# 10年技术开发技术整理

### 数据结构与算法篇

### Linux篇

### 网路篇

### c++篇

### MySQL redis篇

### shell python

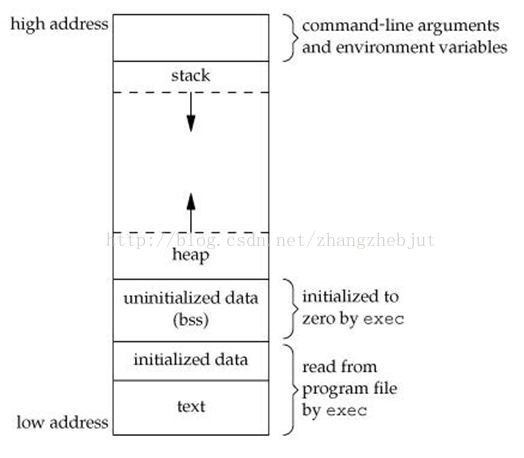
### 大数据&分布式

### 服务器设计

## 数据结构与算法篇

## linux系统篇

#### linux下进程模型



text 代码段 程序二进制代码在内存中的映射

initd data在程序运行初已经对变量进行初始化的数据

uninit data (bss) 在程序运行初未对变量进行初始化的数据

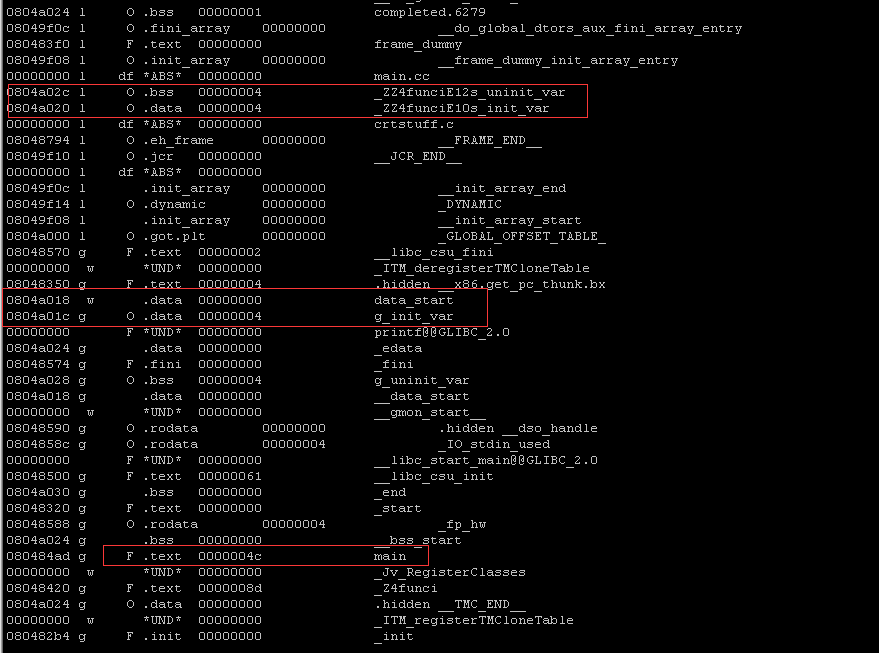
heap 动态分配的内存 new / malloc

stack 1局部变量

2 函数调用时存储函数的返回指针



objdump -x



#### 进程与线程

内核线程、轻量级进程、用户线程

**内核线程**

内核线程就是内核的分身，一个分身可以处理一件特定事情。这在处理异步事件如异步IO时特别有用。内核线程的使用是廉价的，唯一使用的资源就是内核栈和上下文切换时保存寄存器的空间。支持多线程的内核叫做多线程内核(Multi-Threads kernel )。

**轻量级进程**

轻量级线程(LWP)是一种由内核支持的用户线程。它是基于内核线程的高级抽象，因此只有先支持内核线程，才能有LWP。每一个进程有一个或多个LWPs，每个LWP由一个内核线程支持。这种模型实际上就是恐龙书上所提到的一对一线程模型。在这种实现的操作系统中，LWP就是用户线程。

由于每个LWP都与一个特定的内核线程关联，因此每个LWP都是一个独立的线程调度单元。即使有一个LWP在系统调用中阻塞，也不会影响整个进程的执行。

轻量级进程具有局限性。首先，大多数LWP的操作，如建立、析构以及同步，都需要进行系统调用。系统调用的代价相对较高：需要在user mode和kernel mode中切换。其次，每个LWP都需要有一个内核线程支持，因此LWP要消耗内核资源（内核线程的栈空间）。因此一个系统不能支持大量的LWP

**用户线程**

LWP虽然本质上属于用户线程，但LWP线程库是建立在内核之上的，LWP的许多操作都要进行系统调用，因此效率不高。而这里的用户线程指的是完全建立在用户空间的线程库，用户线程的建立，同步，销毁，调度完全在用户空间完成，不需要内核的帮助。因此这种线程的操作是极其快速的且低消耗的。

**POSIX线程实现基于如下要求:**  
1, 查看进程列表的时候, 相关的一组task\_struct应当被展现为列表中的一个节点;  
2, 发送给这个"进程"的信号(对应kill系统调用), 将被对应的这一组task\_struct所共享, 并且被其中的任意一个"线程"处理;  
3, 发送给某个"线程"的信号(对应pthread\_kill), 将只被对应的一个task\_struct接收, 并且由它自己来处理;  
4, 当"进程"被停止或继续时(对应SIGSTOP/SIGCONT信号), 对应的这一组task\_struct状态将改变;  
5, 当"进程"收到一个致命信号(比如由于段错误收到SIGSEGV信号), 对应的这一组task\_struct将全部退出;

#### 进程与线程区别

**联系**：

**1 一个进程可以包括多个线程, 一个线程只能属于一个进程**

**2 一个进程可以生成多个线程,而这些线程之间共享地址空间和相应的资源, 在线程切换时, 并没有太多的开销**

**相同点：**

**进程与线程都可以被操作系统独立调度。**

**区别：**

**进程之间是不共享地址空间的。**

**线程是共享着进程的地址空间，线程崩溃会导致整个进程崩溃。**

**fork() calls clone(least sharing) and pthread\_create() calls clone(most sharing)**

#### Linux 进程与线程调度与优先级

**概念：**

1 进程优先级值越小，优先级越高

2 进程优先级是PRI不是NICE值，但NICE值会影响优先级优**(PRI范围0~MAX\_RT\_PRIO-1,**

**MAX\_RT\_PRIO 100 NICE 范围 -20~19)**

3 三类调度策略：RR 调度和FIFO调度，调度实时进程，分时调度非实时进程

#define SCHED\_NORMAL 0 //非实时进程，基于优先级的轮回法（RoundRobin）

#define SCHED\_FIFO 1 //实时进程，先进先出

#define SCHED\_RR 2 //实时进程，基于优先级的轮回法（Round Robin）

4 实时进程优先级高于普通进程，实时调度使用静态优先级调度，非实时进程使用动态优先 级调度，非实时进程动态优先级通过NICE调整

**查看设置方法：**

**ps top chrt**

**进程设置sched\_setscheduler**

**线程设置pthread\_attr\_setschedpolicy**

nice –n adjustment command option，如果这里不指定adjustment，则默认为10。

实时进程被一个进程替换，可能的情况有：

1. 进程被拥有更高优先级的进程抢占。
2. 进程发生阻塞进入睡眠状态。
3. 进程被终止（状态为TASK\_STOPPED OR TASK\_TRACED）或者被杀死（EXIT\_DEAD OR EXIT\_ZOMBIE）。
4. 进程通过调用sched\_yield()自愿放弃处理器。
5. 进程是轮回实时（SCHED\_RR）且其时间片执行完毕。

**实时进程与非实时进程**

| **类型** | **描述** | **示例** |
| --- | --- | --- |
| 交互式进程(interactive process) | 此类进程经常与用户进行交互, 因此需要花费很多时间等待键盘和鼠标操作. 当接受了用户的输入后, 进程必须很快被唤醒, 否则用户会感觉系统反应迟钝 | shell, 文本编辑程序和图形应用程序 |
| 批处理进程(batch process) | 此类进程不必与用户交互, 因此经常在后台运行. 因为这样的进程不必很快相应, 因此常受到调度程序的怠慢 | 程序语言的编译程序, 数据库搜索引擎以及科学计算 |
| 实时进程(real-time process) | 这些进程由很强的调度需要, 这样的进程绝不会被低优先级的进程阻塞. 并且他们的响应时间要尽可能的短 | 视频音频应用程序, 机器人控制程序以及从物理传感器上收集数据的程序 |

#### 共享内存实现原理

1、多线程。线程之间的内存都是共享的。更确切的说，属于同一进程的线程使用的是同一个地址空间，而不是在不同地址空间之间进行内存共享；  
2、父子进程间的内存共享。父进程以MAP\_SHARED|MAP\_ANONYMOUS选项mmap一块匿名内存，fork之后，其子孙进程之间就能共享这块内存。这种共享内存由于受到进程父子关系的限制，一般较少使用；  
3、mmap文件。多个进程mmap到同一个文件，实际上就是大家在共享文件page cache中的内存。不过文件牵涉到磁盘的读写，用来做共享内存显然十分笨重，所以就有了不跟磁盘扯上关系的内存文件，也就是我们这里要讨论的tmpfs和shmem；

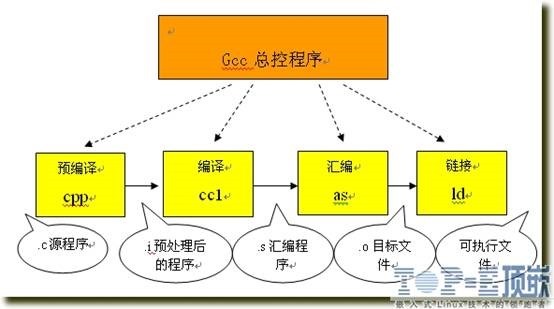
tmpfs是一套虚拟的文件系统，在其中创建的文件都是基于内存的，机器重启即消失。  
shmem是一套ipc，通过相应的ipc系统调用shmget能够以指定key创建一块的共享内存。需要使用这块内存的进程可以通过shmat系统调用来获得它。  
虽然是两套不同的接口，但是在内核里面的实现却是同一套。shmem内部挂载了一个tmpfs分区（用户不可见），shmget就是在该分区下获取名为"SYSV${key}"的文件。然后shmat就相当于mmap这个文件。

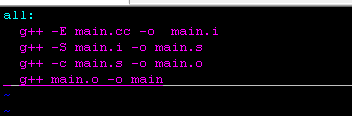
**IPC共享内存和文件映射的区别**  
        1. 文件映射的页框是磁盘文件高速缓存中的页框，内核线程pdflush会将页框中的内容回写进磁盘， 如果是私有映射类型，将会进行写时复制。而IPC共享内存映射的是一种特殊文件系统中的文件高速缓存，它没有相应的磁盘映像。  
        2. IPC共享内存只存在于内存中，系统重新启动，数据将会丢失。而文件共享映射会将数据写回磁盘。  
        3. IPC共享内存的大小是在创建的时候指定，而且大小不能改变，而文件在创建时大小为0，此时还不能建立映射，文件的大小会间接的决定映射区的大小。例如文件的大小是123，而要求映射的区域大小是4096\*2，但实际只会分配4096的映射空间，此时引用4096以后的线性空间将引起缺页异常。  
        4. 当第一次读取共享内存时IPC共享内存对象将分配一个新的页框，而文件映射分配新页框的同时会将磁盘中的数据写入新页框。  
        5. IPC共享内存不需要写回磁盘操作，完全是为共享内存而设计，所以使用效率会更高。  
        6. IPC共享内存对象必须调用shmctl()显示的撤销，否则会一直保留着，使用key或者id号定位一个共享内存对象，key和id号的对应关系并不是固定的。例如，第一次使用key建立一个共享内存对象为shm1对应的id为id1，之后系统重新启动，然后再使用key建立一个共享内存对象shm2，对应的id是id2，此时id2和id1是不同的。而文件映射使用相同的路径将会定位相同的磁盘文件。  
        总结：IPC共享内存和文件映射的实现机制是一样的，文件映射的目的是加快对文件的读写速度，而IPC共享内存就是为了共享内存而设计的，所以效率会高一些。

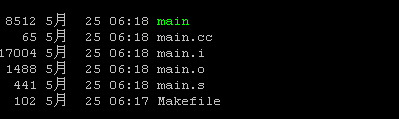
**shm族与mmap共享内存联系**

都是通过文件映射内存实现共享

#### c++程序编译过程







##### 编译过程

编译过程又可以分成两个阶段：编译和汇编。

##### 编译

编译是读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码，源文件的编译过程包含两个主要阶段：

###### 编译预处理

读取c源程序，对其中的伪指令（以# 开头的指令）和特殊符号进行处理。

伪指令主要包括以下四个方面：

1) 宏定义指令，如# define Name TokenString，# undef等。

对于前一个伪指令，预编译所要做的是将程序中的所有Name用TokenString替换，但作为字符串常量的 Name则不被替换。对于后者，则将取消对某个宏的定义，使以后该串的出现不再被替换。

2) 条件编译指令，如# ifdef，# ifndef，# else，# elif，# endif等。

这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉。

3) 头文件包含指令，如# include "FileName" 或者# include < FileName> 等。

在头文件中一般用伪指令# define定义了大量的宏（最常见的是字符常量），同时包含有各种外部符号的声明。

采用头文件的目的主要是为了使某些定义可以供多个不同的C源程序使用。因为在需要用到这些定义的C源程序中，只需加上一条# include语句即可，而不必再在此文件中将这些定义重复一遍。预编译程序将把头文件中的定义统统都加入到它所产生的输出文件中，以供编译程序对之进行处理。

包含到c源程序中的头文件可以是系统提供的，这些头文件一般被放在/ usr/ include目录下。在程序中# include它们要使用尖括号（< >）。另外开发人员也可以定义自己的头文件，这些文件一般与c源程序放在同一目录下，此时在# include中要用双引号（""）。

4) 特殊符号，预编译程序可以识别一些特殊的符号。

例如在源程序中出现的LINE标识将被解释为当前行号（十进制数），FILE则被解释为当前被编译的C源程序的名称。预编译程序对于在源程序中出现的这些串将用合适的值进行替换。

预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。这个文件的含义同没有经过预处理的源文件是相同的，但内容有所不同。下一步，此输出文件将作为编译程序的输入而被翻译成为机器指令。

###### 编译、优化阶段

经过预编译得到的输出文件中，只有常量；如数字、字符串、变量的定义，以及C语言的关键字，如main, if , else , for , while , { , } , + , - , \* , \ 等等。

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

优化处理是编译系统中一项比较艰深的技术。它涉及到的问题不仅同编译技术本身有关，而且同机器的硬件环境也有很大的关系。优化一部分是对中间代码的优化。这种优化不依赖于具体的计算机。另一种优化则主要针对目标代码的生成而进行的。

