中国科学技术大学

高级编程与性能优化

吴锋

Email: wufeng02@ustc.edu.cn

本章内容概述

- 位运算
- 宏操作（带参数、系统宏）
- 静态与动态库
- 编程优化方法

本章内容概述

- 位运算
- 宏操作（带参数、系统宏）
- 静态与动态库
- 编程优化方法

位运算的概念

- 很多算法是按二进制位进行运算的
- 位运算速度快，效率高

- 位运算的运算对象是二进制的位
- 只能对整型数据（包括字符型）进行位运算
- 负数以补码形式参与运算

位运算符

- 注意位运算符与逻辑运算区别

运算符	运算	举例	优先级从高到低
			(逻辑非运算符)
~	按位取反	~flag	
			(算术运算符)
<<	左移	a << 2	
>>	右移	b >> 3	
			(关系运算符)
&	按位与	flag & 0x37	
^	按位异或	flag ^ 0xC4	
|	按位或	flag | 0x5A	
			(赋值运算符)

按位与 (Bitwise AND) &

- 运算规则
  - 0 & 0 = 0;
  - 0 & 1 = 0;
  - 1 & 0 = 0;
  - 1 & 1 = 1;

```
   1010,1101 (0xAD)
 & 0110,1001 (0x69)
-------------------
   0010,1001 (0x29)
```

- 特殊用法
  - 特定位清零
  - 保留其它位

```
   xxxx,xxxx
 & 0110,0010 (0x62)
-------------------
   0xx0,00x0
```

按位或 (Bitwise Inclusive OR) |

- 运算规则
  - 0 | 0 = 0;
  - 0 | 1 = 1;
  - 1 | 0 = 1;
  - 1 | 1 = 1;

```
   1010,1101 (0xAD)
 | 0110,1001 (0x69)
   1110,1101 (0xED)
```

- 特殊用法
  - 特定位置一
  - 保留其它位

```
   xxxx,xxxx
 | 0110,0010 (0x62)
   x11x,xx1x
```

按位异或 (Bitwise Exclusive OR) ^

- 运算规则
  - 0 ^ 0 = 0;
  - 0 ^ 1 = 1;
  - 1 ^ 0 = 1;
  - 1 ^ 1 = 0;

```
   1010,1101 (0xAD)
^  0110,1001 (0x69)
-------------------
   1100,0100 (0xC4)
```

- 特殊用法
  - 特定位取反
  - 保留其它位

```
   xxxx,xxxx
^  0110,0010 (0x62)
-------------------
   xx̄x̄x,xxx̄x
```

按位取反 (Bitwise NOT) ~

- 运算规则
  - ~0 = 1;
  - ~1= 0;

```
 ~ 0110,1001 (0x69)
------------------
   1001,0110 (0x96)
```

左移 (Left Shift) < <

- 运算规则
  - i < < n
  - 把 i 各位全部向左移动 n 位，不会改变 i 的值
  - 最左端的 n 位被移出丢弃
  - 最右端的 n 位用 0 补齐

```
5 << 3 = 40
00000101 (0x05) << 3 → 00101000 (0x28)
```

- 用法
  - 若没有溢出，则左移 n 位相当于乘上 2^n
  - 运算速度比真正的乘法和幂运算快得多

右移 (Right Shift) >>

- 运算规则
  - i >> n
  - 把 i 各位全部向右移动 n 位，不会改变 i 的值
  - 最右端的 n 位被移出丢弃
  - 最左端的 n 位用 0 补齐(逻辑右移)
  - 或最左端的 n 位用符号位补齐(算术右移)

由编译系统的实现者决定。
为了可移植性，最好仅对无符号数进行移位运算

- 用法

```
5 >> 2 = 1
00000101 (0x05) >> 2 → 00000001 (0x01)
```

  - 右移 n 位相当于除以 2^n，并舍去小数部分
  - 运算速度比真正的除法和幂运算快得多

位运算的规则

- 按位运算时，两个操作数长度应相等，
- 否则先扩展，再运算
  - 两个操作数右端对齐
  - 短的数据左端用符号位补齐
    - 正数或无符号数左端用 0 补满
    - 负数左端用 1 补满
- 位运算符都不会改变参与运算变量的值
- 复合赋值运算符（改变参与预算变量的值）

  &=，^=，|=，~=，<<=，>>=

位运算（例一）

- 例：如何判断一个 int 型变量 n 的第7位（从右往左，从 0 开始数）是否是 1？
- 答：只需看 n & 0x80 的值是否等于 0x80 即可。

xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx（n）

& 0000 0000 0000 0000 0000 0000 1000 0000（0x80）

0000 0000 0000 0000 0000 0000 x000 0000

位运算（例二）

- 例：将 16 进制短整数按二进制打印输出
  - 输入：F1E2
  - 输出：1111000111100010
  - 输入：13A5
  - 输出：0001001110100101

```
#include <stdio.h>
int main() {
  int i;
  short a;
  scanf("%X", &a);
  for (i=15;i>=0;i--)
    printf("%1d",
            a&1<<i ? 1 : 0);
  return 0;
}
```

异或运算的特点

- 异或运算的特点是：
  - 如果 a^b=c，那么就有 c^b = a 以及 c^a=b
- 此规律可以用来进行最简单的加密和解密以及信息校验
  - 加密：明文^密钥=密文；解密：密文^密钥=明文
  - 校验：信息（a b c d）+ 校验码（a^b^c^d）
- 还可以用来交换两位数 a 和 b
  - a=a^b
  - b=a^b，即：(a^b)^b = a
  - a=a^b，即：(a^b)^a = b

位运算（例三）

- 例：在一个整数数组中，仅存在一个不重复的数字，其余数字均出现两次（或偶数次），找出不重复数字。

```
for (int i = 0; i < N; ++i) {
  int c = 0;
  for (int j = 0; j < N; ++j)
    if (i != j && a[i] == a[j])
      ++c;
  if (c % 2 != 0)
    return a[i];
}
```

双重循环的解法 $O(N^2)$

```
int r = 0;
for (int i = 0; i < N; ++i)
  r = r ^ a[i];
return r;
```

按位异或的解法 O(N)

0 ^ a = a

a ^ a = 0

a ^ a ^ a = a

a ^ a ^ a ^ a = 0

a ^ b ^ a ^ a ^ b = a

位运算（例四）

int n1 = 15 : 0000 0000 0000 0000 0000 0000 0000 1111

short n2 = 15 : 0000 0000 0000 1111

unsigned short n3 = 15 : 0000 0000 0000 1111

unsigned char c = 15 : 0000 1111

n1 <<= 15，(变成78000)： 0000 0000 0000 0111 1000 0000 0000 0000

n2 <<= 15，(变成-32768)： 1000 0000 0000 0000

n3 <<= 15，(变成 32768)： 1000 0000 0000 0000

c <<= 6 ，(变成 0xc0) ： 1100 0000

int n4 = c << 4 这个表达式是先将 c 转换成整型：

0000 0000 0000 0000 0000 0000 1100 0000

然后再左移，即： (int)(c<<4)=3072

位运算（例五）

int n1 = 15 : 0000 0000 0000 0000 0000 0000 0000 1111

short n2 = -15 : 1111 1111 1111 0001

unsigned short n3 = 0xffe0 : 1111 1111 1110 0000

unsigned char c = 15 : 0000 1111

n1 >>= 2，变成3 : 0000 0000 0000 0000 0000 0000 0000 0011

n2 >>= 3，变成-2 : 1111 1111 1111 1110

n3 >>= 4，变成 0xffe : 0000 1111 1111 1110

c >>= 3 ， 变成1 : 0000 0001

位运算（例六）

- 例：有两个个int型变量 a 和 n（0 ≤ n ≤ 31），求 a 的第 n 位（从右往左，从 0 开始数）的值?
- 答：(a >> n) & 1 或 (a & (1 << n)) >> n。

xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx (a)

