第一部分　改进程序性能

Microsoft C专用开发系统帮助你使用其高级优化程序和增强型存贮器管理功能。

第一章谈谈何时采用某种优化以及Microsoft C怎样产生执行速度快、长度短的代码。第二章叙述了Microsoft C为你提供用来分配和管理程序存贮器（包括新的_based类型）的高级工具。如果程序需要局部优化，可采用第三章描述的内嵌汇编程序。这样可生成最紧凑的代码。如果应用中需要浮点数学运算，你会发现第四章在解释Microsoft C数学软件包可选项方面是很有帮助的，它解释了哪个浮点可选项产生最快、最小、最灵活的代码。

第一章　优化C程序

Microsoft C编译程序将C源语句翻译成机器可执行的指令。另外，汇编程序重写或者"优化"程序中某些部分，使之更为有效，而在源代码级看不出来的。

编译程序执行的优化有三大类型：

1. 修改或移动部分代码，以便使用更少的指令或更为有效地使用处理器。2. 移动代码并合并运算，以便最大限制地利用寄存器。因为对存放在处理器寄存器中数据的运算远远快于对存在存贮器中数据同样的运算。3. 删除多余的和无用的代码。

本章介绍了控制Microsoft C编译程序优化代码的各种途径。

1.1 从程序员工作台控制优化

程序员工作台（PWB）是一个集成开发环境，用来编辑、建立和调试 Microsoft C 编写的应用程序。关于PCW更详尽的内容，参阅《安装和使用 Microsoft C 专用开发系统》。

在程序员工作台内有两种方式编译：

1. Debug编译。若缺省，编译程序不执行任何优化。
2. Release 编译。若缺省，编译程序进行最佳优化。

用修改C Global Build、Ootions、C Debug Build Option和C Release Build Options（在Options菜单）的方式，可使能或非使能编译程序执行的任何优化，这样可进行细微优化。

在Build Options每个对话框的每个优化对应于CL的一个命令行可选项。（实际上，PWB从你的输入建立一命令行传递给CL。）

注解：在本章中，优化可选项的讨论是按照优化的效果、调用优化的命令行可选项和控

制优化的杂注（pragmas）进行的，所有这些优化均可在Build Options对话框的编译单位（文件）级控制。

1.2 从命令行控制优化

要想从命令行控制优化，你必须先确定应用程序需要哪种优化。然后用以/O（有些情况下用/G）打头的命令行可选项说明这些优化。

如果可选项之间发生冲突，编译程序就采用命令行说明的最后一个可选项。命令行

```
CL /Oa /Ol /Ot TEST.C
```

对程序TEST.C编译。它说明编译程序可以：

1) 化化的前提是未做别名使用（/Oa）
2) 进行循环化化（/Ol）
3) 进行加速总化化（/Ot）

前述的命令行也可写作：

```
CL /Oalt TEST.C
```

1.3 用编译指令控制优化

有时，你需要更精细地控制优化范围，命令行可选择使你能够以整个编译单位（文件）控制优化。另外，Microsoft C支持几种编译指令，使你能在每个功能的基础上进行控制。

本章所描述的控制优化的编译指令可以分为以下类型。

在C6.0中，每个优化指令都使用Optimize，跟随一个优化开关。

- 关于别名的优化（a和w）
- 清除局部公用子表达式（c）
- 清除全局公共子表达式（g）
- 全局寄存器的分配（o）
- 循环优化（l）
- 猜测式优化（z）
- 关闭不安全的优化（n）
- 浮点结果的一致性（p）
- 生成更小的代码或更快的速度（t）

任何优化或可选项的结合均可用optimize编译指令控制。例如，你要对某个函数使用，这时如果进行无别名优化，就可能出问题，你这时不需要使用这个优化选择。为此，按下述使用optimize：

```
/* Function (s) that do not do aliasing.*/
.
.
.
#pragma optimize ( "a" , off)
/* Function (s) that do aliasing.*
```

```
#pra ma optimize ( "a" , on)
/* More function (s) that do not do aliasing.*/
```

