·开　篇·

哈尔滨工业大学 计算机科学与技术学院 孙志岗

“计算机科学”是什么？凡混 IT 这碗饭的，恐怕除了做市场、搞管理的（剩下的也就是做技术、搞研究的了），都会思考这个问题。如果您还没思考过，那么就马上想想吧，或者立刻阅读这份刊物，也许它会给你一些思考。

对科学的追求本应该是非功利的，尽管现在评价一个科学家的尺度往往是看拿了多少奖、戴多少头衔、有多少科研经费。对于“计算机科学”，我辈才疏，不敢冒然下定义。但我们深知，真正的科学应该是一种能经受时间考验的东西，它不应该是流行元素，不应该代表时尚，不应该被炒作。当然，一门科学在转变为生产力的时候往往成为“热门”，获得更多的关注，从而发挥更大的功效。但那些不热的科学，那些鲜有人过问的科学，可能才真的能代表未来，可能才真的是热门科学的基石，可能才真的是纯粹的科学。

纯 C 论坛的宗旨，是“追求纯粹的 Computer Science”。纯 C 论坛的电子刊物，也是这个宗旨。当四方诸神正在吸引眼球、创造价值之时，我等凡夫俗子空凭一腔热情、三分才气，曲高和寡也罢，默默无闻也罢，尽倾所有，妄图撑起一片天空，广引天下同道，同心协力，给计算机科学一个不华丽、不浮躁、不排他

的生存空间。

也许无人喝彩，也许窘迫沮丧，也许黔驴技穷，也许热情不再。但我们今天在这里，我们今天有渴望、有梦想，我们今天就要实现理想！

计算机科学，万岁！

目　录

工欲善其事，必先利其器

lex 和 yacc 工具介绍

哈尔滨工业大学 信息检索实验室 高立琦

在编译过程中，词法分析和语法分析是两个重要阶段。lex 和 yacc 是 Unix 环境下非常著名的两个工具，可以生成分别完成词法分析和语法分析功能的 C 代码。在学习编译原理过程中，可以善加利用这两个工具，有利于加深对这两个阶段的理解。即使在平时的工作中，这两个工具也能发挥重要的作用。

1. 词法分词器生成工具 lex

Lex 是 Lexical compiler 的缩写，主要功能是生成一个词法分析器(scanner)的 C 源码。描述词法分析器的文件，经过 lex 编译后，生成一个 lex.yy.c 的文件，然后由 C 编译器编译生成一个词法分析器。词法分析器，简单来说，其任务就是将输入的各种符号，转化成相应的标识符(token)，转化后的标识符很容被后续阶段处理。过程如图 1。

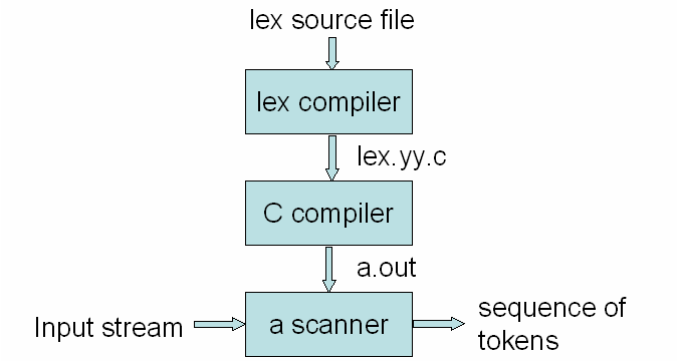


图 1

先让我们来看一个简单的例子：

```
        int num_lines = 0, num_chars = 0;

%%
\n      {++num_lines; ++num_chars;}
.       {++num_chars;}
%%
main()
{
        yylex();
        printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

例 1 sample1.l

然后编译，输入一个文本试试：

```
$ flex sample1.l
$ mv lex.yy.c sample1.c
```

```
$ gcc sample1.c -o sample1 -ll
$ ./sample1 <sample.txt
# of lines = 4, # of chars = 225
```

不错哦，没多少代码就可以实现一个行数统计程序。让我们再加点功能。

```
        int num_lines = 0, num_chars = 0, num_words = 0;

%%
\n              {++num_lines; }
[A-Za-z]*       {++num_words; }
.               {++num_chars; }

%%
main()
{
        yylex();
        printf("lines:%d    \tchars:%d\twords:%d\n",    num_lines,    num_chars,
num_words);
}
```

例 2 sample2.l

现在这个 lex 文件可以用来生成一个统计行数、字符个数和单词个数的工具。

让我们来仔细研究一下这个奇妙的工具吧。先看看 Lex 文件的结构。

Lex 文件结构简单，分为三个部分：

declarations	声明
%%	%%
translation rules	转换规则
%%	%%
auxiliary procedures	其它函数

声明段包括变量的声明、符号常量的声明和正则表达式声明。希望出现在目标 C 源码中的代码，用%{...%}扩在一起。比如：

```
%{
#include <stdio.h>
#include "y.tab.h"
typedef char * YYSTYPE;
char  *  yylval;%}
```

正则表达式声明如下

```
/* regular definitions */
delim   [ \t\n]
ws      {delim}+
letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}{digit})*
number  {digit}+(\.{digit}+)?(E[+\-]?{digit}+}?
```

这段正则表达式描述识别数(number)、标识符(id)的“规则”。过一会儿我们再细说正则表达式。

这里要注意几点：…

规则段是由正则表达式和相应的动作组成的。

p_1　　{$action_1$}
p_2　　{$action_2$}
……
p_n　　{$action_n$}

值得注意的是，lex 依次尝试每一个规则，尽可能地匹配最长的输入流。如果有一些内容根本不匹配任何规则，那么 lex 将只是把它拷贝到标准输出。比如

```
%%
A       {printf("you");}
AA      {printf("love ");}
AAAA    {printf("I ");}
%%
```

编译后运行一下，

```
$ ./sample3
AAAAAAA
I love you
```

可以看出 lex 的确按照最长的规则匹配。

程序段部分放一些扫描器的其它模块，比如一些动作执行时需要的模块。也可以在另一个程序文件中编写，最后再链接到一起。

生成 C 代码后，需用 C 的编译器编译。连接时需要指定链接库。gcc 的连接参数为 -ll。

2. 正则表达式

正则表达式可以描述有穷状态自动机(Finite Automata)接受的语言，也就是定义一个可以接受的串的集合。限于篇幅，我们就不展开关于这方面的话题了。有兴趣的请参考[4]。这里只介绍一下 lex 中用到的正则表达式的一些规则。

转义字符（也称操作符）：

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

这些符号有特殊含义，不能用来匹配自身。如果需要匹配的话，可以通过引号(")或者转义符号(\)来指示。比如

C"++"

C\+\+

都可以匹配 C++。

非转义字符：所有除了转义字符之外的字符都是非转义字符。一个非转义字符可以匹配自身。比如

integer

匹配文本中出现的 integer。

通配符：通配符就是"."（dot），可以匹配任何一个字符。

字符集：用一对[]指定的字符构成一个字符集。比如[abc]表示一个字符集，可以匹配 a、b、c 中的任意一个字符。使用 – 可以指定范围。比如[a-z]表示可以匹配所有小写字母的字符集。

重复：

*	任意次重复
+	至少一次的重复，相当于 xx*
?	零次或者一次

选择和分组：|符号表示选择，二者则一。比如(ab|cd)匹配 ab 或者 cd。

3. 文法分析器生成工具 yacc

简单来说，yacc(Yet Another Compiler-Compiler)就是编译器的编译器。Yacc 是一个通用的工具，能够根据用户指定的规则，生成一个词法分析程序。yacc 能识别 LALR(1)且无歧义的文法，它的输入是词法分析器的输出。我们知道，生成词法分析器是 lex 分内的事，因此 lex 和 yacc 常常珠联璧合。

先让我们看一下 yacc 文件的格式。和前面介绍的 lex 的格式类似：

declarations	声明
%%	%%
rules	规则
%%	%%
programs	其它程序

其中**声明段**声明一些符号常量，可以为空。同 lex 一样，声明段中可以有出现在目标 C 程序中的代码，放在%{…%}中；还有一些 yacc 关键词可以指示出 token 的结合顺序：

%left	左结合
%right	右结合
%nonassoc	不结合
%token	声明 token

俗话说“没有规矩，不成方圆”。 **规则段**描述规则，自然是重中之重了。规则段的结构是如下，

```
A : BODY ;
```

A 表示非终结符名，BODY 表示产生式和动作。产生式包括非终结符和终结符，终结符用”引用。一些转义字符，比如’\r’,’\n’等，和 C 里面的表示是一样的。动作(action)则是在输入被当前规则识别出来时而执行的。动作实际上就是 C 的代码，写在{ }中。为了沟通词法分析器和动作，yacc 引入了形式变量，以$开头。如果希望获得词法分析器和前面的动作返回的值，我们可以使用$1,$2,…。$*i* 表示一条规则右侧第 *i* 个单元的值。比如有这样的一条规则，

```
A : B C D ;
```

C 的返回值为$2，D 为$3。依此类推。

程序段放一些其它的程序，也可以省略，连%%都可以不要。

连接时需要指定连接库，gcc 的参数为-ly。

让我们看一个经典的例子，它实现一个简单的计算器：

```
%{
#  include  <stdio.h>
#  include  <ctype.h>
int  regs[26];
int  base;
%}

%start  list

%token  DIGIT  LETTER

%left  '|'
%left  '&'
%left  '+'  '-'
```

```
%left '*' '/' '%'
%left UMINUS     /* supplies precedence for unary minus */

%%      /* beginning of rules section */

list    :   /* empty */
    |   list stat '\n'
    |   list error '\n'
            {   yyerrok; }
    ;
stat    :   expr
            {   printf( "=%d\n", $1 ); }
    |   LETTER '=' expr
            {   regs[$1] = $3; }
    ;

expr    :   '(' expr ')'
            {   $$ = $2; }
    |   expr '+' expr
            {   $$ = $1 + $3; }
    |   expr '-' expr
            {   $$ = $1 - $3; }
    |   expr '*' expr
            {   $$ = $1 * $3; }
    |   expr '/' expr
            {   $$ = $1 / $3; }
    |   expr '%' expr
            {   $$ = $1 % $3; }
    |   expr '&' expr
            {   $$ = $1 & $3; }
    |   expr '|' expr
            {   $$ = $1 | $3; }
    |   '-' expr        %prec UMINUS
            {   $$ = - $2; }
    |   LETTER
            {   $$ = regs[$1]; }
    |   number
    ;
number  :   DIGIT
            {   $$ = $1;    base = ($1==0) ? 8 : 10; }
    |   number DIGIT
            {   $$ = base * $1 + $2; }
    ;
%%      /* start of programs */

yylex() {   /* lexical analysis routine */
/* returns LETTER for a lower case letter, yylval = 0 through 25 */
/* return DIGIT for a digit, yylval = 0 through 9 */
/* all other characters are returned immediately */

    int c;
    while( (c=getchar()) == ' ' ) {    /* skip blanks */ }
    /* c is now nonblank */
    if( islower( c ) ) {
        yylval = c - 'a';
        return ( LETTER );
        }
    if( isdigit( c ) ) {
```

```
        yylval  =  c  -  '0';
        return(  DIGIT  );
        }
    return(  c  );
    }
```

例 3 example.y

编译、执行：

```
$bison example.y
$gcc example.tab.c  -ly -o example
$./example
20+30*50
=1520
```

4. 小结

lex 是词法分析器的生成工具，yacc 是文法分析器的生成工具。lex 的描述规则采用正则表达式，关于正则表达式的详细讨论，参见文献[1]；yacc 的描述规则采用无歧异文法，进一步讨论请参阅文献[1][4][6]。在 GNU 中有相应 lex 和 yacc 工具：flex 和 bison，与 lex 和 yacc 兼容。

参考文献

[1] Alfred V.Aho, Ravi Sethi, Jeffery D.Ullman: ***Compilers:Principles,Techniques, and Tools***.

[2] Peter Seebach: ***Build code with lex and yacc, Part 1: Introduction***, 2004.
http://www-106.ibm.com/developerworks/library/l-lexyac.html

[3] Ashish Bansal: ***Jumpstart your Yacc...and Lex too! An intro to Lex and Yacc***, 2000.
http://www-900.ibm.com/developerworks/cn/linux/sdk/lex/index_eng.shtml#author

[4] John E.Hopcroft,Rajeev Motwani,Jeffrey D.Ullman: ***Introduction to Automata Theory, Languages, and Computation***,2000.

[5] M. E. Lesk and E. Schmidt : ***Lex - A Lexical Analyzer Generator.***
http://www.combo.org/lex_yacc_page/lex.html

[6] Stephen C. Johnson : ***Yacc: Yet Another Compiler-Compiler***
http://dinosaur.compilertools.net/yacc/index.html

【全文完 • 责任编辑：worldguy@hitbbs】

ACM/ICPC 计算机算法大赛简介

哈尔滨工业大学 数据库研究中心 熊蜀光

如果真正是在计算机领域做学问的话，对 ACM 的大名一定不会陌生，ACM 的全称是 Association for Computing Machinery，建立于 1947 年，是世界上第一个教育和科研的，也是最有影响的计算机组织。今天，ACM 已经有超过 8 万个成员，遍布在世界各地。ACM 的主要活动包括一些专题的兴趣小组（SIGs Special Interesting Groups），每年要组织一系列高水平的学术会议，还有一些面向不同层次的学术竞赛，ACM/ICPC 就是其中之一。

ACM/ICPC（ACM International Collegiate Programming Contest），即 ACM 国际大学生程序设计竞赛，是由 ACM 协会提供给大学生的一个展示和提高解题与编程能力的机会。面向全世界的大学生，分为地区赛和决赛，地区赛的优胜者（通常是前两名）有资格参加决赛，决赛的颁奖仪式将和计算机界权威的学术奖——图灵奖的颁奖仪式同时进行。

ACM 竞赛有着独特的赛制，比赛是以参赛队为单位的，每队三个人，每支队伍至少有两名参赛队员必须是大学本科尚未毕业的学生，所有参赛队员的学历不可以超过研究生两年。在赛场上，为了体现团体协作精神，三个人共用一台计算机，可以携带一切书面材料。比赛时间一般是 4-5 个小时，共有 6-10 道题，按照解答的题目多少和解答所用的时间长短决定名次。

对于每道题目，参赛队必须写出解决该题的程序源代码，提交给裁判，由裁判编译得到可执行程序，如果对于所有的输入数据该程序都能在规定的时间内得到正确的结果，才能够获得通过。测试数据通常极为严格。当某个队通过了一道题时，工作人员会在这个队的计算机前插上一个代表这道题颜色的气球，这样所有比赛的情况一目了然，为了增加比赛的紧张气氛，比赛结束前一个小时，停止公布所有的成绩。

如果参赛队提交的程序代码未获通过，裁判会及时返回错误信息，大约有答案错误(Wrong Answer)，输出格式错误(Presentation Error)，超时(Time Limit Exceeded)等。

竞赛涵盖的范围很广，大致划分如下：Direct（简单题），Computational Geometry（计算几何），Number Theory（数论），Combinatorics（组合数学），Search Techniques（搜索技术），Dynamic Programming（动态规划），Graph Theory（图论），Other（其他）。

目前 ACM/ICPC 由 IBM 赞助，全球地区赛共分若干个赛区，数十个赛点。中国学生可以报名参加亚洲赛区的任何赛点的比赛(从 2004 年开始，大陆各赛点将增加一轮地区赛的预选赛)。比赛支持 C/C++，Java，Pascal 等语言。

至今哈工大已经派对参加了多届 ACM/ICPC：

1998 年，孙广坤，罗浩，陈俊第一次参加了 ACM/ICPC 上海赛区的比赛,获得赛区第 9 名的成绩。

1999 年，在实验学院柳进老师的组织下，王宏志，刘占一，张博参加了上海赛区的比赛，获得赛区第 8 名的成绩。

（图中从左起分别为：刘占一、王宏志、张博）

2001 年，实验学院组织了 ACM/ICPC 的选拔赛，经过选拔，张博，李鹏和李振国赴上海参加比赛，获得赛区第 8 名。

2003 年，季检，肖颖和我赴北京参赛，获得赛区第 8 名。

（上图中从左起分别为：肖颖、我、季检）

如果你对计算机程序设计有浓厚的兴趣，不妨加入到 ACM/ICPC 中来！当然，首先要求的是必须熟练掌握一门计算机语言，因为比赛中的时间大部分不是用来写代码，而是用于思考算法和修改调试的。然后，还应该具备基本的数据结构和算法基础，任何一本数据结构教材都可以作参考资料，此外，推荐的读物有：

刘汝佳，黄亮：《算法艺术与信息学竞赛》

陈志平，徐宗本：《计算机数学》

王晓东：《计算算法设计与分析》

William Ford：《数据结构 C++语言描述》

吴文虎，王建德：《实用算法的分析与程序设计》

然后就是进行实战训练了，网上练习的方式是一个好办法，将源代码提交给 Online Judge 服务器，由服务器端的程序编译执行再判断是否通过，然后将结果返回。哈工大目前有一套 Online Judge 系统，详情请见：http://acm.hit.edu.cn

还有一些其它不错的网上资源：

- ICPC 官方网站：http://icpc.baylor.edu/icpc/
- 在线评测系统(Online Judge)

http://acm.uva.es/

http://acm.zju.edu.cn/

http://acm.pku.edu.cn/

http://acm.jlu.edu.cn/

http://acm.timus.ru/

http://ace.delos.com/usacogate/

- 讨论区

http://oibh.ioiforum.org/

http://bbs.hit.edu.cn/cgi-bin/bbs2/bbsdoc?board=Algorithm

感谢谢煜波，感谢王宏志和肖颖提供的资料

【全文完 • 责任编辑：xiong@hitbbs】

ACM/ICPC 试题解析

哈尔滨工业大学 数据库研究中心 熊蜀光

孔子和耶稣都说过，算法是程序设计的灵魂（开个玩笑），由此可见算法理论的重要性，然而我们也不能纸上谈兵，必须动手实践，这对于编程和思维能力都会有好处。希望本栏目能对大家有所帮助（编者按：同时也迫切地希望大家多多投稿，多提意见）。好了，闲话休提，让我们来看看一个纯算法设计的题目吧。这道题来自浙江大学的 ACM/ICPC 在线评判网站(http://acm.zju.edu.cn)，关于 ACM/ICPC 的介绍请见后文。

Fence

Time limit: 1 Seconds Memory limit: 32768K
Total Submit: 532 Accepted Submit: 134

Workers are going to enclose a new working region with a fence. For their convenience the enclosed area has to be as large as possible. They have N rectangular blocks to build the fence. The length of the i-th block is Li meters. All blocks have the same height of 1 meter. The workers are not allowed to break blocks into parts. All blocks must be used to build the fence.

Input

The first line of the input file contains one integer N (3 <= N <= 100). The following N lines describe fence blocks. Each block is represented by its length in meters (integer number, 1 <= Li <= 100). Process to the end of file.

Output

Write to the output file one non-negative number S - maximal possible area of the working region (in square meters). S must be written with two digits after the decimal point. If it is not possible to construct the fence from the specified blocks, write 0.00.

Sample Input

```
4
10
5
5
4
3
8
5
```

5
3
10
5
4
Sample Output

28.00
12.00
0.00

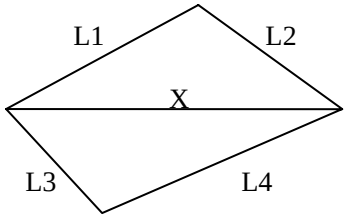
这是一道几何题，已知 N 条线段，每条线段长度为 Li，求它们能围成的多边形的最大面积，并精确到小数点后两位输出，如果不能围成多边形，则直接输出 0.00。

分析：

初看这道题目，似乎无从下手，我们首先对简单情况讨论。

对于 3 条线段，则围成的面积是固定的，只需判定短的两边之和是否大于第三边，然后根据海伦公式可求出面积。

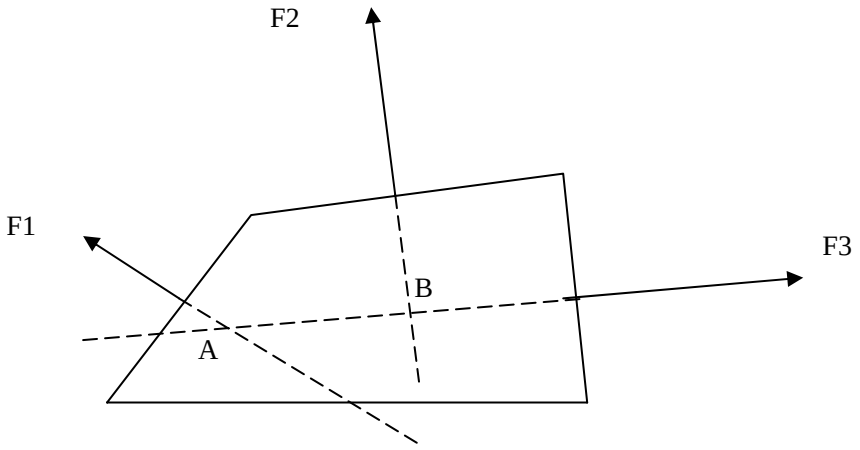
对于 4 条线段，显然能围成四边形的充要条件是最长边小于其余边长的和，那么如果满足此条件，又如何确定面积的最大值呢？



图一

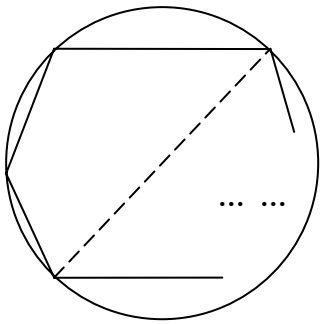
设其中的一条对角线长为 X(图一)，显然四边形面积可以表示为 X 的函数，再对该函数求导得 0 建立方程，可求出 X 的值和最大面积的数值解，但是用这种方法求解时表达式过于繁琐，何况我们只需求出近似解即可，所以考虑其它的方法。

假想由 4 条边围成的一个很薄的盒子，顶点处可以自由转动，里面充满气体，盒子外面为真空，那么由于内部气体的压力，盒子会发生形变，最终会处于静力平衡状态，那么在这个状态下，气体的压强达到可能的最低，倘若不是，那么气体必然会向压强减小的方向膨胀。由于气体的质量不变，所以此时的盒子面积最大。在这种情况下，由于气体的各处压强相同，每边受力是均匀的，所以可以将受力转化为一个作用在各边中点，并垂直于相应边的力。因为不可能出现三边都平行的情况，所以必能找到三条边，使它们的中垂线两两相交，设作用在这三边的力分别为 F1,F2,F3，固定剩下的一边于地面上，设 F1 与 F3 的交点为 A，F2 与 F3 的交点为 B(图二)，对 A 点取力矩，则 F1 和 F3 作用在 A 点的力矩均为 0，F2 作用在 A 点的力矩不为 0，所以不能达到平衡，所以 A 点和 B 点必定重合。所以此三边的中垂线交于一点，所以这个四边形的顶点共圆。



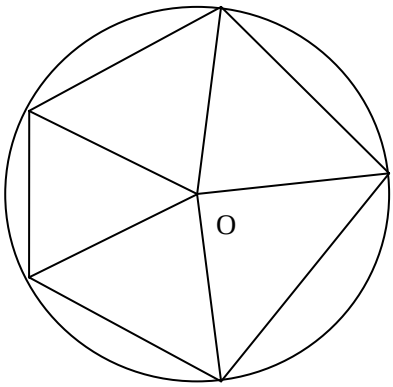
图二

对于 N 条线段，假设它们已经围成了一个面积最大的多边形，那么取相邻的 4 个顶点围成的四边形(图三)，必然也是这 4 边围成的四边形中面积最大的，否则我们还能由这 4 边围成一个面积更大的四边形，那么所形成的新多边形面积大于目前的多边形，矛盾。所以这 4 个相邻顶点共圆，那么接下来的 4 个顶点也共圆，由 3 点确定一个圆可知，这 5 点共圆。因此，所有的 N 个点共圆。

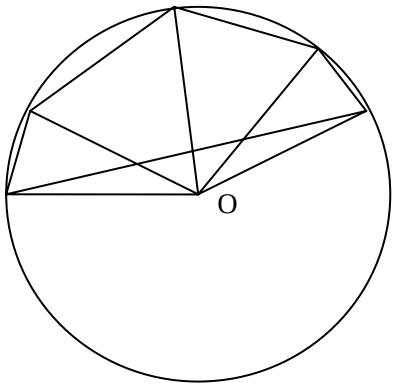


图三

现在就可以有很多方法求面积了，我们采用先求外接圆的半径，再利用海伦公式求各个以半径分割的三角形面积，再求和得到多边形面积的方法(图四)。



图四



图五

求半径可以采用解析方法，但是同样繁琐，我们采用二分逼近的数值方法。首先确定半径的范围，显然大于等于最大边长的 1/2，小于线段长度和的 1/2(当然还可以更小，这里只

考虑了能够围成多边形这一基本条件)。首先令 R 为最大边长的 1/2，要考虑三种情况：圆心在多边形内部(图四)，圆心在多边形外部(图五)，圆心是最大边的中点。可以用其余边对 R 的圆心角之和与π来判断：如果大于，则圆心必定在多边形内部，如若不然，因为实际的半径大于等于 R，而圆心在多边形外部，所以其余边对实际半径的圆心角必小于π，矛盾。同理可证圆心在多边形外部的情况。圆心是最大边中点的情况可直接求出半径。在这个范围内以半径 R 作迭代，R 每次取上下界的中值。

圆心在多边形内部时，由于 R 与圆心角之和增减性相同，所以到算出的所有边的圆心角之和在误差范围内等于 2*π停止，此时的 R 就是所求半径。再根据半径求面积即可。
圆心在多边形外部时,倘若除去最长边的其余各边的圆心角 sum 的和小于最长边的圆心角 a，则根据圆心角的变化规律不难看出，所求半径比当前半径 R 要小；否则所求半径比当前半径 R 大，到算出 sum 与 a 在误差范围内相等时停止。面积为除去最长边的其余各边所对应的三角形面积和减去最长边为底边的三角形面积。

注：2003 年 ACM/ICPC 北京赛区的 B 题跟此题类似。

附：程序源代码

```
///                        求 n 条已知长度的线段围成的最大面积                    ///
///                                                                          ///
#include "stdio.h"
#include "iostream.h"
#include "math.h"
#include <algorithm>
using namespace std;

const double pi=3.1415926535898;
const double small=0.00000000001;
double l[200];
int n;

double halen(double a,double b,double c)
{
    double s=(a+b+c)/2.0;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

double angle_small(double r)
{
    int i;
    double angle=0.0;
    for(i=0;i<n-1;i++)
    {
        angle+=2*asin(l[i]/2.0/r);
    }
    return angle;
}
```