0000 000x xxxx xxxx xxxx xxxx xxxx xxxx (>> 7)

& 0000 0000 0000 0000 0000 0000 0000 0001 (1)

0000 0000 0000 0000 0000 0000 0000 000x

性能优化中的位运算（例）

- 例：求 char 型数据二进制表示中 1 的个数
- 方法1：模拟进制转换，反复除 2 ， 累加余数

```
int count1(unsigned char v)
{
  int num = 0;
  while (v) {
    if (v%2 == 1) num++;
    v /= 2;
  }
  return(num);
}
```

十进制到二进制的转换

性能优化中的位运算（例）

- 例：求 char 型数据二进制表示中 1 的个数
- 方法2：将方法1中运算改为位运算

```
int count2(unsigned char v)
{
  int num=0;
  while (v) {
    num += v & 0x01;
    v >>= 1;
  }
  return(num);
}
```

位运算快于算术运算

性能优化中的位运算（例）

- 例：求 char 型数据二进制表示中 1 的个数
- 方法3：将循环次数降为 1 的个数

思路：设法使得每次循环都能减少一个1

```
int count3(unsigned char v) {
  int num=0;
  while (v) {
    v &= (v-1);
    num++;
  }
  return(num);
}
```

减1会使得最后一个1变为0，与之后1就少了一个

性能优化中的位运算（例）

- 例：求 char 型数据二进制表示中1的个数
- 方法4：枚举法，写出所有分支，“人工手动判断”

```
int count4(unsigned char v)
{
  int num=0;
  switch (v) {
    case 0x00: num=0; break;
      ......
  }
  return(num);
}
```

性能优化中的位运算（例）

- 例：求 char 型数据二进制表示中1的个数
- 方法5：打表法，预存所有结果，“以空间换时间”

```
int count5(unsigned char v)
{
  static int countTable[256]={0, 1, ...};
  return countTable[v];
}
```

本章内容概述

- 位运算
- 宏操作（带参数、系统宏）
- 静态与动态库
- 编程优化方法

程序的编译过程

源代码（.c, .cpp, .h）

预处理　　gcc -E hello.c -o hello.i

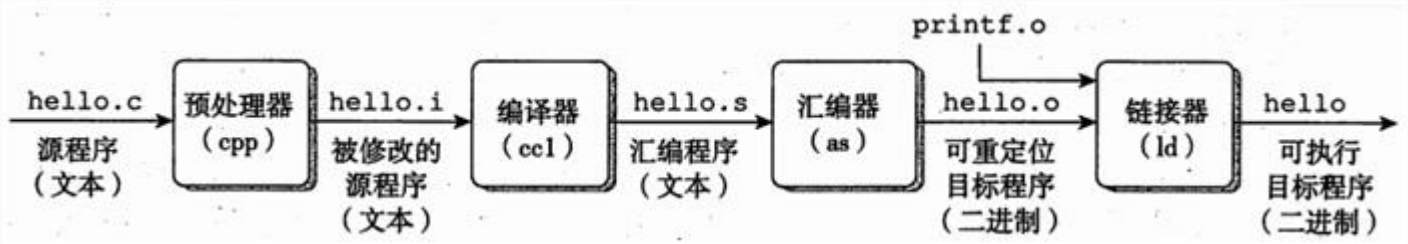
头文件嵌入、宏展开的源代码（.i）

编译、汇编　　gcc -S hello.i -o hello.s
gcc -c hello.s -o hello.o

目标代码（.o .obj）

链接　　函数、变量等进行交叉映射

可执行文件（.exe）



GCC编译系统

预处理

- 预处理器一般内置于C编译器中
- 预处理是处理C源程序中所有以#开头的命令行（称为预处理命令）
- 指令在第一个换行符处结束，除非用“\”明确指出要换行延续

- C语言提供的预处理命令：文件包含、宏定义和条件编译
- 预处理命令的语法与C语言的语法完全独立，预处理器不处理C语言语句
- 为区别程序中的代码行与预处理命令行，所有预处理命令行都以#开头
- 预处理命令遇到换行符就会结束。如果需要换行，可以在预处理命令行的行尾加“\”，并紧跟换行符，表示续行

不带形参的宏

- 宏体中可以引用别的宏名，如

```
#define R    18.75                      // 半径
#define PI   3.1415926                  // 圆周率
#define Circumference 2.0*PI*R          // 圆周长
```

- 同一宏名可以重复定义，如

```
#define PI 3.14
...          // 此范围内使用的 PI 都被替换为 3.14
#define PI 3.1415926
...          // 此范围内使用的 PI 都被替换为 3.1415926
```

- 可以用#undef 终止宏定义的作用范围，如

```
#define N 50
...          // 此范围内 N 被替换为 50
#undef N
...          // 此范围内 N 无效
```

带形参的宏

- 一般形式：#define 宏名(形参列表) 宏体
- 左圆括号“(”必须紧随宏名之后，中间不能有空格，否则就变成不带形参的宏定义（宏体从“(”开始）
- 形参列表中可以出现多个用逗号隔开的不重名的标识符
- 宏调用的展开：分别用宏调用中的实参文本去替换宏体中对应的形参，宏体中的其它文本不变
  - 例： #define AREA(W,H) W*H
    AREA(a,b) 宏展开为 a*b
    AREA(a+1,b+1) 宏展开为 a+1*b+1

    修改为： #define AREA(W,H) (W)*(H)
    AREA(a+1,b+1) 宏展开为(a+1)*(b+1)
- 宏展开仅是文本替换，不会进行表达式计算
- 实参可以为空，但逗号不能省。展开时替换为空串

带形参的宏

- 字符串化操作 #：宏展开后，把宏参数变成字符串字面量

```
#define PRINT_VALUE(x) printf(#x " = %d\n", x)

int age = 25;
PRINT_VALUE(age); // 展开为：printf("age" " = %d\n", age);
```

- 连接操作 ##：宏展开后，把两个符号连接成一个新符号

```
#define CONCAT(a, b) a##b

int value12 = 100;
int result = CONCAT(value, 12); // 展开为：int result = value12;
```

函数与带形参宏的区别

- 函数调用与宏调用的区别
  - 函数调用是在程序运行时处理的，涉及内存单元的分配与回收以及表达式的计算等；宏调用是在预处理阶段进行的，只进行文本替换，既不分配内存单元也不计算
  - 函数调用的形参与实参都有数据类型，存在类型匹配或转换问题；宏调用仅是文本的替换，不存在类型的问题
  - 函数调用往往有返回值；宏调用没有返回值的概念

- getchar() 和 putchar() 实际上都是定义在 stdio.h 中的宏：

```
#define getchar()    fgetc(stdin)        //形参列表为空
#define putchar(x) fputc(x, stdout)
```

带形参的宏（例一）

- 一些带参数宏定义的例子
  - 例：求2个、3个、4个数值中最小值的宏定义

```
#define min(a,b)  (((a)<(b))?(a):(b))
#define min3(x,y,z)  min(min(x,y),z)
#define min4(r,s,t,u) min(min3(r,s,t),u)
```

思考：int a=5, b = 10; c = min(a++, b++); c 的值是多少?

