;

int *pi=new int[]; //指针pi所指向的数组未初始化
int *pi=new int[n]; //指针pi指向长度为n的数组，未初始化
int *pi=new int[](); //指针pi所指向的地址初始化为0
delete [] pi; //回收pi所指向的数组
```

    这里顺便说一句，在cpp中，NULL被定义为了0(不是c中的(void*)(0))，cpp中应该使用nullptr来表示空指针

  - malloc/free基本用法

```
void *malloc(size_t size);
void free(void *pointer);
//直接初始化为0
void *calloc(size_t num_elements,size_t element_size);
//再分配
void realloc(void *tr , size_t new_size);
```

  - 区别
    - new/delete是c++中的关键字，而后面几个是c中的函数

    - new不需要根据类型计算大小，而malloc需要
    - new返回的是对象类型的指针，而malloc返回的是void*，需要类型转换
    - 分配失败，new跑出bad_allod异常，malloc返回NULL
    - 对于自定义类型，new实际上是分为两部分，首先调用operator new申请内存(这步等价于malloc)，然后再调用placement new调用构造函数，初始化。delete首先调用析构，然后operator delete释放内存
      malloc只进行内存的申请和释放
    - new/delete是可以重载的

- 运行时类型检查(RTTI)
  在c++中体现为dynamic_cast和typeid，VS中虚函数表-1的位置存放了type_info指针，二者均去查询type_info

  当typeid操作数是不带有虚函数的类型时，取出的是操作数的类型

- 介绍STL的allocator
  首先STL内存分配阶段，new和delete都是分为内存申请以及内存构造相关的两部分。STL allocator也将两个阶段区分开来，以配置为例分为`alloc::allocate()`和`alloc::construct`两部分。

  在<<STL源码解析>>中，介绍了早期的分配方法，allocator采用了两级的配置器，减小内存碎片问题，当分配空间大于128B时，采用一级空间配置器，直接采用malloc等进行操作，当申请空间小于等于128B时，采用二级配置器。二级配置器从8开始，掌管了16个链表的内存池，配置时采用向上舍入的方式。

  在g++4.8中，默认使用的是new_allocator，只是简单地封装了operator new而已，没有再使用内存池(如果这个时候有内存没有被释放，说明是lib里的cache没有释放，和STL无关，例如vector就不会释放无效的内存，可以调用shrink_to_fit手动释放)。如果选用__pool_alloc，是带有内存池版本的分配器，和上述相同。

  那么为什么在新版的实现中，选用了默认不带内存池的分配器呢?

  首先，新版本没采用内存池，可能速度上会更慢，但是优势是简单并且在各种硬件和操作系统中也能正确工作。

- 迭代器失效问题

    - 对于序列式容器，vector和dequeue，由于会移动后序的内容，所以删除元素之后的迭代器失效
    - 对于map set关联容器，除了删除的元素，其余的迭代器均有效，erase会返回下一个迭代器
    - list类似，因为同样是不连续分配的内存

- c++类的访问权限
  访问权限有三种，分别是public，protected，以及private，区别是

    - public：都可见
    - protected：对类内和派生类可见
    - private：仅类内可见

  继承时，同样有三种继承方式

    - public：不改变基类属性
    - protected：public和protected都为protected，private不变

    - private：均为private

- 右值引用/std::move/完美转发

    - 右值：指的是程序中的临时值，是不能取地址的值，对于纯右值来说，在表达式结束之后，立马会被销毁。
    - 右值引用：方式为`type&& k = rlvate`,这样可以延长临时值的声明，到和k相同，当T需要类型推导时，此时此时既可以是左值，也可以是右值，也叫作万能引用，可以利用std::forward进行完美转发
    - 通过移动构造函数，我们可以充分的利用右值的资源，进行所有权的移动，这样可以减少拷贝的次数。对于左值，std::move可以强制转换为右值，从而调用移动构造，对拷贝次数进行优化。