对于前一种优化，主要的工作是删除公共表达式、循环优化（代码外提、强度削弱、变换循环控制条件、已知量的合并等）、复写传播，以及无用赋值的删除，等等。

后一种类型的优化同机器的硬件结构密切相关，最主要的是考虑是如何充分利用机器的各个硬件寄存器存放有关变量的值，以减少对于内存的访问次数。另外，如何根据机器硬件执行指令的特点（如流水线、RISC、CISC、VLIW等）而对指令进行一些调整使目标代码比较短，执行的效率比较高，也是一个重要的研究课题。

经过优化得到的汇编代码必须经过汇编程序的汇编转换成相应的机器指令，方可能被机器执行。

###### 汇编

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。

目标文件由段组成。通常一个目标文件中至少有两个段：

1) 代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

2) 数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

UNIX环境下主要有三种类型的目标文件：

1) 可重定位文件

其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。

2) 共享的目标文件

这种文件存放了适合于在两种上下文里链接的代码和数据。

第一种是链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个目标文件；

第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。

3) 可执行文件

它包含了一个可以被操作系统创建一个进程来执行之的文件。

汇编程序生成的实际上是第一种类型的目标文件。对于后两种还需要其他的一些处理方能得到，这个就是链接程序的工作了。

##### 链接过程

由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。

例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

根据开发人员指定的同库函数的链接方式的不同，链接处理可分为两种：

1) 静态链接

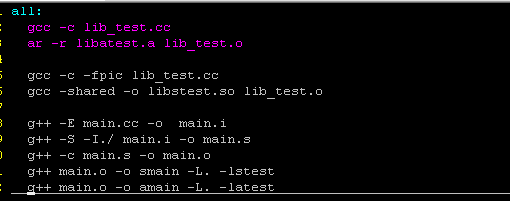
在这种链接方式下，函数的代码将从其所在的静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。

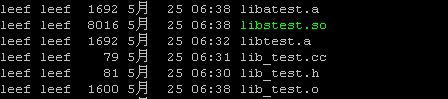
2) 动态链接

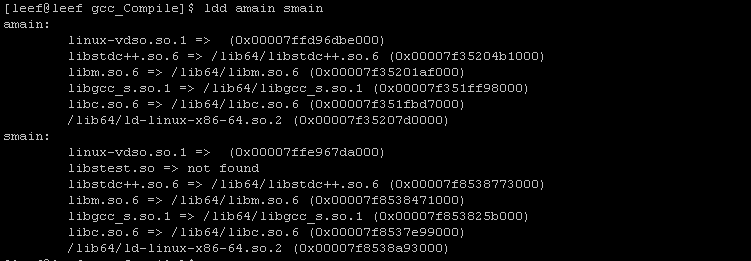
在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小，并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害。

#### 静态库 动态库

****

****

****

#### linxu 系统常用工具

##### pstack

**跟踪进程栈**

##### strace

**跟踪系统调用**

##### ipcs

查看共享内存

##### 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**
  + 09:14:56 ： 系统当前时间
  + 264 days, 20:56 ： 系统开机到现在经过了多少时间
  + 1 users ： 当前2用户在线
  + load average: 0.02, 0.04, 0.00： 系统1分钟、5分钟、15分钟的CPU负载信息
* **第二行**
  + Tasks：任务;
  + 87 total：很好理解，就是当前有87个任务，也就是87个进程。
  + 1 running：1个进程正在运行
  + 86 sleeping：86个进程睡眠
  + 0 stopped：停止的进程数
  + 0 zombie：僵死的进程数
* **第三行**
  + Cpu(s)：表示这一行显示CPU总体信息
  + 0.0%us：用户态进程占用CPU时间百分比，不包含renice值为负的任务占用的CPU的时间。
  + 0.7%sy：内核占用CPU时间百分比
  + 0.0%ni：改变过优先级的进程占用CPU的百分比
  + 99.3%id：空闲CPU时间百分比
  + 0.0%wa：等待I/O的CPU时间百分比
  + 0.0%hi：CPU硬中断时间百分比
  + 0.0%si：CPU软中断时间百分比
  + 注：这里显示数据是所有cpu的平均值，如果想看每一个cpu的处理情况，按1即可；折叠，再次按1；
* **第四行**
  + Men：内存的意思
  + 8175320kk total：物理内存总量
  + 8058868k used：使用的物理内存量
  + 116452k free：空闲的物理内存量
  + 283084k buffers：用作内核缓存的物理内存量
* **第五行**
  + Swap：交换空间
  + 6881272k total：交换区总量
  + 4010444k used：使用的交换区量
  + 2870828k free：空闲的交换区量
  + 4336992k cached：缓冲交换区总量
* **进程信息**
  + 再下面就是进程信息：
  + PID：进程的ID
  + USER：进程所有者
  + PR：进程的优先级别，越小越优先被执行
  + NInice： 优先值
  + VIRT：进程占用的虚拟内存
  + RES：进程占用的物理内存
  + SHR：进程使用的共享内存
  + S：进程的状态。S表示休眠，R表示正在运行，Z表示僵死状态，N表示该进程优先值为负数
  + %CPU：进程占用CPU的使用率
  + %MEM：进程使用的物理内存和总内存的百分比
  + TIME+：该进程启动后占用的总的CPU时间，即占用CPU使用时间的累加值。
  + COMMAND：进程启动命令名称

**htop**

安装：

yum -y install epel-release

yum -y update

yum -y install htop

如果报错： 将/etc/yum.repos.d/epel.repo中的mirrorlist改为baseurl

##### free

total used free shared buffers cached

Mem: 8175320 6159248 2016072 0 310208 5243680

-/+ buffers/cache: 605360 7569960

Swap: 6881272 16196 6865076

**total**：表示物理内存总量(total = used + free)  
**used**：表示总计分配给缓存（包含buffers 与cache ）使用的数量，但其中可能部分缓存并未实际使用。  
**free**：未被分配的内存。  
**shared**：共享内存，一般系统不会用到，这里也不讨论。  
**buffers**：系统分配但未被使用的buffers 数量。  
**cached**：系统分配但未被使用的cache 数量。

：  
第三行所指的是从应用程序角度来看，  
对于应用程序来说，buffers/cached 是等于可用的，  
因为buffer/cached是为了提高文件读取的性能，  
当应用程序需在用到内存的时候，buffer/cached会很快地被回收。  
所以从应用程序的角度来说，可用内存=系统free memory+buffers+cached.

第二行(mem)的used/free与第三行(-/+ buffers/cache) used/free的区别。  
这两个的区别在于使用的角度来看，  
第一行是从OS的角度来看，因为对于OS，buffers/cached 都是属于被使用，所以他的可用内存是16936KB,已用内存是238332KB,其中包括，内核（OS）使用+Application(X,oracle,etc)使用的+buffers+cached.

所以从应用程序的角度来说，可用内存=系统free memory+buffers+cached

**释放cache**

echo 3**>/**proc**/**sys**/**vm**/**drop\_caches

##### vmstat

/root$vmstat 5 5

procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----

r b swpd free buff cache si so bi bo in cs us sy id wa st

6 0 0 27900472 204216 28188356 0 0 0 9 1 2 11 14 75 0 0

9 0 0 27900380 204228 28188360 0 0 0 13 33312 126221 22 20 58 0 0

2 0 0 27900340 204240 28188364

**Procs（进程）:**

* r: 运行队列中进程数量
* b: 等待IO的进程数量

**Memory（内存）:**

* swpd: 使用虚拟内存大小
* free: 可用内存大小
* buff: 用作缓冲的内存大小
* cache: 用作缓存的内存大小

**Swap:**

* si: 每秒从交换区写到内存的大小
* so: 每秒写入交换区的内存大小

**IO：（现在的Linux版本块的大小为1024bytes）**

* bi: 每秒读取的块数
* bo: 每秒写入的块数

**system：**

* in: 每秒中断数，包括时钟中断
* cs: 每秒上下文切换数

**CPU（以百分比表示）**

* us: 用户进程执行时间(user time)
* sy: 系统进程执行时间(system time)
* id: 空闲时间(包括IO等待时间)
* wa: 等待IO时间

##### iostat

通过iostat方便查看CPU、网卡、tty设备、磁盘、CD-ROM 等等设备的活动情况, 负载信息

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

**cpu属性值说明：**

* %user：CPU处在用户模式下的时间百分比。
* %nice：CPU处在带NICE值的用户模式下的时间百分比。
* %system：CPU处在系统模式下的时间百分比。
* %iowait：CPU等待输入输出完成时间的百分比。
* %steal：管理程序维护另一个虚拟处理器时，虚拟CPU的无意识等待时间百分比。
* %idle：CPU空闲时间百分比。

注：如果%iowait的值过高，表示硬盘存在I/O瓶颈，%idle值高，表示CPU较空闲，如果%idle值高但系统响应慢时，有可能是CPU等待分配内存，此时应加大内存容量。%idle值如果持续低于10，那么系统的CPU处理能力相对较低，表明系统中最需要解决的资源是CPU。

**disk属性值说明：**

* rrqm/s: 每秒进行 merge 的读操作数目。即 rmerge/s
* wrqm/s: 每秒进行 merge 的写操作数目。即 wmerge/s
* r/s: 每秒完成的读 I/O 设备次数。即 rio/s
* w/s: 每秒完成的写 I/O 设备次数。即 wio/s
* rsec/s: 每秒读扇区数。即 rsect/s
* wsec/s: 每秒写扇区数。即 wsect/s
* rkB/s: 每秒读K字节数。是 rsect/s 的一半，因为每扇区大小为512字节。
* wkB/s: 每秒写K字节数。是 wsect/s 的一半。
* avgrq-sz: 平均每次设备I/O操作的数据大小 (扇区)。
* avgqu-sz: 平均I/O队列长度。
* await: 平均每次设备I/O操作的等待时间 (毫秒)。
* svctm: 平均每次设备I/O操作的服务时间 (毫秒)。
* %util: 一秒中有百分之多少的时间用于 I/O 操作，即被io消耗的cpu百分比

**查看TPS和吞吐量**

/root$iostat -d -k 1 1

Linux 2.6.32-279.el6.x86\_64 (colin) 07/16/2014 \_x86\_64\_ (4 CPU)

Device: tps kB\_read/s kB\_wrtn/s kB\_read kB\_wrtn

sda 1.95 0.74 35.44 4572712 218559410

dm-0 3.08 0.28 12.17 1696513 75045968

dm-1 5.83 0.46 23.25 2857265 143368744

dm-2 0.01 0.00 0.02 11965 144644

* tps：该设备每秒的传输次数（Indicate the number of transfers per second that were issued to the device.）。“一次传输”意思是“一次I/O请求”。多个逻辑请求可能会被合并为“一次I/O请求”。“一次传输”请求的大小是未知的。
* kB\_read/s：每秒从设备（drive expressed）读取的数据量；
* kB\_wrtn/s：每秒向设备（drive expressed）写入的数据量；
* kB\_read：读取的总数据量；kB\_wrtn：写入的总数量数据量；

**查看设备使用率（%util）和响应时间（await）**

/root$iostat -d -x -k 1 1

Linux 2.6.32-279.el6.x86\_64 (colin) 07/16/2014 \_x86\_64\_ (4 CPU)

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await svctm %util

sda 0.02 7.25 0.04 1.90 0.74 35.47 37.15 0.04 19.13 5.58 1.09

dm-0 0.00 0.00 0.04 3.05 0.28 12.18 8.07 0.65 209.01 1.11 0.34

dm-1 0.00 0.00 0.02 5.82 0.46 23.26 8.13 0.43 74.33 1.30 0.76

dm-2 0.00 0.00 0.00 0.01 0.00 0.02 8.00 0.00 5.41 3.28 0.00

* rrqm/s： 每秒进行 merge 的读操作数目.即 delta(rmerge)/s
* wrqm/s： 每秒进行 merge 的写操作数目.即 delta(wmerge)/s
* r/s： 每秒完成的读 I/O 设备次数.即 delta(rio)/s
* w/s： 每秒完成的写 I/O 设备次数.即 delta(wio)/s
* rsec/s： 每秒读扇区数.即 delta(rsect)/s
* wsec/s： 每秒写扇区数.即 delta(wsect)/s
* rkB/s： 每秒读K字节数.是 rsect/s 的一半,因为每扇区大小为512字节.(需要计算)
* wkB/s： 每秒写K字节数.是 wsect/s 的一半.(需要计算)
* avgrq-sz：平均每次设备I/O操作的数据大小 (扇区).delta(rsect+wsect)/delta(rio+wio)
* avgqu-sz：平均I/O队列长度.即 delta(aveq)/s/1000 (因为aveq的单位为毫秒).
* await： 平均每次设备I/O操作的等待时间 (毫秒).即 delta(ruse+wuse)/delta(rio+wio)
* svctm： 平均每次设备I/O操作的服务时间 (毫秒).即 delta(use)/delta(rio+wio)
* %util： 一秒中有百分之多少的时间用于 I/O 操作,或者说一秒中有多少时间 I/O 队列是非空的，即 delta(use)/s/1000 (因为use的单位为毫秒)
* 如果 %util 接近 100%，说明产生的I/O请求太多，I/O系统已经满负荷，该磁盘可能存在瓶颈。 idle小于70% IO压力就较大了，一般读取速度有较多的wait。 同时可以结合vmstat 查看查看b参数(等待资源的进程数)和wa参数(IO等待所占用的CPU时间的百分比，高过30%时IO压力高)。
* 另外 await 的参数也要多和 svctm 来参考。差的过高就一定有 IO 的问题。
* avgqu-sz 也是个做 IO 调优时需要注意的地方，这个就是直接每次操作的数据的大小，如果次数多，但数据拿的小的话，其实 IO 也会很小。如果数据拿的大，才IO 的数据会高。也可以通过 avgqu-sz × ( r/s or w/s ) = rsec/s or wsec/s。也就是讲，读定速度是这个来决定的。
* svctm 一般要小于 await (因为同时等待的请求的等待时间被重复计算了)，svctm 的大小一般和磁盘性能有关，CPU/内存的负荷也会对其有影响，请求过多也会间接导致 svctm 的增加。await 的大小一般取决于服务时间(svctm) 以及 I/O 队列的长度和 I/O 请求的发出模式。如果 svctm 比较接近 await，说明 I/O 几乎没有等待时间；如果 await 远大于 svctm，说明 I/O 队列太长，应用得到的响应时间变慢，如果响应时间超过了用户可以容许的范围，这时可以考虑更换更快的磁盘，调整内核 elevator 算法，优化应用，或者升级 CPU。
* 队列长度(avgqu-sz)也可作为衡量系统 I/O 负荷的指标，但由于 avgqu-sz 是按照单位时间的平均值，所以不能反映瞬间的 I/O 洪水

**形象的比喻：**

* r/s+w/s 类似于交款人的总数
* 平均队列长度(avgqu-sz)类似于单位时间里平均排队人的个数
* 平均服务时间(svctm)类似于收银员的收款速度
* 平均等待时间(await)类似于平均每人的等待时间
* 平均I/O数据(avgrq-sz)类似于平均每人所买的东西多少
* I/O 操作率 (%util)类似于收款台前有人排队的时间比例

设备IO操作:总IO(io)/s = r/s(读) +w/s(写)