  - 例：使两个参数的值互换的宏

```
#define SWAP(type,x,y) {type temp=x; x=y; y=temp;}
double a1=2.3, a2=4.8;
SWAP(double, a1, a2);
```

带形参的宏（例二）

- 定义交换两个整数的宏：SWAP(a, b)
  - 方法一：#define SWAP(a, b) a = a+b; b = a–b; a = a–b;
    - SWAP(x, y); // OK
    - if (x < 0) SWAP(x, y); // ERROR
      - 替换的结果：if (x< 0) x= x+y; y=x–y; x = x–y;
  - 方法二：#define SWAP(a, b) { a = a+b; b = a–b; a = a–b; }
    - if (x < 0) SWAP(x, y); // OK
      - 替换的结果：if (x< 0) { … };
    - if (x < 0) SWAP(x, y); else SWAP(x, z); // ERROR
      - 替换的结果：if (x< 0) { … }; else { … };
  - 方法三：#define SWAP(a, b) do { a = a+b; b = a–b; a = a–b; } while(0)

带形参的宏（例三）

- 例：如何判断一个变量是有符号数还是无符号数?
  - 例如，ANSI C中，char型变量可以是有符号数，也可以是无符号数，由编译器设计者决定
  - 判断依据：无符号数永远不会是负的

```
// 对变量 a
#define IS_UNSIGNED(a) (a>=0 && ~a>=0)

// 对类型 type
#define IS_UNSIGNED(type) ((type)0 – 1 >0)
```

宏的编程实践（例一）

- 在实践中可以使用宏，实现数组的遍历

```
#define FOREACH(item, array) \
    for(int keep=1,count=0,size=sizeof(array)/sizeof(*(array)); \
        keep && count < size; keep = !keep, count++) \
         for(item = (array) + count; keep; keep = !keep)

// 宏的使用
int nums[] = {1, 2, 3, 4, 5};
FOREACH(int *num, nums) {
    printf("%d\n", *num);
}
```

宏的编程实践（例二）

- 在实践中也可以使用宏，确保动态分配的内存被回收

```
#include <stdlib.h>
#define ALLOC_MEM(T, num, p) \
     for (T *p=(T*)malloc(num*sizeof(T)), *_flag_=(void*)0; \
          (p==NULL?(printf("Allocate Error.\n"), 0):!_flag_); \
          (free(p), _flag_=(void*)1) )

// 宏的使用
ALLOC_MEM(int, 1, p) {
     *p = 5; // 对指针的操作
}
```

宏的编程实践（例三）

- 在实践中可以使用宏，确保文件正常打开，且使用后关闭

```
#include <stdio.h>
#define OPEN_FILE(name, mode, fp) \
  for (FILE *fp=fopen(name,mode), *_flag_= (void*)0; \
     (fp==NULL?(printf("Error:%s.\n", name),exit(0),0):!_flag_); \
     (fclose(fp), _flag_ = (void*)1) )
// 宏的使用
OPEN_FILE("file.txt", "r", fp) {
     ... ... // 对文件的操作
}
```

功能类似 Python 中的：

```
with open("file.txt", "r") as fp:
    ... ... # 对文件的操作
```

条件编译

- 条件编译允许预处理器根据条件，选择性地传递源程序中的文本行给编译器进行处理，或者忽略
- 根据指定的标识符是否定义过

```
#ifdef 标识符
    程序段1
#endif
```

或

```
#ifdef 标识符
    程序段1
#else
    程序段2
#endif
```

例：

```
#define DEBUG  //常用于调试
#ifdef DEBUG
   printf(……);
#endif
```

条件编译

- 根据指定的标识符是否未定义过

```
#ifndef 标识符
    程序段1
#endif
```

或

```
#ifndef 标识符
    程序段1
#else
    程序段2
#endif
```

- 根据常量表达式的值是否非0

```
#if 常量表达式
    程序段1
#endif
```

或

```
#if 常量表达式
    程序段1
#else
    程序段2
#endif
```

或

```
#ifdef 标识符
    程序段1
#elif 常量表达式
    程序段2
#else
    程序段3
#endif
```

条件编译

- 可编写子多台机器或多种操作系统之间可移植的程序
- 可编写用不同的编译器编译的程序
- 为宏提供默认定义

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

- 临时屏蔽包含注释/* … */的代码

```
#if 0
    包含/* … */的代码段
#endif
```

- 保护头文件以避免重复包含

头文件：hello.h

```
#ifndef _HELLO_H_
#define _HELLO_H_
// 增加上述宏定义
// 避免头文件被重复包含

// 头文件的内容
… …

#endif
```

核心系统预定义宏（ANSI C标准）

- 这些宏在现代C编译器中均可用，通常用于打印调试信息
  - __FILE__ ：字符串文字，表示当前正在处理的源文件名。
  - __LINE__ ：整数，表示当前代码行在文件中的行号。
  - __DATE__ ：字符串文字，表示预处理的日期，格式为 "Mmm dd yyyy"。
  - __TIME__ ：字符串文字，表示预处理的时间，格式为 "hh:mm:ss"。
  - __func__ ：字符串，表示当前所在函数的名称（C99标准引入）。
  - __STDC__ ：如果编译器符合ANSI C标准，则定义为1。

```c
#include <stdio.h>
void print_log() {
    printf("File:%s, Line:%d, Function:%s\n",__FILE__,__LINE__,__func__);
}
// 输出示例: File: main.c, Line: 4, Function: print_log
```

编译器特性与平台宏

- 用于条件编译，根据编译器类型或平台调整代码
  - __GNUC__ ：GNU GCC编译器或兼容编译器（如Clang）定义此宏。
  - _MSC_VER ：Microsoft Visual C++ (MSVC) 编译器定义此宏。
  - __linux__ ：在Linux操作系统下定义。
  - _WIN32 / _WIN64: 在Windows操作系统下定义。
  - __cplusplus：如果是C++代码文件，则定义此宏。

```
#ifdef _WIN32
    #include <windows.h>
#else
    #include <unistd.h>
#endif
```

注意事项

- 预处理阶段: 宏定义是在编译之前由预处理器进行简单文本替换的。
- 调试与查看: 可以在GCC中使用 gcc -dM -E - < /dev/null 命令查看当前编译器定义的所有宏。
- 不可重定义: 系统预定义宏通常不能使用 #undef 重新定义。

其它系统预定义宏

- 整形变量的表示范围：#include <limits.h>
  - 字符类型（char）　：CHAR_MIN，CHAR_MAX, UCHAR_MAX
  - 整数类型（int）　　：INT_MIN, INT_MAX, UINT_MAX
  - 长整数类型（long）：LONG_MIN, LONG_MAX, ULONG_MAX
- 浮点型变量的表示范围：#include <float.h>
  - 最大值：FLT_MAX, DBL_MAX, LDBL_MAX
  - 精度　：FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON

系统预定义宏（例）

- 在实践中使用系统宏进行打印错误消息，进行程序调试

```
#ifdef DEBUG
    #define LOG(fmt, ...) \
     printf("[DEBUG] File %s, line %d: " fmt "\n", \
          __FILE__, __LINE__,  ##__VA_ARGS__)
#else
    #define LOG(fmt, ...) // Release版本中空定义
#endif
int main() {
    int b = 20;
    LOG("Variable b is %d", b); // 调试时打印,否则不打印
    return 0;
}
// 调试模式下编译程序: gcc -DDEBUG -g -o main main.c
```

调试断言

- 调试断言：#include <assert.h>
  - void assert(int expr)：如果表达式expr的值为假(即为0),那么它就先向stderr打印一条出错信息，然后通过条用abort来终止程序；否则什么也不做
    - 使用assert时，断言的内容（表达式）要明确，尽量一项内容写一行
    - 在调试程序时，在怀疑有问题的地方插入断言，可阻断Bug的扩散

```
int func(int x, int y, ) {
  assert(x < a1 && x > b1); // 函数执行前，关于参数 x 的断言
  assert(y < a2 && y > b2); // 函数执行前，关于参数 y 的断言
  ...
  assert(...); // 函数执行中，关于临时变量的断言
  ...
  assert(z < a4 && z > b4); // 函数执行后，关于返回值 z 的断言
  return z;
}
```

调试断言中的宏定义

- 调试断言：#include <assert.h>
  - void assert(int expr)：如果表达式expr的值为假(即为0),那么它就先向stderr打印一条出错信息，然后通过条用abort来终止程序；否则什么也不做
    - 使用assert时，频繁的调用会影响程序的性能，增加额外的开销
    - 在调试结束后，可以通过插入 #define NDEBUG 来禁用assert调用

```
#ifdef NDEBUG
// 如果NDEBUG的宏在程序中被定义，则assert定义为空（即什么也不做）
#define assert(e)((void) 0)
#else
// 否则，如果表达式e为假，则终止程序，打印出问题的文件和行号
#define assert(e)\
    ((void)((e) ? ((void) 0) : __assert(#e, __FILE__, __LINE__)))
#endif
```

本章内容概述

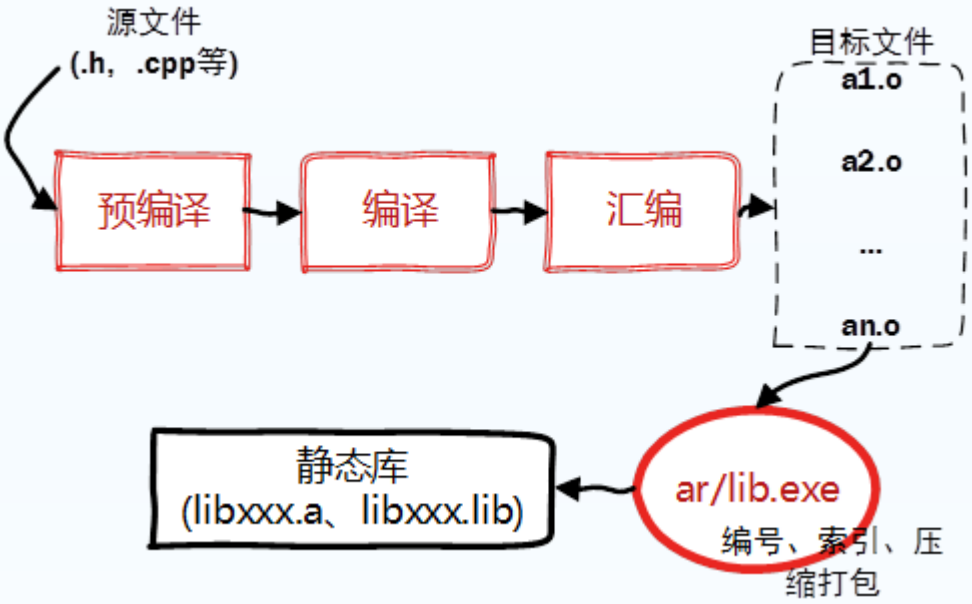
- 位运算
- 宏操作（带参数、系统宏）
- 静态与动态库
- 编程优化方法

库的基本概念

- 在 C 语言开发中，库是一种预先编译好的代码集合（二进制文件），它包含了一系列可复用的函数和变量。根据链接方式分为静态库和动态库
  - 库的作用：避免“重复造轮子”，提高编译效率，保护核心源码
  - 库的本质： 它是 .o (Object file) 文件的打包集合
- 静态库：在程序编译链接阶段会被完整地复制到最终的可执行文件中。一旦链接完成，可执行文件就不再依赖原有的静态库文件
  - 后缀名： Windows 下为 .lib (Library)，Linux/Unix/Mac 下为 .a (Archive)
- 动态库：在编译时并不直接拷贝到可执行文件中，而是在程序运行时由操作系统按需加载。可执行文件中仅保留对库的引用（符号表）
  - 后缀名： Windows 下为 .dll (Dynamic Link Library)，Linux 下为 .so (Shared Object)，Mac 下为 .dylib

静态库

- 静态库在链接阶段，会将汇编生成的目标文件(.o)与引用的库一起链接打包到可执行文件中。因此对应的链接方式称为静态链接
- 静态库与汇编生成的目标文件一起链接为可执行文件，跟.o文件格式相似。一个静态库可以简单看成是一组目标文件（.o/.obj文件）的集合，即很多目标文件经过压缩打包后形成的一个文件



静态库的优缺点

- 静态库的优点：
  - 执行速度快：对函数库的链接是放在编译时期完成的，运行时无需额外寻址或加载过程
  - 独立性强： 生成的可执行文件包含所有代码，分发时不需要携带库文件，程序在运行时与函数库再无瓜葛，方便移植
- 静态库的缺点：
  - 浪费空间：如果多个程序使用同一个库，每个程序都会包含一份副本（所有相关的目标文件与牵涉到的函数库），浪费磁盘和内存
  - 更新麻烦： 一旦库代码修改，所有依赖它的程序都必须重新编译

Linux下创建与使用静态库

- 静态库命名规则
  - Linux 的静态库必须命名为“lib[your_library_name].a”：lib为前缀，中间是静态库名，扩展名为.a，如：libmymath.a
- 创建静态库（.a）
  - 首先，将代码文件编译成目标文件.o（math_utils.o），注意带参数-c，否则直接编译为可执行文件，如： gcc -c math_utils.c
  - 然后，通过ar工具将目标文件打包成.a静态库文件，生成静态库libmymath.a，如： ar rcs libmymath.a math_utils.o
- 使用静态库
  - 只需要在编译的时候，指定静态库的搜索路径（-L选项，如静态库在当前目录或者系统目录可省略）、指定静态库名（不需要lib前缀和.a后缀，-l选项）
  - 如：gcc main.c -L../StaticLibrary -lmymath

Linux下创建与使用静态库（例）

- 准备源码文件
  - 创建头文件（math_utils.h）及函数实现（math_utils.c）

```c
// math_utils.h 文件
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int multiply(int a, int b);

#endif
```

```c
// math_utils.c 文件
#include "math_utils.h"
int add(int a, int b) {
    return a + b;
}
int multiply(int a, int b) {
    return a * b;
}
```