  需要注意，实际上std::move什么都没做，只是强制转换为右值，从而使对象能够调用移动构造，具体的优化，要看是否提供了移动构造 还有一点，右值引用只能延长 prvalue 的生命周期，对于 xvalue(也就是 std::move 产生的，并不能起到延长的作用)

- c++源文件到执行文件过程

    - 预处理：替换头文件、宏定义(#开头的指令)等，生成预编译文件
    - 编译：生成汇编文件
    - 汇编：转换成机器码，生成可重定位目标文件
    - 连接：多个目标文件和库连接，生成可执行目标文件

- include头文件的顺序以及""和<>的区别
  对于include头文件来说，如果a.h中声明了一个b.h中定义的变量，而没有引用b.h，那么a.c中要先引用b.h，在引用a.h，否则会汇报未声明错误

    - **google风格**

        - 本cpp对应的.h
        - C系统文件
        - C++系统文件
        - 其他库的.h
        - 本项目内的.h
          这样可以防止本cpp.h漏了包含必须的头文件，防止其他文件包含的时候报错

    - <>：先去系统目录中找头文件，没有再在当前目录找

    - ""：首先在当前目录查找，再去系统目录查找

- malloc原理

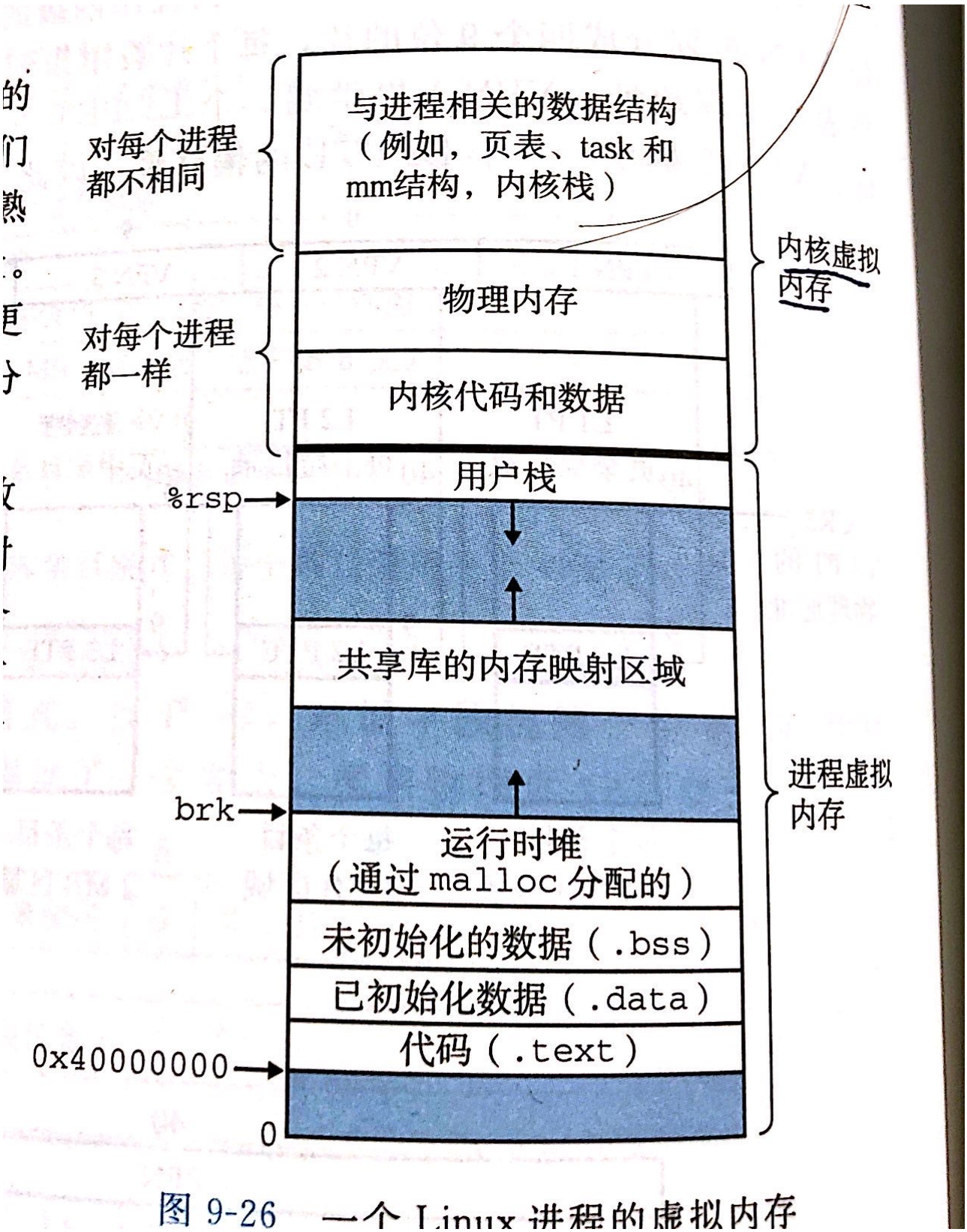


图 9-26 一个 Linux 进程的虚拟内存

  - 首先介绍一下进程的内存布局(从低到高)

    - 代码段：存放机器指令
    - 数据段：初始化的全局数据和静态数据(生命遍布整个程序运行周期)
    - BSS段：未初始化的全局和静态数据
    - 堆：有一个指向堆顶的brk指针，对和映射段都是用于动态申请的内存
    - 映射段：用于大于128k的内存申请，高效IO
    - 栈：存放临时遍历，以及函数调用的上下文保存(利用后入先出的特点)

        - 内核空间：用户进程不能直接访问

    - malloc申请的动态内存是在堆和映射区申请的。当申请小于128k的内存，在heap中进行申请(如果heap大小不够，调用brk或者sbrk扩展heap)，当申请大内存是，利用mmap在映射区进行匿名映射，申请内存。

      malloc采用的是基于分离适配的分离空闲链表，这样可以避免显示空闲链表需要线性时间进行查找的问题。

      ps:分离适配是找到最合适的块后，对其进行分割，并且插入到适合的表中的方法。具体可见csapp(e3)第9章。

- 类的内存分布

    - static static关键字在类内，可以用来修饰成员变量和成员函数
        - 静态成员变量：存放在对象外，即使没有对象也可以被访问
        - 静态成员函数：和非静态成员函数一样，在代码段，但是，因为static不和任何对象由联系，没有this指针，所以只能访问静态数据成员
    - virtual
        - virtual函数：每个class为每个虚函数的指针构成了一个虚函数表，虚函数表在代码段，对象内部通过虚表指针(vptr)指向虚表，此外，virtual table还含有typeinfo，通常在第一项。vptr是在构造和析构函数中被设定和抹消的
        - virtual继承：对于虚继承，因为要实现共享实例，cfront是在派生类中安排指向虚基类的指针，达到共享的目的。MS将虚基类通过虚基类表实现，派生类多一个指针即可。还有一种是在虚函数表中，放置虚基类的offeset，达到共用实体的效果。

- 大小端判断

```
//true 小端，  false 大端
union test {
    int i;
    char c;
}
test t;
t.i =  1;
return t.c == 1;
```

- 布隆过滤器 利用k个哈希函数，映射为mbit数组，通过判断是否被置位来判断元素是否在集合中，复杂度为O(k)

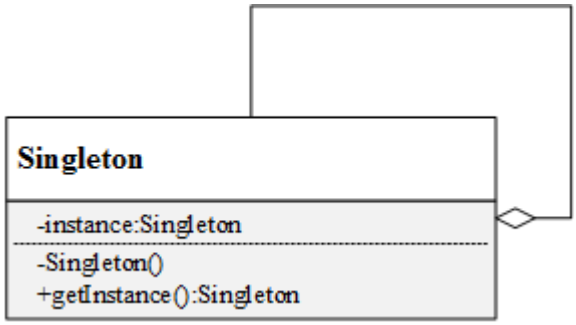
  但是结果只有不在是确定的，在只能是概率上在

c++设计模式

设计模式主要是为了解决某类重复出现的问题而出现的一套成功或有效的解决方案。设计模式提供一种讨论软件设计的公共语言，使得熟练设计者的设计经验可以被初学者和其他设计者掌握。

1. 单例模式(确保一个类只有一个实例，并提供一个全局访问点来访问这个实例)

有些东西是整个程序独占的，比如任务管理器、打印机这种，没必要生成多个一样的实例，既造成资源的浪费，又会让用户困惑。



下面是单例模式的代码，需要特别注意，单例模式要禁用掉构造及拷贝构造等。此外，代码利用的局部静态变量的线程安全性。