平均等待时间=单个I/O服务器时间\*(1+2+...+请求总数-1)/请求总数

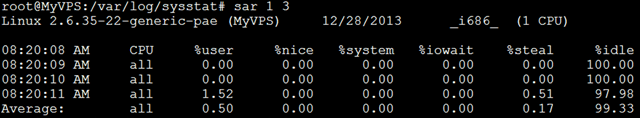
每秒发出的I/0请求很多,但是平均队列就4,表示这些请求比较均匀,大部分处理还是比较及时。

##### sar

**sar 找出系统瓶颈的利器**

查看CPU使用率

sar -u : 默认情况下显示的cpu使用率等信息就是sar -u；



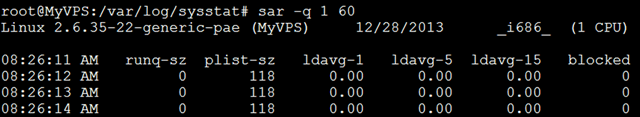
可以看到这台机器使用了虚拟化技术，有相应的时间消耗； 各列的指标分别是:

* %user 用户模式下消耗的CPU时间的比例；
* %nice 通过nice改变了进程调度优先级的进程，在用户模式下消耗的CPU时间的比例
* %system 系统模式下消耗的CPU时间的比例；
* %iowait CPU等待磁盘I/O导致空闲状态消耗的时间比例；
* %steal 利用Xen等操作系统虚拟化技术，等待其它虚拟CPU计算占用的时间比例；
* %idle CPU空闲时间比例；

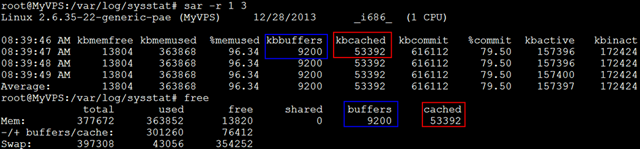
sar -q: 查看平均负载

指定-q后，就能查看运行队列中的进程数、系统上的进程大小、平均负载等；与其它命令相比，它能查看各项指标随时间变化的情况；

* runq-sz：运行队列的长度（等待运行的进程数）
* plist-sz：进程列表中进程（processes）和线程（threads）的数量
* ldavg-1：最后1分钟的系统平均负载 ldavg-5：过去5分钟的系统平均负载
* ldavg-15：过去15分钟的系统平均负载



sar -r： 指定-r之后，可查看物理内存使用状况；



* kbmemfree：这个值和free命令中的free值基本一致,所以它不包括buffer和cache的空间.
* kbmemused：这个值和free命令中的used值基本一致,所以它包括buffer和cache的空间.
* %memused：物理内存使用率，这个值是kbmemused和内存总量(不包括swap)的一个百分比.
* kbbuffers和kbcached：这两个值就是free命令中的buffer和cache.
* kbcommit：保证当前系统所需要的内存,即为了确保不溢出而需要的内存(RAM+swap).
* %commit：这个值是kbcommit与内存总量(包括swap)的一个百分比.

##### readelf elf文件格式分析

##### objdump 二进制文件分析

##### nm 目标文件格式分析

##### size 查看程序内存映像大小

##### wget scp sync

cp local\_file remote\_username@remote\_ip:remote\_folder

##### crontab

分 时 日 月 星期 要运行的命令

* 第1列分钟0～59
* 第2列小时0～23（0表示子夜）
* 第3列日1～31
* 第4列月1～12
* 第5列星期0～7（0和7表示星期天）
* 第6列要运行的命令

实例1：每1分钟执行一次myCommand

\* \* \* \* \* myCommand

实例2：每小时的第3和第15分钟执行

3,15 \* \* \* \* myCommand

实例3：在上午8点到11点的第3和第15分钟执行

3,15 8-11 \* \* \* myCommand

实例4：每隔两天的上午8点到11点的第3和第15分钟执行

3,15 8-11 \*/2 \* \* myCommand

实例5：每周一上午8点到11点的第3和第15分钟执行

3,15 8-11 \* \* 1 myCommand

实例6：每晚的21:30重启smb

30 21 \* \* \* /etc/init.d/smb restart

实例7：每月1、10、22日的4 : 45重启smb

45 4 1,10,22 \* \* /etc/init.d/smb restart

实例8：每周六、周日的1 : 10重启smb

10 1 \* \* 6,0 /etc/init.d/smb restart

实例9：每天18 : 00至23 : 00之间每隔30分钟重启smb

0,30 18-23 \* \* \* /etc/init.d/smb restart

实例10：每星期六的晚上11 : 00 pm重启smb

0 23 \* \* 6 /etc/init.d/smb restart

实例11：每一小时重启smb

\* \*/1 \* \* \* /etc/init.d/smb restart

实例12：晚上11点到早上7点之间，每隔一小时重启smb

0 23-7 \* \* \* /etc/init.d/smb restart

#### MYSQl 安装

yum install wget

wget http://repo.mysql.com/mysql-community-release-el7-5.noarch.rpm

rpm -ivh mysql-community-release-el7-5.noarch.rpm

ls -1 /etc/yum.repos.d/mysql-community\*

yum install mysql-server

启动 systemctl start mysqld

停止 systemctl stop mysqld

重启 systemctl restart mysqld

状态 systemctl status mysqld

配置文件 /etc/my.cnf

1 创建并配置 datadir=/data/mysql

2 init **mysql\_install\_db**

启动失败分析

1 目录权限改为 mysql

2  rm ibdata1  ib\_logfile0  ib\_logfile1

默认的存储引擎，my.ini中修改成MyISAM

**default**-storage-engine=**MyISAM**

然后尝试启动，如果启动OK，命令行连上mysql，然后安装innodb引擎

**INSTALL** **PLUGIN** **InnoDB** **SONAME** 'ha\_innodb.so';

然后在my.ini修改回innodb

修改 root密码

1) mysql -u root -p

2) update user set password=PASSWORD("GIVE-NEW-ROOT-PASSWORD") where User='root';

3) flush privileges

查看默认编码

show variables like '%colla%';

show variables like '%char%';

#### 内存泄漏检查工具

##### 如何发现内存泄漏

top ps

##### 静态分析

包括手动检测和静态工具分析，这是代价最小的调试方法。

代码静态扫描和分析的工具比较多，比如 splint, cppcheck 等。

##### 内嵌程序

可以重载内存分配和释放函数 new 和 delete,然后编写程序定期统计内存的分配和释放，从中找出可能的内存泄漏。或者调用系统函数定期监视程序堆的大小，关键要确定堆的增长是泄漏而不是合理的内存使用。这类方法比较复杂，在这就不给出详细例子了。

##### 动态运行检测

**Valgrind**

valgrind 是帮助程序员寻找程序里的 bug 和改进程序性能的工具。程序通过 valgrind 运行时，valgrind 收集各种有用的信息，通过这些信息可以找到程序中潜在的 bug 和性能瓶颈。

Valgrind 现在提供多个工具，其中最重要的是 Memcheck，Cachegrind，Massif 和 Callgrind。Valgrind 是在 Linux 系统下开发应用程序时用于调试内存问题的工具。它尤其擅长发现内存管理的问题，它可以检查程序运行时的内存泄漏问题。其中的 memecheck 工具可以用来寻找 c、c++ 程序中内存管理的错误。可以检查出下列几种内存操作上的错误：

* 读写已经释放的内存
* 读写内存块越界（从前或者从后）
* 使用还未初始化的变量
* 将无意义的参数传递给系统调用
* 内存泄漏

### 吞吐量（Throughput）、QPS、并发数、响应时间（RT）

1) 响应时间(RT)

响应时间是指系统对请求作出响应的时间

2) 吞吐量(Throughput)

吞吐量是指系统在单位时间内处理请求的数量。

3) 并发用户数

并发用户数是指系统可以同时承载的正常使用系统功能的用户的数量。

4) QPS每秒查询率(Query Per Second)

每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准，在因特网上，作为域名系统服务器的机器的性能经常用每秒查询率来衡量。对应fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。

mysql 与 redis 的QPS

QPS 主要受限于三点：

1 并发数

2 带宽

3 服务器配置

mysql 官方基准测试

https://www.mysql.com/cn/why-mysql/benchmarks/

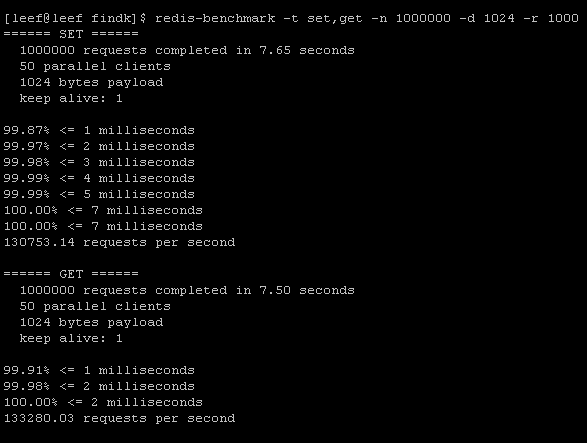
**5.5 ~5.6版本50万QPS**

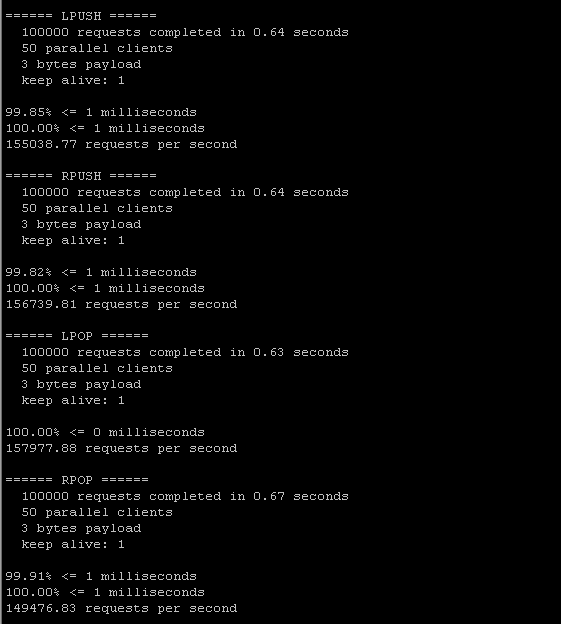
**redis 没有官方测试 民间测试情况看来 基本在几十万**

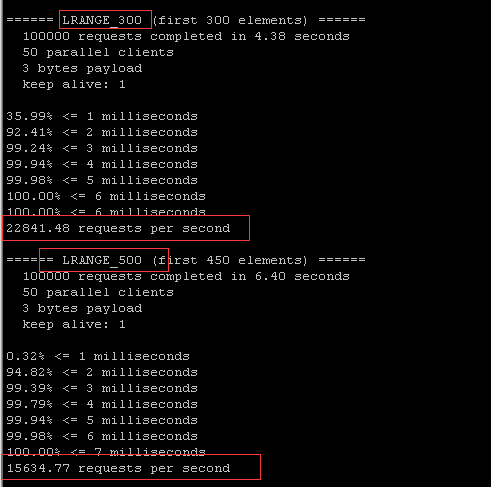
这是在单机虚拟机下跑的数据 SET 15万+ QPS

redis-benchmark -t set,get -n 1000000 -d 1024 -r 1000

1 K数据 1000个连接 每秒13万+的QPS







RabbitMQ 搭建

配置文件 /etc/rabbitmq/rabbitmq.config

查看状态 service rabbitmq-server status

安装插件 /sbin/rabbitmq-plugins enable rabbitmq\_management

### linux 锁

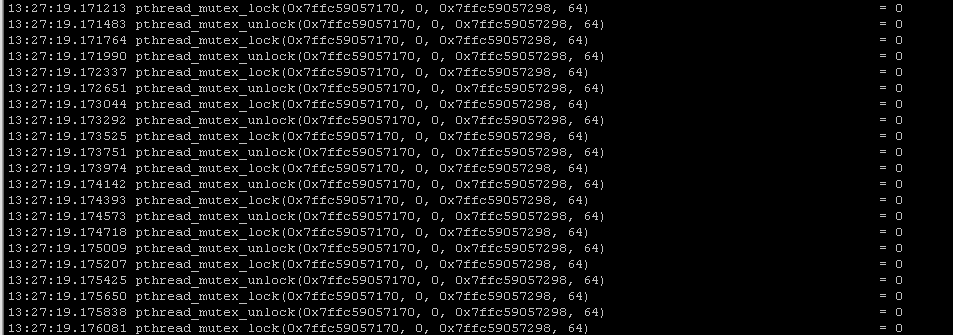
### 1 互斥锁

Mutex

Mutex属于sleep-waiting类型的锁, 调用可能会引起睡眠

pthread\_mutex\_t m\_mutex;

ltrace 跟踪



测试环境 虚拟机 每秒可达2000W+

### 2 自旋锁

**Spin lock** 如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是 否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

**不当使用也会造成死锁**

### 3 可重入锁

同一线程relock可返回成功

pthread\_mutexattr\_init(&attr);

pthread\_mutexattr\_settype(&attr, PTHREAD\_MUTEX\_RECURSIVE);

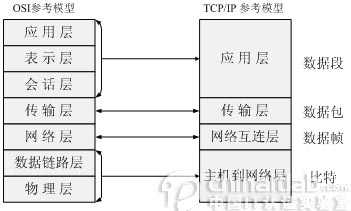
pthread\_mutex\_init(&m\_mutex, &attr);

或者

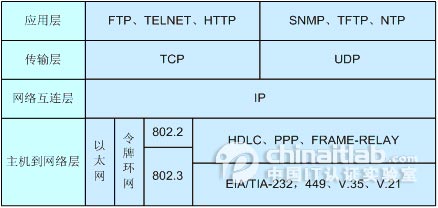
pthread\_mutex\_init(&m\_mutex, null);

## 网络篇

#### 网络层

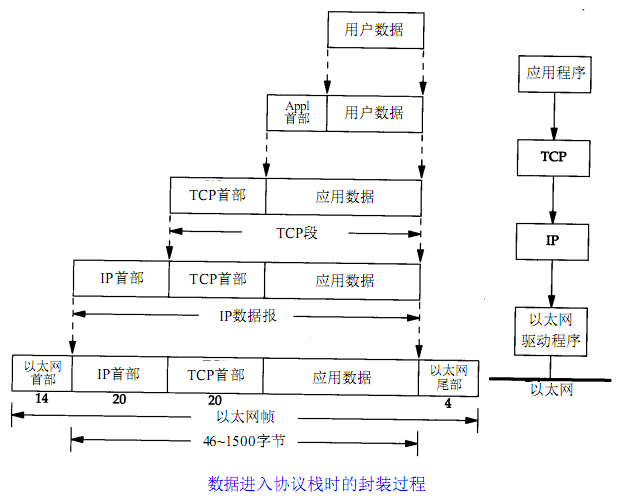
ISO制定的OSI参考模型的过于庞大、复杂招致了许多批评。与此对照，由技术人员自己开发的TCP/IP协议栈获得了更为广泛的应用。如图2-1所示，是TCP/IP参考模型和OSI参考模型的对比示意图。  