- 创建静态库（.a）
  - 编译为目标文件 (.o)： gcc -c math_utils.c -o math_utils.o
  - 使用 ar 工具打包：ar rcs libmymath.a math_utils.o

Linux下创建与使用静态库（例续）

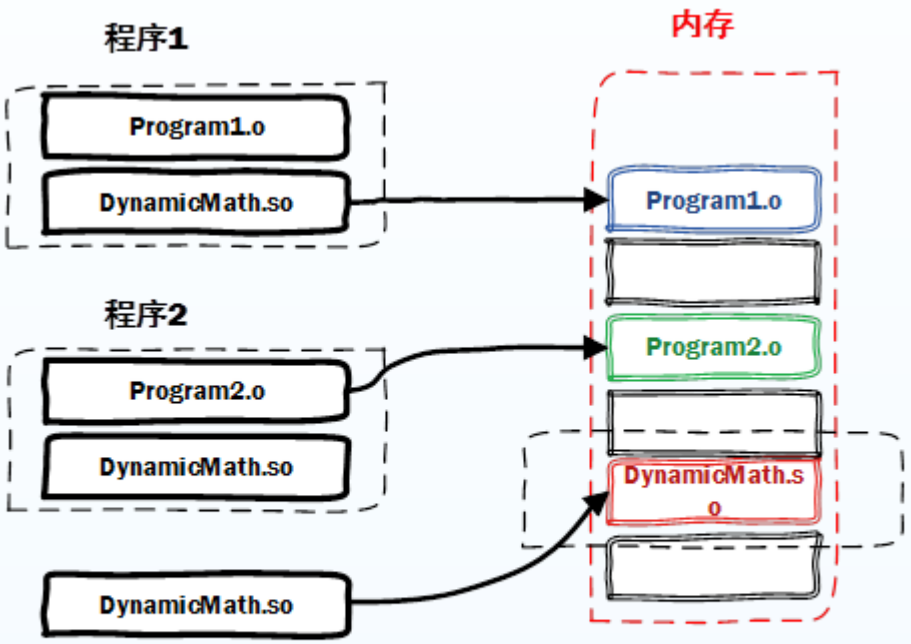
- 使用静态库
  - 编写一个主程序（main.c）来调用这个库

```c
// main.c 文件
#include <stdio.h>
#include "math_utils.h"  // 包含该库的头文件
int main() {
    printf("Add: %d\n", add(10, 5));
    printf("Multiply: %d\n", multiply(10, 5));
    return 0;
}
```

  - 编译并链接：gcc main.c -L. -lmymath -o myapp
  - 库的代码已经已完全嵌入到了程序内部，直接执行程序：./myapp

动态库

- 动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。不同的应用程序如果调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题
- 动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新



动态库在内存中只存在一份拷贝，避免了静态库浪费空间的问题。

动态库的优缺点

- 动态库的优点
  - 节省资源： 多个程序可以共享内存中的同一个库副本，可以实现进程之间的资源共享（因此动态库也称为共享库）
  - 更新方便： 只需替换 .dll 或 .so 文件，无需重新编译主程序（只要接口不变），将程序的升级变得简单，只需要增量更新动态库即可
- 动态库的缺点
  - 依赖性： 运行环境（当前目录或者系统目录）必须存在对应的库文件，否则会出现“找不到链接库”的错误
  - 启动略慢：动态库把对一些库函数的链接载入推迟到程序运行的时期，运行时需要进行符号重定位，会有微小的性能开销

Linux下创建与使用动态库

- 动态库命名规则
  - 动态链接库的名字形式为 “lib[your_library_name].so”，前缀是lib，后缀名为“.so”，程序通过这个名字来告诉动态加载器该载入哪个共享库
- 创建动态库（.so）
  - 首先，生成目标文件，此时要加编译器选项-fPIC， 创建与地址无关的编译程序（PIC，Position Independent Code），是为了能够在多个应用程序间共享，如：gcc -fPIC -c math_utils.c
  - 然后，生成动态库，此时要加链接器选项-shared， 如：
    gcc -shared -o libmymath.so math_utils.o
- 使用动态库
  - 引用动态库编译成可执行文件（跟静态库方式一样），如：
    gcc main.c -L../DynamicLibrary -lmymath

Linux下创建与使用动态库（续）

- 程序执行时系统如何定位动态（共享）库文件：
  - 当系统加载可执行代码时候，能够知道其所依赖的库的名字，但是还需要知道绝对路径，此时就需要加载器（Loader）完成动态库的载入
  - 对于ELF格式的可执行程序，加载器依次搜索ELF文件的：1) DT_RPATH段；2) 环境变量 LD_LIBRARY_PATH；3) /etc/ld.so.cache 文件列表；4) /lib/, /usr/lib 目录；找到库文件后将其载入内存
- 如何让系统能够找到编译好的动态库文件：
  - 编译时指定运行时搜索路径 (-Wl,-rpath)，把搜索路径硬编码进了可执行文件里，如：gcc main.c -L. -lmymath -o myapp -Wl,-rpath=.
  - 设置 LD_LIBRARY_PATH 环境变量，适合测试时临时添加路径：
    export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
  - 编辑 /etc/ld.so.conf 文件，加入库文件所在目录的路径，运行 ldconfig ，该命令会重建 /etc/ld.so.cache 文件
  - 将文件拷贝到 /usr/lib 或 /usr/local/lib，并运行 sudo ldconfig 更新缓存

Linux下创建与使用动态库（例）

- 准备源码文件
  - 创建头文件（math_utils.h）及函数实现（math_utils.c）
- 创建动态库 (.so)
  - 生成位置无关的目标文件：gcc -fPIC -c math_utils.c -o math_utils.o
  - 生成共享库文件：gcc -shared -o libmymath.so math_utils.o
- 编译并链接主程序
  - 与编译静态库时完全一致：gcc main.c -L. -lmymath -o myapp
- 运行程序
  - 默认情况下，加载器只会在系统标准路径（如 /usr/lib）下寻找，而不会看当前目录，需要添加：export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
- 检查工具
  - ldd myapp：查看程序运行需要哪些 .so，以及它们目前被定位到了哪里
  - nm -D libmymath.so：查看动态库导出了哪些函数接口（符号表）

CMake中编译静态库和动态库

- 在 CMakeLists.txt 中通过 STATIC 或 SHARED 关键字来决定库的类型

```
cmake_minimum_required(VERSION 3.10)
project(LibraryExample)

# 设置头文件路径
include_directories(include)

# 1. 编译静态库 (libmy_static.a)
add_library(my_static STATIC src/math_lib.c)

# 2. 编译动态库 (libmy_shared.so)
add_library(my_shared SHARED src/math_lib.c)

# 3. 编译主程序并链接
add_executable(app_static src/main.c)
target_link_libraries(app_static my_static)
add_executable(app_shared src/main.c)
target_link_libraries(app_shared my_shared)
```

CMake中编译静态库和动态库（续）

- 或者通过CMake变量 BUILD_SHARED_LIBS 来全局控制

```
cmake_minimum_required(VERSION 3.10)
project(FlexibleLib)

# 确保在安装后的可执行文件也包含库的搜索路径
set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib")
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)

# 如果不指定类型，CMake 会根据 BUILD_SHARED_LIBS 变量来决定
# 默认通常是 STATIC
add_library(my_lib src/math_lib.c)

add_executable(myapp src/main.c)
target_link_libraries(myapp my_lib)
```