```
// 线程安全的单例模式
class Singleton
{
private:
    Singleton() { }
    ~Singleton() { }
    Singleton(const Singleton &);
    Singleton & operator = (const Singleton &);
public:
    static Singleton & GetInstance()
    {
        static Singleton instance;
        return instance;
    }
};
```

modern c++

- 结构化绑定(c++17)

```
multimap<string, int>::iterator
  lower, upper;
std::tie(lower, upper) =
  mmp.equal_range("four");
auto [lower, upper] =
  mmp.equal_range("four");
```

可以直接用 auto，捕获 pair、tuple 等的元素值，还是很方便的，注意vector 这种编译期长度未知的，不可以捕获。

- lambda

  使用 lambda 表达式可以快速的创建一个匿名的函数对象，在一些操作中可以得到更简洁的代码。

    - 本地变量名标明对其按值捕获（不能在默认捕获符 = 后出现；因其已自动按值捕获所有本地变量）
    - & 加本地变量名标明对其按引用捕获（不能在默认捕获符 & 后出现；因其已自动按引用捕获所有本地变量）
    - this 标明按引用捕获外围对象（针对 lambda 表达式定义出现在一个非静态类成员内的情况）；注意默认捕获符 = 和 & 号可以自动捕获 this（并且在 C++20 之前，在 = 后写 this 会导致出错）
    - *this 标明按值捕获外围对象（针对 lambda 表达式定义出现在一个非静态类成员内的情况；C++17 新增语法）

      go 的闭包和 c++的很不同，go 捕获的都是引用，可以通过显示的拷贝变量来捕获值

- c++函数式编程 许多高阶函数在函数式编程中已成为基本的惯用法，在不同语言中都会出现，虽然可能是以不同的名字。我们在此介绍非常常见的三个，map（映射）、reduce（归并）和 filter（过滤）。
    - Map 在 C++ 中的直接映射是 transform（在 头文件中提供）。它所做的事情也是数学上的映射，把一个范围里的对象转换成相同数量的另外一些对象。这个函数的基本实现非常简单，但这是一种强大的抽象，在很多场合都用得上。

```
template< class InputIt, class OutputIt, class UnaryOperation >
constexpr OutputIt transform( InputIt first1, InputIt last1, OutputIt
d_first, UnaryOperation unary_op );
// 将 [first1, last1)尽心 unary_op 操作，并存取到 d_first 开头的地方
```

    - Reduce 在 C++ 中的直接映射是 accumulate（在 头文件中提供）。它的功能是在指定的范围里，使用给定的初值和函数对象，从左到右对数值进行归并。在不提供函数对象作为第四个参数时，功能上相当于默认提供了加法函数对象，这时相当于做累加；提供了其他函数对象时，那当然就是使用该函数对象进行归并了。

```
template< class InputIt, class T, class BinaryOperation >
constexpr T accumulate( InputIt first, InputIt last, T init,
BinaryOperation op );
// 计算给定值 init 与给定范围 [first, last) 中元素的和。第一版本用 operator+
，第二版本用二元函数 op 求和元素，均将 std::move 应用到其左侧运算数 (C++20
起)。
```

- 并发
    - 提供了 mutex 类，主要提供了 lock、trylock 和 unlock 的方法，和 linux 基本一致，还提供了 lock_guard的 RAII 方式的加减锁。
    - condition_variable条件变量，有 wait，notiy 等方法
- 智能指针

    - unique_ptr：独占的持有一个资源，析构会释放持有的指针的内存。由于资源是独占的，因此拷贝构造是需要被禁用的，且赋值函数，需要用 swap 成员函数实现。