TCP/IP参考模型的层次结构  
　　TCP/IP协议栈是美国国防部高级研究计划局计算机网（Advanced Research Projects Agency Network，ARPANET）和其后继因特网使用的参考模型。ARPANET是由美国国防部（U.S．Department of Defense，DoD）赞助的研究网络。最初，它只连接了美国境内的四所大学。随后的几年中，它通过租用的电话线连接了数百所大学和政府部门。最终ARPANET发展成为全球规模最大的互连网络-因特网。最初的ARPANET于1990年永久性地关闭。　　  
　　TCP/IP参考模型分为四个层次：应用层、传输层、网络互连层和主机到网络层。



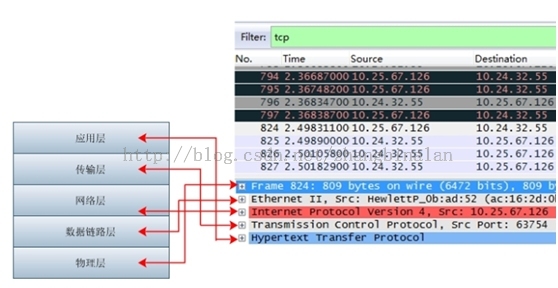
数据包

http://images.cnblogs.com/cnblogs_com/bluetzar/IPPackage.jpg



抓包分析

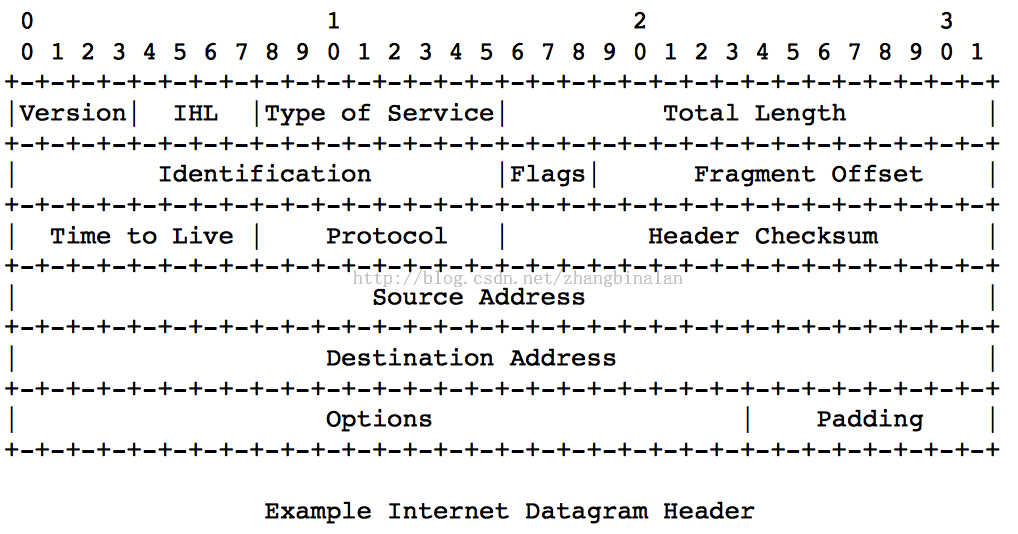
**抓包分析**



1. Frame: 物理层的数据帧概况
2. Ethernet II: 数据链路层以太网帧头部信息
3. Internet Protocol Version 4: 互联网层IP包头部信息
4. Transmission Control Protocol: 传输层的数据段头部信息，此处是TCP
5. Hypertext Transfer Protocol: 应用层的信息，此处是HTTP协议

#### IP协议

**IP协议头部**



英文不好来个中文版的



版本（Version）字段：占4比特。用来表明IP协议实现的版本号，当前一般为IPv4，即0100。　　  
　　●报头长度（Internet Header Length，IHL）字段：占4比特。是头部占32比特的数字，包括可选项。普通IP数据报（没有任何选项），该字段的值是5，即160比特=20字节。此字段最大值为60字节。　　  
　　●服务类型（Type of Service ，TOS）字段：占8比特。其中前3比特为优先权子字段（Precedence，现已被忽略）。第8比特保留未用。第4至第7比特分别代表延迟、吞吐量、可靠性和花费。当它们取值为1时分别代表要求最小时延、最大吞吐量、最高可靠性和最小费用。这4比特的服务类型中只能置其中1比特为1。可以全为0，若全为0则表示一般服务。服务类型字段声明了数据报被网络系统传输时可以被怎样处理。例如：TELNET协议可能要求有最小的延迟，FTP协议（数据）可能要求有最大吞吐量，SNMP协议可能要求有最高可靠性，NNTP（Network News Transfer Protocol，网络新闻传输协议）可能要求最小费用，而ICMP协议可能无特殊要求（4比特全为0）。实际上，大部分主机会忽略这个字段，但一些动态路由协议如OSPF（Open Shortest Path First Protocol）、IS-IS（Intermediate System to Intermediate System Protocol）可以根据这些字段的值进行路由决策。

　　●总长度字段：占16比特。指明整个数据报的长度（以字节为单位）。最大长度为65535字节。　　  
　　●标志字段：占16比特。用来唯一地标识主机发送的每一份数据报。通常每发一份报文，它的值会加1。(**分片是根据此标识判断是否属于同一数据流**)　　  
　　●标志位字段：占3比特。标志一份数据报是否要求分段。　　  
　　●段偏移字段：占13比特。如果一份数据报要求分段的话，此字段指明该段偏移距原始数据报开始的位置。　　  
　　●生存期（TTL：Time to Live）字段：占8比特。用来设置数据报最多可以经过的路由器数。由发送数据的源主机设置，通常为32、64、128等。每经过一个路由器，其值减1，直到0时该数据报被丢弃。　　  
　　●协议字段：占8比特。指明IP层所封装的上层协议类型，如ICMP（1）、IGMP（2） 、TCP（6）、UDP（17）等。

1 ICMP

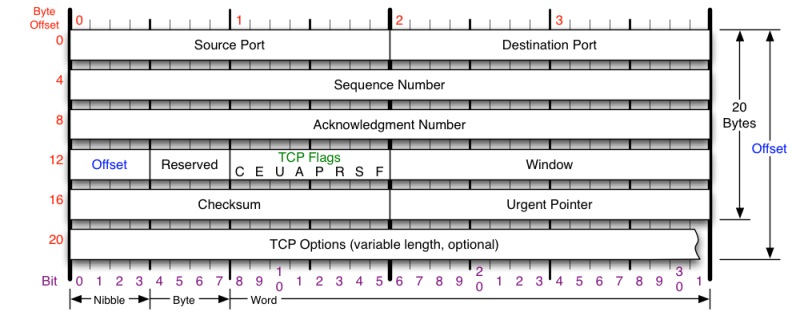
2 IGMP  
 6 TCP  
 17 UDP  
 88 IGRP  
 89 OSPF　  
　　●头部校验和字段：占16比特。内容是根据IP头部计算得到的校验和码。计算方法是：对头部中每个16比特进行二进制反码求和。（和ICMP、IGMP、TCP、UDP不同，IP不对头部后的数据进行校验）。　　  
　　●源IP地址、目标IP地址字段：各占32比特。用来标明发送IP数据报文的源主机地址和接收IP报文的目标主机地址。　　  
　　可选项字段：占32比特。用来定义一些任选项：如记录路径、时间戳等。这些选项很少被使用，同时并不是所有主机和路由器都支持这些选项。可选项字段的长度必须是32比特的整数倍，如果不足，必须填充0以达到此长度要求。

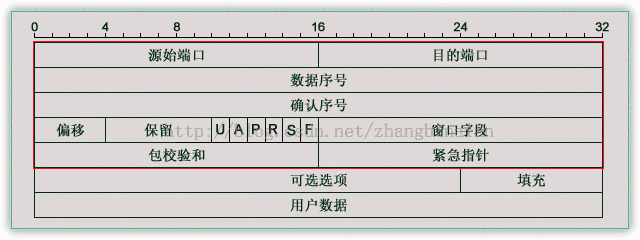
#### TCP协议

tcp 协议满足一下特点

1. 基本数据传输  
   连接是双向通信的；字节流；字节封装成包；TCP自己决定何时停止和发送数据，如果需要确保已经提交的数据及时发送，可以使用push功能
2. 可靠性  
   TCP必须可以恢复以下数据：受损的、丢失的、重复的、乱序的  
   每个字节一个编号，接受者确认，如果确认超时未收到应该重发  
   接收者根据编号来排序错序的报文和去除重复的报文  
   受损的数据可以通过报文段的校验码来控制，丢弃受损的数据  
   超时重发
3. 流量控制  
   流量控制，接收者控制发送者数据发送  
   窗口机制，每一个ACK报文段返回一个窗口值，当前确认的字节以后最多能发送的字节数
4. 多路复用  
   多路复用，同一个机器上可以存在多个TCP连接  
   端口+host=socket;两个socket确认一个连接  
   常用服务特定的端口号；
5. 链接  
   可靠性以及滑动窗口机制都是建立在链接的基础上的  
   连接：描述当前数据流的状态的一些信息的组合，sockets+序列号+窗口  
   建立连接（初始化通信双方的状态信息），创建TCB（Transmission Control Block）建立连接三次握手  
   关闭链接四次握手
6. 优先级和安全性  
   先不关注这部分

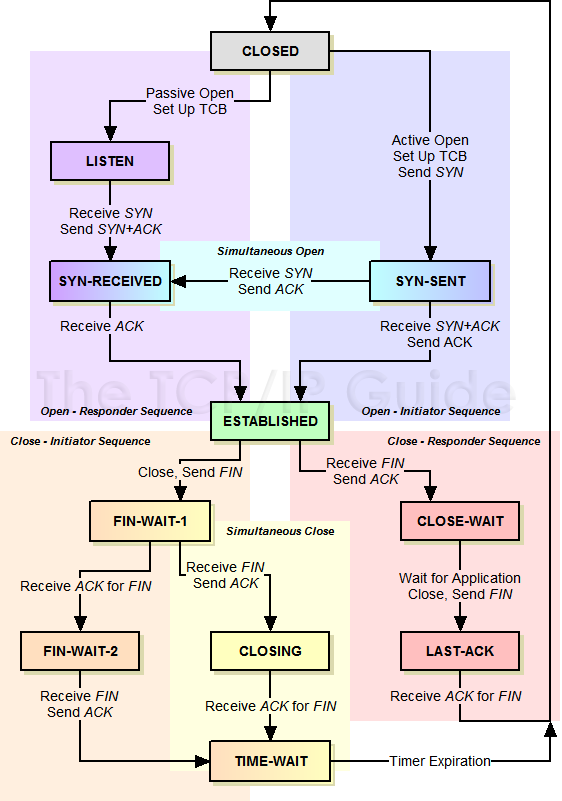
协议头：





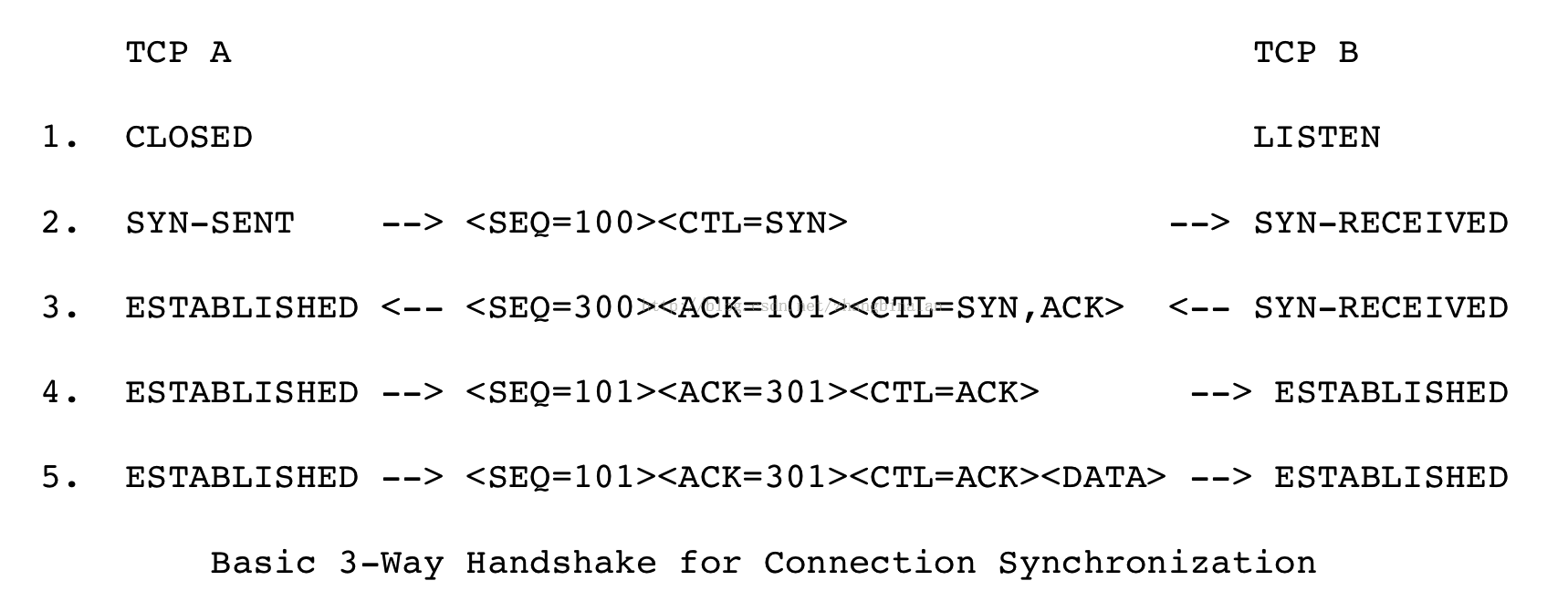
1. 源端口，16bits，范围0~65525。
2. 目的端口，16bits，范围同上。
3. sequence number： 数据序号，32bits，TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。
4. acknoledgement number： 确认号，32bits，期望收到对方的下一个报文段的数据的第一个字节的序号。
5. 数据偏移，4bits，单位为4字节，它指出报文数据距TCP报头的起始处有多远(TCP报文头长度)。
6. 保留字段 6bits，保留今后使用，目前置0处理。
7. URG：紧急比特，1bit，当 URG=1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)  
   ACK：确认比特，1bit，只有当 ACK=1时确认号字段才有效。当 ACK=0 时，确认号无效  
   PSH：推送比特，1bit，接收方 TCP 收到推送比特置1的报文段，就尽快地交付给接收应用进程，而不再等到整个缓存都填满了后再向上交付  
   RST：复位比特，1bit，当RST=1时，表明TCP连接中出现严重差错(如由于主机崩溃或其他原因)，必须释放连接，然后再重新建立运输连接  
   SYN：同步比特，1bit，同步比特 SYN 置为 1，就表示这是一个连接请求或连接接受报文  
   FIN：终止比特，1bit，用来释放一个连接。当FIN=1 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接
8. 窗口大小，16bits，窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。
9. 检验和，16bits，检验和字段检验的范围包括首部和数据这两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。
10. 紧急指针字段，16bits，紧急指针指出在本报文段中的紧急数据的最后一个字节的序号。
11. 选项字段，长度可变。TCP首部可以有多达40字节的可选信息，用于把附加信息传递给终点，或用来对齐其它选项。 这部分最多包含40字节，因为TCP头部最长是60字节（其中还包含前面讨论的20字节的固定部分）  
    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是选项表结束选项  
    kind=1是空操作（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是最大报文段长度选项,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（Max Segment Size，MSS）**。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。