- 编译为静态库： cmake .. -DBUILD_SHARED_LIBS=OFF
- 编译为动态库： cmake .. -DBUILD_SHARED_LIBS=ON

静态库与动态库小结

- 静态库与动态库对比

特性	静态库 (.a / .lib)	动态库 (.so / .dll)
链接时间	编译链接阶段	程序运行阶段
文件大小	生成的 exe 文件较大	生成的 exe 文件较小
内存占用	每个进程独立拷贝，浪费内存	多个进程共享，节省内存
部署便捷性	极高（单文件运行）	一般（需携带库文件）
更新维护	需要重新编译主程序	替换库文件即可生效

- 开发时如何选择
  - 选择静态库： 如果软件比较小，或者希望用户“下载即用”，不希望用户因为缺少环境而报错，静态库是首选
  - 选择动态库： 对于大型系统（如 Linux 内核、大型游戏引擎），或者需要频繁插件化更新的项目，动态库是不二之选

本章内容概述

- 位运算
- 宏操作（带参数、系统宏）
- 静态与动态库
- 编程优化方法

编程优化概述

- 优化是一门平衡艺术，需要在执行速度、内存消耗和代码可读性之间寻找最优解，对高性能或资源受限的嵌入式开发至关重要
- 编程中常用的优化方法，分为四个维度：
  - 数据结构与算法优化：比任何底层的代码微调都要高效，例如大规模数据的二分查找比普通的顺序查找性能要高得多
  - 函数级优化：函数调用的开销虽然微小（入栈、跳转、出栈），但在高性能计算或高频调用的内层循环中，这些开销会积少成多
  - 控制指令级优化：分支和循环是程序逻辑的骨架，由于CPU的指令流水线和分支预测实现，不当的编写方式往往会成为性能瓶颈
  - 运算指令级优化：算术运算指令的执行开销差异巨大，用低开销的指令组合替代高开销的单条指令

函数级优化

- 核心思路是：减少调用开销、优化数据交换方式、以及辅助编译器决策
- 调用级：减少函数调用，消除栈帧开销
  - 使用内联函数 (inline) 或者带参数的宏
  - 尾递归优化 (Tail Recursion)
- 数据级：减少拷贝、消除内存依赖
  - 减少参数传递开销（传值 vs 传址）
  - 利用 restrict 关键字消除别名
- 编译器级：帮助编译器进行死代码删除或循环外提
  - 使用 static 关键字限制函数作用域及 const 关键字优化内存访问
  - 使用编译器属性告知编译器函数的特殊行为

内联函数

- 内联函数（C99）：告诉编译器将函数代码直接“嵌入”到调用点，消除函数调用的压栈、跳转和返回开销
  - 适用场景： 函数体非常小（如 1-5 行）且被频繁调用
  - 与带参数宏的差异：编译器会检查函数的有效性，没有宏的副作用
  - 注意： inline 只是给编译器的建议，并不保证该函数一定会被内联。现代编译器（如开启 -O2）通常会自动内联它认为合适的函数

```
// 内联函数
static inline int max(int a, int b) {
  return (a > b) ? a : b;
}

// 带函数的宏,思考: MAX(a++, b++)会有什么结果?
#define MAX(a, b) (((a) >(b)) ? (a) : (b))
```

尾递归优化

- 如果函数的最后动作是调用自身，编译器可以将其转化为 jmp 指令，而不是建立新的栈帧，防止栈溢出并提升速度

```
// 普通递归：需要 O(n) 栈空间
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial(n - 1);
}

// 尾递归：编译器可优化为循环,O(1) 栈空间
int factorial_tail(int n, int accumulator) {
  if (n == 0) return accumulator;
  return factorial_tail(n - 1, n * accumulator);
}
```

参数地址传递

- 函数参数默认是“值传递”。如果传递大型结构体，会发生内存拷贝。改用指针（地址传递）只需拷贝一个机器字长（4 或 8 字节）
- 传递结构体指针，如不需要修改结构体，可使用 const 修饰以保证安全

```
typedef struct {
  double matrix[100][100];
} BigData;

// 不推荐：拷贝 80,000 字节
void process(BigData data);
BigData update_value();

// 推荐：仅拷贝一个 8 字节指针
void process_optimized(const BigData *data);
void update_value_optimized(BigData *data);
```

消除指针别名

- restrict 关键字（C99）用于修饰指针，表示在该指针的生命周期内，它所指向的内存地址不会被其他指针访问，解决“指针别名”问题
- 此时，编译器不需要在每次操作后重新从内存加载数据，可以直接利用寄存器中的缓存。这是编译器实现 SIMD（向量化） 优化的前提

```
// 告诉编译器 a, b, result 指向的内存绝不重叠
void add_arrays(int *restrict a, int *restrict b, int *restrict
result, int n) {
  for (int i = 0; i < n; i++) {
    result[i] = a[i] + b[i]; // 编译器现在可以放心地使用向量化指令
  }
}

int a[10] = {1, 2, 3, 4, 5};
// 错误用法：让两个 restrict 指针指向了同一个数组重叠的区域
add_arrays(a, a + 1, a, 5); // 错误使用时,会导致结果不确定
```

编译器函数属性

- GCC 和 Clang 提供了一些非标准的属性，告知编译器函数的特殊行为
  - __attribute__((const))：告知编译器该函数不读取也不修改任何全局内存，输出仅依赖于输入参数（如 abs()），如果多次调用该函数且参数相同，编译器会将多次调用合并为一次
  - __attribute__((pure))：告知编译器该函数虽然可能读取全局内存，但不会修改任何内存（无副作用），如果循环中调用该函数且没有修改内存的操作，编译器可以将调用移出循环
  - __attribute__((noreturn))：告知编译器该函数执行后不会返回（如 exit() 或死循环），编译器可以优化调用后的控制流，减少不必要的清理代码

```
int square(int x) __attribute__((const));

for (int i = 0; i < 100; i++) {
  // 编译器发现 square(10) 结果不变,只会计算一次,不会重复调用
  sum += square(10);
}
```

控制指令级优化

- 分支语句优化：由于现代 CPU 采用分支预测技术，一旦分支预测失败，CPU 就必须清空流水线并重新加载指令，这会导致严重的性能损耗
  - 分支排布：排布得当可以显著提升CPU分支预测器的命中率
  - 分支暗示：使用内置函数告诉编译器分支的概率，提高预取成功率
  - 无分支编程：通过其它方式直接计算出结果，彻底消除分支语句
- 循环语句优化：循环通常占据了程序 80% 以上的执行时间，优化循环能减少 CPU 的跳转开销，并为 CPU 提供了更多的指令级并行空间
  - 循环展开与融合：展开或融合小的循环体，减少判断和跳转开销
  - 循环外提与交换：外提不变量或交换多层循环，减少冗余计算
  - 终点判断优化：循环判断条件优化，减少不必要的循环

分支排布优化（一）

- 频率优先原则：C 语言的 if 链是按顺序执行的，一旦首个条件命中，后续所有的 cmp 指令都会被跳过，应当将最可能发生的条件放在最前面

```
// 效率低：如果 80% 的情况是普通用户
if (is_admin(user)) { ... }                    // 5% 概率
else if (is_guest(user)) { ... }               // 15% 概率
else { ... }                                   // 80% 概率（普通用户）

// 效率高：将大概率分支置顶
if (is_normal_user(user)) { ... }              // 80% 概率
else if (is_guest(user)) { ... }               // 15% 概率
else { ... }                                   // 5% 概率
```