    - shared_ptr: 多个共享一个资源，析构时会判断引用计数，如果为 0 则释放资源。
    - weak_ptr: 使用 shared_ptr会有循环引用的问题，weak_ptr可以通过expired方法观察是否失效，且通过lock 可以提升为 shared_ptr。

  此外，最好使用 make_shared/make_unique 进行申请智能指针

- 右值和移动

    - 纯右值 prvalue：没有标识符、不可以取地址的表达式，一般也称之为“临时对象”。最常见的情况有：

        - 返回非引用类型的表达式，如 x++、x + 1、make_shared(42)
        - 除字符串字面量之外的字面量，如 42、true

      右值之前可以绑定在常引用上，现在也可以利用 T&&右值引用捕获，很反常理的是，右值引用本身是一个左值

    - std::move 的可以看做有名字的右值，记为 xvalue

  采用右值引用T&&可以延长prvalue 的生命周期，但不能延长xvalue 的。

- 异常

    - 异常安全：指当异常发生时，既不会发生资源泄漏，系统也不会处于一个不一致的状态 详细内容，查看异常
    - 在go中，panic 相当于 go 的异常，也会层层上报，直到被 recover或者退出程序。

- 成员函数说明符 override 和 final 是两个 C++11 引入的新说明符。它们不是关键词，仅在出现在函数声明尾部时起作用，不影响我们使用这两个词作变量名等其他用途。

    - override 显式声明了成员函数是一个虚函数且覆盖了基类中的该函数。如果有 override 声明的函数不是虚函数，或基类中不存在这个虚函数，编译器会报告错误
    - final 则声明了成员函数是一个虚函数，且该虚函数不可在派生类中被覆盖。如果有一点没有得到满足的话，编译器就会报错。final 还有一个作用是标志某个类或结构不可被派生。同样，这时应将其放在被定义的类或结构名后面。

- 无锁队列见无锁队列

算法和数据结构

1.一致性哈希

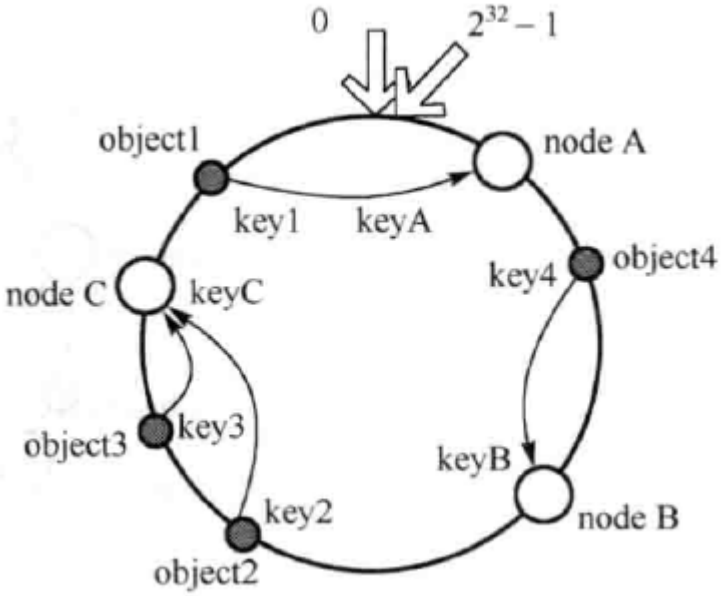
- 普通集群
  将固定的key映射到固定的节点，节点只存放各自key的数据。

  这种方法需要维护一个key和节点关系的表格，当其中一台宕机，节点的数据要进行迁移，表格要重新维护。并且，当查找某个key对应数据，需要遍历所有表格，直到找到存放的节点，然后再去节点读

- hash集群
  很容易想到的一个解决查找表的方法是对key做hash，将keyhash到集群中的各个节点中。

hash集群可以快速找到key对应的节点，然后从中取出数据，但是当集群增减节点的时候，需要将整个集群的数据重新映射一遍才行，工作量太大

- 一致性hash
  一致性hash为了在增加或减少一个节点的时候，尽可能的小的改变已存在key的映射关系

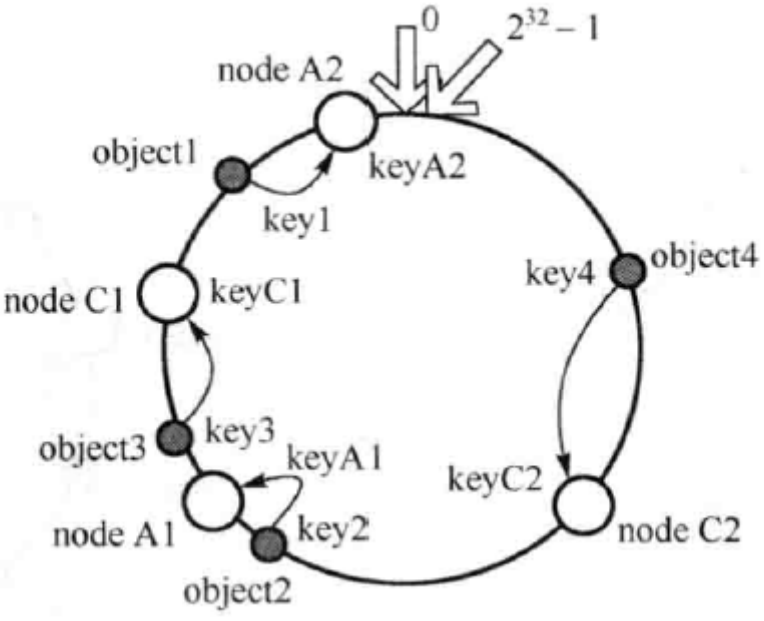


示意图如图，算法将hash的值空间当做一个环形，然后对数据(object)做hash，得出哈希值在圆环中的分布，然后再将集群中的节点也映射到圆环中(可以利用IP或主机名等作为关键字key)，在选取数据映射的节点时，数据选取顺时针方向遇到的第一个key节点机器作为存储。

在增加节点时，显然只会影响环形空间中，新加节点逆时针访问，直到下一个机器节点中间的数据。

在移除节点，受影响的只有删除节点逆时针访问的数据(直到下一个节点)

可以看到，增加和移除节点受影响的范围被大大减少到只有圆环中的一小部分



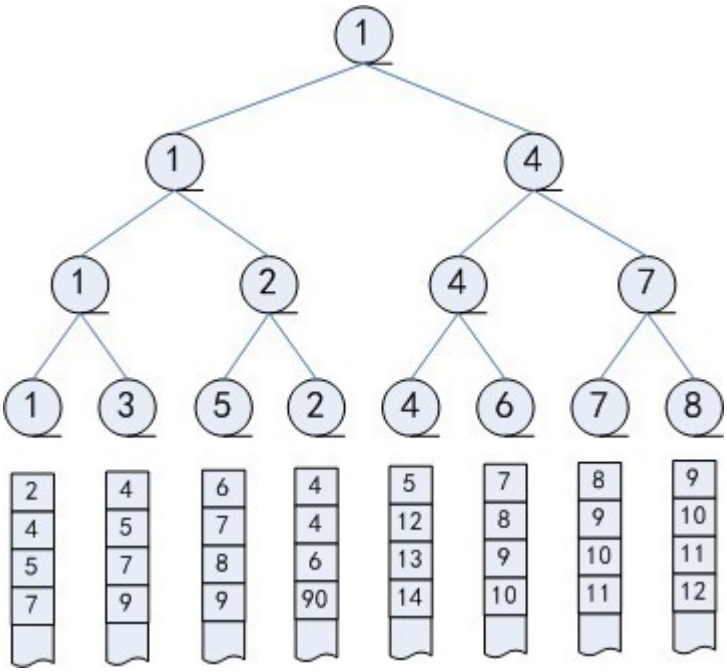
当机器节点比较少，分布不均匀的时候，有可能节点到机器映射的很不平衡，可以为每个机器产生虚拟节点来解决这个问题。

2.胜者树，败者树

采用2路归并排序，每个数据需要进行log2N次IO，访问外存的次数较多，效率很低。

如果采用K路选择，则每个只要进行logKN次，IO，可以有效的减少IO次数，但是增加K值，会降低内部排序的效率，因此引入了胜者树、败者树算法 多路归并

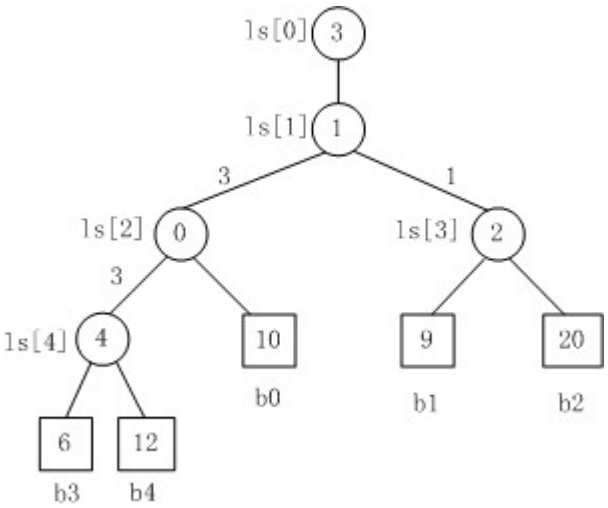
- 胜者树
  胜者树的定义：
  - 胜者树是一颗完全二叉树
  - 胜者树的叶子结点保存我们的一个输入缓冲区（一路归并顺序表）； 叶节点L[1.....n]
  - 胜者树的非叶子节点保存当前比较的胜者的输入缓冲区的指针；非叶子节点B[1.....n-1] //存储的是数组L的索引
  - 胜者树的根节点保存我们的胜者树当前的的一次比较中的冠军（最优值） B[0]



  然后，不断取出根节点，写入缓冲区，并且更新对应的叶子节点，然后不断向上跟新比较结果(跟兄弟比较)。