##### TCP 状态转换图





##### TCP 连接建立



**为什么是三次握手**

client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，客户端重发了建立链接的请求并成功通信，连接释放，这之后最开始的建立链接的请求到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。

**这个问题的本质是, 信道不可靠, 但是通信双发需要就某个问题达成一致. 而要解决这个问题,  无论你在消息中包含什么信息, 三次通信是理论上的最小值. 所以三次握手不是TCP本身的要求, 而是为了满足"在不可靠信道上可靠地传输信息**

##### TCP断开连接

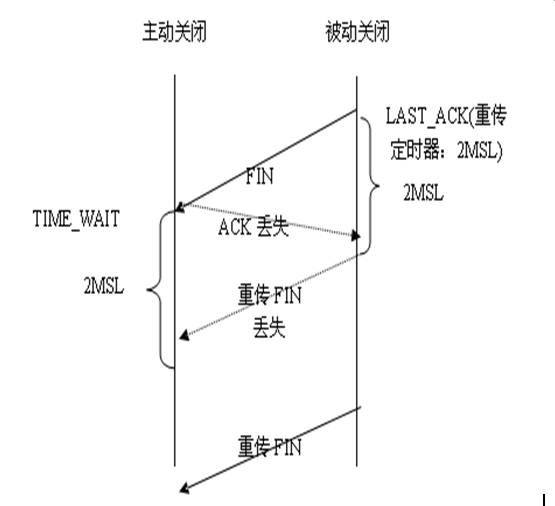
为什么关闭连接却是四次握手

服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的

TIME\_WAIT状态

1 TIME\_WAIT确保有足够的时间让对端收到了ACK，如果被动关闭的那方没有收到Ack，就会触发被动端重发Fin，一来一去正好2个MSL(Maximum Segment Lifetime 报文最大生存时间) ，

2 有足够的时间让这个连接不会跟后面的连接混在一起



上述情况也并没有正常关闭

**在TIME\_WAIT下等待2MSL，只是为了尽最大努力保证四次握手正常关闭。**  
**确保老的报文段在网络中消失，不会影响新建立的连接**

**SYNC----- >**

**<---------FIN**

当上一个连接的重传FIN到达主动关闭方时，被新的连接所接受，这将导致新的连接被复位，很显然，这不是我们希望看到的事情

1 在TIME\_WAIT状态，如果收到SYN报文段，且新的起始序号大于连接上最后收到的序号，说明对端要求在已被关闭且正处于TIME\_WAIT状态的连接上重新连接。

2 如果在TIME\_WAIT状态收到RST报文段，这释放连接资源，跳转到CLOSED状态，提前终止TIME\_WAIT状态

**3 收到ACK报文段**

**此时，连接的两端已发送过FIN,且两个FIN都已被确认。但如果TCP对远端FIN的确认丢失，对端将重传FIN（带有ACK）。TCP丢弃报文端并重传ACK。此外TIME\_WAIT定时器必须被重置，时限等于2MSL。**

**4 收到FIN报文段**

**如果在TIME\_WAIT状态收到FIN,说明这是一个重复报文端。启动TIME\_WAIT定时器，时限等于2MSL，发送ACK。**

##### ****滑动窗口****

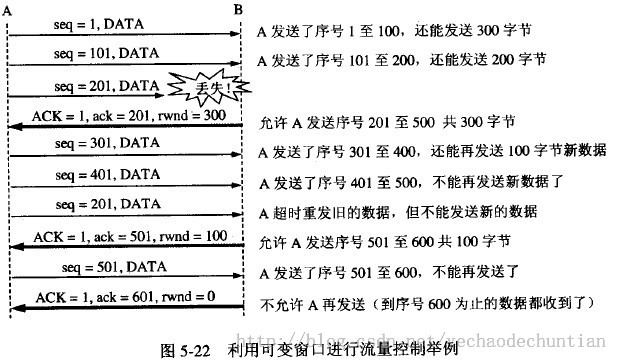
借用一位前辈原话

需要说明一下，如果你不了解TCP的滑动窗口这个事，你等于不了解TCP协议。

**TCP必需要解决的可靠传输以及包乱序（reordering）的问题**，所以，TCP必需要知道网络实际的数据处理带宽或是数据处理速度，这样才不会引起网络拥塞，导致丢包

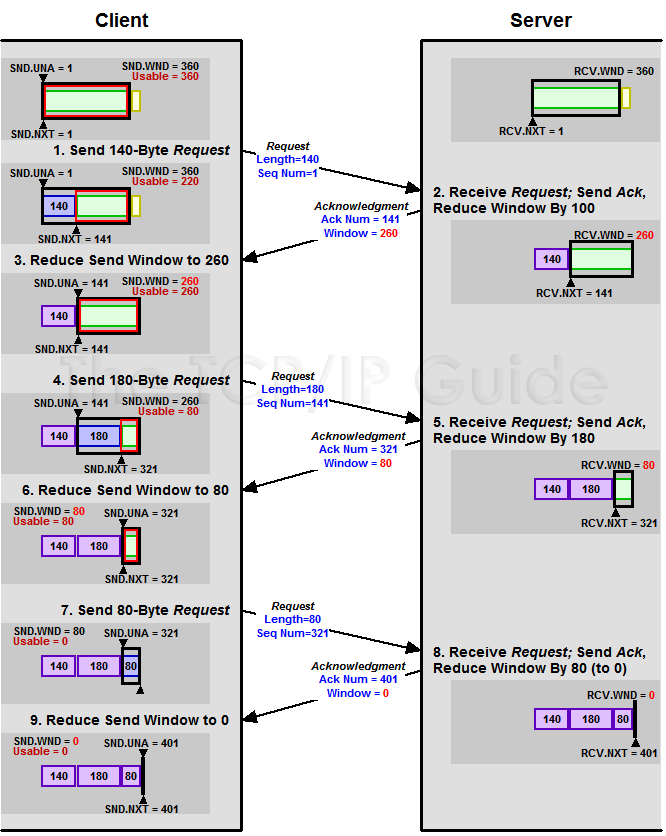
TCP引入了一些技术和设计来做网络流控，Sliding Window是其中一个技术，**这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据**。**于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。**

如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失  
所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收.利用滑动窗口机制可以很方便地在TCP连接上实现对发送方的流量控制  
设A向B发送数据。在连接建立时，B告诉了A：“我的接收窗口是 rwnd = 400 ”(这里的 rwnd 表示 receiver window) 。因此，发送方的发送窗口不能超过接收方给出的接收窗口的数值。请注意，TCP的窗口单位是字节，不是报文段。TCP连接建立时的窗口协商过程在图中没有显示出来。再设每一个报文段为100字节长，而数据报文段序号的初始值设为1。大写ACK表示首部中的确认位ACK，小写ack表示确认字段的值ack



从图中可以看出，B进行了三次流量控制。第一次把窗口减少到 rwnd = 300 ，第二次又减到了 rwnd = 100 ，最后减到 rwnd = 0 ，即不允许发送方再发送数据了。这种使发送方暂停发送的状态将持续到主机B重新发出一个新的窗口值为止。B向A发送的三个报文段都设置了 ACK = 1 ，只有在ACK=1时确认号字段才有意义。 TCP为每一个连接设有一个持续计时器(persistence timer)。只要TCP连接的一方收到对方的零窗口通知，就启动持续计时器。若持续计时器设置的时间到期，就发送一个零窗口控测报文段（携1字节的数据），那么收到这个报文段的一方就重新设置持续计时器。

再来一个例子

c

TCP使用了Zero Window Probe技术，缩写为ZWP，也就是说，发送端在窗口变成0后，会发ZWP的包给接收方，让接收方来ack他的Window尺寸，一般这个值会设置成3次，第次大约30-60秒（不同的实现可能会不一样）。如果3次过后还是0的话，有的TCP实现就会发RST把链接断了。

##### 拥塞控制

拥塞控制：防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提：网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机、路由器，以及与降低网络传输性能有关的所有因素

流量控制：指点对点通信量的控制，是端到端正的问题。流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收

拥塞控制代价：需要获得网络内部流量分布的信息。在实施拥塞控制之前，还需要在结点之间交换信息和各种命令，以便选择控制的策略和实施控制。这样就产生了额外的开销。拥塞控制还需要将一些资源分配给各个用户单独使用，使得网络资源不能更好地实现共享。

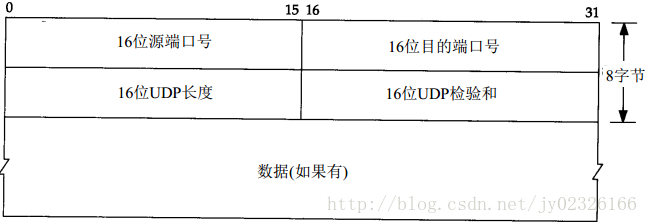
几种拥塞控制方法

慢开始( slow-start )、拥塞避免( congestion avoidance )、快重传( fast retransmit )和快恢复( fast recovery )。

发送方维持一个拥塞窗口 cwnd ( congestion window )的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞  
发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数

#### UDP协议

##### udp协议头



checksum的算法与[IP协议的header checksum算法](http://www.cnblogs.com/vamei/archive/2012/12/02/2796988.html)相类似。然而，UDP的checksum所校验的序列包括了整个UDP数据包，以及封装的IP头部的一些信息(主要为出发地IP和目的地IP) 这样，checksum就可以校验IP：端口的正确性了。在IPv4中，checksum可以为0，意味着不使用checksum。IPv6要求必须进行checksum校验。

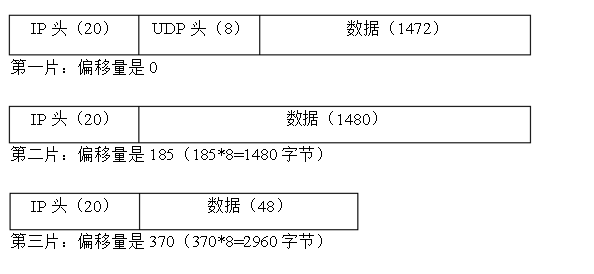
##### UDP分片

当UDP报文的长度大于MTU时，IP层就需要进行分片了。

物理网络层一般要限制每次发送数据帧的最大长度，这个最大长度称为最大传输单元（MTU）。**IP层在发送一个IP数据报之前需要询问下一跳接口的MTU大小，如果待发送的IP数据报大小大于MTU，则需要进行分片。所以，分片既可以发生在原始发送端主机上，也可以发生在中间路由器上。**但数据报的重组只在最终目的端进行。不管是分片还是重载，都是在IP层进行的，对上层来说这些操作是透明的。

分片方法**：**

1. 在ip头中3位标志，13位片偏移，这俩东西就是用来分片的。
2. 3位标志中：一个更多的片标识，除了最后一个分片，其它分片都要有这个标识，还有一个不分片标识，如果置上，则即便IP数据报再长，也不分片，但是会将该IP数据报丢弃，并发送一个ICMP差错报文给发送端。
3. 13位片偏移：用于表示相对于起始的偏移量，就是个offset。
4. 每个分片都有一个独立的IP头，20个字节。
5. IP的分片和组装都是在IP层完成的，其对UDP/TCP网络层是透明的。
6. 如果在传输过程中，某个分片丢失，则需要重传整个IP数据报，不能单独传递某个分片，这主要是，如果是中间某个路由器做的分片，起始端也不知道如何分片的，所以只能重传整个IP数据报。
7. 至于重不重传。则只能有网络层来决定。TCP会丢包重传，但是UDP则不会，IP层是没有丢包重传机制的。
8. 传输单元，IP层叫IP数据报，链路层叫分组



**此处需要注意的是对于分片1的报头，相对于其他两个分片的报头而言，要多出8个字节UDP协议的报头开销，因此，在计算实际传输的数据净载荷时，分片1要多减去8字节UDP报头。最后，接收主机通过此偏移值将数据重组成完整的数据报。**

**如果在IP首部设置了不可分片标志，但是达到MTU上限，必须分片，则会丢弃该IP报，返回ICMP不可达差错**

**分片标识**

第1位作为保留，置0;第2位，分段，有两个不同的取值：该位置0，表示可以分段;该位置1，表示不能分段;第3位，更多分段，同样有两个取值：该位置0，表示这是数据流中的最后一个分段，该位置1，表示数据流未完，后续还有分段，当一个数据报没有分段时，则该位置0，表示这是唯一的一个分段。

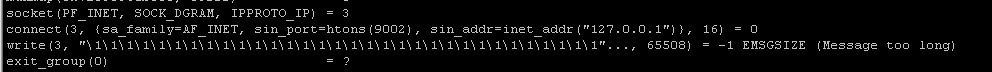
##### 重组

当分片的IP数据报从源地址发送到目的地址的时候，由于网络延迟或者不同的传输路径的关系，在到达目的主机时，这些分片数据报并不总是有序的排列，而是处于一种无序状态，因此，接收主机便用此ID(IP协议头中的标识)判断接收的这些分片数据是否属于同一个数据流，然后再进行重组

##### TCP与UDP区别

TCP传输控制协议,提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个TCP连接，之后才能传输数据。TCP提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端  
UDP---用户数据报协议，是一个简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快