```
double angle_all(double r)
{
    int i;
    double angle=0.0;
    for(i=0;i<n;i++)
    {
        angle+=2*asin(l[i]/2.0/r);
    }
    return angle;
}

int main(void)
{
    int i;
    double r,max,min,sum,area;
    while(cin>>n)
    {
        sum=0.0;
        for(i=0;i<n;i++)
        {
            cin>>l[i];
            sum+=l[i];
        }
        sort(l,l+n);
        if(sum-l[n-1]<l[n-1])
        {
            printf("0.00\n");
            continue;
        }
        min=l[n-1]/2.0;max=sum/2.0;area=0.0;
        if(angle_small(l[n-1]/2.0)>pi-small)
        {
            while(true)
            {
                r=(min+max)/2.0;
                if(angle_all(r)<2.0*pi-small)
                {
                    max=r;
                }
                else if(angle_all(r)>2.0*pi+small)
                {
                    min=r;
                }
```

```
                else
                {
                    break;
                }
            }
            for(i=0;i<n;i++)
            {
                area+=halen(l[i],r,r);
            }
        }
        else if(angle_small(l[n-1]/2.0)<pi-small)
        {
            max=1000000000;
            while(true)
            {
                r=(min+max)/2.0;
                if(angle_small(r)<2*asin(l[n-1]/2.0/r)-small)
                {
                    min=r;
                }
                else if(angle_small(r)>2*asin(l[n-1]/2.0/r)+small)
                {
                    max=r;
                }
                else
                {
                    break;
                }
            }
            for(i=0;i<n-1;i++)
            {
                area+=halen(l[i],r,r);
            }
            area-=halen(l[n-1],r,r);
        }
        else
        {
            r=l[n-1]/2.0;
            for(i=0;i<n-1;i++)
            {
                area+=halen(l[i],r,r);
            }
        }
        printf("%.2f\n",area);
```

```
    }
    return 0;
}
```

【全文完 • 责任编辑：xiong@hitbbs】

WinXP SP2 对病毒和加密技术的影响

中国安天实验室 Killer (killer@uid0.net)

Windows XP SP2 的出现，使得一些软件已经无法在 XP2 上运行了。早在早些时候微软就提醒软件开发商，Windows XP SP2 不同于以往的补丁程序或普通升级。在 SP2 中，微软更多地关注如何增强系统的安全性。微软还警告说，这次升级将放弃某些具有安全缺陷且无法修正的应用。尽管 WinXP SP2 微软说如何如何，在我们看来，微软还有一个目的就是防止盗版。抛开这些不说，拨开 WinXP SP2 的层层迷雾，我们还是看到了一些新东西，这就是可执行文件的数据段防执行(DEP, Data Execution Prevention)机制。今年上半年，微软与 AMD 曾联合宣布，WindowsXP SP2 补丁将开启 AMD64 处理器中的 Enhanced Virus Protection（增强病毒防护）技术。并称：“AMD 的 Enhanced Virus Protection 安全技术将与微软 SP2 中的 Data Execution Prevention 技术相结合，可以监测出已知的病毒，尤其对那些缓冲区溢出病毒以及传播速度快的病毒有很好的抑制效果。”而 AMD Enhanced Virus Protection 技术是通过在转换物理地址和逻辑地址的“page translation table”中增加新的比特位（NX bit）来实现，Intel 也表示未来 Itanium 和 Itanium 2 也会集成该技术。

下面我们就几种非正常的代码执行方式加以简述：

1、代码段改写：

以往的病毒、加壳软件多数利用代码段的可写特性进行一些反跟踪、代码加密操作。加壳软件在给目标加壳后，会有加壳软件对目标文件的代码段进行改写，达到的 replace code 功能，造成壳与目标文件一体的效果。

2、数据段执行：

以往的病毒、加壳软件多数利用数据段执行的办法来进行一些特殊操作。

3、堆栈段执行：

使用了缓冲区溢出的 shellcode 都是在利用这一特性。

以上的示例程序 Masm32 源代码及编译后程序在本文后面提供。经测试，在 DEP 打开的 SP2 正式版中，运行堆栈段执行演示代码时，系统弹出 DEP 提示，如图 1 所示。

（图 1）堆栈段执行演示代码在 SP2 上无法正常运行的错误提示

而微软的 DEP 技术将使得大多数自解密、采用加密壳的病毒蠕虫将无法在 WinXP SP2 上运行和发作。许多采用商业加密手段的工具也将无法正常执行使用，图 1 就是著名的商业 Sniffer 工具 IRIS 4.0 版本在 SP2 上无法正常运行的报错提示，经过检查得知它是通过一款商业保护工具 PCGUARD 4.x 进行的加密保护，我们在脱掉这个加密壳后软件得以正常执行。

（图 2）IRIS 4.0 版本在 SP2 上无法正常运行的报错提示

WinXP SP2 在初期 DEP 保护力度很大，兼容性也不另人满意，其多数保护开关都是默认打开，这使得众多应用程序难以执行。后来微软进行了改进，正式版中的 DEP 默认为关闭，而且兼容性有了很大的改进。用户可以在系统属性中手工确认是否开启 DEP 及针对何种程序进行 DEP，如图 3 所示。

（图 3）WinXP SP2 数据执行保护设置界面

现将目前可在 WinXP SP2 正常使用的常用加壳工具整理如表 1 所示，以方便大家使用。

加壳工具	版本
upx	0.81+
aspack	1.83+
fsg	1.33+
telock	0.92+
pe pack	1.0+
ezip	1.x+
dxpack	0.86+
asprotect	1.23rc1+

(表 1)可在 WinXP SP2 正常使用的常用加壳工具

代码示例：

1、代码段可写：

```
;
###########################################################################
     .386                          ; create 32 bit code
     .model flat, stdcall          ; 32 bit memory model
     option casemap :none          ; case sensitive
; 编译方式：\MASM32\BIN\Link.exe  /SECTION:.text,RWE  /SUBSYSTEM:WINDOWS
Test.obj
;
###########################################################################
;    include files
;    ~~~~~~~~~~~~~
     include \masm32\include\windows.inc
     include \masm32\include\kernel32.inc
     include \masm32\include\user32.inc
;    libraries
;    ~~~~~~~~~
     includelib \masm32\lib\kernel32.lib
     includelib \masm32\lib\user32.lib
.code
;
###########################################################################
start:
      jmp @F
   TestWrite db 0
   @@:
   mov TestWrite, 1 ; 未使用指定编译方式，此处将抛出访问异常
       .if (TestWrite == 1)
         jmp @F
      szMsg db '代码段已被改写! ', 0
```

```
      @@:
    invoke MessageBox, NULL, ADDR szMsg ,NULL, MB_OK
   ret
end start
;
###########################################################################
```

2、数据段可执行：

```
;
###########################################################################
     .386                       ; create 32 bit code
     .model flat, stdcall       ; 32 bit memory model
     option casemap :none       ; case sensitive
; 由于 MASM32 不支持在数据段使用 call、jmp、lea 等指令，这里完全使用 push、ret 实现。
;
###########################################################################
;    include files
;    ~~~~~~~~~~~~~
   include \masm32\include\windows.inc
   include \masm32\include\kernel32.inc
;    libraries
;    ~~~~~~~~~
   includelib \masm32\lib\kernel32.lib
;
###########################################################################
.data
   TestExecute dd $ + 4
   push MB_OK
   push NULL
   push lblMSG
   push @F
   ret
   lblMSG dd szMSG
   szMSG  db '数据段已被执行! ', 0
   @@:
   push NULL
   push @F
   push MessageBox
@@:
   ret
.code
;
###########################################################################
start:
```

```
    push @F
    jmp TestExecute
@@:
    ret
end start
;
#########################################################################
```

3、堆栈段可执行：

```
#########################################################################
      .386                          ; create 32 bit code
      .model flat, stdcall          ; 32 bit memory model
      option casemap :none          ; case sensitive
;
#########################################################################
;     include files
;     ~~~~~~~~~~~~~
      include \masm32\include\windows.inc
      include \masm32\include\kernel32.inc
      include \masm32\include\user32.inc
;     libraries
;     ~~~~~~~~~
      includelib \masm32\lib\kernel32.lib
      includelib \masm32\lib\user32.lib
.data
    TestExecute dd $ + 4
    push MB_OK
    push NULL
    push lblMSG
    push @F
    ret
    lblMSG dd szMSG
    szMSG  db '堆栈段已被执行', 0
    @@:
    push NULL
    push @F
    push MessageBox
@@:
    ret
    ExecuteSize equ $ - TestExecute
.code
start:
    push esp
    mov ebp, esp
```

```
    sub esp, ExecuteSize
    mov esi , TestExecute
    mov ecx , ExecuteSize
    mov edi , esp
    rep movsb
    call esp
    mov esp, ebp
    pop ebp
    ret
end start
```

注：特别感谢 Swordlea 的大力帮助和技术指点！

【全文完 • 责任编辑：swordlea@hitbbs】

剖析 Intel IA32 架构下 C 语言及 CPU 浮点数处理机制

哈尔滨工业大学 计算机体系结构实验室 谢煜波

(xieyubo@126.com)

序

这两天翻看一本 C 语言书的时候，发现上面有一段这样写到：

例：将同一实型数分别赋值给单精度实型和双精度实型，然后打印输出。

```
/* -------------- test1.c ----------------- */
#include <stdio.h>
main()
{
    float a;
    double b;
    a = 123456.789e4;
    b = 123456.789e4;
    printf(“%f\n%f\n”,a,b);
}
```

运行结果如下：

```
1234567936.000000
1234567890.000000
```

为什么同一个实型数据赋值给 float 型变量和 double 型变量之后，输出的结果会有所不同呢？这是因为将一个实型常量赋值给 float 型变量与赋值给 double 型变量，它们所接受的有效数字位数是不同的。

这一段的说法是正确的，但实在是太模糊了！为什么一个输出的结果会比原来的大？为什么不是比原来的小？这之间到底有没有什么内在的根本性原因还是随机发生的？为什么会出现这样的情况？上面都没有对此进行解释。上面的解释是一种最普通的解释，甚至说它只是说出了现象，而并没有很深刻的解释原因，这不免让人读后觉得非常不过瘾！

书中还有下面一段：

“（1）两个整数相除的结果仍为整数，舍去小数部分的值。例如，6/4 与 6.0/4 运算的结果值是不同的，6/4 的值为整数 1，而 6.0/4 的值为实型数 1.5。这是因为当其中一个操作数为实数时，则整数与实数运算的结果为 double 型。”

非常遗憾的说，“整数与实数运算的结果为 double 型”，这样的表述是不精确的，不论从实际程序的反汇编结果，还是从对 CPU 硬件结构的分析，这样的说法都非常值得推敲。然而在很多 C 语言的教程上我们却总是常常看见这样的语句：“所有涉及实数的运算都会先转换成 double，然后再运算”。然而实际又是否是这样的呢？

关于浮点数运算这一部份，绝大多数的 C 教材没有过多的涉及，这也使得我们在使用 C 语言的时候，会产生很多疑问。

先来看看下面一段程序：

```
/* ------------a.c----------------- */
```

```
#include <stdio.h>
double f(int x)
{
        return 1.0 / x ;
}

void main()
{
        double a , b;
        int i ;
        a = f(10) ;
        b = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

这段程序使用 gcc –O2 a.c 编译后，运行它的输出结果是 0，也就是说 a 不等于 b，为什么？

再看看下面一段，几乎同上面一模一样的程序：

```
/*---------------- b.c ----------------------*/
#include <stdio.h>
double f(int x)
{
        return 1.0 / x ;
}

void main()
{
        double a , b , c;
        int i ;
        a = f(10) ;
        b = f(10) ;
        c = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

同样使用 gcc –O2 b.c 编译，而这段程序输出的结果却是 1，也就是说 a 等于 b，为什么？

国内几乎没有一本 C 语言书（至少我还没看见），解释了这个问题，在 C 语言对浮点数的处理方面，国内的 C 语言书几乎都是浅尝即止，蜻蜓点水，而国外的有些书对此就有很详尽的描述，上面的例子就是来源于国外的一本书《Computer Systems A Programmer's Perspective》（本文参考文献 2，以下简称《CSAPP》），这本书对 C 语言及 CPU 处理浮点数描写得非常细致深入，国内很多书籍明显不足的地方，就在于对于某些细节我们似乎并没有某种深入的精神，没有一定要弄个水落石出的气度，这也注定了我们很少出版一些 Bible 级的著作。一本书如果值得长期保留，能成为 Bible，那么我认为它必须把某一细节描述得非

常清楚，以至于在读了此书之后，再也不需要阅读其它的书籍，就能对此细节了如指掌。

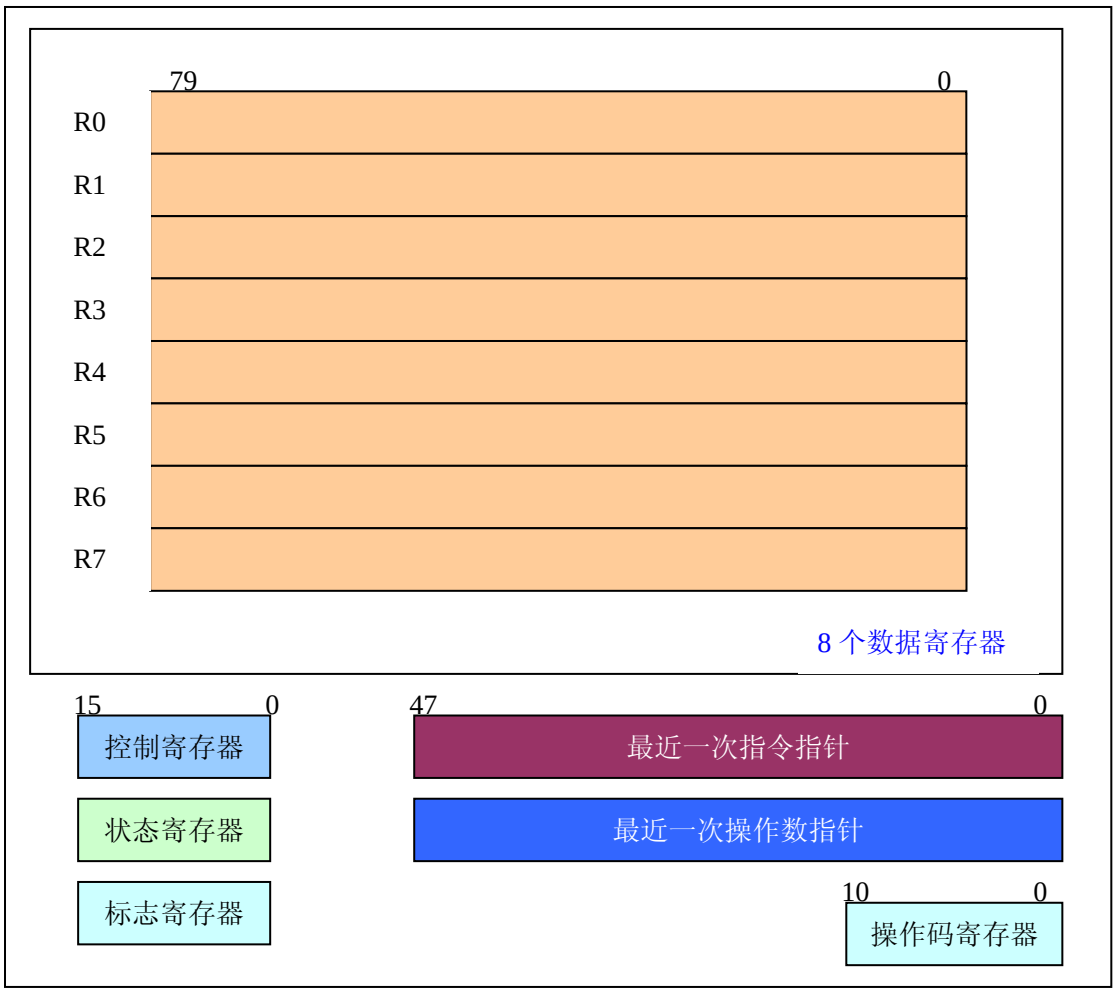
《CSAPP》这本书的确非常经典，遗憾的是此书好像目前还没有电子版，因此我打算以此书为基础（一些例子及描述就来自此书），再加上自己看过的一些其它资料，以及自己对此问题的理解与分析，详细谈一下 C 语言及 Intel CPU 对浮点数的处理，以期望在此方面，能对不清楚这部分内容的朋友们有些许帮助。

要无障碍的阅读此文，你需要对 C 语言及汇编有所了解，本文的所有实验，均基于 Linux 完成，硬件基于 Intel IA32 CPU，因此，如果你想从此文中了解更多，你最好能熟练使用 Linux 下的 gcc 及 objdump 命令行工具（非常遗憾的是，现在少有 C 语言教材会对此进行讲述）。另外，你还需要对堆栈操作有所了解，这在任何一部讲解数据结构的书上都会提到。

由于自身知识及能力有限，如果书中有描述不当的地方或错误，请你与我联系，我也会在哈工大纯 C 论坛上（http://purec.binghua.com）对所有问题进行跟踪及反馈。

一、Intel CPU 浮点运算单元的逻辑结构

在很久以前，由于 CPU 工艺的限制，无法在一个单一芯片内集成一个高性能的浮点运算器，因此，Intel 还专门开发了所谓的协处理器配合主处理器完成高性能的浮点运算，比如 80386 的协处理器就是 80387，后来由于集成电路工艺的发展，人们已经能够在一个芯片内集成更多的逻辑功能单元，因此，在 80486DX 的时候，Intel 就在 80486DX 这个芯片内集成了很强大的浮点处理单元（FPU）。下面，我们就来看看，被集成到主处理器内部之后，这个浮点处理单元的逻辑结构，这是理解 Intel CPU 浮点数处理机制的前提。



（图 1 Intel CPU 浮点处理单元逻辑结构图）

上图就是 Intel IA32 架构 CPU 浮点处理单元的逻辑结构图，从图中我们可以看出它总共有 8 个数据寄存器，每个 80 位（10B）；一个控制寄存器（Control Register），一个状态寄存器（Status Register），一个标志寄存器（Tag Register），每个 16 位（2B）；还有一个最近一次指令指针（Last Instruction Pointer），及一个最近一次操作数指针（Last Operand Pointer），每个 48 位（6B）；以及一个操作码寄存器（Opcode Register）。

状态寄存器用处与常见的主 CPU 的程序状态字差不多，用来标记运算是否溢出，是否产生错误等，最主要的一点是它还记录了 8 个数据寄存器的栈顶位置（这点在下面将会有详细描述）。

控制寄存器中最重要的就是它指定了这个浮点处理单元的舍入的方式（后面将会对此详细描述）及精度（24 位，53 位，64 位）。Intel CPU 浮点处理器的默认精度是 64 位，也称为 Double Extended Precision（中文也许会译为：双扩展精度，但这种专有名词，不译更好，译了反而感觉更别扭）。而 24 位，与 53 位的精度，是为了支持 IEEE 所定义的浮点标准（IEEE 754 标准），也就是 C 语言中的 float 与 double。

标志寄存器指出了 8 个寄存器中每个寄存器的状态，比如它们是否为空，是否可用，是否为零，是否是特殊值（比如 NaN：Not a Number）等。

最后一次指令指针寄存器与最后一次数据指针寄存器用来存放最后一条浮点指令（非控制用指令）及所用到的操作数在内存中的位置。由于包括 16 位的段选择符及 32 位的内存偏移地址，因此，这两个寄存器都是 48 位（这涉及到 Intel IA32 架构下的内存地址访问方法，如果对此不清楚的，可以不用太在意，只需知道它们指明了一个内存地址就行，如果很想弄清楚，可以参看本文的参考文献 1）。

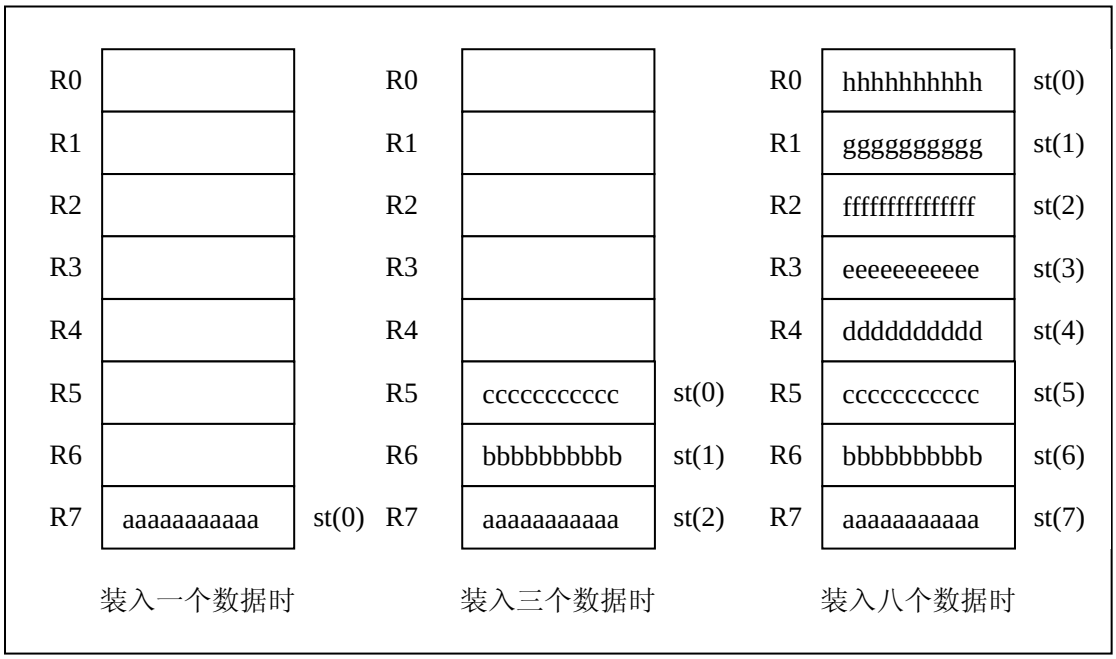
操作码寄存器记录了最后一条浮点指令（非控制用指令）的操作码，这很简单，没什么可多说的。

下面我们将详细描述一下，浮点处理单元中的那 8 个数据寄存器，它们与我们通常用的主 cpu 中的通用寄存器，比如 eax，ebx，ecx，edx 等相比有很大的不同，它们对于我们理解 Intel CPU 浮点处理机制非常关键！

二、Intel CPU 浮点运算单元浮点数据寄存器的组织

Intel CPU 浮点运算单元中浮点数据寄存器总共有 8 个，它们都是 80 位，即 10 字节的寄存器，对于每个字节所带表的含义我将在后面描述浮点数格式的时候详细介绍，这里详细介绍的将是这 8 个寄存器是怎么组织以及怎么使用的。

Intel CPU 把这 8 个浮点寄存器组织成一个堆栈，并使用了状态寄存器中的一些标志位记录了这个栈的栈顶的位置，我们把这个栈顶记为 st(0)，紧接着栈顶的下一个元素是 st(1)，再下一个是 st(2)，以此类推。由于栈的大小是 8，因此，当栈被装满的时候，可以访问的元素为 st(0)~st(7)，如下图所示：

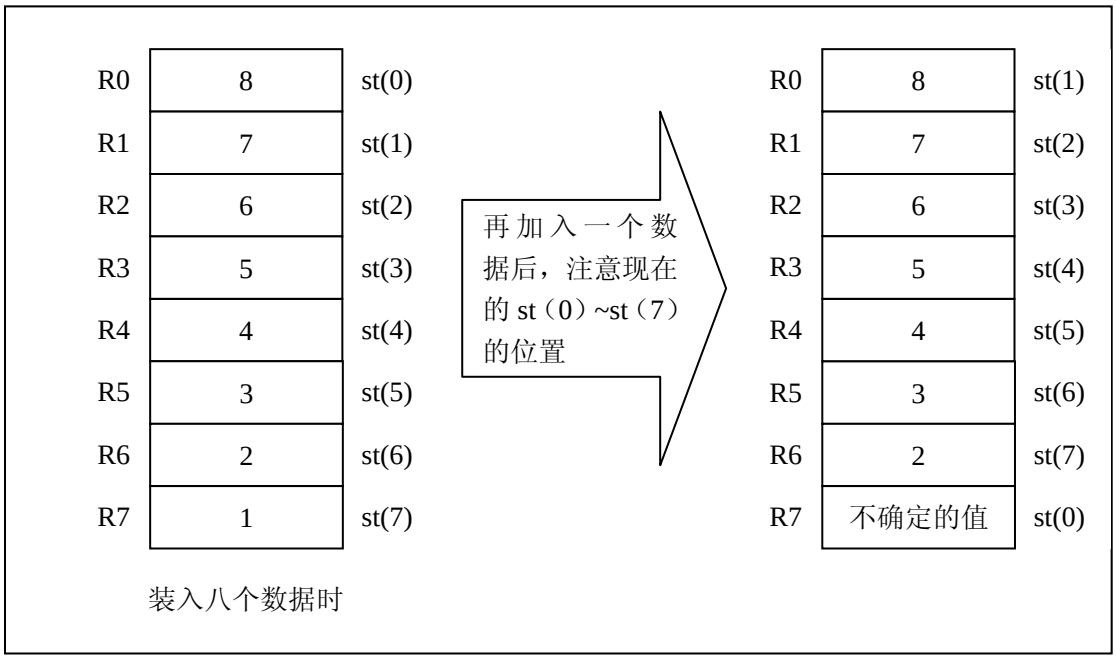


（图 2 装入不同数据时浮点寄存器中栈顶的位置）

由上图可以很明显的看出浮点寄存器是怎样被组织及使用的。需要注意的是，我们并不能通过指令直接使用 R0~R7，而只能使用 st(0)~st(7)，这在下边描述浮点运算指令的时候，会有详细描述。

也许会有朋友对上图产生疑问，当已经放入 8 个数后，也即当 st(0)处于 R0 的时候，再向里面放入一个数会产生什么情况呢？

当已经有 8 个数存入浮点寄存器中后，再向里面放入数据，这会根据控制寄存器中的相应的屏蔽位是否设置进行处理。如果没有设置相应的屏蔽位，就会产生异常，就像产生一个中断似的，通过操作系统进行处理，如果设置了相应的屏蔽位，则 CPU 会简单的用一个不确定的值替换原来的数值。如下图所示：



（图 3 装入数据大于八个数时，浮点寄存器状态）

可见，其实浮点寄存器相当于是被组织成了一个环形栈，当 st(0)在 R7 位置的时候，如

果还有数据装入，则 st(0)会回到 R0 位置，但这个时候装入 st(0)的却是一个不确定的值，这是因为 CPU 将这种超界看做是一种错误。

那么上面的说法倒底对不对呢？别急，在下面描述了浮点运算之后，我将会用一段实验代码验证上面所述。

三、Intel CPU 浮点运算指令对浮点寄存器的使用

在第二节中，我们指出 Intel CPU 将它 8 个浮点寄存器组织成为一个环形堆栈结构，并用 st(0)指代栈顶，相应的，Intel CPU 的相当一部份浮点运算指令也只对栈首的数据进行操作，并且大多数指令都存在两个版本，一个会弹栈，一个不会弹栈。比如下面的一条取数指令：