- 败者树
  败者树的定义：
  - 败者树是一颗完全二叉树（败者树是树形选择排序的一种变形）
  - 败者树的叶子结点保存的是我们的输入缓冲区
  - 败者树的非叶子结点保存我们的当前的比较中败者的对应的输入缓冲区的指针

- 败者树根保存我们的当前比较的亚军，根上面还有一个节点保存我们的冠军



败者树将败者储存，然后让胜者继续进行下一轮的比赛。 将新进入选择树的结点与其父结点进行比赛：将败者存放在父结点中；而胜者再与上一级的父结点比较。 比赛沿着到根结点的路径不断进行，直到ls[1]处。把败者存放在结点ls[1]中，胜者存放在ls[0]中。

胜者树、败者树和堆的相同点：空间和时间复杂度均相同——调整一次的时间复杂度都是 O(logN)，空间复杂度为O(1)。

但是，堆在取出堆顶后，进行跳转，会把最后一个数换到堆顶，然后下滤，下滤过程中，需要左右节点的值来决定下滤的方向。

胜者树相对于此，可以减少一次比较，因为只会在叶子节点更新的分支进行更新。

败者树相对于胜者树，如果更新的节点此前为胜者，那么和父亲节点比较即可，此时可以减少一次外存访问，但是如果是此前就是败者，那么就不得不更新了(但是，如果每次取的是根节点，似乎这种情况是不存在的)

3.海量数据处理

- 分治-hash映射(给定a、b两个文件，存放50亿url，并且占64字节，内存限制4GB，找出相同的url)
  将文件一点一点读入，然后对hash(url)%K，就可以将文件分为K个小文件，这样，相同的url一定被分在了同一个小文件中，可以再对小文件进行读入处理。

- topK问题(海量数据下找出最大的前K个数之类)

  - 如果能读入内存，快排找pivot方案可以在O(n)时间内找到
  - 维护小根堆，可以在O(nlogK)时间内找到
  - 利用hash(x) % K进行分块，在每个小文件统计单词出现的频率，再对文件进行归并排序。

- Bit-map bit-map可以用位数组来表示某些元素是否存在，可以大大节省存储空间。
  当需要判断集合是否重复时，如果数据量比较大，用此方法很合适。能够很好的节省空间。
  统计出现多次的话，可以用2个bit来表示

- 布隆过滤器
  bit-map是申请N(集合中最大整数)位的数组 ， 每一位对应一个特定整数

  布隆过滤将一个m位的数组，k个hash函数，每个hash函数可以将集合中的元素映射到为数组的某一位。

> 当插入一个元素时，用k个hash函数计算置位的k位。查询时，**如果k位都是1，则可能在，否则，一定不在。**

4.B树，B+树

B树也称B-树,它是一颗多路平衡查找树。我们描述一颗B树时需要指定它的阶数，阶数表示了一个结点最多有多少个孩子结点，一般用字母m表示阶数。当m取2时，就是我们常见的二叉搜索树。可以用节点的分支范围表示B数(ceil(m/2), m)树

一颗m阶的B树定义如下：

- 最多m-1个关键字
- 根节点最少可以只有1个关键字
- 非根节点至少有ceil(m/2)向上取整 - 1个关键字
- 节点中关键字升序排列，且关键字的左子树小于他，右子树大于他
- 所有叶子节点在同一层

因为B树是多路版本的平衡BST，所以即使有大量的数据，高度也可以很低。

- 插入和分裂
  当插入元素后，节点数超过m-1，则进行分裂，去中位数s=floor(m/2)，进行划分，将key_s上升一层，逐层上传，如果根节点还上溢，则整棵树高度上升一层。
- 删除和下溢
  将要删除节点和后继互换，然后在后继(右的最左，一定是叶子节点)的分支中删除
  当左右兄弟存在，且-1不下溢的话，可以通过旋转，来进行补齐
  当兄弟不满足删除条件，可以进行合并，这会导致父亲节点减少一个节点，因此也需要逐层上传。

B+树和B树的区别在于

- n个子树的节点中有n个关键字
- 非叶子节点是索引部分，数据域只保存在叶子节点中

由于这些特点，B+树与B树相比

- key的大小变小了，可以支持更多的key，IO效率高
- B+遍历全部信息只需要遍历叶子节点即可
- 查询效率稳定，每次查询效率相当

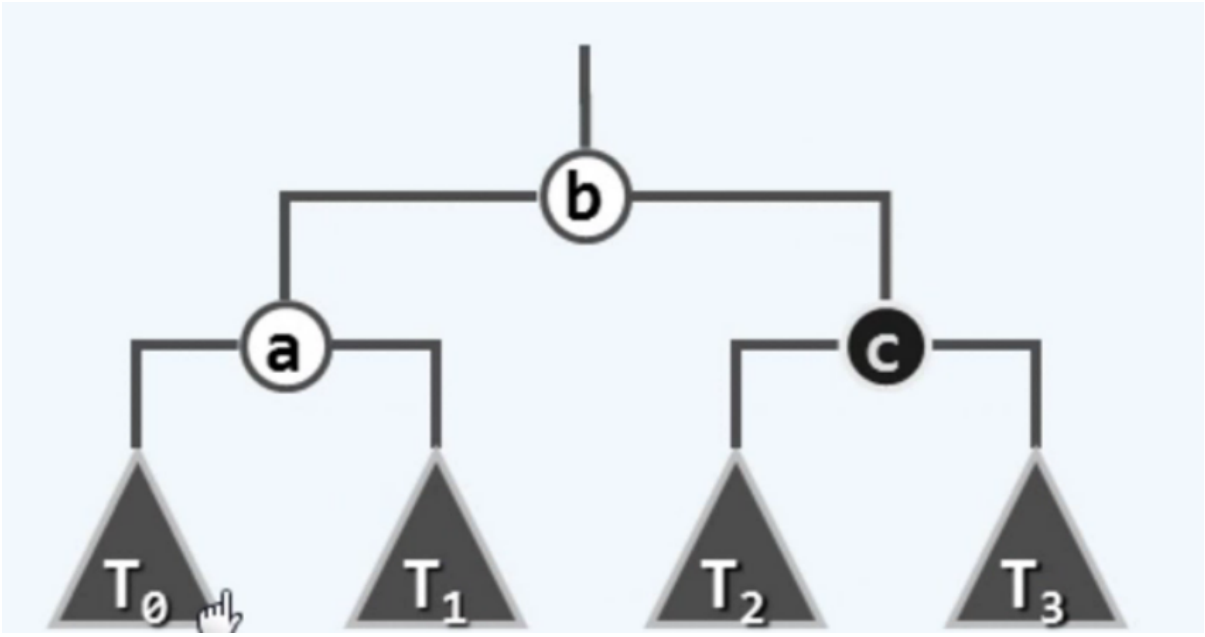
5.AVL和红黑树

AVL是左右子树平衡因子(高度差绝对值)小于等于1的树。

插入删除都可能造成失衡，可以通过3+4重构进行平衡。但是，删除操作会逐步上滤，至多需要logn次调整。

- 3+4 重构
  - 设 g 为最低的失衡节点，p 是 g 更高的那个孩子，v 是 p 更高的那个孩子。
  - 按中序排序，重命名为 a < b < c
  - 他们总共拥有互不相交的四颗子树

- 按中序遍历重命名为 T0 < T1 < T2 < T3

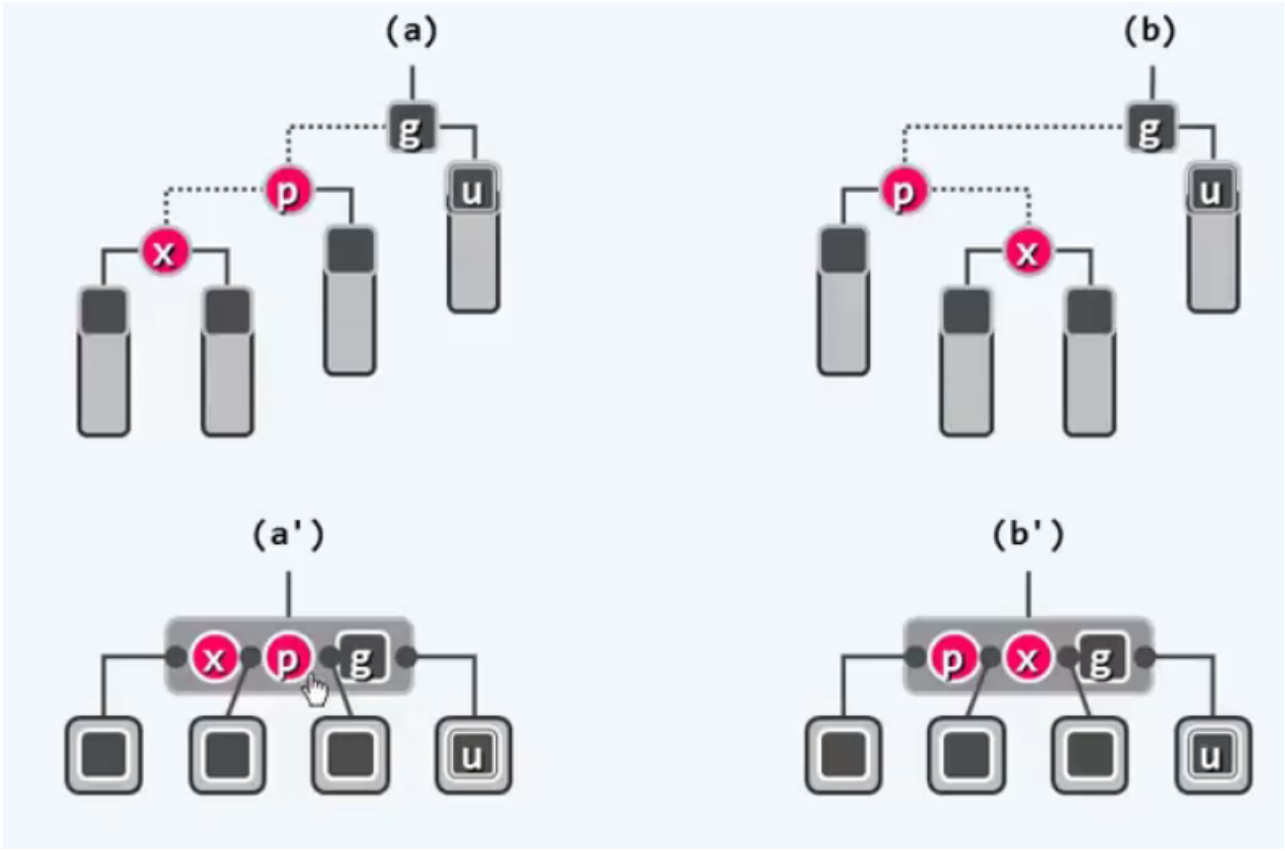


红黑树希望减小AVL删除操作的**logn次结构调整**，红黑树定义：