分支排布优化（二）

- 先过滤“少数派”：当逻辑中包含异常处理或边界检查，使用卫语句 (Guard Clauses) 提前退出，而不是将核心逻辑嵌套在庞大的 if 块中

```
// 不推荐：深层嵌套
void process(data) {
  if (data != NULL) {
    if (is_ready(data)) {
      // 核心逻辑...
    }
  }
}

// 推荐：卫语句,快速失败
void process(data) {
  if (data == NULL) return;       // 异常情况提前退出
  if (!is_ready(data)) return;      // 异常情况提前退出
  // 核心逻辑保持在函数主层级
}
```

分支排布优化（三）

- 逻辑短路的布局：
  - && (与)：将最可能为假或计算开销最小的表达式放在左侧
  - || (或)：将最可能为真或计算开销最小的表达式放在左侧
- switch-case 的排布：
  - 紧凑数值：尽量让 case 的值连续（如 1, 2, 3），编译器会生成跳转表 (Jump Table)，查找效率极高 O(1)，与分支排布顺序无关
  - 离散数值：如果值很分散（如 1, 100, 5000），编译器不会生成跳转表，而会将其转为类似二分查找树，将它们集中在数值较近的范围内会有所帮助
    - 手动分组：如果分散数值有明显的“集群”特征，可以通过一个简单的 if 分组判断，人为减少了每个 switch 内部的搜索深度分组
    - 热点优先：如果知道某个值（比如 1）占了大量的调用频率，那么把它留在 switch 树里是不划算的，可以通过一个 if 判断移出 switch

编译器分支暗示

- 如果非常确定某个条件在大多数情况下都会发生（或不发生），可以使用 GCC/Clang 提供的内置函数告诉编译器，帮助编译器优化汇编代码的排布，提高指令预取的成功率。

```
// __builtin_expect: 告诉 CPU 哪条路径是"高速公路"
#define LIKELY(x) __builtin_expect(!!(x), 1)
#define UNLIKELY(x) __builtin_expect(!!(x), 0)

if (UNLIKELY(ptr == NULL)) {
  // 这里的汇编代码会被编译器移出热点区域
  return ERROR;
}
// 主逻辑保持指令流的连续性
```

无分支编程（一）

- 使用查表法 (Lookup Table) 代替分支：如果分支的作用只是根据一个输入值映射到一个输出值，那么数组查表是最高效的方案

```
// 优化前：冗长的 switch
int get_days(int month) {
  switch(month) {
    case 1: case 3: case 5: return 31;
    case 4: case 6: return 30;
      // ... 其他 case
  }
}

// 优化后：查表法
static const int days_table[] = {0, 31, 28, 31, 30, 31, 30,
31, 31, 30, 31, 30, 31};
int get_days_fast(int month) {
  return days_table[month]; // O(1) 时间复杂度,零分支
}
```

无分支编程（二）

- 使用三元运算符 (a ? b : c)：在许多现代编译器中，三元运算符 会被编译成 cmov（条件移动）指令，不涉及分支预测和 jmp（跳转）指令，因此在处理逻辑简单但结果随机的情况时，效率更高

```
// 将数值限制在 [min, max] 区间内
if (x < min) x = min; else if (x > max) x = max; // 传统做法：使用分支语句
x = (x < min) ? min : ((x >max) ? max : x); // 无分支版：使用三元运算符
```

- 利用算术运算、位运算或逻辑运算来消除 if 语句，在处理大规模随机数据时效率极高，可以避免 CPU 分支预测失败导致的指令流水线清空

```
// 大规模排序算法中整数值的条件交换
if (a > b) { int temp = a; a = b; b = temp; } // 传统做法：使用分支语句

// 无分支版：使用位运算
int mask = -(a > b); // 如果 a > b,mask 为全 1
int t = (a ^ b) & mask; a ^= t; b ^= t;
```

循环展开与融合

- 循环展开：通过减少循环次数，增加单次循环内的操作量，从而减少循环计数和条件跳转的次数

```
// 展开前
for (int i = 0; i < 1000; i++) {
  sum += data[i];
}

// 展开后：减少了 75% 的循环开销
for (int i = 0; i < 1000; i += 4)
{
  sum += data[i];
  sum += data[i+1];
  sum += data[i+2];
  sum += data[i+3];
}
```

- 循环融合：如果两个循环遍历相同的范围，合并它们可以减少循环控制开销，并提高数据重用率

```
// 融合前：遍历两次
for (int i=0; i<n; i++) {
  a[i] = b[i] + 1;
}
for (int i=0; i<n; i++) {
  c[i] = a[i] * 2;
}

// 融合后：遍历一次
for (int i=0; i<n; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] * 2;
}
```

循环外提

- 循环外提：将循环中不随循环变量变化的代码（常量计算、重复的指针寻址、函数调用等）移到循环外面，减少冗余计算

```
// 外提前：每次循环都在重复计算
for (int i = 0; i < n; i++)
  data[i] = i * (a + b) / scale;

// 外提后：提炼出常量因子
double factor = (a + b) / scale;
for (int i = 0; i < n; i++)
  data[i] = i * factor;
```

```
// 外提前：每次循环都要写入内存
for (int i = 0; i < n; i++)
     *res += arr[i];

// 外提后：只写一次内存
int temp = 0; // 使用寄存器暂存
for (int i = 0; i < n; i++)
     temp += arr[i];
*res = temp;// 最后一次性写回内存
```

```
int my_strchr(char * s, char c) { // 思考：该实现有什么问题？该如何优化？
  for (int i = 0; i < strlen(s) - 1; i++)
    if (s[i] == c) return 1;  // 看s中是否包含 c
  return 0;
}
```

循环交换

- 循环交换：在嵌套循环中，改变内外循环的顺序，最忙的循环放最里面，并确保内存访问是连续的，提高缓存命中率

```
// 交换前：内存跳跃极大,Cache Missing 严重
for (int column = 0; column < 1000; column ++) {
  for (int row = 0; row < 5; row++) {
    sum += table[row][column];
  }
}

// 交换后：连续访问内存,速度提升可达 10 倍以上
for (int row = 0; row < 5; row++) {
  for (int column = 0; column < 1000; column ++) {
    sum += table[row][column];
  }
}
```

终点判断优化（一）

- 将循环判断条件优化为与 0 比较，大多数CPU都有 jz 指令来检查计算结果是否为 0，比两个非零变量的比较（i < n）少一条 cmp 指令

```
// 传统做法：需要执行 i < n 的比较操作
for (int i = 0; i < n; i++) {
  process(data[i]);
}

// 优化做法：如果顺序不重要,递减到 0
for (int i = n; i-- > 0; ) {
  process(data[i]);
}
```

终点判断优化（二）

- 哨兵位技术：在数组搜索等逻辑中，通过在数组末尾放置一个“哨兵”，将“越界检查”和“值检查”合并为一个检查，减少了大量的分支判断

```
// 传统做法：当 n 非常大时,每次都要判断 i < n 和是否找到目标
int find(int *arr, int n, int target) {
  for (int i = 0; i < n; i++)
    if (arr[i] == target) return i;
  return -1;
}

// 优化做法：哨兵位（假设数组空间足够）
int find_optimized(int *arr, int n, int target) {
  int last = arr[n-1]; arr[n-1] = target; // 设置哨兵
  for (int i = 0; arr[i] != target; i++); // 循环内只需一次判断
  arr[n-1] = last; // 恢复数据
  if (i < n - 1 || last == target) return i; else return -1;
}
```