```
fsts 0x12345678
```

这就是一个不会弹栈的指令，它只是将栈顶，即 st(0)的数据存到内存地址为 0x12345678 的内存空间中，其中 fsts 最后的字母 s 表明这是对单精度数进行操作，也就是说它只会把 st(0)中四个字节的数据存入以 0x12345678 开始的内存空间中。具体是那四个字节，这就涉及到从 80 位的 st(0)到单精度（float）的一个转换，这将在下面介绍浮点数格式的小节中详细描述。

上面的指令执行后，不会进行弹栈操作，即 st(0)的值不会丢失，而下面就是同种指令的弹栈版本：

```
fstps 0x12345678
```

这条指令的功能几乎与上面一条指令完全相同，唯一不同的地方就在于这是一个会引起弹栈操作的指令，其中 fstps 中的字母 p 指明了这一点。此条指令执行后，原来 st(0)中的内容就丢失了，而原 st(1)中的内容成为 st(0)中的内容，这种堆栈的弹压栈操作我想对大家是再熟悉不过了，因此，这里将不再对其进行描述，不清楚的可以参看任一本讲数据结构的书。

本文主旨在于描述一下 Intel CPU 浮点数处理机制的基本原则，而并非浮点指令的资料，因此本文不再对众多的浮点指令进行描述，在下面的描述中，本文仅对所用到的指令进行简单的解释，如果你想完整的了解浮点指令，可以参看本文的参考文献 1。

下面，我们将用一个例子结束本节的讲述，这个例子将涉及上节及本节所讲述的内容，它验证了上面的描述是否正确。

请在 Linux 下输入下面的代码：

```
/* ---------------------------- test.c ---------------------------------- */
void f(int x[])
{
        int f[] = {1,2,3,4,5,6,7,8,9} ;
/*----------------------------- A 部分 -----------------------------*/
        __asm__( "fildl %0"::"m"(f[0]) ) ;
        __asm__( "fildl %0"::"m"(f[1]) ) ;
        __asm__( "fildl %0"::"m"(f[2]) ) ;
        __asm__( "fildl %0"::"m"(f[3]) ) ;
        __asm__( "fildl %0"::"m"(f[4]) ) ;
        __asm__( "fildl %0"::"m"(f[5]) ) ;
        __asm__( "fildl %0"::"m"(f[6]) ) ;
        __asm__( "fildl %0"::"m"(f[7]) ) ;
        // __asm__( "fildl %0"::"m"(f[8]) ) ;                    (*)
        // __asm__( "fst %st(3)" ) ;                             (**)
```

```
/* ------------------------------ B 部分 ---------------------------------*/
        __asm__( "fistpl %0"::"m"(x[0]) ) ;
        __asm__( "fistpl %0"::"m"(x[1]) ) ;
        __asm__( "fistpl %0"::"m"(x[2]) ) ;
        __asm__( "fistpl %0"::"m"(x[3]) ) ;
        __asm__( "fistpl %0"::"m"(x[4]) ) ;
        __asm__( "fistpl %0"::"m"(x[5]) ) ;
        __asm__( "fistpl %0"::"m"(x[6]) ) ;
        __asm__( "fistpl %0"::"m"(x[7]) ) ;
}

void main()
{
        int x[8] , j ;
        f(x) ;
        for( j = 0 ; j < 8 ; ++j )
                printf( "%d\n" , x[j] ) ;
}
```

上面的代码通过内嵌汇编，在 A 部分把一个整数数组中的整数压入浮点寄存器中（fildl 指令用于把整数压入浮点寄存器），而后又在 B 部分将浮点寄存器中的数取到另一个数组中（fistpl 指令用于把浮点寄存器中的栈顶数据存入指定内存单元中，指令中的字母 p 表明这是一个弹栈指令，每次都会弹栈）。程序中我们只压入了 f[0]~f[7]的 8 个数据，而压入的数据顺序是 1，2，3，4，5，6，7，8，因此，取出的顺序应当是 8，7，6，5，4，3，2，1，在 Linux 下编译并运行我们会得到这样的结果。

下面，我们将（*）语句处的注释符号“//”去掉，这个时候我们压入了 f[0]~f[8]共 9 个数据，这将会引起超界，按照上面的描述，当发生这种情况的时候，st(0)会从 R0 的位置变到 R7，并在其中存入一个不确定的值，那么实际情况是不是这样呢？同样请在 Linux 下编译并运行此程序，并将结果与图 3 进行比较。这里需要注意的时，我们总是按照 st(0)，st(1)，……，st(7)的顺序取出数据的。

最后，我们再将（**）语句处的注释符号去掉，“fst %st(3)”这条指令的作用是把 st(0)中的内容存入 st(3)，指令中并没有 p 字母，因此，这并不是一条会引起弹栈的指令。同样请在 Linux 下编译运行，并对照图 3 观察它的结果，以验证前文所述内容。

四、浮点数格式

C 语言及 CPU 所使用的浮点数格式均遵从 IEEE 754 标准，下面我们就对此详细的讨论一下。

IEEE 标准指出，一个数可以表示为：$V = (-1)^S \times 2^E \times M$ 。其中 S 在 V 是正数时取 0，在 V 是负数是取 1。对应到程序中，这显然就是一个符号位，所以 S 占 1 位，而 V 及 M 的位数由数据类型来决定。如果是单精度型（float），那 E 占 8 位（e = 8），M 占 23 位（m = 23），如果是双精度型（double），E 占 11 位（e = 11），M 占 52 位（m = 52），如下图所示：

S’ 1 位	E’ e 位	M’（M’作为纯小数） m 位

（图 4 IEEE 745 浮点数格式）

这里需要注意的是，我们用的是 S’、E’、M’而非 S、E、M，这是因为它们之间存在着一种转换关系。请看下面的描述。

S 在任何时候都等于 S’，但 E 与 M 就不一样了。当 E’=0 时，$E=1-\left(2^{e-1}-1\right)$，M = 0.M’，举个例子：0x00 50 00 00 这个数所表示的浮点数是多少呢？我们把这个数展开

0000 0000 0101 0000 0000 0000 0000

最开头红色的 1 位是符号位 S’，故 S=S’=0，中间 8 位蓝色的是 E’位，由于 E’=0，所以按上面公式可算得 E = -126，最后的是 M’，所以 M = 0. 101 0000 0000 0000 0000=0.625，所以这个小数应当是 $V=(-1)^0\times 2^{-126}\times 0.625$=7.346840e-39，那么实际是不是这样的呢？我们同样通过下面一个程序进行一下验证：

```
/* ------------------- test2.c ---------------------- */
union U{
        float f ;
        struct{
                unsigned char x1 ;
                unsigned char x2 ;
                unsigned char x3 ;
                unsigned char x4 ;
        } ;
} u ;

int main()
{
        u.x1 = 0x00 ;
        u.x2 = 0x00 ;
        u.x3 = 0x50 ;
        u.x4 = 0x00 ;

        printf( "%e\n " , u.f ) ;
        return 0 ;
}
```

程序非常简单，你可以在 Linux 下编译并执行，以检查前文所述。

上面谈到了当 E’ = 0 时的情况，那么当 E’不为 0，且不全为 1（即 $E'\neq 2^e-1$）时又是什么情况呢？这个时候，$E = E'-(2^{e-1}-1)$，M = 1.M’，举个例子，0xbf 20 00 00 这个数表示的浮点数是多少呢？同样把这个数展开：

1011 1111 0010 0000 0000 0000 0000

下面我们来按上述的要求计算：

S = S’ = 1，E = -1，M = 1.M = 1. 010 0000 0000 0000 0000 0000 = 1.25

所以

$$V = (-1)^1 \times 2^{-1} \times 1.25 = -0.625$$

同样，我们通过一个程序验证一下，不过这次我们把这个程序变一下，直接输入-0.625，看它的输出字节是多少：

```
/* ---------------------- test3.c ------------------- */
union U{
        float f ;
        struct{
                unsigned char x1 ;
                unsigned char x2 ;
                unsigned char x3 ;
                unsigned char x4 ;
        } ;
} u ;

int main()
{
        u.f = -0.625 ;

        printf( "%2x %2x %2x %2x\n",u.x4,u.x3,u.x2,u.x1) ;
        return 0 ;
}
```

编译并运行之后，我们得到的输出是：bf 20 0 0 这与我们前面的分析完全一致。

这里还想提请大家注意一点的是，通过这个例子，我们很容易可以看见，IEEE 格式中，负数并不是用补码来表示的！

下面只剩下最后一种情况了：当 E’全为 1 即 $E' = 2^e - 1$ 时。这个时候，如果 M’ = 0，此数表示无穷（inf，当 S = 0 时是正 inf，当 S = 1 时是负 inf），如果 M’ 不是 0，此数是 NaN（Not a Number），比如，你让计算机计算-1 的平方根，就会得到 NaN，表明它无法用一个数表示。

上面描述的是 IEEE 所定义的浮点格式，而在 Intel CPU 中使用的是一种扩展精度的浮点数形式，它的 E 有 15 位（e = 15），M 有 63（m = 63）位，加上 1 位的符号位，恰好等于 80 位，这同 Intel CPU 中浮点寄存器的长度是一样的。但是 E，M 位的确定方法还是同 IEEE 标准兼容，也即本节所描述的方法完全适用于 Inte CPU 的 80 位的扩展精度格式。

五、浮点数的舍入及所带来的程序设计上的问题

由于存在多种格式的浮点数，因此，从一个高精度的格式转换到一个低精度的格式就会出现舍入，而由于舍入的存在，就会造成程序中出现很多非常有意思的错误，现在我们就来谈谈这个问题。这里将最终解决前言中提到的那个问题。

首先谈谈从低精度格式转换到高精度格式。这并不会引起精度的丢失，因此，这样的转换是很安全的，随着表示数据的位数的增加，高精度格式可以完全把低精度格式的相应数据

位复制过来，并不会丢失任何信息，然而从高精度向低精度进行转换，就会丢失信息，因为低精度格式的数据位数比高精度的要少，容纳不下高精度的所有信息。

现在我们就来看一下各种精度格式可表示的最大正数或最小正数是多少。

首先来看一下最小正数，最小正数时 E'=0，而 M'=000…..1，故由前面的描述可知，对于单精度数（float），最小正数为$V_{\min} = 2^{-126} \times 2^{-23}$，对于双精度数（double），最小正数为$V_{\min} = 2^{-52} \times 2^{-1022}$；最大正数时 E'=111…10，而 M'=111…1，故可知单精度数的最大正数为$V_{\max} = 2^{127} \times (2 - 2^{-23})$，双精度数（double）的最大正数为$V_{\max} = 2^{1023} \times (2 - 2^{-52})$。很明显的可以看到双精度数所能表示的范围，远远大于单精度数所表示的范围。

当一个双精度数所表示的数比单精度所能表示的最大的数还要大时，CPU 会让单精度数等于无穷，此时，单精度数的 E'全为 1，而 M'=0（见上节所述）。

当一个双精度数所表示的数比单精度所能表示的最小数还要小时，CPU 会让单精度数等于 0，此时，单精度数的 E'、M'全为 0。

注意这样一个事实，M 只有两种可表达形式，一种是 M=1.M'，一种是 M=0.M'，当 M=0.M'的时候，指数 E 只能为$E = 1 - (2^{e-1} - 1)$，这个特点决定了任何一个非零的数只有一种确定的表示。这种确定性使计算机的处理变得非常方便。

比如一个 double 型的浮点数，它在用 float 型来表示时只能有一种型式，这使我们能很快的确定出 float 型的数据表示，也一眼就能确定这个数是否造出了 float 表示数的范围。同样，对于计算机来说，这极大的方便了不同精度之间的转换，使这种转换有唯一而确定的形势。

在 CPU 的内部，所有的浮点数在被浮点指令装入浮点寄存器的时候都会发生转换，从单精度、双精度、整数转换为 80 位的扩展精度，当从浮点寄存器存入内存的时候又会发生转换，从扩展精度转换为相应的精度格式，这种转换是由 CPU 硬件自动完成的，然而正是由于从扩展精度转换为低精度格式这一行为的存在，会让我们的程序出现某些很奇怪的问题。请看下面一段代码：

```
/* ------------------- test4.c ----------------- */
int main()
{
        float f = 3.25 + 1e10;
        f = f - 1e10 ;
        printf( "%f\n",f ) ;
        return 0 ;
}
```

这段代码就是一个典型的精度丢失的例子，我来现在就来认真的分析一下它：

3.25 + 1e10 这个数我们用二进制可以表示为：

1001 0101 0000 0010 1111 1001 0000 0000 11.01

由于 M'只能是 0.M'的形式或 1.M'的形式，如果是 0.M'的形式，则上式可以表示为：

$$0.\ 1001\ 0101\ 0000\ 0010\ 1111\ 1001\ 0000\ 0000\ 1101 \times 2^{34}$$

如果是 1.M'的形式，则可以表示为：

$$1.001\ 0101\ 0000\ 0010\ 1111\ 1001\ 0000\ 0000\ 1101 \times 2^{33}$$

现在我们将它们转换为单精度数，按照 IEEE 单精度数的格式要求，如果是 0.M'的形式，

则指数只能是 $E = 1 - (2^{8-1} - 1) = -126$，而上面的 E=34，故上面的数只能采用 1.M 的形式，这个时候，M’=001 0101 0000 0010 1111 1001 0000 0000 1101，但由于 IEEE 指出，在单精度格式中，M’只能有 23 位，因此，最后的 0000 0000 1101 将会被截断，M’实际为：

M’=001 0101 0000 0010 1111 1001

也就是说，原数中的 3.25 被丢失了，因此，实际上 f=3.25+1e10 计算后 f=1e10，所以，上面这段代码的输出结果是 0。

这里需要提到一点上面所涉及到的截断方法，专业一点的术语称为舍入（Round）。IEEE 总共定义了四种舍入规则，分别是“Round to Even”，“Round Toward Zero”，“Round Down”及“Round UP”，到底使用哪一种舍入规则可由程序员在浮点单元的控制寄存器中指定（参见前文所述），默认是使用“Round to Even”，下面我们就来看看这些规则。

先看“Round to Even”，假若有个数 x.yyyy0….那么舍入为 x.yyyy；如果有个数为 x.yyyy1….且 1 后面的所有位不全为零，那么舍入为 x.yyyy+0.0001；如果原数为 x.yyyy100….00 那么这个时候的舍入就需要分情况讨论了。如果此时最后一个 y 是 1，那么舍入为 x.yyyy+0.0001，如果此时最后一个 y 是 0，那么舍入为 x.yyyy。比如 1.0110 100 这个数舍入后为 1.0110，而 1.0111 100，这个数舍入后就为 1.0111+0.0001=1.1000。

在来看看“Round Toward Zero”，这个很简单，它要求舍入后的数的绝对值不大于原数的绝对值。

“Round Down”要求：舍入后的数不大于原数。

“Round UP”要求：舍入后的数不小于原数。

上面关于精度损失的例子是个比较明显而比较简单的例子。但事情不总是这么明显的，下面我们就来解剖一下前言中提到的那个程序：

先来回顾一下前言中提到的那个问题：

先来看看下面一段程序：

```
/* -------------a.c------------------ */
#include <stdio.h>
double f(int x)
{
        return 1.0 / x ;
}

void main()
{
        double a , b;
        int i ;
        a = f(10) ;
        b = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

这段程序使用 gcc –O2 a.c 编译后，运行它的输出结果是 0，也就是说 a 不等于 b，为什么？

再看看下面一段，几乎同上面一模一样的程序：

```
/*--------------- b.c ---------------------*/
```

```
#include <stdio.h>
double f(int x)
{
        return 1.0 / x ;
}

void main()
{
        double a , b , c;
        int i ;
        a = f(10) ;
        b = f(10) ;
        c = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

同样使用 gcc –O2 b.c 编译，而这段程序输入的结果却是 1，也就是说 a 等于 b，为什么？

我们现在将第一段代码，也即 a.c 反汇编，下面是反汇编的结果：

```
08048328 <f>:
 8048328:       55                      push   %ebp
 8048329:       89 e5                   mov    %esp,%ebp
 804832b:       d9 e8                   fld1
 804832d:       da 75 08                fidivl 0x8(%ebp) //计算，并将结果存入 st(0)中
 8048330:       c9                      leave
 8048331:       c3                      ret
 8048332:       89 f6                   mov    %esi,%esi
```

上面代码中我们只需要注意一点：fidivl 指令表示用 st(0)中的数除以某一内存地址中的数，并将结果存在 st(0)中。

注意这样一个事实，现在的运算结果是存在 st(0)中的，而且这是一个 80 位的值。下面我们来看看 main 中的代码：

```
08048334 <main>:
 8048334:       55                      push   %ebp
 8048335:       89 e5                   mov    %esp,%ebp
 8048337:       83 ec 08                sub    $0x8,%esp
 804833a:       83 e4 f0                and    $0xfffffff0,%esp
 804833d:       83 ec 0c                sub    $0xc,%esp
 8048340:       6a 0a                   push   $0xa
 8048342:       e8 e1 ff ff ff          call   8048328 <f>
 8048347:       dd 5d f8                fstpl  0xfffffff8(%ebp)
 804834a:       c7 04 24 0a 00 00 00    movl   $0xa,(%esp,1)
 8048351:       e8 d2 ff ff ff          call   8048328 <f>
 8048356:       dd 45 f8                fldl   0xfffffff8(%ebp)
 8048359:       58                      pop    %eax
```

```
804835a:       da e9                   fucompp
804835c:       df e0                   fnstsw %ax
804835e:       80 e4 45                and    $0x45,%ah
8048361:       80 fc 40                cmp    $0x40,%ah
8048364:       0f 94 c0                sete   %al
8048367:       5a                      pop    %edx
8048368:       0f b6 c0                movzbl %al,%eax
804836b:       50                      push   %eax
804836c:       68 d8 83 04 08          push   $0x80483d8
8048371:       e8 f2 fe ff ff          call   8048268 <_init+0x38>
8048376:       c9                      leave
8048377:       c3                      ret
```

代码很长，但我们实际只需关心其中的一小部份，请看：

```
8048342:       e8 e1 ff ff ff       call   8048328 <f>        // 计算 f(10)，这个时候的
                                                              // 计算结果在 st(0)中
8048347:       dd 5d f8             fstpl  0xfffffff8(%ebp)   // 把计算结果存回内存 a 中
804834a:       c7 04 24 0a 00 00 00 movl   $0xa,(%esp,1)
8048351:       e8 d2 ff ff ff       call   8048328 <f>        // 计算 f(10)，对应于 b=f(10)
                                                              // 计算结果在 st(0)中
8048356:       dd 45 f8             fldl   0xfffffff8(%ebp)   // 直接载入 a 中的值
                                                              // 这时 st(0)= a
                                                              // st（1）方才计算的 b 的值
8048359:       58                   pop    %eax
804835a:       da e9                fucompp                   // 将 st(0)与 st(1)进行比较
804835c:       df e0                fnstsw   %ax
```

这里我们已经能够看出问题所在了！它先计算了 a=f(10)，然后把这个结果存回到内存中了，由于 0.1 没办法用二进制精确表示，因此，从 80 位的扩展精度存到内存的 64 位的 double 中，产生了精度损失，这之后，计算 b=f(10)，而这一值并没有被存回内存中，这个时候，gcc 就直接将内存中的 a 值装入到 st(0)中，与方才计算的 b 值进行比较，由于 b 值并没有被存回内存中，因此，b 值并没有精度损失，而 a 值是损失了精度的，因此 a 与 b 不相等！

下面我们来把第二段代码反汇编，这里我们只贴出我们最需要关心的那部份代码：

```
8048342:       e8 e1 ff ff ff       call   8048328 <f>       // 计算 a
8048347:       dd 5d f8             fstpl  0xfffffff8(%ebp) // 把 a 存回内存
                                                             // a 产生精度损失
804834a:       c7 04 24 0a 00 00 00 movl   $0xa,(%esp,1)
8048351:       e8 d2 ff ff ff       call   8048328 <f>       // 计算 b
8048356:       dd 5d f0             fstpl  0xfffffff0(%ebp) // 把 b 存回内存
                                                             // b 产生精度损失
8048359:       c7 04 24 0a 00 00 00 movl   $0xa,(%esp,1)
8048360:       e8 c3 ff ff ff       call   8048328 <f>       // 计算 c
8048365:       dd d8                fstp   %st(0)
8048367:       dd 45 f8             fldl   0xfffffff8(%ebp)  // 从内存中载入 a
804836a:       dd 45 f0             fldl   0xfffffff0(%ebp)  // 从内存中载入 b
804836d:       d9 c9                fxch   %st(1)
```