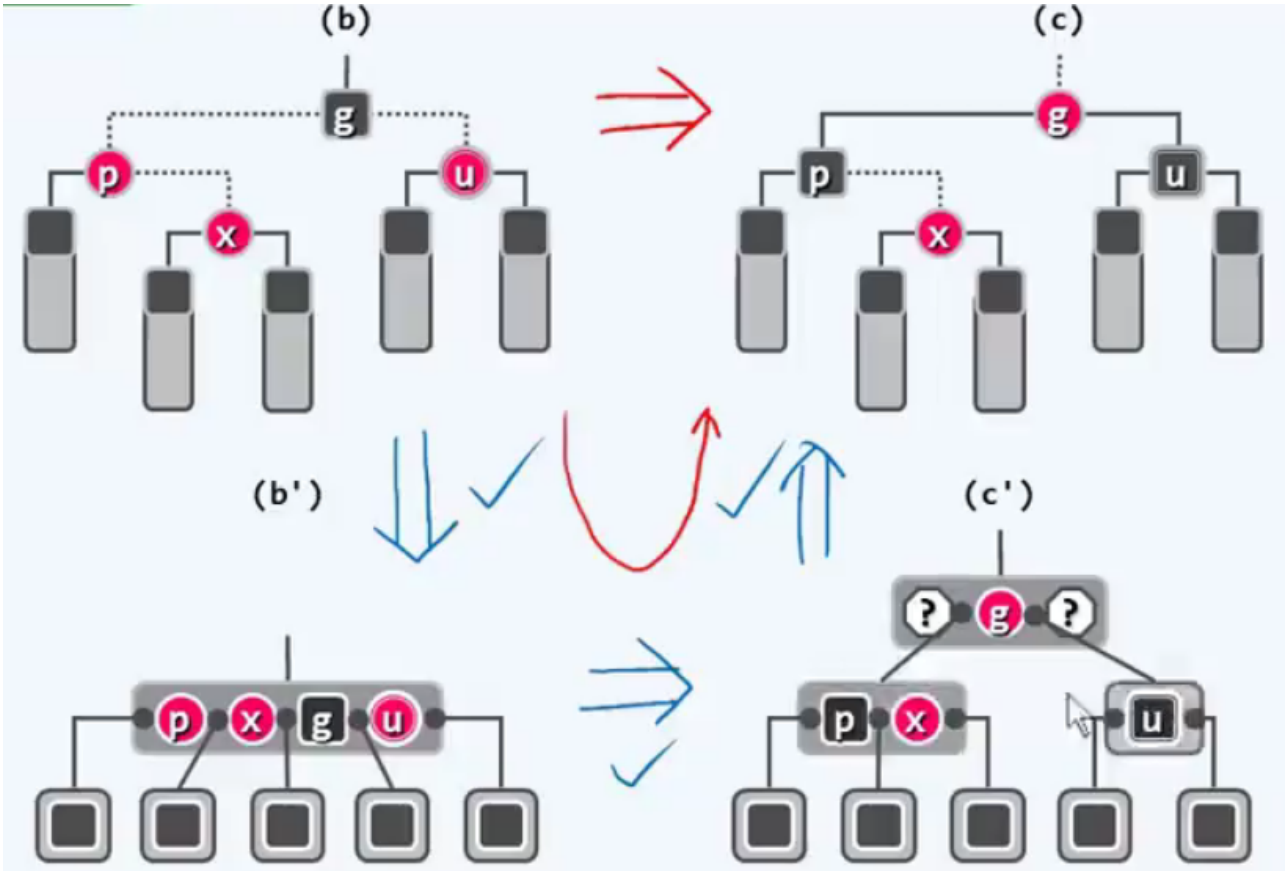
- 树根为黑色
- 外部节点均为黑色
- 其余节点，若为红，则只能有黑孩子(红的父，子均为黑)
- 外部节点到根的路径途中黑色节点数目相同

如果将红黑树的红节点上移一层，其实就是(2, 4)树

- 红黑树的插入和双红缺陷
  红黑树插入的新节点以红色染色进行插入，这样只有父子节点为黑这一条不满足。



如果双红上溢对应的等效B树节点没有溢出，那么只需要做3+4重构，改变染色，即可全局恢复



如果会发生溢出，那么就要进行分裂，将中间节点染红后，移入上一层，这种情况有可能上溢到根节点，但是，过程中只有染色，没有结构调整，而一旦进行结构调整，红黑树必将重新平衡

- 红黑树的删除和双黑缺陷 删除情况太多，过于复杂，先不考虑了。。。 但是要知道删除的结构调整次数也是只有常数次

go语言部分

go语言部分看笔记 go 语言实战 Go程序设计语言 Go源码 go 注意事项

1. GC

- 标记清除法
    - 标记：从程序的根节点开始，递归的遍历所有对象，将能遍历到的对象打上标记
    - 清楚：将所有未标记的对象当做垃圾销毁。

  这个算法缺陷在于 STW 问题，因为算法在标记及清除的时候，必须暂停其他所有程序，防止其他并发程序新增加的对象没来得及标记，就被清除了。因为遍历复杂度随着对象的增多而增大，在大型程序中STW 可能要达到毫秒级别，这是非常可怕的。

- 三色标记法，可以看GoGC 正常流程如下：
    - 新创建的对象，默认颜色为白色
    - 每次GC回收开始, 然后从根节点开始遍历所有对象，把遍历到的对象从白色集合放入“灰色”集合
    - 遍历灰色集合，将灰色对象引用的对象从白色集合放入灰色集合，之后将此灰色对象放入黑色集合
    - 重复第三步, 直到灰色中无任何对象
    - 回收所有的白色标记表的对象. 也就是回收垃圾

  如果三色标记只采用正常流程，那么在没有 STW 的情况下，是有可能出现问题的
    - 白色对象被黑色对象引用
    - 灰色对象和它可达的白色对象的关系遭到破坏(灰色丢了那个白色) 当上述两个问题成立时，对象就会丢失掉

  为了避免上述情况，满足下面两种方式，均可以破坏掉上面的两个条件
    - 强三色不变式：不存在黑色对象引用到白色对象的指针
    - 弱三色不变式：所有被黑色对象引用的白色对象都处于灰色保护状态

  为了解决上述问题，Go 利用了两种屏障：
    - 插入屏障：在A对象引用B对象的时候，B对象被标记为灰色。(将B挂在A下游，B必须被标记为灰色) 满足: 强三色不变式. (不存在黑色对象引用白色对象的情况了， 因为白色会强制变成灰色)，这种方式在结束时，需要 STW，来标记栈上的对象
    - 删除屏障：在 GC 过程中被删除的对象，如果自身为灰色或者白色，那么被标记为灰色。 满足: 弱三色不变式(保护灰色对象到白色对象的路径不会断)，这种方式的回收精度低，一个对象即使被删除了最后一个指向它的指针也依旧可以活过这一轮，在下一轮GC中被清理掉。

  为了解决屏障的问题，Go 1.8 提出了混合写屏障的机制：
    - GC 开始时将栈上对象标记为黑色
    - GC 期间，栈上创建对象标记为黑色
    - 被删除的对象标记为灰色
    - 表添加的对象标记为灰色
      满足变形的弱三色不变式，主要改进标记后需要重新 STW，标记栈上的问题。

mysql部分

mysql 必知必会 mysql

rpc

rpc 和 http 的区别，为什么要用 rpc

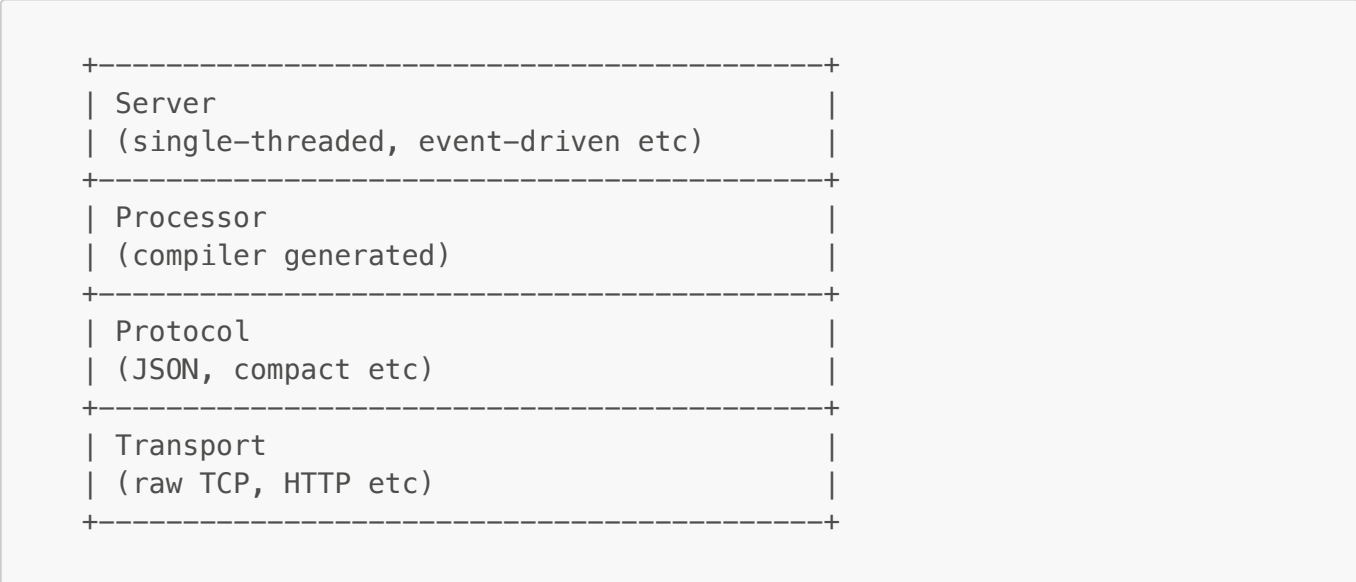
首先说下，http是一个基于TCP/IP通信协议来传递数据的协议，rpc 指的是远程过程调用，分为很多部分，通信协议部分可以采用 http(grpc)，也可以框架自行在 TCP 上进行封装。

在来说说为什么要做 rpc，这主要是因为在微服务化之前，服务端代码都是在一台机器上运行的(虽然可能有多个机器跑同样代码做负载均衡)，这时候的调用都是本地调用。后来由于单机代码太大，不好开发及维护，所以采用了微服务的架构，对代码进行拆分，原来本地的调用变成了远程的，也就需要 rpc 这种方式了。

如何设计 rpc 框架，thrift 介绍

1. 通信：要解决通讯的问题，主要是通过在客户端和服务器之间建立TCP连接，远程过程调用的所有交换的数据都在这个连接里传输。连接可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接。
2. 服务间寻址：A服务器上的应用怎么告诉底层的RPC框架，如何连接到B服务器（如主机或IP地址）以及特定的端口，方法的名称名称是什么，这样才能完成调用。