Optimize的参数可以放在一个字符串中。例如，

```
#pragma optimize ( "lge" , off)
```

将禁止循环优化、全局公共子表达式优化和全局寄存器分配。

1.4 缺省优化

如果没有明显地在命令行上禁止优化（/Od），编译器会进行缺省的优化。它们包括：

- 小范围公共子表达式的删除
- 无用存储删除
- 常量传播

1.4.1 公共子表达式删除

在公共子表达式删除中，编译器如果在程序中找到重复的子表达式，就命修改代码，使公用的表达式只计算一次。公用子表达式的例子如下：

```
a=b+c*d;
x=c*d/y;
```

上面两行中都包命公共子表达式c*d。编译程序只计算一次c*d，并把结果放在一中间变量里（通常是一个寄存器）：

```
tmp=c*d;
a=b+tmp;
x=tmp/y;
```

1.4.2 无用存储的删除

无用存贮的删除是公共子表达式删除的推广。在小范围内的变量如果值不变，就被合并到临时变量中。

在下列代码段中，编译程序检测出表达式func(x) 等价于func(a+b)：

```
x=a+b;
x=func (x) ;
```

这样，编译程序可重写代码如下：

```
x=func (a+b) ;
```

1.4.3 常量传播

做常量传播时，编译程序分析变量赋值并确定是否可将它们改变成常量赋值。在下面的例子中，先把7赋给i，再把i赋给j：

```
i=7;
j=i;
```

它被改成：

```
i=7;
j=7;
```

尽管可以在源文件中做这些修改，但这样做可能会降低程序的可读性。在许多情况下，优化不仅提高了程序的效率，而且允许你写出易读的代码。

在某些情况下，你甚至想关闭缺省优化。因为优化可能在目标文件中重新组织代码。在调试中，你的某些代码可能就难以辨认了。因此，在使用符号优化程序之前通常最好去除一切优化。可以用/Od（关闭优化）达到这个目的。

对函数来说，可以用语句#pragma optimize (`` '', off) 关闭一切优化。要将优化恢复成以前状态，用语句#pragma optimize (`` '', on)。

1.5 调整优化

对许多应用来说，缺省的优化就足够了。但有时你还要更进一步地进行优化。优化任选项可以使编译器达到这个目的。

1.5.1 选择速度或尺寸（/Ot和/Os）

除缺省优化外，Microsoft C编译程序还自动使用/Ot可选项进行速度优化。任选项/Ot提高速度，但也可能增加程序的尺寸。如果你宁愿面向缩短程序长度进行优化，就采用/Os任选项。/Os任选项能缩短程序尺寸，但也可能会减慢程序的速度。

如果以函数为单位控制速度和尺寸，就使用带t的optimize。值为on表示进行速度优化；值为off表示生成更小的代码。。例如，

```
#pragma optimize ( "t" , off)          /* Optimize for smallest code.*/
.
.
.
#pragma optimize ( "t" , on)           /* Optimize for fastest code.*/
```

1.5.2 生成内部函数（/Oi）

在某些常规函数调用的位置，C编译程序可以插入运行更快的“内部函数”。每执行一次函数调用，必须执行一组指令来存贮参数并为局部变量保存空间。在函数返回时，也要用一些指令释放局部变量和参数使用的空间，并将值返回到调用程序。这些指令的执行需要时间。对正常大小的函数来说，这些指令影响不大，但如果函数仅有一两行，这些指令几乎会占用函数一半的代码。

为防止这种类型的代码扩展，一种办法是不编写这样的短函数，尤其在速度关键的经常使用代码部分。不过许多库函数仅有一两行代码。编译程序提供两种库函数形式。一种是标准C函数，它进行函数调用的所有操作。另一种完成同样操作，但却不需要函数调用。第二种形式称为内部函数。内部函数总是比标准的函数调用快，它在优化时非常有用。

例如，函数strcpy实际上为：

```
int strcpy (char *dest, char *source)
{
```

```
    while (*dest++ = *source++)
}
```

编译程序含有一内部形式的strcpy。如果你指示了编译程序生成内部函数，任何对strcpy的调用均可用这个内部形式代替。

注解：尽管上例用C写的程序是为清晰起见的，大多数库函数采用汇编语言书写，以便充分利用80×86指令集的功能。内部函数不是定义成宏的简单库函数。

用任选项/Oi编译会使编译程序使用下列函数的内部形式：

abs	labs	outp	strcpy
_disable	lrotl	outpw	strlen
_enable	lrotl	rotl	strset
fabs	memcmp	rotr	
inp	memcpy	strcat	
inpw	memset	strcmp	