```
804836f:       58                      pop    %eax
8048370:       da e9                   fucompp                 // 比较 a , b
8048372:       df e0                   fnstsw  %ax
```

从上面的代码看出，a 与 b 在计算完成之后，都被 gcc 存回了内存，于是都产生了精度损失，因此，它们的值就是完全一样的了，于是此后再把它们调入浮点寄存器进行比较，得出的结果就是 a = b。这主要是因为程序中多了一个 c = f(10) 的计算，它使 gcc 必须把先前计算的 b 值存回内存。

我想，现在你应当对此问题有比较清楚的了解了吧。顺带说一句，gcc 中提供了一个 long double 的关键字，对应于 Intel CPU 中的 80 位的扩展精度，它是 10 字节的。

六、总纲

据说《九阴真经》在最后有一个总纲，对全书进行总结，这里，我也借鉴一下，对前面的行文进行一下总结。

这篇文章比较完整的描述了 Intel CPU 对浮点数的基本的处理机制，希望能对大家理解 CPU 及在运用 C 语言进行浮点编程时产生一定有益的影响。在最后，我想对几个比较模糊的说法谈谈自己的看法。

首先，浮点数不能比较是否相等。其实这个说法是不精确的，从本质上说，浮点数完全可以比较是否相等，CPU 也提供了相应的指令，但由于存在舍入问题，因此，浮点数在比较是否相等的时候是不安全的，比如上面分析的那个程序，同样计算的是 f(10)，但第一个程序在比较是否相等时是不成立的，而第二个却是成立的，这主要是由于舍入问题的存在，如果我们能够从本质上理解这个问题，那么我们完全可以放心大胆的在程序中比较两个浮点数是否相等。对于上面的这个程序我还想说明一点就是，我们必须在 gcc 打开优化的前提下才能得出上面的结果，这是因为 gcc 在非优化的时候每次计算完结果后，都会把结果存回内存。程序优化的最首要的目标就是不能影响程序的执行结果，但非常遗憾的是，gcc 并没有做到这一点。

其次，我们常常在书上看见：“所有的涉及浮点数的运算，都会转成 double 进行”，从上面的分析中我们也可以看出，这是不精确的。从硬件的角度说，Intel CPU 在默认条件下使用的是扩展精度（gcc 中的 long double 类型），因此，它们都是转换成扩展精度进行的。从 C 语言本身来说，C 语言作为一个同硬件非常贴近的语言，它最终的结果就是产生出硬件可以识别的代码，而硬件到底怎么执行是由硬件决定的，C 语言无法对此进行控制与干涉，相反 C 语言必须去适应硬件的规定。因此，不能说 C 语言本身会将所有浮点运算都转换成某种精度进行计算，到底使用哪种精度这是由 cpu 决定也是由 cpu 来完成的。

第三，由于浮点运算后，结果在存入内存中会产生舍入，这很可能会带来误差，因此，我们应当尽量使用高精度的数据类型，比如用 gcc 的时候，我们尽量使用 long double 而非 double、float，这会减少相当多的错误机会。不要认为使用它们会带来性能上的下降，它们最主要的弱点在于占用的空间比较大，但现在空间已经不是一个主要考虑的因素。

七、推荐阅读

C 语言是一个与硬件贴得很近的语言，要想真正精通 C 语言，你应当对硬件结构及指令系统有相当的了解，这你可以参看本文的参考文献 1。

如果你认为参考文献 1 写得太过繁复，你可以参看一下本文的参考文献 2，另外，它还对在编程中可能出现的问题做了很多分析，非常深入细致，在某些细节上描述得非常清楚，具有 Bible 的品质。

如果你对 C 语言不是很了解，你可以参考一下参考文献 3，它对 C 语言有比较简单的

描述，特别是书中有一章《C 程序设计常见错误及解决方案》对初学者有很大好处。

参考文献：

1. 《IA-32 Intel Architecture Software Develop's Manual》Vol.1，Vol.2，Intel Corp
2. 《Computer Systems A Programmer's Perspective》（Randal E.Bryant，David R.O'Hallaron）电子工业出版社（影印版）
3. 《C 语言大学实用教程》（苏小红，陈惠鹏，孙志岗）电子工业出版社

【全文完 • 责任编辑：sun@hitbbs】

操作系统引导探究 (Version 0.02)

哈尔滨工业大学 计算机体系结构实验室 谢煜波

(xieyubo@126.com)

Version 0.02 修改记录:

对与 GDT 有关的段描述符方面的描述进行了修订，更正了上一个版本中出现的一些错误，增加了一些描述，使其更完善。

与上个版本中不同的地方均用红色标记。

前言

本篇文章并不旨在完整的讨论一个多引导系统程序怎样去引导不同的操作系统，而只打算从编写操作系统的角度出发，谈谈计算机怎样从加电开始，从无到有，将操作系统运行起来，在其中将尽量详尽的描述从实模式到保护模式的过渡，目的只在于能将所学与广大爱好者共享，为希望开发操作系统的朋友留下一点资料，也为自己留下一点心得。

本篇文章将以开发中的 pyos 系统引导程序为例，pyos 是一个正在开发中的实验型操作系统，它并不打算以目前任何一种运行中的操作系统为模式，而只想通过自己编写一个从头到尾的操作系统来学习知识，积累技术，如果你有兴趣，非常欢迎你的加入！

本篇纯属学习过程中的一点心得体会，如果你发现其中有错误或不当之处，非常希望你来信指教。

一、计算机从加电开始都做了什么？

当机算机的电源键被按下时，同这个键相联的电信号线就会送出一个电信号给主板，主板将此电信号传给供电系统，供电系统开始工作，为整个系统供电，并送出一个电信号给 BIOS，通知 BIOS 供电系统已经准备完毕。随后 BIOS 启动一个程序，进行主机自检，主机自检的主要工作是确保系统的每一个部分都得到了电源支持，内存储器、主板上的其它芯片、键盘、鼠标、磁盘控制器及一些 I/O 端口正常可用，此后，自检程序将控制权还给 BIOS。接下来 BIOS 读取 BIOS 中的相关设置，得到引导驱动器的顺序，然后依次检查，直到找到可以用来引导的驱动器（或说可以用来引导的磁盘，包括软盘、硬盘、光盘等），然后调用这个驱动器上磁盘的引导扇区进行引导。BIOS 是怎么知道或说分辨哪一个磁盘可以用来引导的呢？

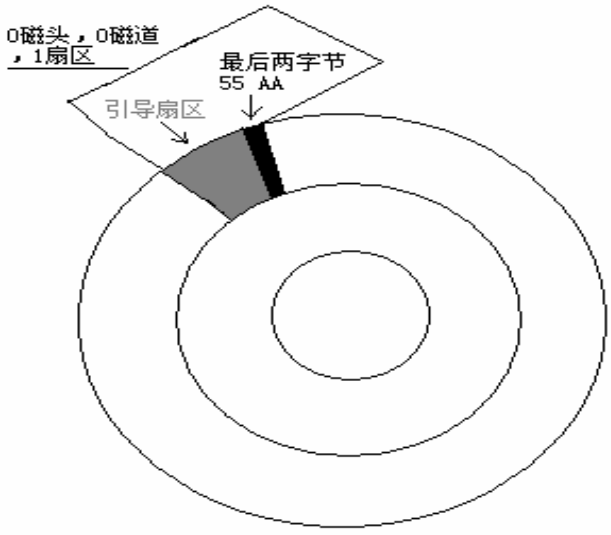
二、认识引导程序

BIOS 将磁盘的第一个扇区（磁盘最开始的 512 字节）载入内存，放在 0x0000:0x7c00 处（见图三），如果这个扇区的最后两个字节是“55 AA”，那么这就是一个引导扇区，这个磁盘也就是一块可引导盘。通常这个大小为 512B 的程序就称为引导程序（boot）。如果最后两个字节不是“55 AA”，那么 BIOS 就检查下一个磁盘驱动器。

通过上面的表述我们可以总结出如下三点引导程序所具有的特点：

1. 它的大小是 512B，不能多一字节也不能少一字节，因为 BIOS 只读 512B 到内存中去。
2. 它的结尾两字节必须是“55 AA”，这是引导扇区的标志。
3. 它总是放在磁盘的第一个扇区上（0 磁头，0 磁道，1 扇区），因为 BIOS 只读

第一个扇区。



（图一）

因此，在我们编写引导程序的时候，我们也必须注意上面的三点原则，符合上面三点原则的程序都可以看作是引导程序，至少 BIOS 是这样认为的，虽然它也许可能是你随意写的一段并没有什么实际意义的代码。

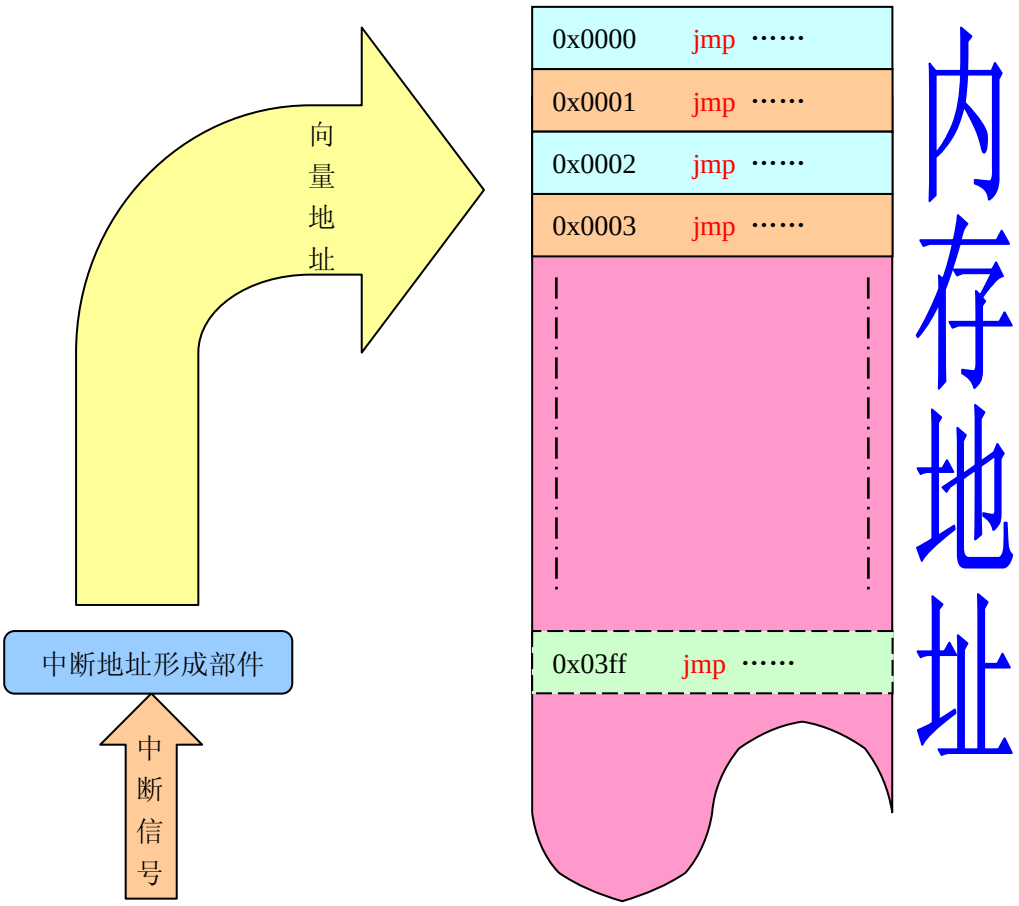
因为 BIOS 一次只读一个扇区也即 512 字节的数据到内存中，这显然是不够的，现在操作系统都比较庞大，因此我们必须在引导扇区里，将存在磁盘上的操作系统的核心部分读进内存，然后再跳转到操作系统的核心部分去执行。

三、通过 BIOS 读磁盘扇区

从上面的描述我们可以知道，引导程序需要将存在于磁盘上的操作系统读入内存，因此这里我们不得不再讲一讲，怎样不通过操作系统（因为现在还没有操作系统）去读取磁盘上的内容。一般说来这有两种方法可以实现，一种是直接读写磁盘的 I/O 端口，一种是通过 BIOS 中断实现。前一种方法是最低层的方法（后一种方法也是在它的基础上实现的），具有极高的灵活性，可以将磁盘上的内容读到内存中的任意地方，但编程复杂。第二种方法是前一种方法稍微高层一点的实现，牺牲了一点灵活性，比如，它不能把磁盘上的内容读到 0x0000:0x0000 ~ 0x0000:0x03FF 处。为什么不能读到此处呢？这里我们将不得不描述一下 CPU 在加电后的中断处理机制。

3.1 BIOS 的中断处理

中断是什么？相信学过计算机的人都不会陌生，如果你对中断一点都不了解建议你翻看一下《计算机组成原理》（高等教育出版社　唐朔飞），上面有非常详尽的描述，而一般的汇编教材也多有谈及，因此这里只打算讲讲 BIOS 对中断的处理。



（图二）

由上图（图二）我们可以清楚的看到，当中断信号产生时，中断信号通过“中断地址形成部件”产生一个中断向量地址，此向量地址其实就是指向一个实际内存地址的指针，而这个实际内存地址中往往安排一条跳转指令（jmp）跳转到实际处理此中断的中断服务程序中去执行。这一块专门用于处理中断跳转的内存就被称为中断向量表。在内存中，这块中断向量表被放在什么地方呢？而实际的中断处理程序又在什么地方呢？

3.2 系统的内存安排（1M）

要回答上面的两个问题，我们需要看看系统中内存是怎么安排的。在 CPU 被加电的时候，最初的 1M 的内存，是由 BIOS 为我们安排好了的，每一字节都有特殊的用处。

0x00000~0x003FF：	中断向量表
0x00400~0x004FF：	BIOS 数据区
0x00500~0x07BFF：	自由内存区
0x07C00~0x07DFF：	引导程序加载区
0x07E00~0x9FFFF：	自由内存区
0xA0000~0xBFFFF：	显示内存区
0xC0000~0xFFFFF：	BIOS 中断处理程序区

(图三)

由上图我们现在可以很方便的问答上面提出的两个问题。由于 0x00000~0x003FF 是中断向量表所在，因此不能将磁盘中的操作系统读到此处，因为这样会覆盖中断向量表，就无法再通过 BIOS 中断读取磁盘内容了。你也许会说：我是先调用中断，再读的啊。但事实上 BIOS 在读的过程中自己会多次调用其它中断辅助完成。

3.3 利用 BIOS 13 号中断读取磁盘扇区

有了前面的描述作为基础，下面我们可以正式描述怎样通过 BIOS 中断读取磁盘扇区了。要读取磁盘扇区，我们需要使用 BIOS 的 13 号中断，13 号中断会将几个寄存器的值作为其参数，因此，我们在调用 13 号中断的过程中需要首先设置寄存器。那么当怎样设置寄存器呢？会用到哪些寄存器呢？请往下看：

AH 寄存器：存放功能号，为 2 的时候，表示使用读磁盘功能
DL 寄存器：存驱动器号，表示欲读哪一个驱动器
CH 寄存器：存磁头号，表示欲读哪一个磁头
CL 寄存器：存扇区号，表示欲读的起始扇区
AL 寄存器：存计数值，表示欲读入的扇区数量

在设置了这几个寄存器后，我们就可以使用 int 13 这条指令调用 BIOS 13 号中断读取指定的磁盘扇区，它将磁盘扇区读到 ES:BX 处，因此，在调用它之前，我们实际上还需要

设置 ES 与 BX 寄存器，以指出数据在内存中存放的位置。

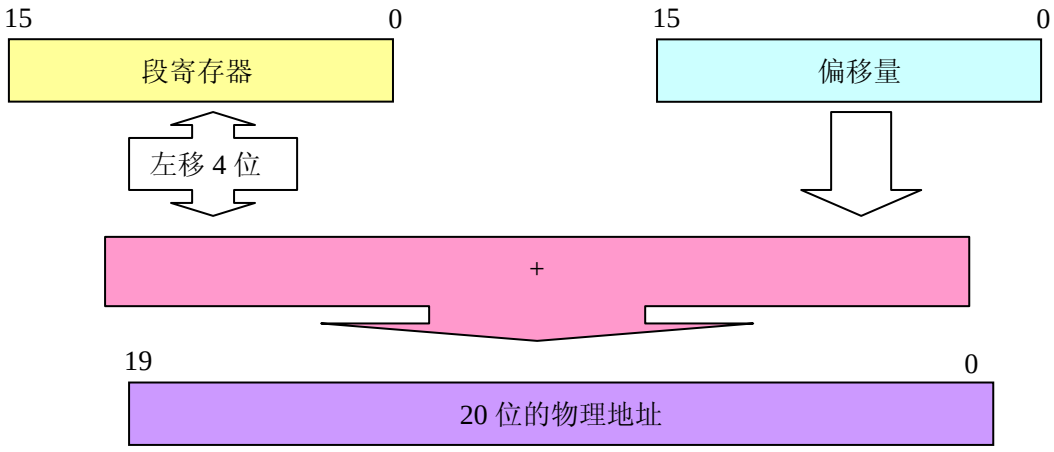
四、保护模式下，段模式内存地址的访问

写程序离不开对内存的访问，然而在保护模式下内存的访问与在实模式下内存的访问完全不同，这里我们将详细描述一下保护模式下内存的访问方法。当然，这里并不打算完整的介绍保护模式下所有的内存访问方法与机制，只介绍从实模式转到保护模式下所需要进行的转换，完整的内存访问请你参见《Intel 用户手册》。当然，随着 pyos 的实验进行，我也会在后面的实验报告与心得体会中渐渐描述。

4.1 实模式下的内存访问

计算机在加电时，处于“实模式”，在计算机中有一个 CR0 寄存器，又称为 0 号控制寄存器，在这个寄存器中，最低位也即第 0 位，被称为 PM（Protected Modle：保护模式）位，当它被清零的时候表示 CPU 在“实模式”下工作，当它被置位的时候表示 CPU 在“保护模式”下工作。在计算机加电的时候，它是被清零的，所在这个时候的计算机，处于“实模式”。

“实模式”下的内存访问通过段寄存器与偏移量构成，比如前面描述中常常出现的 0x:0000:0x0001 就是一个实模式下的内存地址。分号前面的值表示段寄存器中的值，分号后面的值表示偏移量，实际物理地址的形成如下图所示：



（图四）

然而在保护模式下，内存地址却不是如上图所示的方法形成的。那么它又是怎样形成的呢？

4.2 保护模式下的内存地址形成

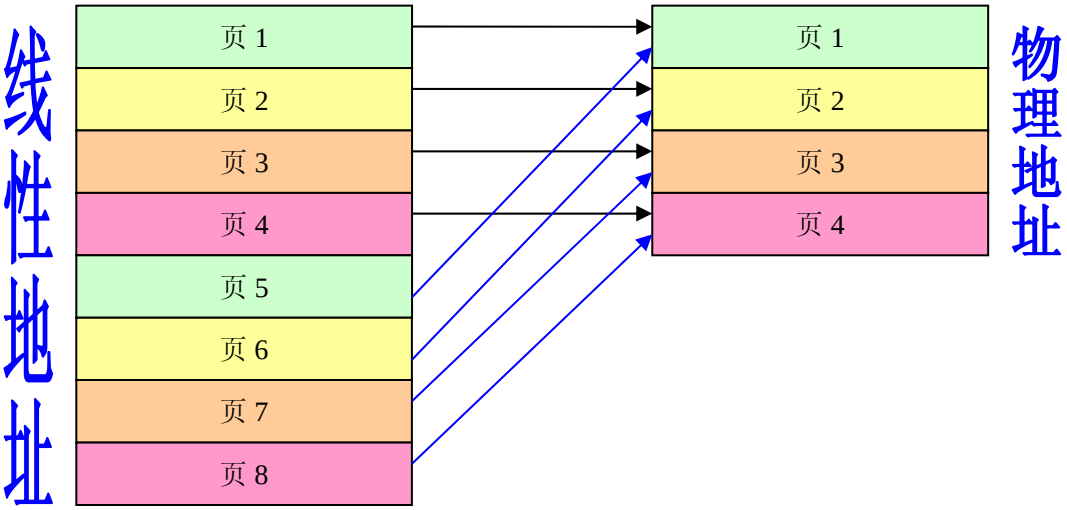
保护模式下内存地址就复杂多了， 我们首先要分清三个概念：逻辑地址、线性地址与物理地址。物理地址很好理解，逻辑地址也好理解，就是程序所使用的地址。那么什么是线性地址呢？

其实如果不使用分页机制的话，线性地址就是物理地址，它与物理地址是一一对应的，线性地址 0，也就是物理地址 0。但我们知道，32 位的 CPU 拥有 32 根地址线，也就是可以访问：

$$2^{32} = 4GB$$

的内存空间，这实在是一个太大的空间了！现在很少有机器的物理内存能有这么大。那怎么

在有限的物理空间中使用 4GB 的空间呢？人们把物理内存分成许多页，同样也把整个 4GB 的线性地址空间分成大小相同的许多页。在线性地址空间中，当某些页被使用的时候，某些页可能没有被使用，操作系统可以让 CPU 将没有被使用的页调出物理内存（存放在磁盘的某个地方，以备需要的时候再次调入），而把需要使用的页调入，这样，虽然物理内存空间有限，但也几乎可以使用所有的线性地址空间了。这就称为从线性地址到物理地址的映射，这是一个多对一的映射，也就是说多个线性空间中的页对应一个物理空间中的页，希望下面一幅图能有助于你理解这样的分页机制。

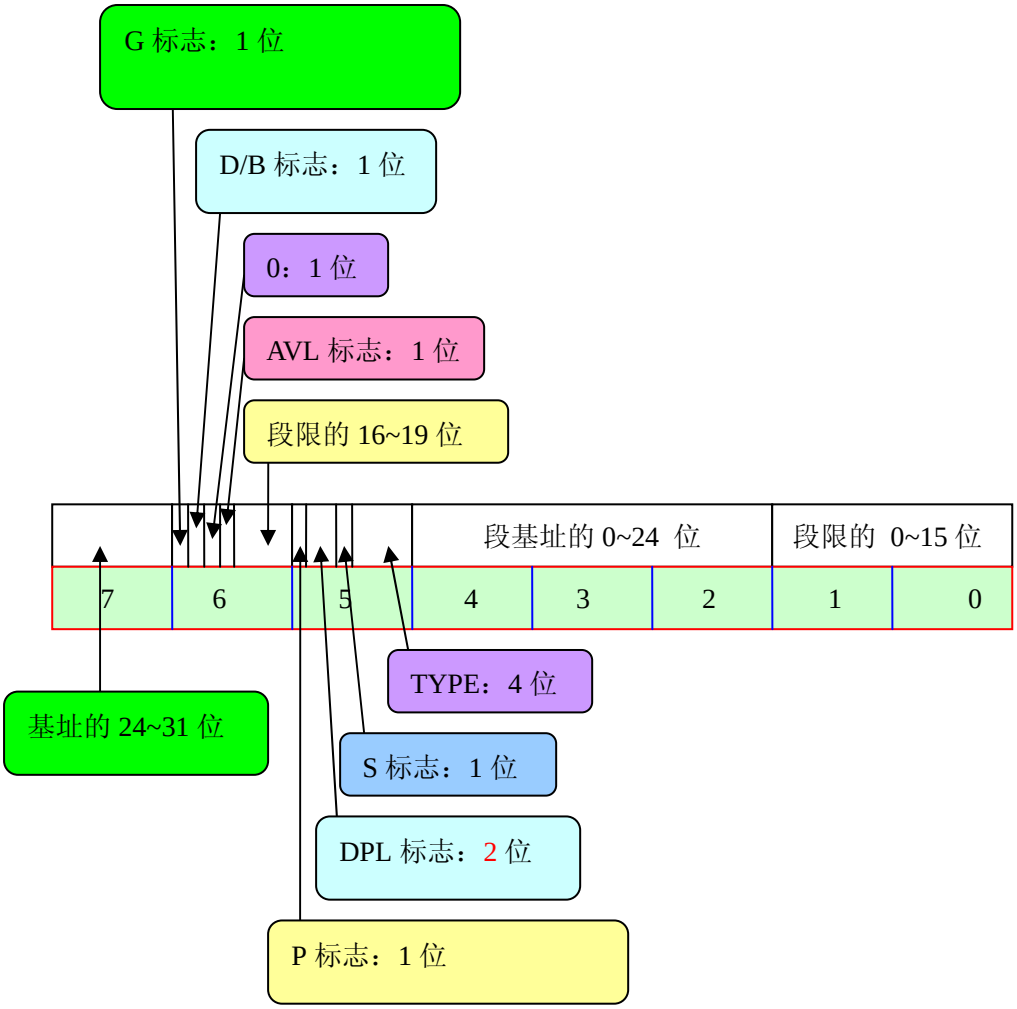


（图五）

上面是一种最简单的映射方式，术语称作“直接相连”映射，它大约只能用来说明问题，而在一个实际的操作系统中通常是“全相联相连”映射，也就是说线性地址中的页可以是映射到物理地址中的任何一个页中，只要那块物理地址空间现在是空闲的。不过，通过上图也能说明问题，当线性地址中的页 5 需要被访问时，CPU 通过地址映射机制将其转换到物理地址，发现其对应物理地址中的页 1。于是 CPU 会产生一个所谓的缺页中断来通知操作系统进行处理，操作系统响应这个中断，并在中断服务程序中将物理地址页 1 中的内容放到磁盘上的一个地方（虚拟内存），然后将线性地址中的页 5 载入物理内存页 1 中。这里就当可以比较明显的区别什么是线性地址，什么是物理地址了。

然而，当不使用分页机制的时候，线性地址就会被 CPU 当做物理地址来使用，线性地址会被直接放在 CPU 的地址信号线上。不过，在编写应用程序的时候，我们通常使用的却是另一种地址——逻辑地址，从逻辑地址到线性地址也存在着与上述机制类似的一种映射机制，不过这个机制常常称为“段模式”，它是由操作系统与 CPU 硬件共同完成的。操作系统的任务就是分配映射表，而 CPU 硬件的任务就是按着映射表进行映射。而这样的映射表在操作系统编写中又称之为“描述符表”，有两种重要的描述符表，一种是“全局描述符表（GDT）”，另一种是“局部描述符表（LDT）”，这两种表的用途不同，但它们的用法却是近似的，下面我们就来描述一下全局描述符表。

说到表，学过数据结构的人都知道，其实它就是一种数据结构，全局描述符表也是一种数据结构，当这种结构放在一块连续的内存之中就称之为表了。表由表项组成，全局描述符表由它的表项“全局描述符”组成，其实单纯的术语就叫“描述符”，只因为它放在全局描述符表中就成了全局描述符了。这个描述符由 8 个字节组成，下面我们就来看看它的结构：



(图六)

TYPE： 表明此段的类型，4 位中的最高位被清 0 的时候表是它是数据段，相应的余下的三位，从左到右依次为 E、W、A，即数据段的 TYPE 为：0EWA。其中 E 表示向下增长位，置 1 时表示向下增长（这主要是在大小需要动态改变的堆栈段中使用，如果是向下增长的段，动态的改变段的大小限制，会让堆栈空间加到堆栈的底部。如果堆栈的大小不需要改变，那么这个段既可以是向下增长的段，也可以是非向下增长的段）；W 表示可写位，置 1 表示可写；A 表示被访问位（如果 CPU 访问了它，此位将会被置 1）。

4 位中的最高位被置 1 时表示它是代码段，相应的余下的三位，从左到右依次为 C、R、A，即代码段的 TYPE 为：1CRA。其中 C 是表明此代码段是否是一致代码段，如果 C 被置 1，表明此是一致代码段。一致代码段主要是用于特权级访问控制，这在以后的实验报告中会详细论述；R 表明此段是否可读，置 1 表示可读；A 表示被访问位，这与前述一样。

S：为 1 时表示这是一个代码段或数据段描述符，为 0 时表示这是一个系统段描述符。系统段描述符又称为特殊段描述符，包括：局部描述符表（LDT）描述符，任务状态段（TSS）描述符，调用门描述符，中断门描述符，陷阱门及任务门描述符等。

DPL：表示特权级，从 00~11，共 0，1，2，3 四个特权级

P：为 0 是表示此描述符无效，不能被使用

AVL：留给程序员随便用的

D/B：为 0 的时候表示它是一个 16 位的段，为 1 时表示它是一个 32 位的段

G：为 0 时，表示段限的单位是 1 字节，为 1 时表示段限的单位是 4KB，并且段偏移量的最低 12 位将不被检测是否在段限之中。（这一点现在可能不好理解，但我下面马上会解释）。

这里面有两个部份比较有意思，一个是“基址”，一个是“段限”。基址应当比较好理解，它给出的是一个段在线性内存中的起始地址，对于“段限”，顾名思义，就是段大小的限制。不过它有点特别，对于一个段的最大可访问的地址 CPU 是通过下面的公式计算得到的：

段基址 + 段限值 * 段限单位 = 此段最大可访问地址

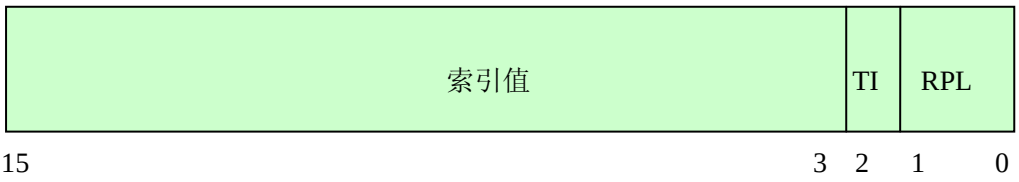
如果一个偏移地址大于了此段最大可访问地址的话，CPU 就将产生一个错误中断，这样一来就可以防止一个程序非法访问另一个程序的内存空间，这对内存起到了保护作用，“保护模式”由此得名。

所以，如果段限是 0，那么此段最大可访问地址就是段的基址，因此，当段限单位为一字节时，此段的段大小就是 1 字节；当段限单位为 4KB 时，因为 CPU 将不检测偏移量的最低 12 位，而这 12 位最大可能为 0xFFF，因此，这时此段的可访问范围就为 4KB，所以：

（段限值 + 1）* 段限单位 = 此段大小

现在我们可以正式开始描述在保护模式下段模式是怎样访问内存的了。这里之所以要强调“段模式”是因为在保护模式下还有一种前面叙述过的内存访问模式——页模式，它负责将线性地址再按某种映射转换为物理地址。“页模式”也是基于段模式的，在不使用它的情况下，线性地址会被直接放到地址线上当做物理地址使用。“段模式”是不可避免的，所谓的“纯页模式”只是将整个线性地址当作一个整段，没有什么方法可以真正绕过“段模式”，因为这是由 CPU 内存访问机制所规定的。本篇只描述段模式。

我们已经知道从程序使用的逻辑地址到线性地址的映射是通过“描述符”来完成的，而“描述符”又是放在描述符表中的，那么，一个描述表中有许多描述符，到底选用哪一个描述符呢？这就由一个索引来决定，这个索引将指出是表中的第几个描述符，这个索引有一个专门的术语来描述，常常称它为“段选择子”。“段选择子”由 2 个字节共 16 位组成，下面，我们就来看看它提供了哪些信息：



（图七）

其中：

RPL：指示出特权级，00~11，共 0、1、2、3 四个特权级，与前述一样。

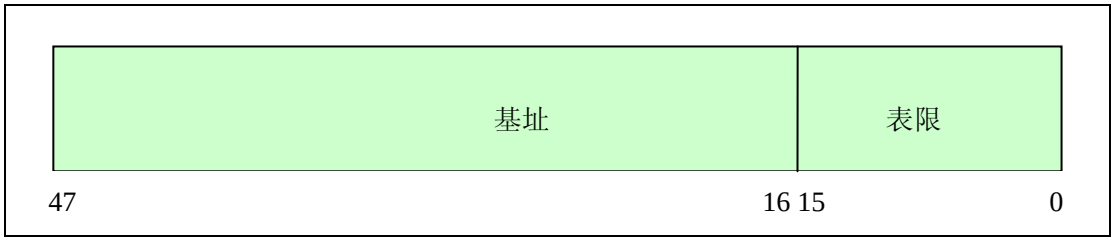
TI：为 0 时表明这是个用于全局描述符表的选择子，为 1 时表明用于局部描述符表。

索引值：用来指示表中第几个描述符。索引值共有 13 位，因此，每张描述符表共可有 8K 个表项，而一个表项如前所述，占 8 个字节，因此一张描述符表最大可达 64K。

不知道大家是否注意到这样一个事实，如果将“段选择子”的最后 3 位置 0，这整个段选择子其实就是一个描述符在描述符表中的偏移量！这里我们可以发现 Intel 的工程师在设计的时候真的是非常精巧，如此的安排，可以使选取一个描述符的速度极大加快，因为将一个段选择子最后 3 位清零后与描述符表的基址相加，就立即可以得到一个描述符的物理地

址，通过这个地址就可以直接得到一个描述符。那么这个描述符表的基址又是放在哪儿的呢？

所谓描述符表的基址也就是此描述符表在内存中的起始地址，也即表中第一个描述符所在的内存地址，系统中用两个特殊的寄存器来存放，一个用于存放全局描述符表的基址，称之为“全局描述符表寄存器（GDTR）”，另一个用来存放局部描述符表的基址，称之为“局部描述符表寄存器（LDTR）”，它们的结构如下图所示：



（图八）

其中表限也即表的大小限制，它的使用与前面所描述的段限是类似的，因此，这里就不在描述了。

在保护模式下，以前实模式下的段寄存器还是有用的，不过它不再用来存放段的基址，而是用来存放“段选择子”，它的名字也变成了“段选择子寄存器”，在访问内存的时候，我们需要给出的是“段选择子”，而不是段基址了。

比如，我现在想使用全局描述符表中第二个表项，即其中的第二个“段描述符”，这个“段选择子”就需按如下的方式构成：

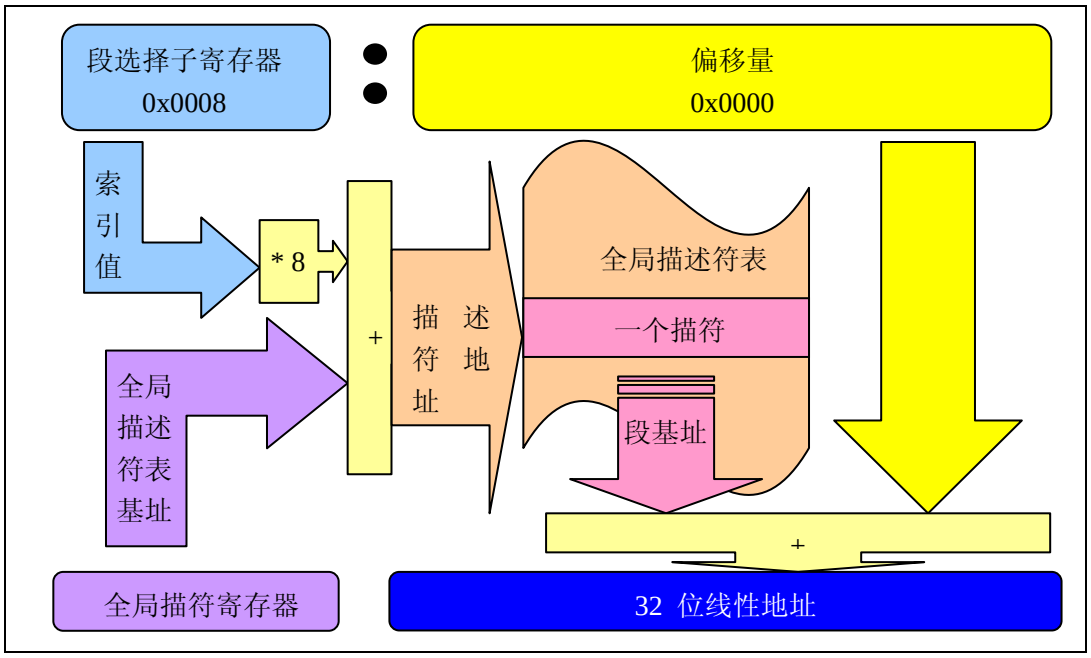
RPL：00，因为我们现在是在写操作系统，工作在 0 特权级

TI：0，我们使用全局描述符

索引值：1，我们使用第二个全局描述符，第一个全局描述符编号为 0，第二个为 1

因此，我们的“段选择子”为：0000 0000 0000 1000，也即 0x0008，因此，对于 0x0008:0x0000 这样一个逻辑地址，在保护模式下就应看成是使用全局描述符段中第二个描述符所描述的段，并且偏移量为 0 的内存地址。

这个逻辑地址的线性地址是怎样形成的呢？请看如下的图示：

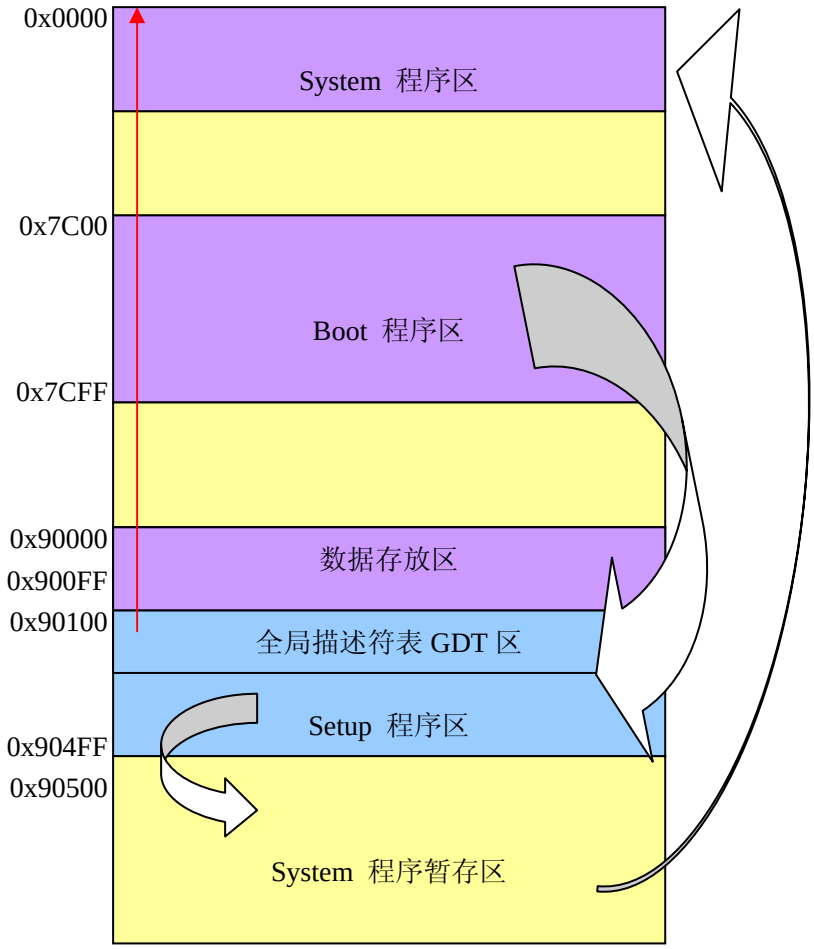


（图九）

相信，从上图中你可以清楚的看出一个逻辑地址是怎样转换为一个 32 位的线性地址的。

五、pyos 引导程序编写

pyos 是一个正在编写中的操作系统，是一个实验中的项目，关于编写的目的与动机我已在前言中谈论过了，这里，仅就此篇所讲述的内容，谈谈 pyos 引导程序的编写，在编写期间参考了 linux 0.11 内核引导程序的编写，不过 pyos 并不是基于 linux 的，就它们的引导程序之间也有许多不一样的地方。下面我们先来看看 pyos 的整个引导区的内存安排：



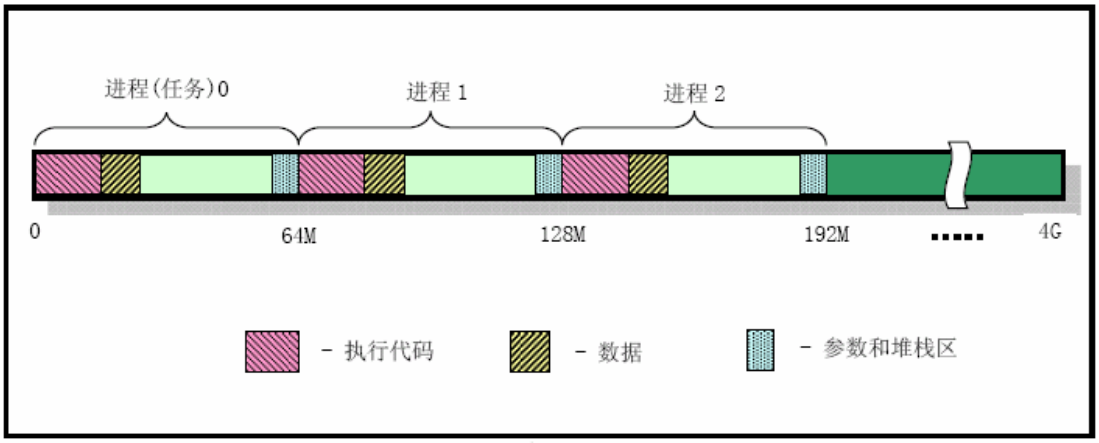
（图十）

上面一幅图就是 pyos 的内存安排图，也是引导程序流程图，pyos 是两级引导系统，首先是 boot 被 BIOS 读入，随后 boot 读入 setup，setup 读入 system 程序到暂存区，然后把 system 程序搬到内存顶部，并建立指向 system 程序所在段的段描述符及建立 GDT，然后切换 CPU 到保护模式，最后跳转到 system 程序中执行，至此 pyos 系统引导完毕，system 程序将是 pyos 真正的系统内核。图中的数据存放区用来存放在 boot、setup、pyos 三者程序间需要传递的参数。

之所以做成两级引导主要是考虑到以后扩展时的方便，各程序间都差不多是独立的，以后可以重写 boot 或者 setup 以提供更多可选择的引导方式。System 程序暂存区是因为如

前所述，不能直接将数据读到中断向量表中覆盖原中断向量表，当数据读完之后，不再调用中断了，才将程序搬到内存顶部覆盖原中断向量表，对于保护模式下的中断向量表，将由 system 程序负责建立，交给 system 使用的是一块完整而干净的内存。

对于 pyos 进程内存安排，准备参照 Linux 0.11 进行，内存安排如下：



（图十一，来源《Linux 0.11 内核代码完全注释》）

一个进程享有 64M 空间，4GB / 64M = 64，也即系统最大进程数为 64。因此一个段的段限为：64MB，每个进程占用全局描术符表中两个描述符，一个为数据段描述符，一个为代码段描述符，段限均为 64MB。

六、pyos 引导程序源代码

下面将提供 pyos 引导程序的全部源代码，因为 system 还未完全完成，因此这里只是让它简单的打印一个字符以示引导工作完成，代码中已有较为详尽的注释，如果仍有不太清楚的地方，可以去 http://purec.binghua.com（纯 C 论坛）操作系统实验专区，查看 pyos 以前的实验报告，上面有非常详尽的注释及相关原理说明，并详细描述了怎样编译及实验。