##### udp最大数据包

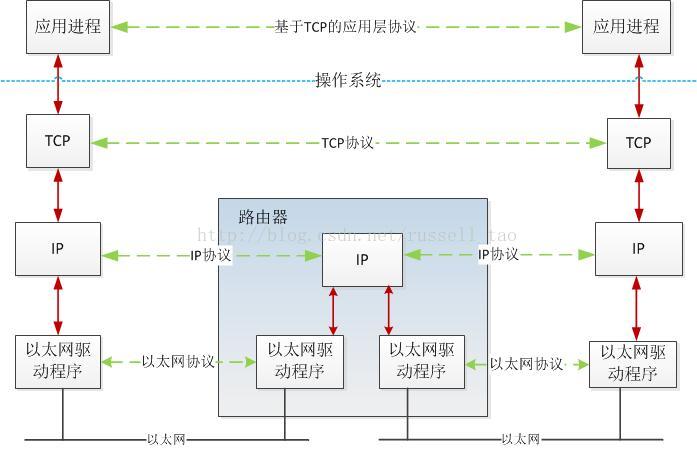




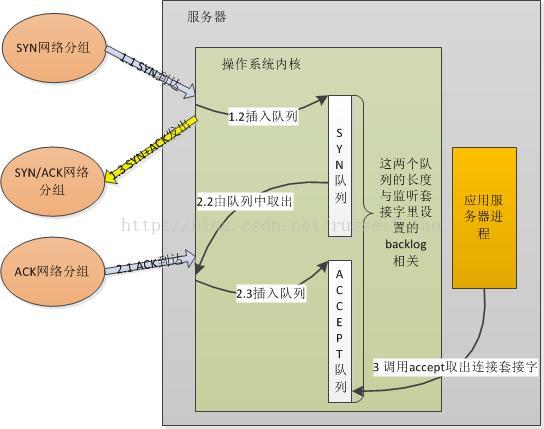
#### TCP网络交互详解

#### accept

**网络、协议、应用服务器间的关系**



TCP 三次握手过程

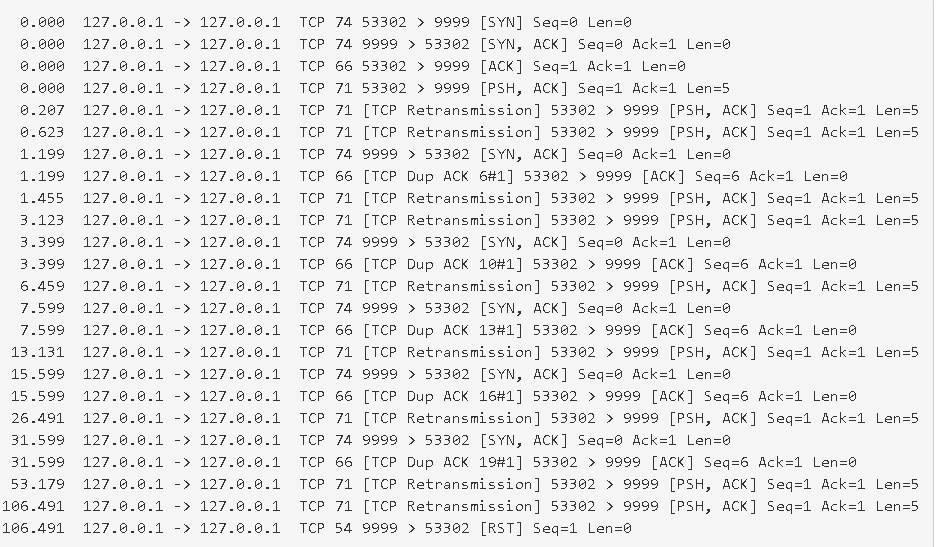


http://veithen.github.io/2014/01/01/how-tcp-backlog-works-in-linux.html

基于上文所述 sync\_queue 和内核参数相关 **/proc/sys/net/ipv4/tcp\_max\_syn\_backlog**

**accpet\_queue** 队列和listen backlog 相关

如果accept 队列满了会怎么样



accept 队列之后收到客户端 ack of the 3-way 不会移动到 accept 队列，当客户端发送数据时继续回复终端一个 sync-ack。终端就这样被调戏死了，服务最终看不下去了扔一个 RST断开连接。

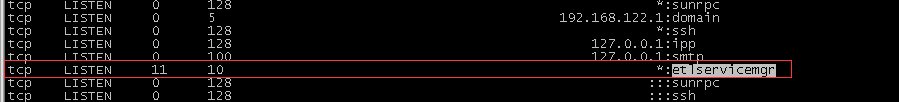
如果sync queue 满了会怎么样？ 直接忽略sync

**注：**

**accept 队列大小由 listen backlog 和 /proc/sys/net/core/somaxconn 值决定**

**accept 队列大小 = min(backlog, /proc/sys/net/core/somaxconn)**

**ss -l**

****

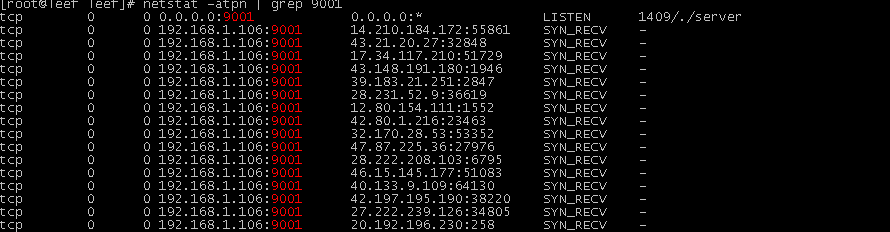
**Recv-Q** accept 队列未处理数据

**Send-Q** accept队列大小

**sync 队列长度**

**/proc/sys/net/ipv4/tcp\_max\_syn\_backlog**

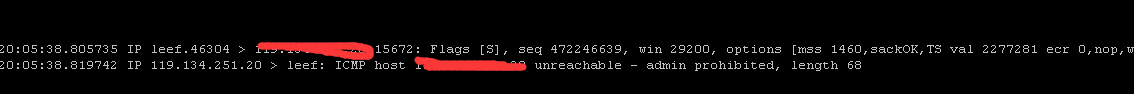
****

****

**实际测试中队列一直未达到最大值，这点不知道什么问题**

##### No route to host

当天工作未完成,晚上回家继续做。在公司能够连接上的服务回到家连接变成了**No route to host，**ping域名有正常返回。平时偶尔有遇到No route to host但从未深究。这个错误什么时候返回的了？猜测是防火墙限制了访问网络，上服务查看果不其然。最初猜测应该有TCP的回包,但是抓包只发现有SYNC没任何返回，猜测应该是回了一个IP包，TCP和IP包同时抓。



**好奇Connection refused是什么数据包**

****

**Connection refused 返回的是RST包，至少这是一个TCP包，No route to host 返回的是IP 包**

SYNC---RST ==> Connection refused

SYNC----ICMP ===> No route to host

#### [TCP洪水攻击（SYN Flood）的诊断和处理](http://tech.uc.cn/?p=1790)

**1 诊断**

tail -f /var/log/messages

netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a]}'

**netstat -na |grep SYN\_RECV|more**

**查看疑似攻击源IP段**

**2 处理**

**1) iptables -A INPUT -s  173.0.0.0/8  -p tcp  –dport 80 -j DROP**

**防火墙封闭**

**2) 硬件防火墙做代理**

**3) 调节参数 增大tcp\_max\_sync\_backlog 关闭 sync\_ack重试 tcp\_synack\_retries 设置为 0**

#### send

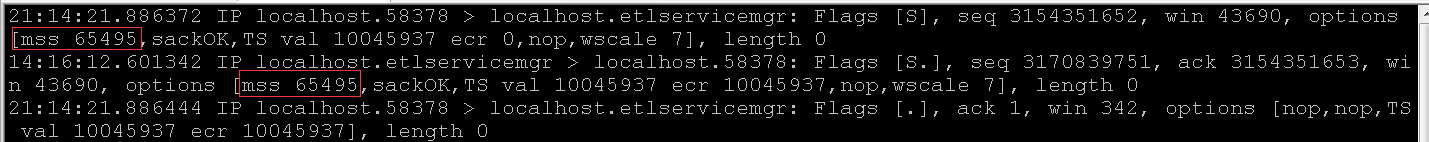
几个概念

MTU: 最大传输单元

MSS：最大分段大小

谁来决定 MTU？ MTU是有物理网络决定的，主机A 到主机B之间经过了N个网络设备，相邻两个网络设备之间都有一个MTU，大小可能都不同。

MMS是在SYNC 分节中协商的



1） 应用程序调用send

2） 内核调用tcp\_sendmsg完成

3）数据从用户态拷贝到内核态 把待发送数据按照MSS分片成报文段存放在tcp\_write\_queue队列中

注：发送缓冲区大小 /proc/sys/net/core/wmem\_default 将受到ACK确认的数据移除，移动滑动窗口

4）内核调用tcp\_push等方法，最终调用IP层的方法来发送tcp\_write\_queue队列中的报文

5） 阻塞socket 返回

解析：send 调用返回了说明了什么？

1 非阻塞 调用只完成了数据从用户态到内核态的拷贝。

2 阻塞socket只完成了数据发送到网络。

都不能说明对方收到了。

Nagle算法、滑动窗口、拥塞窗口对发送方法的影响 http://blog.csdn.net/russell\_tao/article/details/9370109

#### recv

recv主要几个重点

1） 分片数据重组 流程

receive、out\_of\_order、prequeue、backlog 队列

receive：收到的完整的顺序队列 receive队列是允许用户进程直接读取的，它是将已经接收到的TCP报文，去除了TCP头部、排好序放入的、用户进程可以直接按序读取的队列

out\_of\_order: 乱序队列

**阻塞读 队列**

2） socket 读取加锁

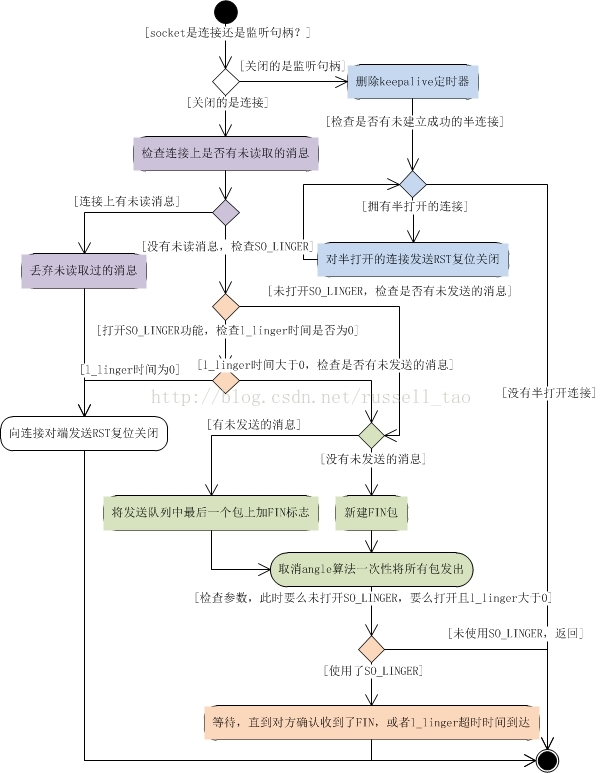
http://blog.csdn.net/russell\_tao/article/details/9950615

#### close

close 与shutdown

当这个socket的引用计数f\_count不为0时，是不会触发到真正关闭TCP连接的tcp\_close方法的。

shutdown则不然，这里是没有引用计数什么事的，只要调用了就会去试图按需关闭连接。

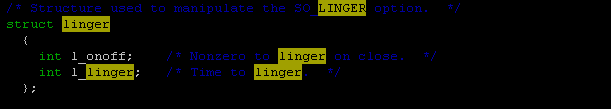


so\_linger

so\_linger用来保证对方收到了close时发出的消息，即，至少需要对方通过发送ACK且到达本机。

怎么保证呢？等待！close会阻塞住进程，直到确认对方收到了消息再返回。然而，网络环境又得复杂的，如果对方总是不响应怎么办？所以还需要l\_linger这个超时时间，控制close阻塞进程的最长时间。注意，务必慎用so\_linger，它会在不经意间降低你程序中代码的**执行速度（close的阻塞）。**

**linger**

****

**1 l\_onoff = 0 关闭 SO\_LINGER 功能**

**2 l\_onoff 非0**

**1) l\_linger = 0 调用close不会发送FIN包，直接回复RST**

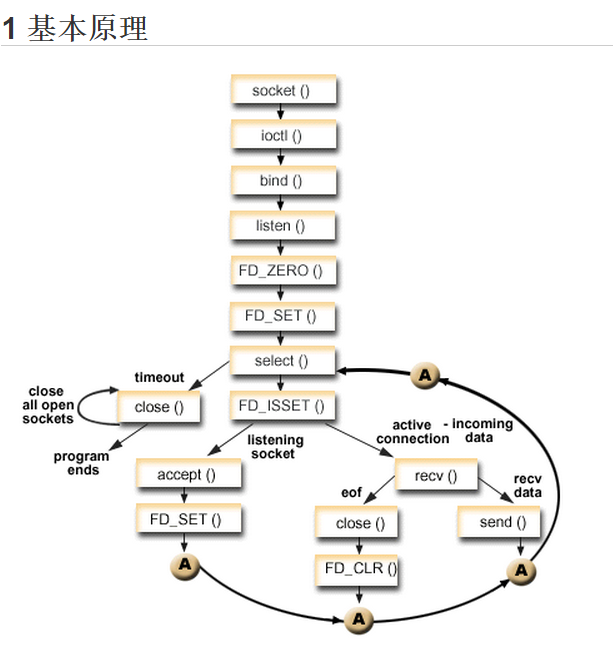
**2) l\_linger = n > 0 调用close时最多等待 n 秒 (若此时socket发送缓冲区有待发送数据会引起阻塞)**

**此时等待对方回复了FIN包的ACK才会返回。**

#### 多路复用

**select**

程序模型



（1）使用copy\_from\_user从用户空间拷贝fd\_set到内核空间

（2）注册回调函数\_\_pollwait

（3）遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock\_poll，sock\_poll根据情况会调用到tcp\_poll,udp\_poll或者datagram\_poll）

（4）以tcp\_poll为例，其核心实现就是\_\_pollwait，也就是上面注册的回调函数。

（5）\_\_pollwait的主要工作就是把current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp\_poll来说，其等待队列是sk->sk\_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。

（6）poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd\_set赋值。

（7）如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule\_timeout是调用select的进程（也就是current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间（schedule\_timeout指定），还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。

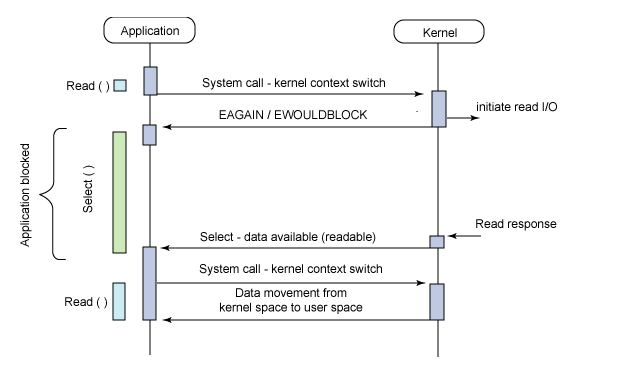
（8）把fd\_set从内核空间拷贝到用户空间。

**总结：**

**select的几大缺点：**

**（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大**

**（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大**

****

poll

int poll (struct pollfd \*fds, unsigned int nfds, int timeout);

不同与select使用三个位图来表示三个fdset的方式，poll使用一个 pollfd的指针实现。

struct pollfd {

int fd; /\* file descriptor \*/

short events; /\* requested events to watch \*/

short revents; /\* returned events witnessed \*/

};

pollfd结构包含了要监视的event和发生的event，不再使用select“参数-值”传递的方式。同时，pollfd并没有最大数量限制（但是数量过大后性能也是会下降）。 和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。

epoll

epool 对于select的几点改进

**epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。**

**epoll不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）**

**epoll\_create** ==> epoll文件系统里建了个file结点, 在内核cache里建了个红黑树用于存储以后epoll\_ctl传来的socket，还会再建立一个list链表，用于存储准备就绪的事件

**epoll\_ctl** ==> 将socket 添加进红黑树，同时注册内核终端函数。告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。所以，当一个socket上有数据到了，内核在把网卡上的数据copy到内核中后就来把socket插入到准备就绪链表里了。

**epoll\_wait**==> 仅仅观察这个list链表里有没有数据即可。有数据就返回，没有数据就sleep，等到timeout时间到后即使链表没数据也返回

**LT模式 和 ET模式：**

LT模式下，只要一个句柄上的事件一次没有处理完，会在以后调用epoll\_wait时再次返回这个句柄，而ET模式仅在第一次返回。

这件事怎么做到的呢？当一个socket句柄上有事件时，内核会把该句柄插入上面所说的准备就绪list链表，这时我们调用epoll\_wait，会把准备就绪的socket拷贝到用户态内存，然后清空准备就绪list链表，最后，epoll\_wait干了件事，就是检查这些socket，如果不是ET模式（就是LT模式的句柄了），并且这些socket上确实有未处理的事件时，又把该句柄放回到刚刚清空的准备就绪链表了。所以，非ET的句柄，只要它上面还有事件，epoll\_wait每次都会返回。而ET模式的句柄，除非有新中断到，即使socket上的事件没有处理完，也是不会次次从epoll\_wait返回的。