运算指令级优化

- 不同的算术运算对 CPU 产生的开销差异极大，通常情况下，加法、减法、位运算只需要 1 个 CPU 时钟周期，而乘法可能需要 3-10 个周期，除法和求余则可能需要 20-80 个周期
- 指令级优化的核心思想是：用低开销的指令组合替代高开销的单条指令
  - 极低开销运算（通常1个机器周期或更少）：位运算、赋值（=）、加减法
  - 低开销运算（1-数个机器周期）：乘法（*）、逻辑运算（&&、||）
  - 中开销运算（数十个机器周期）：除法与取模（/, %）、分支判断（if-else）
  - 高开销运算（函数调用及内存访问开销）：sin()、cos()、log()、pow()
  - 极高开销运算（动态内存分配与文件操作）：malloc()、printf()、fread()

位运算 < 整数加减 < 乘法 < 除法 < 函数调用 < 内存读取 < 系统调用

乘除法转位运算

- 对于 2^n 的乘除法，位移运算几乎是瞬间完成的
  - 乘法： $x * 2^n$ 转化为 $x << n$
  - 除法： $x / 2^n$ 转化为 $x >> n$
  - 取模： $x \% 2^n$ 转化为 $x \& (2^n - 1)$

```
// 优化前：使用乘、除和取模
int a = b * 8;
int c = d / 4;
int e = f % 16;

// 优化后：使用位运算中的移位和与
int a = b << 3;
int c = d >> 2;
int e = f & 15; // 仅限正数
```

除法转乘法

- 除法是 CPU 运算中的“性能杀手”，需要多次除以同一个常数，可以将其转化为乘以该常数的倒数，即：x/y = x*(1/y)

```
// 优化前：在循环中频繁除法
for (int i = 0; i < n; i++) {
  data[i] /= 1.25;
}

// 优化后：预计算倒数,转为乘法
double inv = 1.0 / 1.25; // 0.8
for (int i = 0; i < n; i++) {
  data[i] *= inv;
}
```

- 现代编译器在处理整数常数除法时（如 n / 3），会自动将其转化为复杂的乘法加位移组合，不需要手动优化

代数恒等式简化

- 避开平方根： 在比较长度时，比较平方值

```
// 优化前
if (sqrt(x*x + y*y) > radius) { ... }
// 优化后：消除昂贵的 sqrt()
if ((x*x + y*y) > (radius * radius)) { ... }
```

- 合并公共项

```
// 优化前
y = a * b * c + a * b * d;
// 优化后
y = (a * b) * (c + d); // 减少了一次乘法
```

- 定点数运算代替浮点数

```
// 处理货币,不使用 float 1.25 元,使用 int 表示"分"
// 在没有浮点运算单元的设备中,float 运算由软件库模拟,速度极慢
int price_in_cents = 125;
int total = price_in_cents * quantity;
```

程序性能分析工具

- 在 开发中，“过早优化是万恶之源” -- Donald Knuth，进行优化的前提是找到瓶颈。gprof 和 perf 是 Linux 下最常用的两款性能分析利器
- 性能分析工具 gprof 与 perf 的特性对比

特性	gprof	perf
原理	编译时插桩（修改代码）	内核采样（不修改代码）
开销	较大，可能影响运行逻辑	极小，适合生产环境
维度	仅限用户层函数调用	用户层 + 内核 + 硬件 (Cache Miss 等)
安装	随 gcc 携带，几乎都有	需要额外安装 linux-tools
优点	调用次数统计非常精准	能看到 CPU 周期、指令跳转等深层信息

- 简单来说：gprof 适合查看函数调用频率，perf 能深入内核和硬件层面

使用 gprof 进行性能分析

- gprof 的原理是插桩，即在编译时往每个函数里塞入统计代码
- 使用步骤：
  - 编译（必须加上 -pg 参数）：gcc -pg test.c -o my_gprof_app
  - 正常运行程序，运行结束后，目录下会生成一个 gmon.out 的二进制文件
  - 使用 gprof 进行分析：gprof my_gprof_app gmon.out > analysis.txt
- 查看结果（打开 analysis.txt 文件）：
  - Flat Profile：每个函数耗时占比（函数运行时间百分比）
  - Call Graph：函数调用链，报告谁调用了谁，以及调用了多少次

使用 perf 进行性能分析

- perf 是基于采样的分析工具，不修改程序代码，而是利用 Linux 内核周期性地查看 CPU 正在执行哪行代码
- 使用步骤：
  - 编译： 建议加上 -g（保留调试信息），方便看源码行号
    gcc -g test.c -o my_perf_app
  - 实时查看（类似 top）： sudo perf top -p $(pidof my_perf_app)
  - 记录并报告（最常用）：
    - 录制（生成一个 perf.data 文件）： sudo perf record -g ./my_perf_app
    - 分析：sudo perf report
      进入交互界面后，你可以选中某个函数按 Enter，选择 Annotate（注释），perf 会直接把源码展示出来，并在每一行前面标出耗时百分比

使用 gcc 进行性能优化

- gcc 编译器通过不同的优化选项，你可以指示它在编译时间、程序运行速度、二进制文件大小以及调试易用性之间进行权衡

选项	目的	描述
-O0	**不优化**	默认选项。编译最快，生成的代码最易于调试（变量与源码一一对应）。
-O1	**基础优化**	尝试减少代码大小和运行时间，但不进行耗时较长的优化。
-O2	**推荐级别**	**大多数生产环境的选择。**开启了几乎所有不涉及“空间换时间”的优化，平衡了性能与文件大小。
-O3	**激进优化**	开启循环向量化（Vectorization）和更积极的内联。可能显著增加二进制文件体积。
-Os	**大小优化**	专门针对文件体积优化。它会关闭 -O2 中那些会增加代码大小的优化，适合嵌入式设备。
-Ofast	**极限速度**	开启 -O3 且**无视数学标准（如 IEEE 754）**。虽然极快，但可能导致浮点运算精度丢失。
-Og	**调试优化**	优化性能的同时，保证调试器（GDB）能准确追踪源码。推荐在开发阶段使用。

注意：优化级别越高，并不总是意味着程序越快，需要使用工具进行实际测量

使用 gcc 进行性能优化（续）

- gcc 性能优化实践
  - 开发阶段（保留调试信息）：
    - gcc -g -Og main.c -o app
    - -Og：在不破坏调试信息的前提下，让程序跑得稍微快一点
  - 发布阶段（极致压榨 CPU 性能）：
    - gcc -O3 -march=native -flto main.c -o app
    - -march=native：探测当前机器的 CPU 型号，开启支持的所有指令集
    - -flto：跨文件全局优化，实现跨文件内联函数，从而大幅提升整体性能
    - 注意：有时候 -O3 带来的指令缓存失效反而会让程序慢于 -O2；有时候 -O3（激进优化）编译程序无法正常运行，需要降级为-O2（标准发布）

谢谢！

中国科学技术大学
University of Science and Technology of China