```
;文件名: boot.asm
;作　者: 谢煜波
;Emailv: xieyubo@126.com
;
;内存分配如下
;内存起始地址为 0x90000
;最大结束地址为 0x9ffff
;最大共 64KB
;所有启动代码在一个段内，方便调用
;启动代码共分两部分，一是 boot，一是 setup，这点照搬 linux 0.11 的设计
;但与其不同的是，boot 不会将自己搬到 0x90000 处，而直接跳到 0x90100 处运行
;0x90000~0x900ff (256B) 系统保留来存放一些从 BIOS 中取出的关键数据
;0x90100~0x904ff (1KB)：此处开始存放 setup，setup 大小为 1KB

[BITS 16]                                                    ;编译成 16 位的指令
[ORG 0x7C00]
;-------------------------------------------------------------------------------
```

```
jmp  Main
;-------------------------------------------------------------------------------------------
;数据定义
MSG                db              "Loading pyos ..."  ;输出信息
                   db              13 , 10 , 0         ;13 表示回车，10 表示换行，
                                                       ;0 表示字符串结束
BOOTSEG            equ             0x0000              ;boot 所在的段基址
SETUPSEG           equ             0x9000              ;setup 所在的段基址
SETUPOFFSET        equ             0x0100              ;setup 所在的偏移量
SETUPSIZE          equ             1024                ;setup 的大小，必须是 512 的倍数
BOOTDRIVER         db              0                   ;保存启动的驱动器号
;-------------------------------------------------------------------------------------------
ShowMessage:
  ;以下程序行为显示输出信息
  mov              ah , 0x0e                          ;设置显示模式
  mov              bh , 0x00                          ;设置页码
  mov              bl , 0x07                          ;设置字体属性

  .nextchar:
  lodsb
  or               al , al
  jz               .return
  int              0x10
  jmp              .nextchar

  .return:
  ret
;-------------------------------------------------------------------------------------------
Main:
  mov              [BOOTDRIVER] , dl                  ;得到启动的驱动器号

  ;以下程序设置数据段
  mov              ax , BOOTSEG
  mov              ds , ax
  mov              si , MSG

  call             ShowMessage                        ;显示信息

  ;读入 setup
  ;从磁盘的第二个扇区读到 0x90100 处
  .readfloopy:
  mov              ax , SETUPSEG
  mov              es , ax
  mov              bx , SETUPOFFSET
```

```
  mov               ah , 2
  mov               dl , [BOOTDRIVER]
  mov               ch , 0
  mov               cl , 2
  mov               al , SETUPSIZE / 512              ;读入扇区数( 2 个共 1KB )
  int               0x13
  jc                .readfloopy

  ;把启动驱动器号保存在 0x90000 处
  mov                al , [BOOTDRIVER]
  mov                [0] , al

  ;跳转
  jmp                SETUPSEG : SETUPOFFSET
;-------------------------------------------------------------------------
times 510-($-$$) db 0
db 0x55
db 0xAA
```

```
;文件名: setup.asm
;作　者; 谢煜波
;Emailv: xieyubo@126.com

;此 setup 程序完成 boot 未完成的启动工作，
;包括从 BIOS 中读出系统信息存放在指定位置
;初始化 GDT,LDT 表，完成从保护模式到实模式的转换
;实模式的代码也由此程序读入

[BITS 16]
[ORG 0x0100]
;-----------------------------------------------------------------------------
jmp                Main
;-----------------------------------------------------------------------------
SETUPSEG           equ                0x9000
SETUPOFFSET        equ                0x0100
SETUPSIZE          equ                1024          ;setup 的大小 1KB,必须是 512 的倍数
SYSTEMSEG          equ                0x0000
SYSTEMOFFSET       equ                0x0000
SYSTEMSIZE         equ                1024          ;SYSTEM 的大小 1KB,此值必须是 512 的倍
数
                                                    ;实际值可以不符

;下面定义临时 GDT 表的描述符
;总共定义三个段，一个空段由 intel 保留，一个代码段，一个数据段
```

```
gdt_addr:
                dw              0x7fff          ;GDT 表的大小
                dw              gdt             ;GDT 表的位置
                dw              0x0009
gdt:
  gdt_null:
                dw              0x0000
                dw              0x0000
                dw              0x0000
                dw              0x0000
  gdt_system_code:
                dw              0x3fff          ;段限(0x3fff+1)*4KB=64KB
                dw              0x0000
                dw              0x9a00
                dw              0x00c0
  gdt_system_data:
                dw              0x3fff
                dw              0x0000
                dw              0x9200
                dw              0x00c0
;-------------------------------------------------------------------------------
;等待键盘控制器空闲的子程序
Empty_8042:
  in            al , 0x64
  test          al , 0x2
  jnz           Empty_8042
  ret
;-------------------------------------------------------------------------------
Main:
  ;初始化寄存器，因为 Bios 中断及 call 会用到堆栈或 ss 寄存器
  ;在 CPU 启动或复位时是由 BIOS 初始化的，而现在进行了段转移，需要我们重新设置
  mov           ax , SETUPSEG
  mov           ds , ax
  mov           es , ax
  mov           ss , ax
  mov           sp , 0xffff

  ;-----------------------------------------------------------------------
  ;从 BIOS 中到底应读出哪些有用信息，现在还不确定，因此暂时跳过此功能块
  ;-----------------------------------------
  ;0x90000 (1B): 保存启动驱动器号，由 boot 程序存入
  ;-----------------------------------------------------------------------

  ;下面读入 system 到 setup 程序的后面
```

```
;因为 0x00000 现在是放 BIOS 中断的地方，因此还不能直接将 system 读到 0x00000 处，
;否则将无法调用 BIOS 中断读入磁盘
.readfloopy:
mov                 ax , SETUPSEG
mov                 es , ax
mov                 bx , SETUPOFFSET + SETUPSIZE
mov                 ah , 2
mov                 dl , [0]
mov                 ch , 0
mov                 cl , 1 + 1 + SETUPSIZE / 512        ;system 所在的启始扇区
                                                        ;第一个 1 是指从 1 开始记数
                                                        ;第二个 1 是 boot 所占扇区数
mov                 al , SYSTEMSIZE / 512               ;读入扇区数( 2 个扇区共 1KB )
int                 0x13
jc                  .readfloopy

;下面将读入的 system 搬移到 0x00000 位置
cld
mov                 si , SETUPOFFSET + SETUPSIZE
mov                 ax , SYSTEMSEG
mov                 es , ax
mov                 di , SYSTEMOFFSET
mov                 cx , SYSTEMSIZE / 4
rep movsd

;下面开始为进入保护模式而进行初始化工作
cli                                                     ;关中断
lgdt                [gdt_addr]                          ;载入 gdt 的描述符

;下面打开 A20 地址线
call                Empty_8042
mov                 al , 0xd1
out                 0x64 , al
call                Empty_8042
mov                 al , 0xdf
out                 0x60 , al
call                Empty_8042

;下面设置进入 32 位保护模式运行
mov                 eax , cr0
or                  eax , 1
mov                 cr0 , eax

jmp dword           0x8:0x0
```

```
;-------------------------------------------------------------------------------
times 1024-($-$$) db 0
```

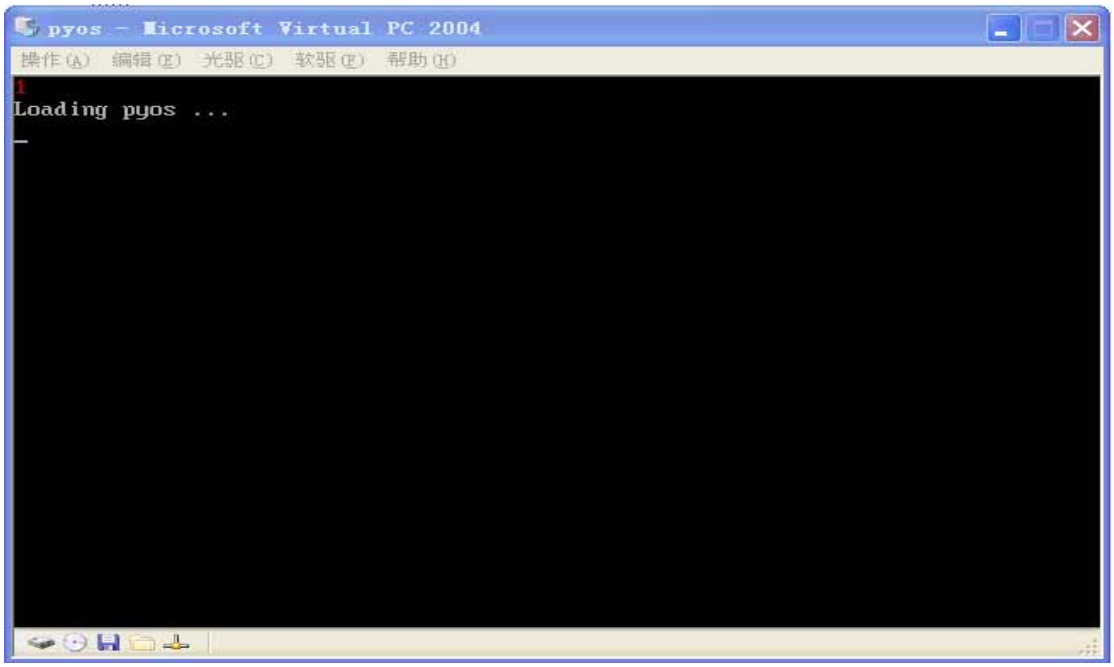
```
;文件名: kernel.asm
;作　者: 谢煜波
;Emailv: xieyubo@126.com

[BITS 32]
[ORG 0x0]
;-------------------------------------------------------------------------------
jmp  Main
;-------------------------------------------------------------------------------
Main:
  ;设置寄存器
  mov                ax , 0x10
  mov                ds , ax

  mov                cl , '1'
  mov                [0xb8000] , cl
  mov                cl , 0x04
  mov                [0xb8001] , cl
  jmp $
```

以上程序中有一个地方本篇及以前的实验报告中也未提到，这就是 A20 地址线的问题，对于有关 A20 地址线的问题，在《Linux 0.11 内核源代码完全注释》中有非常详细的描述，作者还列举了其它几种打开 A20 地址线的方法，并分析了可能存在的问题。这是一本非常好的书，推荐大家阅读。纯 C 论坛上（http://purec.binghua.com）可以下载本书的电子版（PDF 格式），也可以在上面找到另外一些相关资源。

下面就是运行时的截图，现在它只能引导，什么也干不了，希望下次它能多干一点~~



参考资料

1. 《IA-32 Intel® Architecture Software Developer's Manual Volume 3:System Programming Guide》 （Intel 2001）
2. 《Linux 内核 0.11 完全注释》（赵炯，2003）
3. 《计算机组成原理》（唐朔飞，高等教育出版社）

【全文完 • 责任编辑：iamxiaohan@hitbbs】

有空的时候，多读读书吧

哈尔滨工业大学 计算机体系结构实验室 王凯峰

今天闲来无事，整理了一下我的书籍，发现买的书虽多，但真正完全读过一遍的很少，不禁痛恨自己，暴殄天物，愧对这么多大师不说也对不起我可怜的 money 呀，：（。

于是乎想起自己在本科时代的那段浑浑噩噩的时光，那时候倒是也想读书，读好书，但是不知道什么书好，什么书该读，什么不该读。我想现在的不少同学也可能有一样的感觉，于是我觉得把我这几年来读过的书推荐一下，还是很有必要的，希望学弟学妹们能够充分利用时间，把自己培养的棒棒的：-）

Ok，闲话少说，let's begin……

1) 操作系统方面：

如果你对操作系统原理很好奇，想一探究竟，推荐你必读的几本书，记住不要在汤子赢的书上浪费精力，看过以下的书，就知道什么是水平上的差距了：）

Abraham Silberschatz 的两本书：

1.《实用操作系统概念（影印版）》高教出版社

2.《操作系统概念（第六版影印版）》高教出版社

这个作者的 level 是顶尖级的，来自贝尔实验室，是目前世界上操作系统方面的领军人物，我个人认为比坦尼伯姆要强，虽然坦也很强：），这两本书第二本和第一本很多地方相似，区别在于第二本理论偏重一些，第一本实例讨论的更多一些。这两本书别看很厚，但是写的非常流畅，属于比较易读的一类。

3.《现代操作系统》---坦尼伯姆

这本书我没完整看过，只是大略的翻过，是第 4 本书的升级版，里面添加了一些新的操作系统方面的讨论，原理部分比第四本稍有增强。个人认为，是除了前面两本之外的最好的书。

4.《操作系统-设计与实现》

这个比较有名了，主要是分析 minix 源代码的书，顺带着讲了下原理，应该说是偏于实践的，可能当年的 linus 的教材就是这本，影响较大，可以帮助你了解一个文件系统或系统调用之类是如何实现的。当然，和目前的操作系统来讲，稍简单了一些。想做 linux kernel hacker 的同学此书可以一读。

5.《操作系统：现代观点（第二版试验更新版）》

《Operating Systems: A design-Oriented Approach》

这两本一般，但是还是比国内抄袭的教材强很多，可以作为补充阅读的书籍。

6．与特定操作系统相关的书：

《understanding linux kernel》：

千万别买中文版，那叫一个烂

《linux internal》：

这两本我都有电子版，是打印出来看的：（

《linux 内核情景分析》：

这三本是最 nb 的 linux 内核分析书籍，前两本讲 2.2 内核，第三本讲 2.4，好是好，就是钱遭罪，赫赫，要 100 多块吧。

7.《4BSD 操作系统设计与实现》：

不说少了，原来 berkeley 那帮写 bsd 的其中几个人写的，经典就是它了，前几年我恨不得直接花$去 amazon 买了：（，不过这几年国内出版业发展真快，原来很多梦寐以求的书，现在都摆上书架了，haha，很有成就感呢~~~

8.《unix 操作系统设计》：

古老的 unix 设计方面的书籍，应该说这本书在 unix 世界里面的影响是十分巨大的，很多后来的 unix 分枝，思想都是缘于此书。里面主要讲解 unix 各个部分实现时所用的算法，其中一些目前还在使用中。想了解一下 unix 实现但又没什么时间扣 minix 或 linux 内核的朋友可以看看，在这本书上花费几十个小时，绝对超值：）

2） 计算机系统结构：

我是搞体系结构的，所以对这方面还是比较了解，不免又要批判一下国内的书籍了，李学干的书，我怎么看都像是上古作品，讨论的东西基本上是 80 年代以前的玩意儿，看了也是白看，它里面介绍的东西，基本上我们是没有可能遇到的。除非你去计算机历史博物馆。清华的郑伟民有两本书，一厚一薄，都叫计算机体系结构，薄的讲的太浅，反正我看完了还是不清楚体系结构是干啥的，做什么用处。厚的我没全看完，但是感觉和李学干的书一样，不少都是 copy 黄凯的那本高级计算机体系结构，至少我就看见过一张一模一样的图。讲到这里，不免要推荐一下哈工大计算机系唐朔飞老师的**《计算机组成原理》**，（虽然不能算是体系结构），这本书相当好，我想这和唐老师严谨认真的治学作风十分不开的，计算机组成原理的书，别的都不用读了，我还没见过比这本更好的。

国外体系方面的书：

1． hennessy 和 patternson 的《计算机硬软接口》和《计算机量化研究方法》：

作者一个是斯坦福的校长，一个是伯克里的资深教授，体系方面最牛会议的审稿人。这两本书可谓是千锤百炼，绝世好书，呵呵。我都推荐给进入体系实验室的师弟师妹们。《硬软接口》里面讲的是简单的处理器体系结构，最难的地方也只讲到流水线，本科水平阅读应该没有问题。它与国内的书不同，并不是从理论方面泛泛而谈，而是针对某个问题，深入讨论，不弄个水落石出决不罢休，这点使得读起来非常过瘾。第二本《量化研究方法》比较难读，不仅仅是因为英文写的有点晦涩（相对于硬软接口），而且讨论的东西也是最先进的，新接触这一领域的读者比较难于理解，这本书基本上涵盖了 2000 年以前的微处理器体系结构方面的先进技术，如果你想了解 2000 年左右微处理器体系结构已经发展到什么程度，可以从这本书读起，它以前的书可以不用读拉。对了，忘了说明，这两本书都是主要讨论微处理器体系结构的，呵呵，这也是它们的新颖之处，我个人也觉得现在还大讲特将什么向量机有点不合时宜。强烈推荐，看完整两本书，你就会知道处理器到底是如何工作的。另外，千万别买这本书的翻译版，看着闹心，翻译的往往和原意差出好几百里地，浪费时间不说，也对身心健康不好哦。

2． William stalling 的《计算机体系结构-性能设计》：

由于看了前两本书，所以这本就不太起眼了。想比较而言，这本书讨论的问题太过于表面化，缺乏对问题的深入剖析，看完之后没有那种刺破肉皮见骨头的快感。不过我觉得这也很正常，因为 william stalling 只是个科技作家(别看 china-pub 上好像对他的吹捧好像是个神，其实他也是人)，他和前两位世界级的教授根本不在一个数量级上。不过老威倒是也真够狠，什么都写，os 阿，网络阿，体系阿，看着比坦尼伯姆还 nb。看过这本书之后，我发誓再也不看他的书了。这本书当作科普读物可以，了解一下体系结构。

3． 黄凯的《高等计算机系统结构》：

这本书影响非常大，国内的凡是体系结构的书，基本上很多都是 copy 这本的。但是，我不得不说，他太老了，老的已经不再适合做体系的教材了。计算机技术发展多么迅速阿，

这个 10 几年前的东西现在怎么应用啊？里面讲的机器估计以后没人能见到，呵呵，说到这里不得不向大牛 cray 致敬（虽然大师已经挂了），他的 cray 机真是太牛了，不让 ibm 专美于前啊。Sorry，扯远了，这本书如同鸡肋，弃之可惜（毕竟是好书），但食之无味。喜欢跳出微处理器的圈子，试图俯瞰一下计算机体系结构各个领域在 1990 年以前是什么情况的同志可以 look through 一下。

4. 坦尼伯母的《结构化计算机组成》：

这本书不是专讲体系结构的，它连从 c 程序，到编译，汇编，连接加上硬软接口 ISA 和处理器内部实现统统讲到。一气贯通，让你了解这个程序是咋在处理器上刺溜刺溜跑起来的，对大家对于计算机系统的宏观认识非常有帮助。

3)编译原理：

编译这块我了解不多，应该说不怎么了解，我甚至还打算从头学一边编译原理：（，不过这方面的好书我可是注意好久了：现在隆重推荐：

1）《编译原理：技术与工具》：

作者之一就是著名的 ullman，我记得是计算机界目前论文被引用最多的一个活人（fix me），他在编译和数据库方面地位很高，这方面我估计哈工大李建中老师的门下弟子比我知道的多多了，我就不多说了。这本书据说是讲解经典编译原理的宝典，也被称为“龙书”。想学编译？ok，别的书先抛在一边，把这本先读了再说。但是要提醒一下，目前的编译器设计基本上和体系结构结合的很紧密了，有的编译器甚至连程序运行时功耗都考虑进去了，所以，要想一窥现在编译器内部构造，光看这本就不够了。

2）《Optimizing Compilers for Modern Architectures: A Dependence-based Approach》

这本书能弥补第一本的不足，讲的都是如何让编译器利用体系结构方面的技术。不过，国内没有出版，不过，别灰心，呵呵，我这里有电子版。如果实在感兴趣，看看编译器是如何根据体系结构进行优化的，可以打印出来读。

3）最近出的影印版《高级编译器设计与实现》

也和第二本书一样，能弥补第一本龙书的不足，对于编译优化和相关分析等奖的比较深入,也是非常值得读的一本好书。

Ok，累死我了，先推到这里把，对了，希望在别的领域很有研究的同学也推荐些书，让我们知识共享，经验共享，共同进步，共同提高。

【全文完 • 责任编辑：kylix@hitbbs】

Linux 下 SOCK_RAW 的原理和应用

哈尔滨工业大学 网络与信息安全实验室 肖颖

近些天看了一些关于网络捕包/发包的文章，现在跟大家分享一下吧。

网络捕包常用的方式有写驱动程序（内核模块）和用原始套接字的方法，本文着重讨论后者。

创建 socket 连接的时候，domain 参数设为 PF_PACKET，它可以提供低层次的数据包接口，第二个参数设为 SOCK_RAW，第三个参数 protocol 应该为 htons(ETH_P_ALL)，这样才可以得到传到网卡的所有数据包，那么为什么设置为 SOCK_RAW 就能捕到原始数据包呢，让我们来看看 Linux 的内核代码（内核版本 2.4.20-8）。

drivers/net 下包含驱动程序，当网卡收到数据包之后，会产生一个中断，交给接收函数，接收函数首先检查数据包的正确性，如果数据包正确无误，驱动程序则会为该数据包开辟内存空间，调用 netif_rx 函数（core/dev.c）进行处理，netif_rx 通过中断信号 NET_RX_SOFTIRQ 调用 net_rx_action，而在初始化函数 net_dev_init 中，中断 NET_TX_SOFTIRQ 和 NET_TX_SOFTIRQ 已经被打开。

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

net_rx_action 根据数据包协议类型，在 ptype_base Hash 链表中找到相应协议的接收函数，然后调用。以 ip 协议为例，其调用 ip_rcv 函数（net/ipv4/ip_input.c）

ip_rcv 通过简单的错误校验之后，调用 ip_rcv_finish 函数，该函数调用 ip_route_input 获取路由信息，如果该包不是转发的，就调用 ip_local_deliver 进行处理。

下面着重分析 ip_local_deliver 函数。

声明： int ip_local_deliver(struct sk_buff *skb)

此函数专门负责 ip 分片重组，然后把网络数据包传到上层处理。