#### 惊群

惊群效应就是当一个fd的事件被触发时，所有等待这个fd的线程或进程都被唤醒。

许多进程被内核重新调度唤醒，同时去响应这一个事件，当然只有一个进程能处理事件成功，其他的进程在处理该事件失败后重新休眠（也有其他选择）在这种情况下，程序的效率会变得低的很多。

linux内核在2.6.18版本解决了accept的惊群现象。

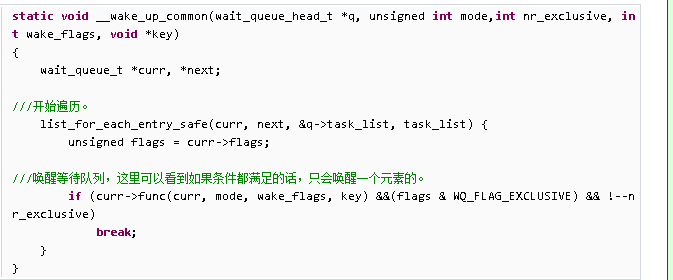
对于一些 linux 内核中已知的惊群问题，内核开发者增加了“互斥等待”选项：

当一个等待队列入口如果 WQ\_FLAG\_EXCLUSEVE 标志置位，它被添加到等待队列首部，否则添加到队列尾部

当 wake\_up 被在一个队列上调用时，它在唤醒第一个有 WQ\_FLAG\_EXCLUSEVE 标志的进程后终止

因此，对于互斥等待的行为，内核只会唤醒队列中的第一个进程

**内核被唤醒代码**



内核对于 accept的惊群现象已经做了处理，但是目前所使用的多路复用都会使用seclet或者epool，对于这样惊群现象依然存在。

nginx 使用 ngx\_accept\_mutex 作为互斥锁让所有 worker 进程竞争，最终只有一个进程可以获得锁，以保证只有一个进程执行 epoll\_wait 操作，并 accept

它的负载均衡也很简单，当达到最大connection的7/8时，本worker不会去试图拿accept锁，也不会去处理新连接，这样其他nginx worker进程就更有机会去处理监听句柄，建立新连接了。而且，由于timeout的设定，使得没有拿到锁的worker进程，去拿锁的频繁更高。

http://www.cnblogs.com/549294286/p/6058811.html\

## C++

#### 指针&引用

从概念上讲。指针从本质上讲就是存放变量地址的一个变量，在逻辑上是独立的，它可以被改变，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。

而引用是一个别名，它在逻辑上不是独立的，它的存在具有依附性，所以引用必须在一开始就被初始化，而且其引用的对象在其整个生命周期中是不能被改变的（自始至终只能依附于同一个变量）。

**区别**：

1 初始化 引用是对某一个对象的别名，因此引用不能为空。指针可以初始化为空，也可指向某一个对象

2 引用一旦指向某一个对象就不可以改变再指向其他对象，指针则可以随时改变指向的对象。

3 指针有自己的地址，引用没有自己的地址。

4 作为函数参数传递

1）指针属于值传递，对指针**值**(不是地址指向的值)的改变不影响主调函数实参的值

2）应用传递的是实参变量的地址，对引用的操作通过间接寻址作用到实参。

(**这是教科书的标准答案，作为一个做过项目的人来说，指针作为参数传递去操作指针而不是操作指针指向的值，对于这样的使用深表遗憾。如果面试官期望你回答这个答案，你不妨鄙视他下**)

5、引用的创建和销毁不会调用类的拷贝构造函数和析构函数。

(有看到这样的回答，这也从另一个方面了解应用，引用不创建对象)

**联系**：

从汇编代码看，引用和指针是一样的，但是编译器在对待引用时强制初始计划绑定到一个具体的对象。

#### static

1 限定了生命周期

2 限定了作用域

1) 修饰局部变量 在第一次访问时初始化，生命周期为整个程序生命周期

2) 修饰全局变量 在main函数之前被初始选，生命周期为整个程序生命周期， 作用域限定为当前文件

3) 修饰类成员变量 只能通过类的命名空间访问。成员不属于任何一个类实例

4) 修饰成员函数 只能通过类的命名空间访问。不属于任何一个类实例

#### const

1) const 修饰基本数据

const int a=10;               等价的书写方式：     int const a=10;  
const int arr[3]={1,2,3};                        int const arr[3]={1,2,3};

2) **const修饰指针变量\*及引用变量&**

const int\* a = & [1]          //非常量数据的常量指针    指针常量  
int const \*a = & [2]          //非常量数据的常量指针     a is a pointer to the constant char variable

int\* const a = & [3]          //常量数据的非常量指针指针常量 常量指针 a is a constant pointer to the (non-constant) char variable

const int\* const a = & [4]    //常量数据的常量指针

如果const位于星号\*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位于星号的右侧，const就是修饰指针本身，即指针本身是常量。

const int\* a  ==》等价于 int const \*a

int\* const a 为指针本身是常量，而指针所指向的内容不是常量，这种情况下不能对指针本身进行更改操作

int\* const a 为指针本身和指向的内容均为常量

int const &a=x;

 const int &a=x;

 int &const a=x;//这种方式定义是C、C++编译器未定义，虽然不会报错，但是该句效果和int &a一样。

          这两种定义方式是等价的，此时的引用a不能被更新。如：a++ 这是错误的。

3) **const应用到函数中**

**作为参数 (不讲)**

**作为返回值**

**4) 类相关CONST**

1> const修饰成员变量

2> const修饰成员函数

void function()const; //常成员函数, 它不改变对象的成员变量.

3> const修饰类对象/对象指针/对象引用

#### 指针

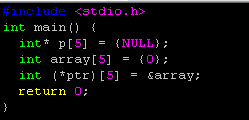
int \*p1[10];

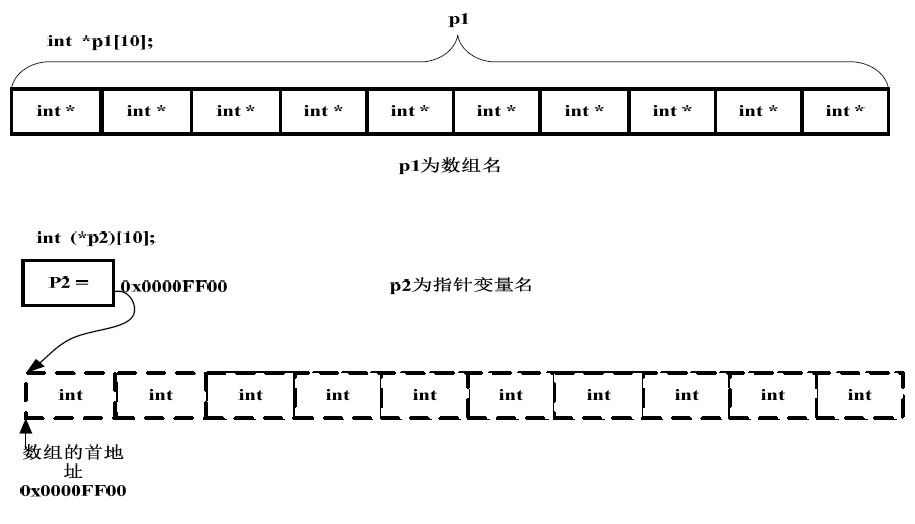
int (\*p2)[10];

**gdb 查看类型**

**ptype p1[0] === > int \***

**ptyp2 p2[0] ===> int [5]**

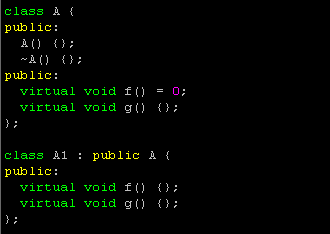
****

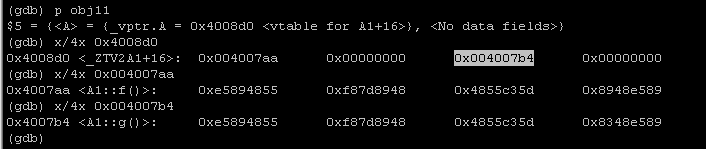


### 多态

多态性可以简单地概括为“**一个接口，多种方法**”，程序在运行时才决定调用的函数，它是面向对象编程领域的核心概念。多态(polymorphism)，字面意思多种形状。

C++多态性是通过虚函数来实现的，虚函数允许子类重新定义成员函数，而子类重新定义父类的做法称为覆盖(override)，或者称为重写。（这里我觉得要补充，重写的话可以有两种，直接**重写成员函数和重写虚函数**，只有重写了虚函数的才能算作是体现了C++多态性）而重载则是允许有多个同名的函数，而这些函数的参数列表不同，允许参数个数不同，参数类型不同，或者两者都不同。编译器会根据这些函数的不同列表，将同名的函数的名称做修饰，从而生成一些不同名称的预处理函数，来实现同名函数调用时的重载问题。但这并没有体现多态性。





#### [C语言中volatile关键字的作用](http://blog.csdn.net/tigerjibo/article/details/7427366)

1.编译器优化介绍：

由于内存访问速度远不及CPU处理速度，为提高机器整体性能，在硬件上引入硬件高速缓存Cache，加速对内存的访问。另外在现代CPU中指令的执行并不一定严格按照顺序执行，没有相关性的指令可以乱序执行，以充分利用CPU的指令流水线，提高执行速度。以上是硬件级别的优化。再看软件一级的优化：一种是在编写代码时由程序员优化，另一种是由编译器进行优化。编译器优化常用的方法有：将内存变量缓存到寄存器；调整指令顺序充分利用CPU指令流水线，常见的是重新排序读写指令。对常规内存进行优化的时候，这些优化是透明的，而且效率很好。由编译器优化或者硬件重新排序引起的问题的解决办法是在从硬件（或者其他处理器）的角度看必须以特定顺序执行的操作之间设置内存屏障（memory barrier），[**Linux**](http://lib.csdn.net/base/linux) 提供了一个宏解决编译器的执行顺序问题。

void Barrier(void)

这个函数通知编译器插入一个内存屏障，但对硬件无效，编译后的代码会把当前CPU寄存器中的所有修改过的数值存入内存，需要这些数据的时候再重新从内存中读出。

2.volatile总是与优化有关，编译器有一种技术叫做数据流分析，分析程序中的变量在哪里赋值、在哪里使用、在哪里失效，分析结果可以用于常量合并，常量传播等优化，进一步可以消除一些代码。但有时这些优化不是程序所需要的，这时可以用volatile关键字禁止做这些优化。

volatile的本意是“易变的” 因为访问寄存器要比访问内存单元快的多,所以编译器一般都会作减少存取内存的优化，但有可能会读脏数据。当要求使用volatile声明变量值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。精确地说就是，遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问；如果不使用valatile，则编译器将对所声明的语句进行优化。（简洁的说就是：volatile关键词影响编译器编译的结果，用volatile声明的变量表示该变量随时可能发生变化，与该变量有关的运算，不要进行编译优化，以免出错）

#### i++与++i的重载函数

class CInt {

public:

CInt() {};

CInt(int c):m\_value(c) {};

~CInt() {};

public:

int Value(){return m\_value;}

CInt operator++() {

m\_value++;

return \*this;

}

CInt operator++(int) {

CInt node(m\_value);

m\_value++;

return node;

}

private:

int m\_value;

};

后置++ 通常写法中加一个int类型。为什么？

自定义类型中 后置++会产生临时变量。效率不高。

#### STL

C++ STL（标准模板库）是一套功能强大的 C++ 模板类

|  |  |
| --- | --- |
| **组件** | **描述** |
| 容器（Containers） | 容器是用来管理某一类对象的集合。C++ 提供了各种不同类型的容器，比如 deque、list、vector、map 等。 |
| 算法（Algorithms） | 算法作用于容器。它们提供了执行各种操作的方式，包括对容器内容执行初始化、排序、搜索和转换等操作。 |
| 迭代器（iterators） | 迭代器用于遍历对象集合的元素。这些集合可能是容器，也可能是容器的子集。 |

**1、顺序性容器**

**vector**

vector是一种动态数组，在内存中具有连续的存储空间，支持快速随机访问.

vector 内存管理

vector会配置比其所容纳的元素所需更多的内存，这一特性可以很大程度上提高vector的性能。vector中有一个与大小有关的函数capacity()，它返回vector实际能够容纳的元素数量。

1) 分配一块大小为当前容量的某个倍数的新内存，在大多数是实现中，vector和string的容量每次以2倍的次数增加。

2) 把容器的所有元素从旧的内存复制到新的内存中。

3) 析构掉内存中的对象。

4) 释放掉就内存。

**重新分配内存会大大影响vector的性能 现实中可通过两组方法**

1.使用reserve()保留适当容量:

2初始化期间就向构造函数传递附加参数，构造出足够的空间:

**内存释放**

**vector 在删除数据后并不会释放内存。**

**可使用swap删除数据**

**vector<string>(v).swap(v);**

**deque**

 deque是双向开口的连续性存储空间。虽说是连续性存储空间，但这种连续性只是表面上的，实际上它的内存是动态分配的，它在堆上分配了一块一块的动态储存区，每一块动态存储去本身是连续的，deque自身的机制把这一块一块的存储区虚拟地连在一起。

template<class \_Ty, class \_A= allocator<\_Ty> >

class deque

{

...