3. 序列化：传递参数的序列化及反序列化过程
4. 错误处理：重试、熔断等

下面根据上面几部分，介绍公司用的 thrift相关的内容，下面是 thrift 整体网络栈



技术栈分为四层，自底向上分别为：传输层(Transport Layer)、协议层(Protocol Layer)、处理层(Processor Layer)和服务层(Server Layer)。(thrift network stack)

1. 传输层(Transport Layer)：传输层负责直接从网络中读取和写入数据，它定义了具体的网络传输协议；比如说TCP/IP传输等，并进行了抽象，使上层对传输协议解耦。
2. 协议层(Protocol Layer)：协议层定义了数据传输格式，负责网络传输数据的序列化和反序列化；比如说JSON、XML、二进制数据等。

3. 处理层(Processor Layer)：处理层是由具体的IDL（接口描述语言）生成的，封装了具体的底层网络传输和序列化方式，并委托给用户实现的Handler进行处理。
4. 服务层(Server Layer)：整合上述组件，提供具体的网络线程/IO服务模型，形成最终的服务。选用TThreadSelectorServer采用的就是多线程 reactor 模型的 server。

可以看到thrift可以解决前面提到的问题 1 和 3，但是对于服务间寻址(如用 tcp)还是需要给出 host 和 port，对于错误处理对于微服务来说，也完全不够。

因此，kite 框架是，基于 thrift，封装一个 kite 框架，以实现 2 和 4 的功能。 为了实现服务注册以及服务发现的功能，Kite之间可以互相通信，通过Kontrol的服务发现机制，一个Kite可以发现其它的Kites。也就是说一个Kite可以在Kontrol注册自己，从而让其它的kites能找到它； Kontrol本身也是一个Kite，它用于对服务进行注册和鉴权，使用 etcd 作为存储。服务熔断，也可以在这里实现

redis

基础见 redis笔记

分布式及架构，见redis服务端

消息队列

- nsq nsq
- kafka 暂无

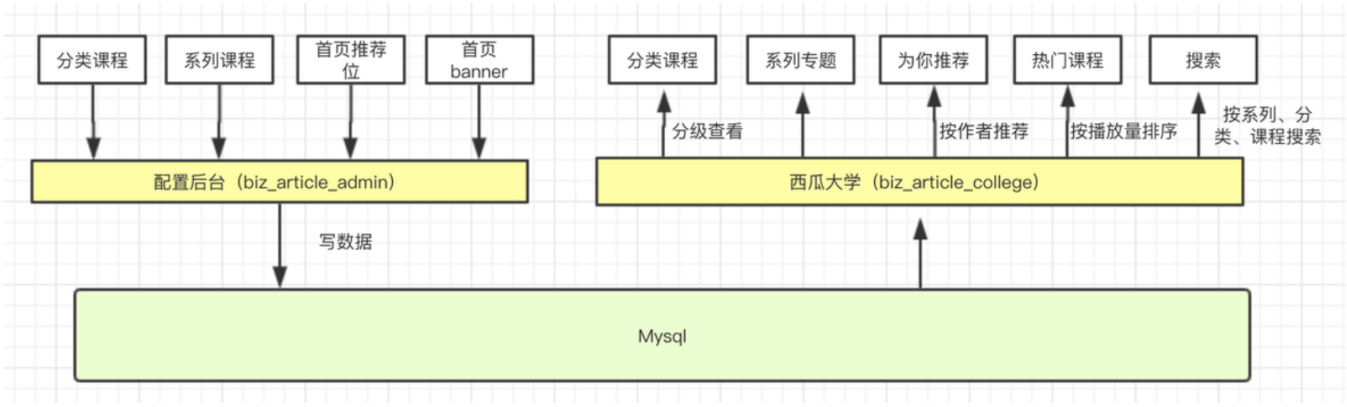
缓存方式

缓存使用 缓存更新 localcache小计

实习项目总结

西瓜大学主要是对标B站有对新人up 主的培养，西瓜大学需要相应的对新人培养的项目，这对吸引新人 up 主以及对品牌的塑造有着很好的作用。

项目结构



项目流程如图，分为两个部分，admin 负责课程配置，讲更改写入 mysql，college 负责跟前端交互，读取相应的配置

数据表结构设计

分类表：

```
CREATE TABLE `admin_college_course_category` (
`id` int(10) NOT NULL AUTO_INCREMENT COMMENT '主键，分类Id',
`name` varchar(30) NOT NULL DEFAULT '' COMMENT '分类名称',
`parent_id` int(10) NOT NULL DEFAULT '0' COMMENT '父分类id',
`level` int(10) NOT NULL DEFAULT '0' COMMENT '分类层级',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='西瓜大学课程分类表';
```

课程表：

```
CREATE TABLE `admin_college_course` (
`id` int(10) NOT NULL AUTO_INCREMENT COMMENT '课程 id',
`group_id` bigint(20) NOT NULL COMMENT '视频gid',
`first_category_id` int(4) NOT NULL DEFAULT '0' COMMENT '第一分类id',
`first_category_name` varchar(30) NOT NULL DEFAULT '' COMMENT '第一分类名称',
`second_category_id` int(4) NOT NULL DEFAULT '0' COMMENT '第二分类 id',
`second_category_name` varchar(30) NOT NULL DEFAULT '' COMMENT '第二分类名称',
`difficulty_type` int(4) NOT NULL DEFAULT '0' COMMENT '难度0: 1: 2: ',
`title` varchar(200) NOT NULL DEFAULT '' COMMENT '标题',