```
int ip_local_deliver(struct sk_buff *skb)
{
    /*
     *  Reassemble IP fragments.
     */
    if (skb->nh.iph->frag_off & htons(IP_MF|IP_OFFSET)) {
        skb = ip_defrag(skb);
        if (!skb)
            return 0;
    }
    return NF_HOOK(PF_INET, NF_IP_LOCAL_IN, skb, skb->dev, NULL,
                   ip_local_deliver_finish);
}
```

该函数对重组好的数据包执行 NF_HOOK。

NF_HOOK 在 linux/netfilter.h 里声明

```
#ifdef CONFIG_NETFILTER_DEBUG
#define NF_HOOK nf_hook_slow
```

```
#else
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn)          \
(list_empty(&nf_hooks[(pf)][(hook)])                         \
 ? (okfn)(skb)                                               \
 : nf_hook_slow((pf), (hook), (skb), (indev), (outdev), (okfn)))
#endif
```

先判断钩子列表 nf_hooks[(pf)][(hook)]是否为空，如果有响应的钩子调用，则先处理钩子函数。接下来执行(okfn)(skb)，在 ip_local_deliver 中，相应的就执行 ip_local_deliver_finish(skb)。ip_local_deliver_finish 也在 net/ipv4/ip_input.c 实现。

```
static inline int ip_local_deliver_finish(struct sk_buff *skb)
{
    int ihl = skb->nh.iph->ihl*4;
#ifdef CONFIG_NETFILTER_DEBUG
    nf_debug_ip_local_deliver(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/
    __skb_pull(skb, ihl);
#ifdef CONFIG_NETFILTER
    /* Free reference early: we don't need it any more, and it may
           hold ip_conntrack module loaded indefinitely. */
    nf_conntrack_put(skb->nfct);
    skb->nfct = NULL;
#endif /*CONFIG_NETFILTER*/
       /* Point into the IP datagram, just past the header. */
       skb->h.raw = skb->data;
    {
         /* Note: See raw.c and net/raw.h, RAWV4_HTABLE_SIZE==MAX_INET_PROTOS */
         int protocol = skb->nh.iph->protocol;
         int hash = protocol & (MAX_INET_PROTOS - 1);
         struct sock *raw_sk = raw_v4_htable[hash];
         struct inet_protocol *ipprot;
         int flag;
         /* If there maybe a raw socket we must check - if not we
          * don't care less
          */
         if(raw_sk != NULL)
              raw_sk = raw_v4_input(skb, skb->nh.iph, hash);
         ipprot = (struct inet_protocol *) inet_protos[hash];
         flag = 0;
         if(ipprot != NULL) {
              if(raw_sk == NULL &&
                 ipprot->next == NULL &&
                 ipprot->protocol == protocol) {
                   int ret;
                   /* Fast path... */
```

```
                    ret = ipprot->handler(skb);
                    return ret;
                } else {
                    flag = ip_run_ipprot(skb, skb->nh.iph, ipprot, (raw_sk != NULL));
                }
            }
            /* All protocols checked.
             * If this packet was a broadcast, we may *not* reply to it, since that
             * causes (proven, grin) ARP storms and a leakage of memory (i.e. all
             * ICMP reply messages get queued up for transmission...)
             */
            if(raw_sk != NULL) {    /* Shift to last raw user */
                raw_rcv(raw_sk, skb);
                sock_put(raw_sk);
            } else if (!flag) {         /* Free and report errors */
                icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0);
                kfree_skb(skb);
            }
        }
    return 0;
}
```

我们看到有这样两行

```
            if(raw_sk != NULL)
                raw_sk = raw_v4_input(skb, skb->nh.iph, hash);
```

这表明，系统内核会先对 SOCK_RAW 进行处理，调用响应的函数 raw_v4_input，在其中调用 raw_recv。

弄清楚 SOCK_RAW 的原理之后，我们来捕包吧。

正常状态下，系统会丢弃不是发给自己的网络包，如果我们需要得到经过网卡的所有数据包，需要把网卡设置为混杂模式(promisc mode)，此时需要调用 ioctl 函数设置。

int ioctl(int d, int request, ...);

ioctl 控制与文件描述符 d 相关的设备，其中 d 必须是已被打开。

request 是文件相关的请求代码。

对网卡设置需要的结构体在 net/if.h 中定义，

```
struct ifreq
  {
# define IFHWADDRLEN    6
# define IFNAMSIZ  IF_NAMESIZE
    union
      {
    char ifrn_name[IFNAMSIZ];   /* Interface name, e.g. "en0".  */
      } ifr_ifrn;
    union
      {
    struct sockaddr ifru_addr;
```

```
    struct sockaddr ifru_dstaddr;
    struct sockaddr ifru_broadaddr;
    struct sockaddr ifru_netmask;
    struct sockaddr ifru_hwaddr;
    short int ifru_flags;
    int ifru_ivalue;
    int ifru_mtu;
    struct ifmap ifru_map;
    char ifru_slave[IFNAMSIZ];  /* Just fits the size */
    char ifru_newname[IFNAMSIZ];
    __caddr_t ifru_data;
     } ifr_ifru;
 };
# define ifr_name  ifr_ifrn.ifrn_name /* interface name  */
```

此时我们需要两个步骤，
1　设置 ifrn_name，表明当前设置的网卡名
2　设置 ifru_flags，变为 promisc 模式

一切搞定之后就可以捕包了。
调用标准的 socket 接收函数就可以了。接收到的包是“RAW”数据包，你可以在里面看到每一层协议的具体内容。
下面就是连接到 bbs.hit.edu.cn 的第一个包。

00000000h: 00 08 E2 83 A0 0A 00 30 48 21 B9 81 08 00 45 10
00000010h: 00 3C C4 B7 40 00 40 06 0D 16 CA 76 F3 EE CA 76
00000020h: E0 02 80 0A 00 17 35 93 3B 4F 00 00 00 00 A0 02
00000030h: 16 D0 D6 49 00 00 02 04 05 B4 04 02 08 0A 01 12
00000040h: FF F8 00 00 00 00 01 03 03 00

ethernet 的头部
00 08 E2 83 A0 0A 目标 MAC，此时为局域网里的网关 MAC
00 30 48 21 B9 81 源 MAC
08 00 上层协议类型，对于 ip 协议而言，该值为 0x0800。
接下来就是该 **ip 包**的内容了
4 ipv4
5 ip 头部长度 5 * 4 = 20 个字节
10 tos
00 3C IP 包长度 3C 个字节
C4 B7 该 IP 包标识，便于 ip 分片处理，一般每发出一个 ip 包，系统会自动加 1
40 00 Flag ip 分片偏移量，由于 Flag 为 3 个字位，4 为 00000100，所以该 ip 没有被分片。
40 TTL，每通过一个路由器，改值减 1
06 上层协议，此包上层为 TCP 协议，下面我们会详细讨论。
0D 16 16 IP 头部 checksum

CA 76 F3 EE 源 IP 地址
CA 76 E0 02 目的 IP 地址 202.118.224.2
80 0A 源端口
00 17 目的端口 23 (telnet)
35 93 3B 4F TCP 序号
00 00 00 00 ACK 序号
A TCP 头部长度 0xA * 4 = 40 个字节
02 TCP Flag，此时的 02 表示 syn 包，请求建立连接
16 D0 接收窗口长度
D6 49 TCP 头部的 checksum
00 00 urgdata 的指针
剩下的全是 Options，因为此包为 syn，无需另外的 data。

在 netinet/in.h 中可以看到 ip 上层（传输层）协议的类型

```
  IPPROTO_HOPOPTS = 0,     /* IPv6 Hop-by-Hop options.  */
  IPPROTO_ICMP = 1,        /* Internet Control Message Protocol.  */
  IPPROTO_IGMP = 2,        /* Internet Group Management Protocol. */
  IPPROTO_IPIP = 4,        /* IPIP tunnels (older KA9Q tunnels use 94).  */
  IPPROTO_TCP = 6,         /* Transmission Control Protocol.  */
  IPPROTO_EGP = 8,         /* Exterior Gateway Protocol.  */
  IPPROTO_PUP = 12,        /* PUP protocol.  */
  IPPROTO_UDP = 17,        /* User Datagram Protocol.  */
  IPPROTO_IDP = 22,        /* XNS IDP protocol.  */
  IPPROTO_TP = 29,         /* SO Transport Protocol Class 4.  */
  IPPROTO_IPV6 = 41,       /* IPv6 header.  */
  IPPROTO_ROUTING = 43,    /* IPv6 routing header.  */
  IPPROTO_FRAGMENT = 44,   /* IPv6 fragmentation header.  */
  IPPROTO_RSVP = 46,       /* Reservation Protocol.  */
  IPPROTO_GRE = 47,        /* General Routing Encapsulation.  */
  IPPROTO_ESP = 50,        /* encapsulating security payload.  */
  IPPROTO_AH = 51,         /* authentication header.  */
  IPPROTO_ICMPV6 = 58,     /* ICMPv6.  */
  IPPROTO_NONE = 59,       /* IPv6 no next header.  */
  IPPROTO_DSTOPTS = 60,    /* IPv6 destination options.  */
  IPPROTO_MTP = 92,        /* Multicast Transport Protocol.  */
  IPPROTO_ENCAP = 98,      /* Encapsulation Header.  */
  IPPROTO_PIM = 103,       /* Protocol Independent Multicast.  */
  IPPROTO_COMP = 108,      /* Compression Header Protocol.  */
  IPPROTO_RAW = 255,       /* Raw IP packets.  */
```

从中可以知道，上层协议类型不足 256 个。
系统通过上层协议字段确定应该交给什么类型的程序处理，一般的防火墙也只关注已知的协议，如果传输的时候把该字段设置为未知的类型，然后创建一个响应的 socket 就可以接收了，而且它不会在 netstat –an 中出现，这是一种隐藏端口的常用方法。

OK，至此捕包已经差不多了，我们再看看发包方面的。

创建了 SOCK_RAW 之后，我们可以就用该 socket 发出 raw 数据包。要发送的数据包需要用户自行构造，类似于捕到的包，自下向上构造出数据包，应当注意的是，checksum 不能有误，如果接受方收到数据包发现 checksum 不对就会简单的 drop 掉。

我们将构造好的发向 bbs.hit.edu.cn:23 的 syn 数据包存入变量 sendbuf 中

```
char sendbuf[] =
"\x00\x08\xE2\x83\xA0\x0A\x00\x30\x48\x21\xB9\x81\x08\x00\x45\x10\x00\x3C\xF1\x09\x40\x00\x40\x06\xE0\xC3\xCA\x76\xF3\xEE\xCA\x76\xE0\x02\x80\x05\x00\x17\x73\x61\x99\x46\x00\x00\x00\x00\xA0\x02\x16\xD0\xD2\x8D\x00\x00\x02\x04\x05\xB4\x04\x02\x08\x0A\x00\xB7\x68\x4F\x00\x00\x00\x00\x01\x03\x03\x00";
```

然后设置 sockaddr_ll 结构体，

```
dst.sll_family = PF_PACKET;
dst.sll_protocol = htons(ETH_P_ALL);
dst.sll_ifindex = 2; /* interface "eth0" */
dst.sll_pkttype=PACKET_HOST;
dst.sll_hatype = IP_HDRINCL ;
```

需要注意的是，sll_ifindex 表明网卡编号，此时 lo 为 1。

最后掉用 sendto(sock,buf,size,0,(struct sockaddr*) &dst,sizeof(dst));就大功告成了。

用 tcpdump 可以看出