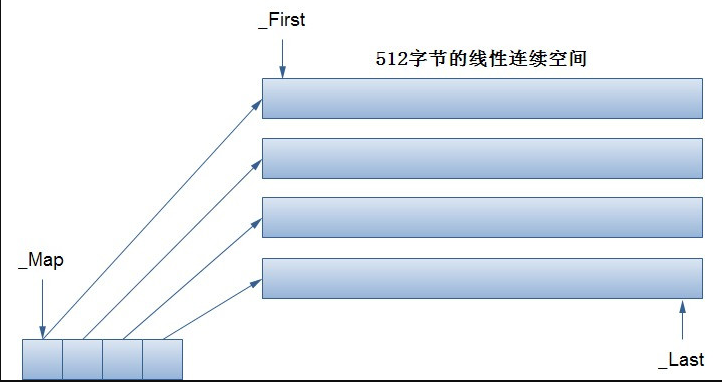
protected:

iterator \_First, \_Last;

\_Mapptr \_Map;

size\_type \_Mapsize, \_Size;

};



**list**

list是一个双向链表，因此它的内存空间是可以不连续的，通过指针来进行数据的访问，这使list的随机存储变得非常低效，因此list没有提供[]操作符的重载。但list可以很好地支持任意地方的插入和删除，只需移动相应的指针即可。

1) 如果你需要高效的随即存取，而不在乎插入和删除的效率，使用vector  
2) 如果你需要大量的插入和删除，而不关心随即存取，则应使用list  
3) 如果你需要随即存取，而且关心两端数据的插入和删除，则应使用deque

**2、关联容器**

**map**

map是一种关联容器，该容器用唯一的关键字来映射相应的值，即具有key-value功能。map内部自建一棵红黑树（一种自平衡二叉树），这棵树具有数据自动排序的功能，所以在map内部所有的数据都是有序的，以二叉树的形式进行组织。

**unordered\_map**

**内部存储采用hash表**

**set**

set也是一种关联性容器，它同map一样，底层使用红黑树实现，插入删除操作时仅仅移动指针即可，不涉及内存的移动和拷贝，所以效率比较高。set中的元素都是唯一的，而且默认情况下会对元素进行升序排列。所以在set中，不能直接改变元素值，因为那样会打乱原本正确的顺序，要改变元素值必须先删除旧元素，再插入新元素。不提供直接存取元素的任何操作函数，只能通过迭代器进行间接存取。

**3、容器适配器**

**queue**

queue是一个队列，实现先进先出功能，queue不是标准的STL容器，却以标准的STL容器为基础。queue是在deque的基础上封装的。之所以选择deque而不选择vector是因为deque在删除元素的时候释放空间，同时在重新申请空间的时候无需拷贝所有元素。

**stack**

stack是实现先进后出的功能，和queue一样，也是内部封装了deque，这也是为啥称为容器适配器的原因吧（纯属猜测）。自己不直接维护被控序列的模板类，而是它存储的容器对象来为它实现所有的功能。stack的源代码原理和实现方式均跟queue相同。

**priority\_queue 优先队列**

**例如 实现最大堆**

**priority\_queue<int, vector<int>, less<int>>**

#### [C++中的空类，默认产生哪些类成员函数？](http://www.cnblogs.com/ltang/archive/2010/10/08/1861145.html)

class Empty

{

  public:

      Empty(); // 缺省构造函数

      Empty( const Empty& ); // 拷贝构造函数

      ~Empty(); // 析构函数

       Empty& operator=( const Empty& ); // 赋值运算符

       Empty\* operator&(); // 取址运算符

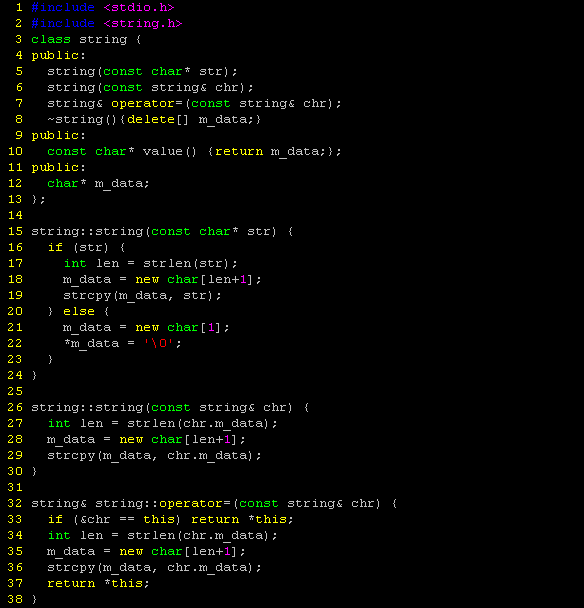
       const Empty\* operator&() const; // 取址运算符 const

};

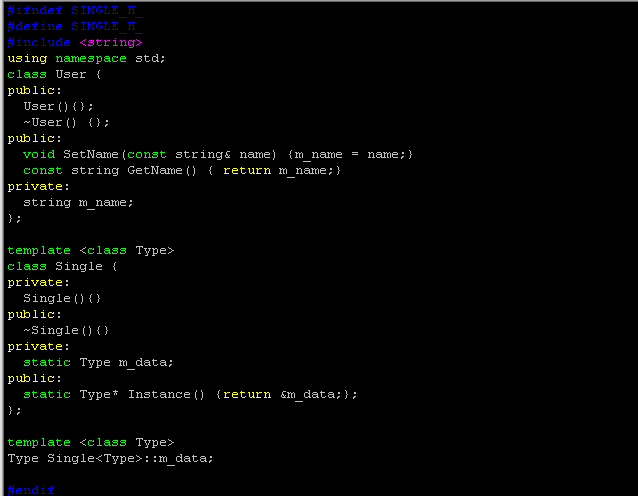
默认构造函数   
析构函数   
拷贝构造函数   
赋值运算符（operator=）   
取址运算符（operator&）（一对，一个非const的，一个const的）

**(未完待续)**

**string**

****

#### ****模板单例****



## MySQL redis篇

### Mysql

select sum(price), count(1), count(distinct(uid)) from table where....

查询 总消费价格，PV ，UV

#### 从B 树、B+ 树、B\* 树谈到R 树

http://blog.csdn.net/v\_july\_v/article/details/6530142

#### 认识mysql 索引

**MySQL索引背后的数据结构及算法原理**

http://www.uml.org.cn/sjjm/201107145.asp

#### ****MYSQL主从****

 mysql支持的复制类型

1:基于语句的复制： 在主服务器上执行的SQL语句，在从服务器上执行同样的语句。MySQL默认采用基于语句的复制，效率比较高。一旦发现没法精确复制时，会自动选着基于行的复制。

2：基于行的复制：把改变的内容复制过去，而不是把命令在从服务器上执行一遍. 从mysql5.0开始支持

3：混合类型的复制: 默认采用基于语句的复制，一旦发现基于语句的无法精确的复制时，就会采用基于行的复制。

**复制如何工作**

        整体上来说，复制有3个步骤：

       (1)    master将改变记录到二进制日志(binary log)中（这些记录叫做二进制日志事件，binary log events）；

       (2)    slave将master的binary log events拷贝到它的中继日志(relay log)；

 (3)    slave重做中继日志中的事件，将改变反映它自己的数据。



#### 脏读、幻读和不可重复读 + 事务隔离级别

1. 脏读 ：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。  
 e.g.  
        1.Mary的原工资为1000, 财务人员将Mary的工资改为了8000(但未提交事务)  
        2.Mary读取自己的工资 ,发现自己的工资变为了8000，欢天喜地！  
        3.而财务发现操作有误，回滚了事务,Mary的工资又变为了1000  
          像这样,Mary记取的工资数8000是一个脏数据。

2. 不可重复读 ：是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。  
    e.g.  
    1.在事务1中，Mary 读取了自己的工资为1000,操作并没有完成  
    2.在事务2中，这时财务人员修改了Mary的工资为2000,并提交了事务.  
    3.在事务1中，Mary 再次读取自己的工资时，工资变为了2000

 解决办法：如果只有在修改事务完全提交之后才可以读取数据，则可以避免该问题。

3. 幻读 : 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。  
   e.g.   
   目前工资为1000的员工有10人。  
   1.事务1,读取所有工资为1000的员工。  
   2.这时事务2向employee表插入了一条员工记录，工资也为1000  
   3.事务1再次读取所有工资为1000的员工 共读取到了11条记录，   
   
 解决办法：如果在操作事务完成数据处理之前，任何其他事务都不可以添加新数据，则可避免该问题

**不可重复读的重点是修改** :  
 同样的条件, 你读取过的数据,再次读取出来发现值不一样了  
**幻读的重点在于新增或者删除** 同样的条件, 第 1 次和第 2 次读出来的记录数不一样

 2、在一个程序中，依据**事务的隔离级别**将会有三种情况发生。  
　　  
　　◆脏读：**一个事务会读进还没有被另一个事务提交的数据，所以你会看到一些最后被另一个事务回滚掉的数据。**　　◆不可重复读：**一个事务读进一条记录，另一个事务更改了这条记录并提交完毕，这时候第一个事务再次读这条记录时，它已经改变了**。  
  
　　◆ 幻影读：**一个事务用Where子句来检索一个表的数据，另一个事务插入一条新的记录，并且符合Where条件，这样，第一个事务用同一个where条件来检索数据后，就会多出一条记录**。

3、数据库提供了四种事务隔离级别, 不同的隔离级别采用不同的锁类开来实现.   
在四种隔离级别中, Serializable的级别最高, Read Uncommited级别最低.   
大多数数据库的默认隔离级别为: Read Commited,如Sql Server , Oracle.   
少数数据库默认的隔离级别为Repeatable Read, 如MySQL InnoDB存储引擎



ReadCommitted：   
假设A事务对正在读取数据Data放置了共享锁，那么Data不能被其它事务改写，所以当B事务对Data进行读取时总和A读取的Data数据是一致的，所以避免了脏读。由于在A没有提交之前可以对Data进行改写，那么B读取到的某个值可能会在其读取后被A更改从而导致了该值不能被重复取得；或者当B再次用相同的where字句时得到了和前一次不一样数据的结果集，也就是幻像数据。

ReadUncommitted：  
假设A事务即不发布共享锁，也不接受独占锁，那么并发的B或者其它事务可以改写A事务读取的数据，那么并发的C事务读取到的数据的状态和A的或者B的数据都可能不一致，那么。脏读、不可重复读、幻象数据都可能存在。

RepeatableRead：  
（注意MSDN原文中的第一句话：在查询中使用的所有数据上放置锁，所以不存在脏读的情况）。  
假设A事务对读取的所有数据Data放置了锁，以阻止其它事务对Data的更改，在A没有提交之前，新的并发事务读取到的数据如果存在于Data中，那么该数据的状态和A事务中的数据是一致的，从而避免了不可重复的读取。但在A事务没有结束之前，B事务可以插入新记录到Data所在的表中，那么其它事务再次用相同的where字句查询时，得到的结果数可能上一次的不一致，也就是幻像数据。

Serializable：  
 在数据表上放置了排他锁，以防止在事务完成之前由其他用户更新行或向数据集中插入行，这是最严格的锁。它防止了脏读、不可重复读取和幻象数据。

#### mysql 注入

**1、SQL注入简介**

SQL注入是比较常见的网络攻击方式之一，它不是利用操作系统的BUG来实现攻击，而是针对程序员编程时的疏忽，通过SQL语句，实现无帐号登录，甚至篡改数据库。

**2、SQL注入攻击的总体思路**

1.寻找到SQL注入的位置

2.判断服务器类型和后台数据库类型

3.针对不通的服务器和数据库特点进行SQL注入攻击

**3、SQL注入攻击实例**

1） or 1 = 1

2）单引号 (') 双引号 (") 反斜杠 (\) NULL。

3）符串转为16进制编码数据或使用char函数进行转化

#### [MYSQL--事务处理](http://www.cnblogs.com/in-loading/archive/2012/02/21/2361702.html)

事务由一条或者多条sql语句组成，在事务中的操作，这些sql语句要么都执行，要么都不执行，这就是事务的目的。

对于事务而言，它需要满足ACID特性，下面就简要的说说事务的ACID特性。

1. A，表示原子性；原子性指整个数据库事务是不可分割的工作单位。只有使事务中所有的数据库操作都执行成功，整个事务的执行才算成功。事务中任何一个sql语句执行失败，那么已经执行成功的sql语句也必须撤销，数据库状态应该退回到执行事务前的状态；
2. C，表示一致性；也就是说一致性指事务将数据库从一种状态转变为另一种一致的状态，在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏；
3. I，表示隔离性；隔离性也叫做并发控制、可串行化或者锁。事务的隔离性要求每个读写事务的对象与其它事务的操作对象能相互分离，即该事务提交前对其它事务都不可见，这通常使用锁来实现；
4. D，持久性，表示事务一旦提交了，其结果就是永久性的，也就是数据就已经写入到数据库了，如果发生了宕机等事故，数据库也能将数据恢复。

### ****redis****

reis 持久化方式

#### 1）RDB持久化

工作原理

* Redis调用fork()，产生一个子进程。
* 子进程把数据写到一个临时的RDB文件。
* 当子进程写完新的RDB文件后，把旧的RDB文件替换掉。

优点

* RDB文件是一个很简洁的单文件，它保存了某个时间点的Redis数据，很适合用于做备份。你可以设定一个时间点对RDB文件进行归档，这样就能在需要的时候很轻易的把数据恢复到不同的版本。
* 基于上面所描述的特性，RDB很适合用于灾备。单文件很方便就能传输到远程的服务器上。
* RDB的性能很好，需要进行持久化时，主进程会fork一个子进程出来，然后把持久化的工作交给子进程，自己不会有相关的I/O操作。
* 比起AOF，在数据量比较大的情况下，RDB的启动速度更快。

缺点

* RDB容易造成数据的丢失。假设每5分钟保存一次快照，如果Redis因为某些原因不能正常工作，那么从上次产生快照到Redis出现问题这段时间的数据就会丢失了。
* RDB使用fork()产生子进程进行数据的持久化，如果数据比较大的话可能就会花费点时间，造成Redis停止服务几毫秒。如果数据量很大且CPU性能不是很好的时候，停止服务的时间甚至会到1秒。

**RDB 持久化相当于某一时间点的快照**

#### 2）AOF持久化

快照并不是很可靠。如果你的电脑突然宕机了，或者电源断了，又或者不小心杀掉了进程，那么最新的数据就会丢失。而AOF文件则提供了一种更为可靠的持久化方式。每当Redis接受到会修改数据集的命令时，就会把命令追加到AOF文件里，当你重启Redis时，AOF里的命令会被重新执行一次，重建数据。

优点

* 比RDB可靠。你可以制定不同的fsync策略：不进行fsync、每秒fsync一次和每次查询进行fsync。默认是每秒fsync一次。这意味着你最多丢失一秒钟的数据。
* AOF日志文件是一个纯追加的文件。就算是遇到突然停电的情况，也不会出现日志的定位或者损坏问题。甚至如果因为某些原因（例如磁盘满了）命令只写了一半到日志文件里，我们也可以用redis-check-aof这个工具很简单的进行修复。
* 当AOF文件太大时，Redis会自动在后台进行重写。重写很安全， ，同时Redis会继续往旧的文件追加数据。新文件上会写入能重建当前数据集的最小操作命令的集合。当新文件重写完，Redis会把新旧文件进行切换，然后开始把数据写到新文件上。
* AOF把操作命令以简单易懂的格式一条接一条的保存在文件里，很容易导出来用于恢复数据。例如我们不小心用FLUSHALL命令把所有数据刷掉了，只要文件没有被重写，我们可以把服务停掉，把最后那条命令删掉，然后重启服务，这样就能把被刷掉的数据恢复回来。

缺点

* 在相同的数据集下，AOF文件的大小一般会比RDB文件大。
* 在某些fsync策略下，AOF的速度会比RDB慢。通常fsync设置为每秒一次就能获得比较高的性能，而在禁止fsync的情况下速度可以达到RDB的水平。
* 在过去曾经发现一些很罕见的BUG导致使用AOF重建的数据跟原数据不一致的问题。

#### redis 数据结构

String——字符串  
Hash——字典  
List——列表  
Set——集合  
Sorted Set——有序集合

## shell python

shell 创建进程

shell脚本里实现并发操作，需要用到

1. for循环
2. &后台运行符
3. wait等待所有子进程结束

argc == > $?

argv[0]$ == >?

如果将命令行下输入的命令用()括号括起来，那么也会fork出一个子Shell执行小括号中的命令，一行中可以输入由分号;隔开的多个命令

(cd ..;ls -l)

### AWK

awk BEGIN{// 初始化 } {// 运行}END {// 统计结果}