`rank` int(10) NOT NULL DEFAULT '0' COMMENT '排名',
`status` int(4) NOT NULL DEFAULT '0' COMMENT '状态1: 2: ',
PRIMARY KEY (`id`),
UNIQUE KEY `group_id` (`group_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT=' ';
```

接口设计

1. 保存分类课程

包括新增，修改，单个、批量，都走这个

```
struct SaveCourseReq {
1: required list<Course> Courses; //

255: optional base.Base Base
}

struct SaveCourseResp {
1: required list<Course> SaveFailCourses // Course
255: optional base.BaseResp BaseResp,
}

//
SaveCourseResp SaveCourse(1:SaveCourseReq request)
```

2. 获取分类课程

```
21 struct GetCourseReq {
22 1: optional string KeyWord // group_id
23 2: optional i32 FirstCategoryId // Id
24 3: optional i32 SecondCategoryId // Id
25 4: optional i32 Offset = 0 //
26 5: optional i32 Limit = 20 // , 20
27
28 255: optional base.Base Base
29 }
30
31 struct Course {
32 1: required i32 Id // Id
33 2: required i64 GroupId // GroupId
34 3: optional string Title //
35 4: optional string FrontCover //
36 5: required i32 FirstCategoryId // Id
37 6: required string FirstCategoryName //
38 7: required i32 SecondCategoryId // Id
39 8: required string SecondCategoryName //
40 9: required i32 DifficultyType //
41 10: optional string CreateTime //
42 11: required string StartTime //
43 12: required i32 Status // 1: 2:
44 13: required i32 Rank //
45 14: required string Operator //
46 15: optional string PublishTime //
47 16: optional string CreateUser //
48 17: optional string UpdateTime //
49 }
50
51 struct GetCourseResp {
52 1: required list<Course> Courses
53 2: optional bool HasMore = false
54 3: optional i32 TotalNum
55
56 255: optional base.BaseResp BaseResp,
57 }
58
59 //
60 GetCourseResp GetCourse(1:GetCourseReq request)
```

由于课程较多，后台接口不可能，也不合适，一次性返回所有数据，而合理的做法和最佳实践是：分页 分页一般有两种方式：

- 页数表示法：
    - cur_page + per_page
- 偏移量表示法

    - offset + limit

返回时，需要标志位标记是否还有数据，以及数据的总数

数据一致性

由于之前设计时，每个 course 都带了分类的名字，但是又是可能需要对分类名称进行更改，因此需要更新一下回复课程的名字。 当时我们考虑了两种方案：

- 两表连接查询 这个方案有几个问题，一个是公司不鼓励使用 join，主要是由于 join 的优化可能有性能上的问题，且可能会同时锁住多个表
- 读取两次，在 service 层做处理 需要读取两个表，网络 io 次数更多。但由于分类名称实际上很少更改，且短时的不同影响较小，我们采用 localcache 进行缓存。关于缓存的选取，见缓存选取