```
202.118.243.238.32773 > bbs.hit.edu.cn.telnet: S 1935776070:1935776070(0) win 5840 <mss 1460,sackOK,timestamp 12019791 0,nop,wscale 0> (DF) [tos 0x10]
0x0000   4510 003c f109 4000 4006 e0c3 ca76 f3ee        E..<..@.@....v..
0x0010   ca76 e002 8005 0017 7361 9946 0000 0000        .v......sa.F....
0x0020   a002 16d0 d28d 0000 0204 05b4 0402 080a        ................
0x0030   00b7 684f 0000 0000 0103 0300 00               ..hO.........
bbs.hit.edu.cn.telnet > 202.118.243.238.32773: S 3827335961:3827335961(0) ack 1935776071 win 5792 <mss 1460,sackOK,timestamp 34455988 12019791,nop,wscale 0> (DF)
0x0000   4500 003c 0000 4000 3f06 d2dd ca76 e002        E..<..@.?....v..
0x0010   ca76 f3ee 0017 8005 e420 8319 7361 9947        .v.........sa.G
0x0020   a012 16a0 a7b0 0000 0204 05b4 0402 080a        ................
0x0030   020d c1b4 00b7 684f 0103 0300                  ......hO....
```

我们向 bbs.hit.edu.cn 发出的 syn 包，被回应 syn ack，证明主机已经收到。类似的原理，我们还可以自己构造很多类型的数据包发出去。

SOCK_RAW 不借助其他第三方类库，编出来程序较小，不过分析/构造数据包比较复杂，本文目的在于让大家熟悉 SOCK_RAW 的方法和原理，以便进一步作相关的研究和工作。

【全文完 • 责任编辑：true@hitbbs】

连接器和加载器

原著: J.R.Levine

北京工业大学 刘彦博(lyb_dotnet@hotmail.com) 译

译序

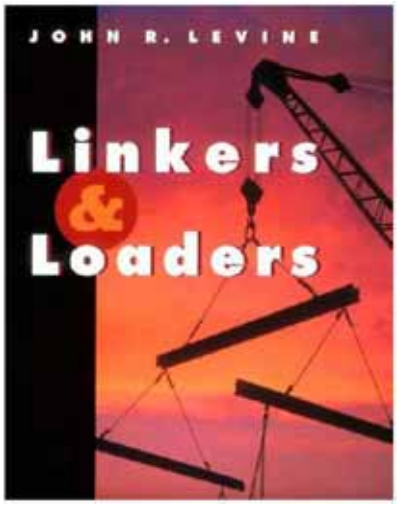
搞底层开发的至臻境界我想大概要算操作系统开发和编译器开发了。计算机，就其字面意义，无非是冷血废铁一块，之所以有血有肉，正是操作系统从中支撑，之所以有经有脉，那便是编译器的功劳了。然而，这血肉之躯如何令周身经脉游走自如，这周身经脉又如何牵动血肉之躯动停行止，却有不少不为人知的秘密，令许多初涉底层的人感到神秘异常。这便是连接器和加载器的职责了。熟悉汇编语言程序设计的人应该知道，每当我们 xasm 后，必当 link 一番，源文件变成的二进制代码才肯乖乖地在特定的操作系统之上运行。而即使是 link 过的程序，也只是存在于磁盘或其他存储介质上的指令，如何依照操作系统的意愿将它们们搬运到内存中的指定位置并开始执行，loader 功不可没。

不幸的是，现在介绍操作系统和编译原理的书籍很多，但无论哪一方面的书籍、无论哪一本书，都忽略了连接和加载这一细节问题。偶尔提到，也不过是“编译之后经过连接即可生成可执行文件”或“操作系统将可执行文件加载到内存中的指定位置开始运行”云云。令狗狗发动灵鼻，爬遍”遍布世界的蜘蛛网（World Wide Web）”也难寻有关连接器和加载器的只言片语，只有少数文章介绍了.net 加载器或 Java 加载器，也无非是宣传其如何如何安全、如何如何快速等等，毫无实际意义可言。

然而本书却可称得上是“唯一”一本极详细介绍连接器和加载器的文献了。本书说不上偏重理论还是实践，抑或是偏重方法，总之是一本极为实用的书。

原著

John R. Levine: *Linkers and Loaders*



第 0 章 前言

原著: J.R.Levine

北京工业大学 刘彦博(lyb_dotnet@hotmail.com) 译

简介

连接器和加载器几乎在计算机出现伊始就作为了软件工具包的一部分，因为若允许以多个模块的形式建成程序而不是将作为一个整体，连接器和加载器就是不可或缺的工具。

早在 1947 年，程序员们就开始使用原始的加载器来获取存贮在不同的磁带上的程序例程并将它们组合、重定向到一个程序中。二十世纪 60 年代早期，这些加载器进化为羽翼丰满的连接编辑器。由于程序存储器仍然昂贵和优先、计算机（以现在的标准来看）仍然很慢，这些连接器往往具有复杂的特性，用来建立复杂的存储 复用结构以将大程序塞到小存储器中，以及用来重新编译已经连接的程序以节约从头开始建立一个程序所学的时间。

二十世纪 70 年代到 80 年代之间，连接技术没有什么进展。连接器愈加简单，虚拟存储将存储管理的工作从应用程序和占位程序中转移到操作系统中，同时，计算机更快了，磁盘也更大了，通过替换模块来重新建立一个已连接的程序也比仅重连接改变的部分要容易得多了。90 年代，连接起又复杂起来，添加了对现代特性包括动态连接的共享库和 C++中不同寻常的需求的支持。具有宽指令字和编译器调度存储访问的新型处理器架构，如 IA64，还为连接器提出了新的需求以确保已连接的程序中代码的复杂要求得以实现。

谁应该阅读本书？

本书是和很多读者：

- **学生**：编译器构造和操作系统课程通常缺乏对连接和加载的重视，这通常是由于连接过程看起来是微不足道的或显而易见的。尽管当使用的语言是 Fortran、Pascal 或 C，并且操作系统没有使用存储映射和共享库时确实是这样，但如今看来却不再是了。C++、Java 和其他面向对象语言需要一个更为复杂的连接环境。存储映射的可执行程序、共享库和动态连接影响着一个操作系统的方方面面，而操作系统的设计者却不愿意担连接问题的风险。
- 从业**程序员**也应该知道连接器都做什么，尤其对于现代语言。C++ 对连接器有着独特的需求， 连接期间出现的非预期情况通常导致大型 C++程序产生难于诊断的 BUG（最著名的就是静态构造器，它以程序员所不期望的顺序执行）。连接特性，如共享库和动态连接能够提供巨大的灵活性和效率，当然是在适用的条件下。
- **语言设计者和开发者**在建立语言和编译器时要了解连接器做什么以及能够做什么。以掌握 30 余年的编程任务在 C++中却是自动的，这依赖于连接器处理的细节。（考虑要在 C 中获得 C++模板的等价物，一个程序员都需要做什么；或者如何在一个有

上百个 C 源文件的程序开支运行之前确保调用了每一个初始化历程。）将来的语言将通过更为强大的连接器来自动完成更多的程序管理任务。连接器还将参与到全局程序优化中，因为连接器是编译过程中唯一能够处理整个程序的代码并将它们作为一个单元进行转换的阶段。

（当然，书写连接器的人也应该需要这本书。但这世上所有的连接器作者或许都聚集在一间屋子中并且它们中的一半都有着这样一份副本，因为他们审查着原稿。）

章节预览

第 1 章，**连接和加载**，对连接器的历史作了一个简短的概述，并讨论了连接过程中的各个阶段。这一章以一个简短但完整的例子——从输入的目标文件到一个可执行的“Hello、world”程序的过程中连接器的运行过程——结束。

第 2 章，**架构问题**，从连接器设计的角度回顾计算机架构。这一章研究了 APARC——一个典型的精简指令集架构、IBM360/370——一个古老但仍然有用的寄存器-存储器架构。以及自成一家的 Intel x86。重要的架构方面包括存储架构、程序定址架构和单个指令的地址域布局。

第 3 章，**目标文件**，研究了目标（对象）文件和可执行文件的内部结构。这一章从非常简单的文件——MS-DOS .COM 文件——开始，进一步研究了更复杂的文件，包括 DOS EXE、Windows COFF 和 PE（EXE 和 DLL）、Unix a.out 和 ELF 以及 Intel/Microsoft OMF。

第 4 章，**存储分配**，涵盖了连接的第一步——为连接后的程序的段分配存储空间，并辅以真实编译器例子。

第 5 章，**符号管理**，涵盖了符号绑定和解析，这个过程将一个文件中对另一个文件中的名字进行的符号引用解析为一个机器地址。

第 6 章，**库**，涵盖了目标代码库的建立和使用，以及库的结构和性能问题。

第 7 章，**重定位**， 涵盖了地址重定位、调整程序中的目标代码已在运行时反映实际地址的过程。这一章还涵盖了地址无关的代码（PIC，Position Independent Code），以这种方式建立的代码可以避免重定位，以及这样做所需要的开销和可以获得的利益。

第 8 章，**加载和复用**，涵盖了加载的过程，从文件获取一个程序并装入到计算机内存中进行运行。这一章还涵盖了树状的复用结构——一个古老但仍然有效的节约地址空间的技术。

第 9 章，**共享库**，研究在多个程序之间共享一个库代码的一份单独的副本都需要什么。这一章集中讨论了静态连接共享库。

第 10 章，**动态连接和加载**，继续第 9 章，对动态连接的共享库进行讨论。这一章详细地介绍了两个例子，Windows32 动态连接库（DLL，Dynamic Link Libraries）和 Unix/Linux ELF 共享库。

第 11 章，**高级技术**，研究精密的现代连接器所做的很多事情。这一章涵盖了 C++所要求的新特性，包括”名字管理”、全局构造器和析构器、模板展开以及重复代码的清除；其他技术，包括增量连接、连接时垃圾收集、连接时代码生成和优化、加载时代码生成，以及剖面和工具。这一章以对一种比其他连接器予以更复杂的连接模型——Java 连接模型的概述告终。

第 12 章，**参考**，一个带有注解的参考条目。

项目

从第 3 章到第 11 章有一个持续的项目——用 Perl 开发一个很小但功能俱全的连接器。尽管 Perl 并不是一个成品连接器的实现语言，但它对于团队项目来说却是一个卓越的选择。Perl 可以解决低级语言如 C 或 C++中一些小毛病所导致的程序终止，这可以使学生们集中精力于项目所需的算法和数据结构。Perl 可以免费地用在很多现代计算机上，包括 Windows 95/98 和 NT、Unix 和 Linux，并且还有很多优秀的书籍可以教会新手使用 Perl。（参考第 12 章可以获得一些建议。）

第 3 章中最初的项目是建立一个连接器的骨架，可以按照一个简单但完整的目标格式读取和写入文件，接下来的各章中依次向这个编译器中添加功能，直到最后产生一个羽翼丰满的连接器，能够支持共享库并产生动态可连接的对象。

Perl 能够处理任意的二进制文件和数据结构，因此项目所建立的连接器将可以适用于处理本地目标格式。

感谢

很多很多的人都慷慨地将他们的时间投入到对本书原稿的阅读和评论中，这些人既有出版社的评论员，也有 comp.compilers usenet 新闻组中阅读并评论过原稿的在线版本的读者。他们包括（按字母顺序）：Mike Albaugh、Rod Bates、Gunnar Blomberg、Robert Bowdidge、Keith Breinholt、Brad Brisco、Andreas Buschmann、David S. Cargo、John Carr、David Chase、Ben Combee、Ralph Corderoy、Paul Curtis、Lars Duening、Phil Edwards、Oisin Feeley、Mary Fernandez、Michael Lee Finney、Peter H. Froehlich、Robert Goldberg、James Grosbach、Rohit Grover、Quinn Tyler Jackson、Colin Jensen、Glenn Kasten、Louis Krupp、Terry Lambert、Doug Landauer、Jim Larus、Len Lattanzi、Greg Lindahl、Peter Ludemann、Steven D. Majewski、John McEnerney、Larry Meadows、Jason Merrill、Carl Montgomery、Cyril Muerillon、Sameer Nanajkar、Jacob Navia、Simon Peyton-Jones、Allan Porterfield、Charles Randall、Thomas David Rivers、Ken Rose、Alex Rosenberg、Raymond Roth、Timur Safin、Kenneth G Salter、Donn Seeley、Aaron F. Stanton、Harlan Stenn、Mark Stone、Robert Strandh、Bjorn De Sutter、Ian Taylor、Michael Trofimov、Hans Walheim、and Roger Wong。

这些人为这本书中大多数正确的陈述负有责任。其他的错误仍旧是作者的责任。（如果你发现了一些，请按照后面的地址和我们联系，他们可以将其添加到后续版本中。）

我要特别感谢我的编辑 Morgan-Kaufmann Tim Cox 和 Sarah Luger，他们容忍了我在写作过程中过多的耽搁，并把这本书的片断集中到一起。

联系我们

这本书有一个支持站点 http://linker.iecc.com。这里包含本书的示例章节、perl 代码示例和项目的目标文件，以及更新和勘误表。

你可以为作者发送电子邮件，地址为 linker@iecc.com。作者会阅读所有的邮件，但由于数量庞大，可能不会迅速地回答所有的问题。

【待续 • 责任编辑：worldguy@hitbbs】

第1章 连接和加载

原著: J.R.Levine

北京工业大学 刘彦博(lyb_dotnet@hotmail.com) 译

内容

连接器和加载器都做些什么？

任何连接器或加载器的基本工作都很多简单：将更加抽象的名字绑定（binding）到更加具体的名字，以允许程序员可以使用更加抽象的名字来编写程序。也就是说，它可以将程序员写的一个名字如 getline 绑定到“从模块 iosys 中的可执行代码的开始处定位 612 字节”。或者可以将一个更加抽象的数值地址如“从该模块的静态数据之后定位 450 个字节”绑定到一个具体的数值地址上。

地址绑定：一个历史性观点

观察连接器和加载器都做什么的一个有用的方法是研究它们在计算机程序系统的开发中所处的地位。

最早的计算机程序完全用机器语言编写。程序员将符号化的程序写在纸张上，再将它们汇编为机器代码并将这些机器代码制成计算机中的触发器，或者可能将它们打孔到纸带或卡片上。（真正刺激的是直接用开关构成代码。）如果程序员使用了符号地址，程序员必须通过他自己的手动翻译将这些符号绑定到地址上。如果发现一条指令必须被添加或删除，整个程序都必须手动地进行检查并且调整所有受指令添加或删除影响的地址。

这里的问题是将名字绑定到地址的时机太早了。汇编器通过让程序员使用符号名字来编写程序，而由汇编器将名字绑定到机器地址来解决这一问题。如果程序发生了变化，程序员只需重新汇编它，而地址分配的工作由程序员转到了计算机。

代码库使得地址分配的问题更加复杂。由于计算机可以执行的基本操作非常简单，有用的程序通常由子程序组成以执行更高级和更复杂的操作。计算机中通常保存了预先编写好并通过调试的子程序库，这样程序员在编写新程序时就可以利用它们，而不是自己编写这些子程序。程序员将这些子程序加载到主程序中就可以得到一个完整的工作程序。

程序员使用子程序库甚至先于使用汇编器。1947 年，主持过 ENIAC 项目的 John Mauchly 写了一些可以加载从磁带上存储的程序目录中选择的子程序的加载程序，这需要重定位子程序代码来反映它们加载的地址。这也许很令人吃惊，两个基本的连接器功能——重定位和库搜索，居然先于汇编器出现，因为 Mauchly 希望程序和子程序都是用机器语言编写的。带重定位的加载器允许自程序的作者和用户在编写每一个子程序时都可以假设它们从位置 0 开始，并且将实际地址的绑定推迟到子程序被连接到一个特定的主程序中时。

随着操作系统的出现，带重定位的加载器有必要从连接器和库中分离出来。在操作系统出现之前，每个程序在其处理过程中都拥有机器的整个内存，计算机中的所有地址都是可用的，因此程序可以使用固定的内存地址来汇编和连接。但操作系统出现之后，程序必须和操作系统甚至可能是和别的程序共享计算机的内存。这意味着直到操作系统将程序加载到内存中以前是不可能知道程序运行的实际地址的，最终的地址绑定从连接时推迟到了加载时。连接器和加载器现在分割了这个工作，连接器负责部分的地址绑定——在每个程序中分配相对地址，而加载器完成最终的重定位步骤以分配实际地址。

由于系统变得越来越复杂，它们要求连接器完成越来越复杂的名字管理和地址绑定。Fortran 程序使用多个子程序和公共块，数据区域由多个子程序共享，它要求连接器来布置存储并为子程序和公共块分配地址。连接器越来越需要对目标代码库进行处理。这既包括用 Fortran 和其他语言编写的应用库，又包括通过调用已编译的代码来处理 I/O 和其他高级操作的编译器支持库。

程序很快变得比可用内存还要大，因此连接器提供复用——一种允许程序员协调程序的不同部分来共享相同内存的技术，每个复用都在程序的其他部分调用它们的时候才加载。从 1960 年左右磁盘出现，到二十世纪 70 年代中期虚拟存储的推广，存储复用一直广泛应用于大型机；之后在二十世纪 80 年代早期又再次以完全相同的形式出现在微型机中，最后再二十世纪 90 年代虚拟存储出现在 PC 机上之后慢慢隐退。它依然存在于内存有限的嵌入式环境中，并且可能由于考究的程序员或编译器需要控制存储以提高性能而出现在其他地方。

随着硬件重定位和虚拟存储的出现，连接器和加载器变得不再复杂，因为每个程序又可以获得整个的地址空间了。可以按照使用固定地址加载的形式来连接程序，通过硬件而不是软件重定位来处理加载时重定位。但是具有硬件重定位的计算机总是要运行多于一个的程序，常常是一个程序的多个副本。当一台计算机运行一个程序的多个实例时，在程序所有的运行实例之间有某些部分是相同的（特别是可执行代码），而其他部分对于每个实例是唯一的。如果不变的部分可以从变化的部分中分离出来，操作系统就可以只使用不变部分的一个副本，这可以节省相当可观的内存。编译器和汇编其变为可以用不同的节（section）来建立目标代码，一个节中只放有只读代码，而其他节中放入可写代码，连接器必须能够合并各种类型的所有节以使得连接后的程序中所有的代码在一个地方而所有的数据在另一个地方。这里尽管没有出现复用地址绑定，但它确实存在，因为地址仍然是在连接时分配的，但更多的工作推迟到了连接器为所有的段分配地址的时候。

甚至当不同的程序运行于同一台计算机上时，这些不同的程序也常常产生共享很多公共代码的情况。例如，几乎每一个用 C 写的程序都要用到如 fopen 和 printf 之类的例程、数据库应用程序都要用一个很大的访问库来连接到数据库，以及在一个诸如 X Window、MS Windows 或 Macintosh 这样的 GUI 下运行的所有程序都要用到一些 GUI 库。很多操作系统现在

都提供共享库（shared library）给程序使用，因此使用了一个库的所有程序可以共享该库的一份单独的副本。这既提高了运行时性能又节省了很多磁盘空间；在一些小型程序中，这些公共库例程甚至比程序本身要占用更多的空间。

在较为简单的静态共享库中，每个库在其建立的时候就被绑定到特定的地址，而连接器也是在连接的时候就将程序中对库例程的引用绑定到这些特定的地址上。静态库显得很不方便，因为每当库发生变化的时候，都有可能必须重新连接程序，并且建立静态共享库的过程显得非常枯燥。系统又添加了动态的连接库，其中的节和符号并未绑定到实际的地址，直到使用了该库的程序开始运行。有的时候这个绑定甚至推迟到这（使用了该库的程序开始运行）之后很久；使用完全的动态连接，对调用过程的绑定直到第一次调用才完成。此外，程序可以在开始运行时绑定到库，而当程序执行到一半时再加载库。这为扩展程序的功能提供了一种强大而高效的方式。Microsoft Windows 广泛地利用了共享库的运行时加载（著名的 DLL，Dynamically Linked Library）来构造和扩展程序。

连接 VS 加载

连接器和加载器执行很多相关但概念上独立的动作。

- **程序加载**：从次要存储器（直到大约 1968 年才特指磁盘）上将程序复制到主要存储器中以准备运行。有些情况下加载只包括将数据从磁盘复制到内存中，其他情况下则还包括重定位存储、设置保护位或安排虚拟内存将虚拟地址映射到磁盘页上。
- **重定位**：编译器和汇编其一般在建立目标代码文件的时候都令程序的地址从零开始，但很少有计算机允许你将你的程序加载到零地址。如果一个程序由多个子程序组成，所有的子程序必须被加载到不交叉的地址中。重定位就是为程序的各个部分分配加载地址，并调整程序的代码和数据以反映已分配的地址的过程。在很多系统中，重定位发生不止一次。一个连接器从多个子程序建立一个程序并且从零开始连接输出程序非常常见，多个子程序会重定位到大程序中的指定位置。之后在程序加载时，系统会决定实际的地址，连接后的程序会作为一个整体重定位到加载地址。
- **符号确定**：当一个程序由多个子程序构成时，一个子程序对其他子程序的引用由符号（symbol）完成；一个主程序可能要用到一个称为 sqrt 的平方根例程，而数学库中定义了 sqrt。连接器通过计算 sqrt 在库中分配的位置并根据调用者的目标代码来修正这个位置，最后给 call 指令提供正确的地址。

尽管连接和加载看起来似乎有所重复，但将加载程序的程序定义为加载器而将确定符号的程序定义为连接器是有原因的。它们都可以完成重定位，当然也存在一体化的连接加载器，可以完成所有三个功能。

重定位和符号确定之间的界限可能是模糊的。 这是由于连接器已经能够确定对符号的引用。处理代码重定位的一种方法就是按照程序每个部分的基地址来分配符号，然后将可重定位地址视为对基于基地址的符号的引用。

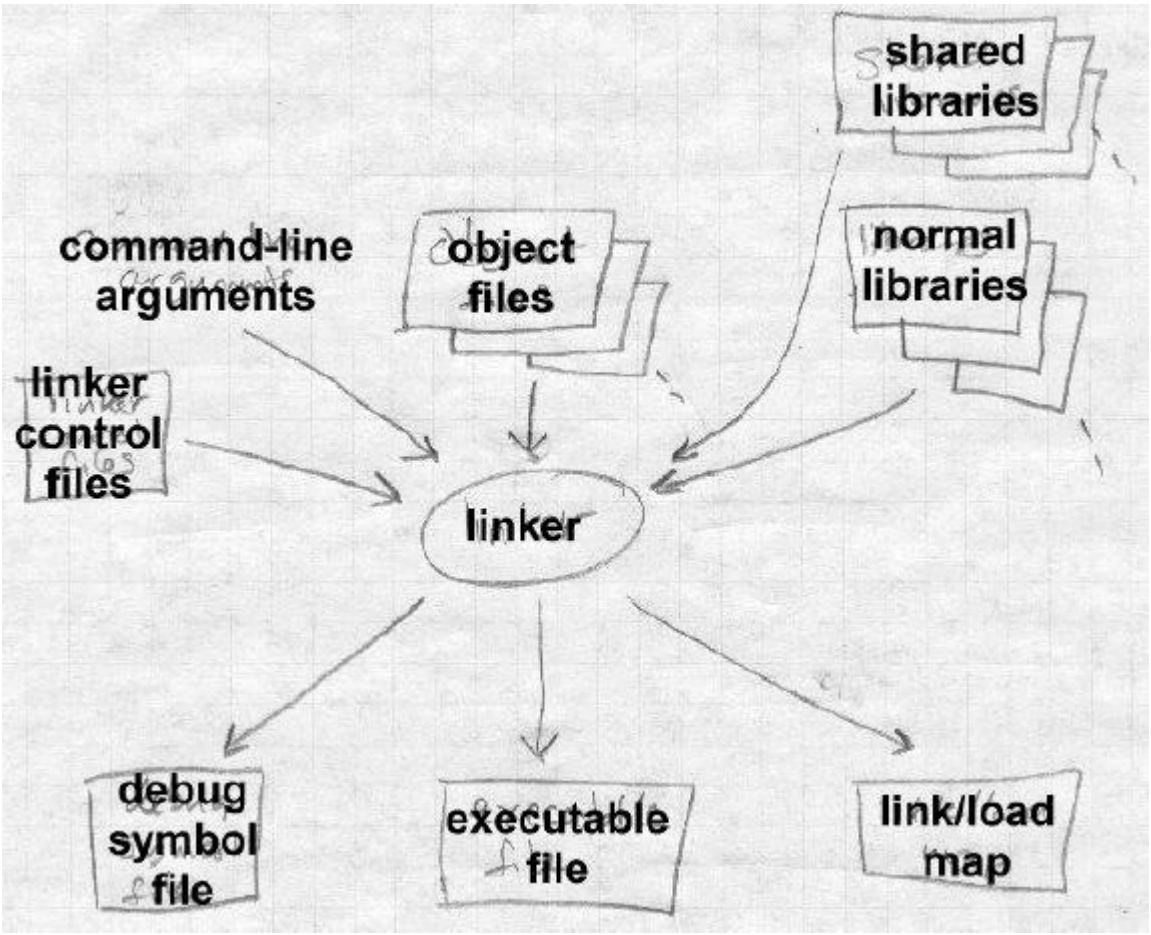
连接器和加载器共同具有的一个重要特性是它们都会修正目标代码，其他能够完成这种工作且广泛使用的工具可能也就是调试器（debugger）了。这是一种唯一且强大的特性，尽管其细节都是因机器特定的，并且一旦发生错误将引起莫名其妙的 BUG。

两遍连接

现在我们转入连接器的一般结构。连接，和编译或汇编类似，基本上是一个两遍（two-pass）的过程。连接器以一组输入目标文件、库，以及可能的命令文件作为输入；并产生一个输出目标文件，以及可能的辅助信息如加载图或包含了调试器符号的文件。如图 1-1 所示。

图 1-1 连接过程

连接器获取输入文件、产生输出文件和其他冗余文件的图解



每个输入文件中都包含了一组段（segment），其中大块相邻的数据或代码被放到输出文件中。每个输入文件还包含至少一个符号表（symbol table）。一些符号是导出的，在一个文件中定义而在其他文件中使用，如通常的例程名字在一个文件中定义后可以在其他文件中调用。其他符号是导入的，在文件中用到但没有定义，如通常一个文件中可能要通过一个没有定义过的名字来调用一个例程。

当一个连接器运行时，它必须首先扫描输入文件以确定段的大小并收集对所有符号的定义和引用。它建立一个罗列输入文件中定义的所有段的段表，以及一个包含所有导入导出符号的符号表。

根据这一遍所得的数据，连接器为符号分配数值地址、检测输出文件中的段的大小和位置，并且指出输出文件中都有什么。

第二遍使用了第一遍所收集的信息，用以确定实际的连接过程。它读取并重定位目标代码，用符号引用来替换数值地址，并调整代码和数据中的内存地址以反映重定位段地址，最后将从定位代码写入到输出文件中。接下来它写出这个输出文件，通常还要加上头信息、重定位段和符号表信息。如果程序使用了动态连接，符号表还要包含能够提供信息以供运行时连接器确定动态符号所需。在很多情况下，连接器本身会在输出文件中产生少量的代码或数据，诸如用于在复用或动态连接库中调用例程的“粘贴代码（glue code）”，或者指向用于在程序开始时执行的初始化例程的指针数组。

不论程序是否使用动态连接，输出文件中都会包含一个符号表用以重新连接或调试，这个符号表并不会有程序本身使用，但是其他处理输出文件的程序可能会用到。

一些目标格式是可重新连接的，也就是说，一个连接器的输出文件可以用作后续连接器的输入文件。这就要求输出文件包含一个和输入文件中类似的符号表，以及出现在输入文件中的其他辅助信息。

几乎所有的目标格式都提供调试符号，以使得当程序运行在一个调试器的控制之下时，调试器能够使用这些符号，以使程序员能够通过源程序中所使用的行号和名字来控制程序。依据目标类型的细节，调试符号可能和待连接的符号混杂在一个单独的符号表中，或者可能是连接器之外的一个单独的表，有时还可能是调试器中的冗余表。

一少部分连接器看起来是一遍的。它们通过在连接过程中将输入文件的部分或全部的内容缓存到内存或磁盘中，然后在读取这些缓存过的材料。由于这种实现只是一种技巧，并没有影响到连接的两遍本质，因此我们在后面将不予讨论。

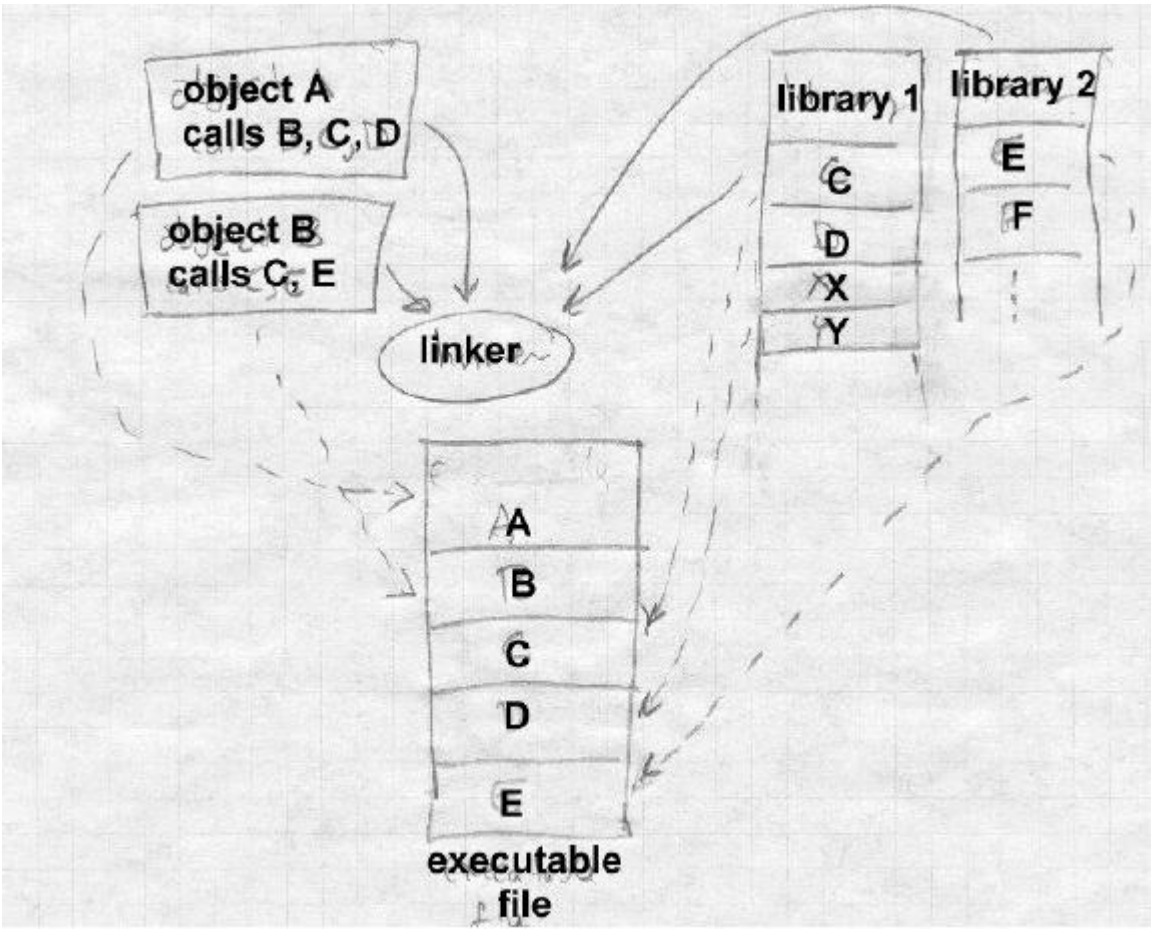
目标代码库

所有的连接器都以一种形式或另一种形式支持目标代码库，很多连接器还提供对多种共享库的支持。

目标代码库的基本原则非常简单，如图 2 所示。一个库和一组目标代码文件差不多。（甚至，在一些系统上你可以逐字地将一组目标文件绑定到一起并作为一个连接库。）在连接器处理完所有的常规输入文件后，如果仍然有某些导入的名字是未定义的，它会继续处理库并连接那些能够导出这些未定义名字的库。

图 1-2：目标代码库

目标文件首先进入连接器，接着是包含很多文件的库。



共享库使得这个任务变得有些复杂，因为它将连接时的一部分工作转移到了加载时。连接器在运行的时候识别能够解决未定义名字的共享库，但不想程序中连接任何东西，连接器向输出文件中注明能够在哪个库中找到相应的符号，以使得相应的库能够在程序加载的时候被绑定。细节请参见第 9 章和第 10 章。

重定位和代码修正

连接器和加载期的动作的核心就是重定位和代码修正。当一个编译器或汇编器生成一个目标文件时，它所生成的代码使用文件中定义的代码和数据的非重定位地址，而且通常用零表示在其他地方定义的代码和数据。作为连接过程的一部分，连接器修正目标代码以反映实际的地址分配。例如，考虑这段用于通过使用 eax 寄存器将变量 a 中的内容移到变量 b 中的 x86 代码。

```
mov a, %eax
mov %eax, b
```

如果 a 在同一个文件的十六进制位置 1234 处定义而 b 是从其他地方导入的，生成的目标代码将是：

```
A1 34 12 00 00 mov a, %eax
A3 00 00 00 00 mov %eax, b
```

每个指令都包含了一个单字节的操作码后跟一个四字节的地址。第一条指令有一个对 1234（字节颠倒的，因为 x86 使用自优至左的字节顺序）的引用，而第二条指令有一个对 0

的引用，因为 b 是未知的。

现在假设连接器连接了这段代码，导致 a 所在的节被定位在十六进制位置 10000 字节处，而 b 出现在十六进制位置 9A12。连接器将代码修正为：

```
A1 34 12 01 00 mov a, %eax
A3 12 9A 00 00 mov %eax, b
```

也就是说，它将 10000 加到第一条指令的地址上使其引用 a 的重定位地址 11234，并且修正了 b 的地址。这些调整不仅影响到指令，一个目标文件的数据部分中的所有指针都要同时调整。

在老式的、具有很小的地址空间并且是直接寻址的计算机上，这个修正过程相当简单，连接器仅仅仅仅需要处理一种或两种地址格式。现代计算机，包括所有的 RISC，都要求相当复杂的代码修正。没有哪个单独的指令能够包含足够的位以保存一个直接地址，因此编译器和连接器必须使用更加复杂的寻址技巧以处理任意地址上的数据。在某些情况下，它可能需要两条或三条指令来合成一个地址，其中每一条指令包含地址的一部分，然后使用位操作来将这些部分组合为一个完整的地址。在这种情况下，连接器必须准备好对每一条指令进行适当的修正，如向每条指令中插入一个地址的某些位。其他情况下，一个或一组例程中用到的所有地址被放到一个数组中作为一个“地址池（address pool），初始化代码将一个机器寄存器设置为指向该数组的指针，而代码在需要加载该地址池之外的指针时将以该寄存器作为一个基址寄存器。连接器可能必须通过一个程序中用到的所有地址来建立这个数组，然后修正指令使得它们能够引用地址池的适当入口。我们将这些内容放到了第 7 章。

一些系统要求不论加载到哪些地址空间都能正确工作的地址无关代码（position independent code）。连接器通常必须提供附加的技巧以支持它，如将不能做到地址无关的部分分离出来，以及安排这两部分进行通信。（参见第 8 章）

编译驱动器

在很多情况下，连接器的操作对于程序员来说是不可见的，或者几乎是这样，因为它作为编译过程的一部分被自动地运行了。很多编译系统具有一个编译驱动器（compiler driver），可以根据需要自动地调用编译器的各个阶段。例如，如果程序员有两个 C 语言源文件，Unix 系统上的编译驱动器将会像这样运行一系列程序：

- 在文件 A 上运行 C 预处理器，创建经过预处理的 A
- 在经过预处理的 A 上运行 C 编译器，创建汇编文件 A
- 在汇编文件 A 上运行汇编器，创建目标文件 A
- 在文件 B 上运行 C 预处理器，创建经过预处理的 B
- 在经过预处理的 B 上运行 C 编译器，创建汇编文件 B
- 在汇编文件 B 上运行汇编器，创建目标文件 B
- 在目标文件 A 和 B 以及系统 C 库上运行连接器

也就是说，它将每一个文件编译为汇编代码然后是目标代码，并且将这些目标代码包括需要的系统 C 库中的例程连接到一起。

编译驱动器通常比这要聪明。它们通常比较源文件和目标文件的创建日期，并且只编译更改过的源文件。（Unix 中的 make 程序是一个典型的例子。）尤其是当编译 C++和其他面向对象语言时，编译驱动器运用各种技巧来围绕着连接器或对象格式的限制工作。例如，C++模板定义了一个可能无穷的相关例程组，因此通过找到程序实际使用的有限的模板例程组，

一个编译驱动器能够不使用模板例程而将程序的目标文件连接到一起、通过读取连接器的错误消息来得知什么是未定义的、调用 C++编译器来产生必要的模板例程的目标代码以及重连接。我们将在第 11 章涵盖了一些这样的技巧。

连接器命令语言

每个连接器都有一些命令语言来控制连接过程。至少连接器需要包含目标文件和要连接的库的列表。通常还需要一个包含可能的选项的很长的列表：是否要保存调试符号、是使用共享库还是非共享库、使用多种可能的输出格式中的哪一个等等。很多连接器允许通过一些途径来指定连接后的代码所绑定的位置，这在使用一个这样的连接器来连接一个系统内核或其他不需要受操作系统控制的程序时是派得上用场的。在支持多重代码和数据段的连接器中，连接器命命令语言可以指定段被连接的顺序、对某些段进行特殊对待以及其他一些应用指定的选项。

有四种通用技术用于将命令传递给连接器：

- **写在命令行中**：很多系统具有一个命令行或其等价物，通过它我们可以传递混合了的文件名和选项开关。这是 Unix 和 Windows 连接器的常用途径。在那些具有命令行长度限制的系统上，通常可以指示连接器去从一个文件中读取命令并将它们视为是通过命令行读取的。

  混合在目标文件中：有些连接器，如 IBM 大型机连接器，在一个单独的输出文件中可以接受可选的目标文件和连接器命令。这要追溯到打孔卡片的年月，当时的人们积累目标卡片并在一个读卡机上对命令卡片进行手工打孔。
- **嵌入到目标文件中**：有些连接器，特别如 Microsoft 的，允许将连接器命令嵌入到目标文件中。这允许一个编译器把对于一个文件所需要的连接命令放在这个文件自身中进行传递。例如，C 编译器可以传递命令以搜索标准 C 库。
- **分离的配置语言**：很少一部分连接器具有一个羽翼丰满的用于控制连接的配置语言。GUN 连接器，可以处理数量庞大的目标文件格式、机器架构以及地址空间转换，它有一个复杂的控制语言，能够允许程序员指定要连接的段的顺序、相似段的结合规则、段地址以及范围广阔的其他选项。其他连接器具有不那么复杂的语言来处理特定的特性如程序员定义的复用。

连接：一个真实的例子

我们以一个很小但真实的连接实例来结束对连接的介绍。图 3 显示了一对 C 语言源文件，m.c 有一个主程序，调用了名为 a 的例程，而 a.c 包含了这个例程，它又调用了库例程 strlen 和 write。

图 1-3：源文件

源文件 m.c

```
extern void a(char *);
int main(int ac, char **av)
{
    static char string[] = "Hello, world!\n";
    a(string);
}
```

源文件 a.c

```
#include <unistd.h>
#include <string.h>
void a(char *s)
{
    write(1, s, strlen(s));
}
```

主程序 m.c 在我的 Pentium 机上被 GCC 编译为一个 165 字节的目标文件，具有典型的 a.out 目标格式，如图 4 所示。这个目标文件包含一个固定长度的头、16 字节的包含只读程序代码的 text 段和 16 字节的包含了 string 的 data 段。这些之后是两个重定位入口，其中一个标记了在准备调用 a 时用来将 string 的地址放到栈顶的 pushl 指令，令一个标记了用于将控制转移到 a 中的 call 指令。符号表导出对_main 的定义，导入_a，并为调试器包含了两个其他符号。（每个全局符号都带有一个前导下划线，其原因在第五章中讲述。）注意 pushl 指令引用了十六进制地址 10——string 的暂时地址，因为它在同一个目标文件中；而 call 引用了地址 0，因为_a 的地址是未知的。

图 1-4：m.o 的目标代码

```
Sections:
Idx Name    Size      VMA       LMA       File off  Algn
  0 .text   00000010  00000000  00000000  00000020  2**3
  1 .data   00000010  00000010  00000010  00000030  2**3

Disassembly of section .text:

00000000 <_main>:
   0:  55              pushl  %ebp
   1:  89 e5           movl   %esp, %ebp
   3:  68 10 00 00 00 pushl  $0x10
   4:  32  .data
   8:  e8 f3 ff ff ff call   0
   9:  DISP32  a
   d:  c9              leave
   e:  c3              ret
...
```

子程序文件 a.c 被编译为一个 160 字节的目标文件，如图 5 所示，具有头、一个 28 字节的 text 段并且没有 data 段。两个入口标记了对 strlen 和 write 的调用，符号表导出_a 并且导入_strlen 和_write。

图 1-5：a.o 的目标代码

```
Sections:
Idx Name    Size      VMA       LMA       File off  Algn
  0 .text   0000001c  00000000  00000000  00000020  2**2
               CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data   00000000  0000001c  0000001c  0000003c  2**2
               CONTENTS, ALLOC, LOAD, DATA

Disassembly of section .text:

00000000 <_a>:
   0:  55              pushl  %ebp
   1:  89 e5           movl   %esp, %ebp
```

```
   3:  53             pushl  %ebx
   4:  8b 5d 08       movl   0x8(%ebp), %ebx
   7:  53             pushl  %ebx
   8:  e8 f3 ff ff ff call   0
   9: DISP32 _strlen
   d:  50             pushl  %eax
   e:  53             pushl  %ebx
   f:  6a 01          pushl  $0x1
  11:  e8 ea ff ff ff call   0
  12: DISP32 _write
  16:  8d 65 fc       leal   -4(%ebp), %esp
  19:  5b             popl   %ebx
  1a:  c9             leave
  1b:  c3             ret
```

为了产生可执行程序,连接器要合并这两个目标文件和一个 C 程序的标准启动初始化例程，以及 C 库中必要的例程。产生的部分可执行文件如图 6 所示。

图 1-6：可执行程序选段

```
Sections:
Idx Name    Size      VMA       LMA       File off  Algn
  0 .text   00000fe0  00001020  00001020  00000020  2**3
  1 .data   00001000  00002000  00002000  00001000  2**3
  2 .bss    00000000  00003000  00003000  00000000  2**3

Disassembly of section .text:

00001020 <start-c>:
    ...
    1092:  e8 0d 00 00 00 call   10a4 <_main>
    ...
000010a4 <_main>:
    10a4:  55            pushl  %ebp
    10a5:  89 e5          movl   %esp, %ebp
    10a7:  68 24 20 00 00 pushl  $0x2024
    10ac:  e8 03 00 00 00 call   10b4 <_a>
    10b1:  c9            leave
    10b2:  c3            ret
    ...
000010b4 <_a>
    10b4:  55             pushl  %ebp
    10b5:  89 e5          movl   %esp, %ebp
    10b7:  53             pushl  %ebx
    10b8:  8b 5d 08       movl   0x8(%ebp), %ebx
    10bb:  53             pushl  %ebx
    10bc:  e8 37 00 00 00 call   10f8 <_strlen>
    10c1:  50             pushl  %eax
    10c2:  53             pushl  %ebx
    10c3:  6a 01          pushl  $0x1
    10c5:  e8 a2 00 00 00 call   116c <_write>
    10ca:  8d 65 fc    leal   -4(%ebp), %esp
    10cd:  5b             popl   %ebx
    10ce:  c9             leave
    10cf:  c3             ret
    ...
000010f8 <_strlen>:
```

```
    ...
0000116c <_write>:
    ...
```

连接器合并了每个文件中的对应段，因此这里有一个合并了的 text 段、一个合并了的 data 段以及一个 bss 段（初始化为 0 的段，两个输入文件都没有使用）。每个段都被填充至 4K 边界以匹配 x86 页面尺寸，因此 text 段为 4K（减去一个出现在文件中但不是短的逻辑部分的 20 字节的 a.out 头），data 和 bss 段也分别是 4K。

合并了的 text 段包含成为 start-c 的库启动代码；然后是来自 m.o 的 text 段，被重定位到 10a4；来自 a.o 的，被重定位到 10b4；以及连接自 C 库的例程，被重定位到 text 的更高的地址处。 合并后的 data 段这里没有显示，它的合并次序和 text 段的合并次序相同。由于_main 的代码被重定位到十六进制地址 10a4,因此这个地址被填到了 start-c 中的 call 指令里。在_main 例程中,对 string 的引用被重定位到十六进制地址 2024——string 在 data 段中的最终地址，其中的调用地址被修正为 10b4——_a 的最终地址。在_a 中，对_strlen 和_write 的调用地址也被修正为这两个例程的最终地址。

最后的可执行程序中还包括了很多其他来自 C 库的例程，这里没有显示，它们直接或间接地由启动代码或_write 调用（如出错时调用的错误处理例程）。可执行程序不包含重定位数据,因为文件格式不是可重连接的,而且操作系统会将它加载到一个可知的固定地址处。它还会包含一个符号表以备调试器所用,尽管这个可执行程序并不使用符号而且符号表可以被去除以节省空间。

在这个例子中，连接自库的代码比程序本身的代码要大很多。这很平常，尤其是当程序使用了巨大的图形或窗口库时，这正是促进共享库（参见第 9 章和第 10 章）出现的原因。连接后的程序为 8K，而同样的程序使用共享库则只有 264 字节。当然，这只是一个玩具性的例子，但真实的程序也能够同样戏剧性地节省空间。

练习

将连接器和加载器划分为分离的程序有什么好处？在哪些情况下一个组合的连接加载器才是有用的？

过去的 50 年里产生的几乎所有操作系统都包含了一个连接器。为什么？

在这一章里我们讨论了对汇编或编译过的代码所进行的连接和加载。在一个能够直接解释源语言代码的纯解释型系统中,连接器和加载器是否有用呢？在一个能够将源代码转换为中间表示的系统中，如 P-code 或 Java 虚拟机中呢？

【待续 • 责任编辑：worldguy@hitbbs】

现代信息检索技术简介

哈尔滨工业大学 信息检索研究室 车万翔

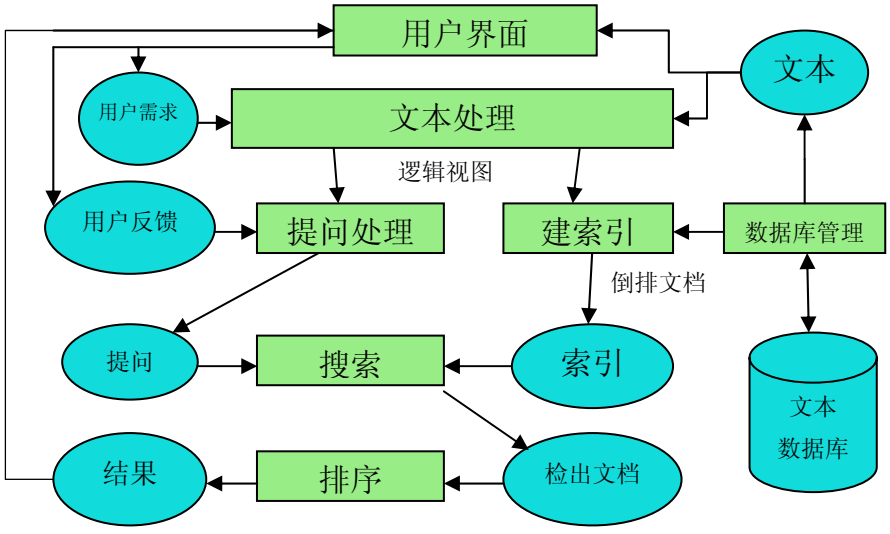
http://ir.hit.edu.cn/~car/

经常和一些人谈起信息检索(Information Retrieval, IR)，多数人想到的一个名词就是搜索引擎，如 Google，Yahoo，百度等。还有一部分人认为信息检索就是教人们如何查找资料的学问。但是这些理解却是未必详尽和有失偏颇的。我们这里说的信息检索是计算机应用学科的一个研究方向，搜索引擎只是其一重要的应用领域，更与情报检索非同一概念。那么信息检索的具体定义是什么，它的发展历史是怎样的，信息检索涉及哪些相关的领域，有哪些应用呢？希望我下面的文字能给读者一个答案。

信息检索指的是从非结构化的数据中找出与用户需求相关的信息。非结构化数据不但包括文本格式新闻、科技论文等，甚至还包含多媒体数据，如图像、视频、图形、音频等等。而由于图像、语音处理技术的限制，目前的信息检索对象主要是文本数据，其载体通常是 HTML、XML 格式等。

典型的信息检索任务(ad hoc)是给定了自然语言的文档集合，对于不同的用户需求(Query)找到相关的信息。而另外一种具有广泛应用的信息检索任务被称作信息过滤(filtering)，与信息检索的使用模式不同，信息过滤中用户的需求相对固定，面对实时变化的文档，同样找出与用户需求相关的文档。两种任务通常使用相似的检索技术，因此未经特殊说明，本文中信息检索通常指的是典型的信息检索任务。

典型的信息检索体系结构如图 1 所示，基本包括用户界面、文本处理、建索引、搜索、排序等。随着 Internet 的发展，基于 Web 的信息检索越来越受到人们的普遍重视，出现了以 Google 为代表的众多优秀互联网搜索引擎。与典型的信息检索系统不同的是，基于 Web 的检索系统必须通过在互联网上“爬行”的网络蜘蛛(spider)搜集网页，而且可以利用 Web 页面结构布局的信息和网页之间的链接结构信息，同时文档的更新又是频繁且不可控的。



图(1)：典型的信息检索体系结构

上个世纪六七十年代，随着计算机应用领域的扩展，简单的信息检索系统出现了。最初的信息检索系统面向小型的科学文摘数据库、法律和商业文档，此时信息检索处理的文档数量有限，普遍使用的检索模型为基本的布尔模型和向量空间模型等。到了八十年代，信息检索技术被应用到大型文档数据库中。在上个世纪九十年代，随着 Internet 的高速发展，信息不再是稀有金属，而是汪洋大海。如何在信息洪流中快速的找到需要的信息成为人们急需解决的问题，迫于用户的强烈需求，信息检索技术得到了迅猛的发展。

据统计，互联网上的网页数量由 1995 的 50M 张，达到了现在单 Google 就索引的 4G 张，在巨大的数据量向信息检索技术提出挑战的同时，激增的网络用户群也是给信息检索技术提供了机遇。来自中国互联网络信息中心的消息，截至 2004 年 6 月 30 日的统计报告显示，代表互联网发展规模的中国网民数量半年时间内激增 750 万，达到 8700 万人，同上一次调查（2003 年 12 月 31 日）相比增长 9.4%，和去年同期相比增长 27.9%。同时调查显示，中国网民中至少有 79%会经常使用搜索引擎，98%会使用搜索引擎。据预测，到 2006 年中国搜索引擎的产值将达到 23 亿人民币。

中文搜索引擎市场的争夺目前也已趋于白热化。2003 年底以前，中国搜索引擎市场的格局是：雅虎和 Google 都提供中文搜索服务，但没有正式进入中国。中国本土的搜索引擎服务商主要是百度、3721、中国搜索(慧聪搜索)。然而，这一切在 2004 年发生了彻底的变化。2003 年 11 月 21 日，雅虎中国收购 3721 公司。3721 的搜索服务成为了 YHAOO 中国的重要组成，YHAOO 正式进军中国搜索引擎服务市场。2004 年 6 月 15 日，Google 与其他七家共同投资者一起，收购了有全球最大中文搜索引擎之称的百度的部分股份。Google 在上市前终于有了中国搜索的概念。2004 年 6 月 21 日，雅虎中国除了坚固其门户搜索、3721 之外，推出了专门的中文搜索门户网站“一搜(www.yisou.com)”。2004 年 7 月 1 日，微软公司董事长比尔·盖茨在北京含蓄地表示，要加强 MSN 搜索开拓中国市场的力度。

可见，信息检索的前途一片光明，相关领域的科研人员也是任重而道远。

同时信息检索技术涉及的领域及其广泛，几乎涵盖了计算机科学的所有研究方向，包括，人工智能、自然语言处理、机器学习、数据库技术、并行计算、网络技术、图书和情报科学等等，甚至还需要图像、语音处理等支持。但是又与这些方向有所不同，可谓是包罗万象的一门学科。其中人工智能、自然语言处理以及机器学习为信息检索提供了知识表示、推理和用户建模的手段，数据库技术和并行计算技术为信息检索大规模的数据处理提供了强有力的支持，网络技术是基于 Web 的信息检索不可或缺的前端，并且随着网络技术的发展，网络带宽的增大，将来图像、语音处理等技术也必将为多媒体信息检索所用。

也有一些人认为目前的信息检索技术是数据库技术与自然语言处理技术的结合。其中数据库专注于研究结构化数据，比如关系表，而不是自由文本，它只能处理定义好了的查询式，并且查询式和数据的语义都非常清晰相比。而自然语言处理技术使检索能够在意义层面而不是仅仅在关键词层面进行，极大地增强了信息检索系统可用性。

除了互联网搜索引擎外，信息检索技术具有非常广泛的应用，在数字图书馆、内容安全、智能商务、电子政务、远程教育等方面都有其用武之地。

目前，国内外都有大批的科研机构投身于搜索引擎的研究中来，如美国 CMU 大学的语言技术中心(LTI)，美国南加州大学的信息科学研究所(ISI)，中科院计算所，复旦大学等。

哈尔滨工业大学信息检索研究室(HIT-IRLab)自 2001 年 3 月 1 日正式成立之日起，在研究室主任李生教授以及常务负责人刘挺教授领导下，一直致力于信息检索及其相关技术的研究与开发。并承担着国家自然科学基金、国家 863、国家信息关防等多项国家级课题。在 2003 年国家 863 举办的技术评测中获得了自动文摘项目第一名。

哈工大信息检索研究室的研究重点是信息过滤和信息抽取技术以及支撑这两项技术的自然语言处理技术。信息过滤旨在根据用户的兴趣偏好对信息进行分类，信息抽取旨在将非

结构化信息转化为结构化信息，自然语言处理将使这两项任务能够在真正的内容层面展开。信息过滤和信息抽取将为人类驾驭网络时代高速膨胀的信息资源提供强有力的工具。

信息过滤的研究重点在于用户兴趣建模、相似度计算、匹配、分类、聚类、相关反馈算法等，信息抽取的研究重点在于对文本中的实体、关系、事件、关键词和关键句的抽取，自然语言处理的研究重点在于句法分析 、词义消歧和多层次信息融合机制。

研究室坚持理论研究与技术开发互动同步发展的原则，一方面在向技术极限挑战的过程中撰写高质量的论文，一方面将陆续完成的阶段性成果适时地转化为实用技术。信息过滤和信息抽取在数字图书馆、内容安全、商务智能、电子政务、电子学习、移动计算、军事情报等各个领域均有巨大的应用价值。研究室主要通过与企业合作，采取将技术嵌入企业的产品中的方式，实现研究的价值，回报社会。

研究室努力营造浓厚的学术氛围，悉心培养优秀学子。目前实验室有 6 名博士生，13 名硕士生，这 13 名硕士生中 8 名为免试推荐生，5 名为考研成绩优异者，其中多人次获美国数学建模竞赛奖、IBM 奖学金、计算机世界奖学金等。

哈工大信息检索研究室是一个年轻的团队，她渴望向国内和国际的同行们学习，也愿意尽自己的努力在计算机应用领域做出贡献。

【全文完 • 责任编辑：car@hitbbs】

投稿指南

非常高兴而隆重的宣布，在大家顶力支持及热情支援下，《纯 C 论坛·电子杂志》编辑部于公元 2004 年 9 月 28 日正式成立了!本刊定位为相对底层而纯粹的计算机科学及技术研究，着眼于各专业方向基本理论，基本原理的研究，重视基础，兼顾应用技术，以此形成本刊独特的技术风格。

本刊将于每月 28 号通过网络发行，任何人均可在这一天从互联网上下载本刊，并将通过电子邮件，向纯 C 论坛的注册用户寄送。

目前本刊有十大骨干技术版块：

栏目	责任编辑	投稿信箱
计算机组成原理及体系结构	kylix@hitbbs	purec_coa@126.com
算法理论与数据结构	worldguy@hitbbs	purec_compile@126.com
C 与 C++语言	sun@hitbbs	purec_cpp@126.com
汇编语言	ogg@hitbbs	purec_asm@126.com
数据库原理	pineapple@hitbbs	purec_db@126.com
网络与信息安全	true@hitbbs	purec_network@126.com
计算机病毒	swordlea@hitbbs	purec_virus@126.com
人工智能及信息处理	car@hitbbs	purec_ai@126.com
操作系统	iamxiaohan@hitbbs	purec_os@126.com

从现在开始，本刊面向全国、全网络公开征集各类稿件，你的投稿将由本刊各栏目的责任编辑进行审校，对于每一稿件我们都会认真处理，并及时通知您是否选用，或者由各位责任编辑对稿件进行点评。

所有被本刊选用的稿件，或者暂不适合通过电子杂志发表的稿件，将会在纯 C 论坛网站上同期发表。所有稿件版权完全属于各作者自己所有，非常欢迎您积极向本刊投搞，让你的工作被更多的人知道，让自己同更多的人交流、探讨、学习、进步！

为了确保本刊质量，保证本刊的技术含量，本刊对稿件有如下一些基本要求：

1. 主要以原创（包括翻译外文文献）为主，可以不需要有很强的创新性，但要求有一定的技术含量，注重从原理入手，依据原理解决问题。描述的问题可以很小但细致，可以很泛但全面，最好图文并貌，投稿以 word 格式发往各栏目的投稿信箱，或直接与各栏目责任编辑联系。本刊各栏目均为活动性栏目，会随时依据稿件情况新开或暂定各栏目，因此，只要符合本刊采稿宗旨的稿件，本刊都非常欢迎，如果您一时拿不准您的稿件应投往哪一栏目可直接将稿件投到本刊通用联系信箱（purec@126.com）。

2. 本刊对稿件的风格或格式没有特殊要求，注重质量而非形式，版权归各作者所有，不限一稿多投但限制重复性投稿（如果稿件没有在本刊发表，则不算重复性投稿）。稿件的文字、风格除了在排版时会根据需要有必要的改动及改正错别字外，不会对稿件的描述风格、观点、内容、格式进行大的改动，以期最大限度的保留各作者原汁原味的行文风格，因此，各作者在投稿时最好按自己意愿自行排版。也可以到本刊网站（http://purec.binghua.com）下载稿件模板。

3. 来稿中如有代码实现，在投稿时最好附带源代码及可执行文件，本刊每期发行时，除了一个 pdf 格式的杂志外，还会附带一个压缩文件，其中将包含本期所有的源代码及相应资源。当然，提不提供源代码，随各位作者自便。

4. 如果稿件是翻译稿件，请最好附带英文原文，以便校对。另外本刊在刊发翻译稿的同时，如有可能，将随稿刊发英文原文，因此请在投稿前确认好版权问题。

5. 来稿请明确注明姓名、电子邮箱、作者单位等信息，以便于编辑及时与各位作者交

流，发送改稿意见，选用通知及寄送样刊等，如果你愿意在发表你稿件的同时，提供一小段的作者简介，我们非常欢迎。

6. 所有来稿的版权归各作者所有，也由各作者负责，切勿抄袭，如果在文中有直接引用他人观点结论及成果的地方，请一定在参考文献中说明。每篇稿件最好提供一个简介及几个关键词，以方便读者阅读及查询。由于本刊有可能被一些海外朋友阅读，所以非常推荐您提供英文摘要。

7. 所有来稿编辑部在处理后，会每稿必复，如果您长时间没有收到编辑部的消息，请您同本编辑部联系。

8. 由于本刊是纯公益性质，没有任何外来经济支持，所有编辑均是无偿劳动，因此，本刊暂无法向您支付稿筹，但在适当的时候，本刊会向各位优秀的作者赠送《纯 C 论坛资料（光盘版）》，以感谢各位作者对本刊支持！

9. 如果您想转载(仅限于网络)本刊作品请注明原作者及出处，如果您想出版本刊作品，请您与本刊编辑部联系。

本刊联系地址：

网　　址：http://purec.binghua.com

联系信箱：purec@126.com

通信地址：哈尔滨工业大学 计算机科学与技术学院 综合楼 520 室

邮　　编：150001

再一次诚恳邀请您加盟本刊，为本刊投稿！

《纯 C 论坛·电子杂志》编辑部

读者俱乐部

1. 本刊在纯 C 论坛网站上建立了读者俱乐部（由纯 C 论坛→BBS 讨论区→读者俱乐部进入），各位读者可以在上面发表对每期刊物的看法及建议，包括对刊物稿件质量、内容，刊物风格、格式等各方面的建议，对于热心读者我们会邀请他成为本刊特邀请评刊员，并在适当的时候赠送《纯 C 论坛资料（光盘版）》以示感谢！
2. 读者俱乐部中还会对每期刊物所刊发稿件的内容进行根踪，因此，您可以在上面取得每期刊物最新的勘误表，及更正后的最新的刊物。

欢迎您访问“纯 C 论坛 • 电子杂志”读者俱乐部！

【本期信息汇总】

期数：2004 年第 10 期（总第 1 期）
发行时间：2004 年 10 月 28 日
修订时间：2005 年 1 月 13 日(SP1)
本期责任编辑：iamxiaohan@hitbbs
本期特邀评刊员：cliff@hitbbs，ssos@hitbbs
主题文章数：10 篇
字数（约）：50,000

SP1 版对原版的修订说明

1. 修订了如下错误（注：页号均为原版页号）：

页：30
第二段第一行：E=1-(2^e-1) 改为 E=1-(2^(e-1)-1)
最后一段第二行：E=E'-(2^e-1) 改为 E=E'-(2^(e-1)-1)
[感谢 薛风(xf_cau@163.com) 指正]

页:31
第五行：“同样，我们通过一个程序验证一下，不过这次我们把这个程序变一下，直接输入 0.625”
其中的“0.625”改为“-0.625”
[感谢 timw 指正]

页：32
第三段第二行最后：＂最小正数数＂改为＂最小正数为＂

页:43
图 5 下面第 6 行“于是 CPU 会产生一个所谓的缺页中断来通知操作系统进行处理，操作系统相应这个中断”中“相应”改为“响应”
[感谢 CrazyWind 指正]

页：46
第三行开头两个字：“所为”改为“所谓”
[感谢 CrazyWind 指正]

图九上面倒数 4 行：“我们的“段选择子”为：0000 0000 0000 10000 ”其中段选择子的最后一个 0 去掉
[感谢 CrazyWind 指正]

页:47 页
图十下面，“图中的数据存放区用来存发在……” 其中的“存发”改为“存放”
[感谢 CrazyWind 指正]

2. 重编编排了页号。