下列浮点函数没有真正的内部形式，虽然它们不使用标准的栈传送参数，但却直接向浮点芯片传送数据。

acos	fmod	acosl	fmodl
asin	log	asinl	logl
atan	log10	atanl	log10l
atan2	pow	atan2l	powl
ceil	sin	ceill	sinl
cos	sinh	cosl	sinhl
cosh	sqrt	coshl	sqrtl
exp	tan	expl	tanl
floor	tanh	floorl	tanhl

警告：编译程序执行优化的前提是数学内部函数无副作用。如果你编写了自己的matherr函数，且该函数影响了全局变量，就要使用function编译，使编译器不生成内部函数。

当你不需要前面所列的所有函数都采用内部形式时，就用 intrinsic 代替任选项/Oi。编译指示intrinsic的形式如下：

```
#pragma intrinsic (function1) ...
```

如果你想大部分函数都用内部形式，只有少数函数用标准形式，就在编译时用/Oi而在源文件中使用function编译指示。它具有下列格式：

```
#pragma function (function1...)
```

下列代码显示了intrinsic编译指示的用法：

```
#pragma intrinsic (abs)

void main (void)
{
```

```
        int i, j;

        i=ibg_routine_1 ( ) ;
        j=abs (i) ;
        big_routine_2 (j) ;
}
```

这个程序中的abs函数用一段汇编代码计算。由于没有函数调用的开销，所以程序执行更快。

在上面的例子中，速度的损失不是很明显，因为只调用了一次abs。在下面的例子中，对abs的调用处在循环中，且被调用很多次，使用内部函数则可节省大量执行时间。

```
#pragma intrinsic (abs)
void main (void)
{
int i, j, x;
        for (j=0; j<1000; j++)
        {
                for (i=0; i<1000; j++)


                        x=abs (i-j) ;


    {
    printf ( "The value of x is %d\n" , x) ;
 {
```

下面列的是使用内部形式的函数调用的一些限制：

- 在使用替换数学库时（MLIBCAY.LIB）不能使用内部形式的浮点函数。
- 在Os/2 DLL中不能使用内部形式的浮点函数，因为LLIBCDLL.LIB实际上用了替换数学库。
- 若使用/Ox（最大优化）任选项，就等于开放了/Oi（产生内部函数）。所以/Ox不要与上面所列的几点冲突。

注意：内部形式的_enable、_disable、inp、outp、inpw和outpw在OS/2下不能工作。必须使用库形式。可以用function编译指示强制说明。

1.5.3 没有别名（/Oa或/Ow）

“别名”是用来指示已经有另一个不同名字所指存贮器位置的名字。因为存贮器存取比起CPU寄存器存取占用更多时间，编译程序就设法将常用变量存入寄存器。然而，别名的使用减少了编译程序将变量保存在寄存器中的机会。

指针是引用存贮的一种方法。因为指针的值要到程序执行时才能确定，所以编译程序没法知道程序执行时哪些变量将通过指针修改，因此，编译器只好认为指针会修改任何位置的变量。这样便限制了编译程序在存贮器中长时间保存变量。

任选项/Oa告诉编译程序存贮器中没有多个别名。在下面所列几点中，赋值语句的左

右都可以是变量。任何调用函数的变量都使用的是变量本身。如果编译程序假定没有使用别名，任何没用volatile的变量，都必须遵守下列规则：

- 如果变量直接使用，就没有指向它的指针。
- 如果用指针引用变量，就不会直接引用变量。
- 如果一个指针指向某个区域，就不会用其他指针指向同一个区域。

要说编楚这些就则是怎样影响代码的，考虑下面例子：

```
char  p;
char  *ptr_p;

ptr_p=&p;          /* Take the address of p.*/
```

可以使用*ptr_p或p，但不就在同一个函数中使用。如果引用了两个变量名，便是使用了别名。例如，

```
char    *p_buf;
char    *p_alias;

if ((p_alias=p_buf=malloc (5000)) ==NULL)
     return;
else
{
     .
     .
     .
}
```

上面例子中的代码是很常见的。它示范了从堆中动态分配一块存贮器并将原来的地址保存在p_buf中。程序然后在别名p_alias上执行所有的指针算术。当函数用完了存贮器块后，对于free函数，p_buf是一有效参数，因为它仍然包含了原来的地址。

可选项/Oa和/Ow告诉编译程序你没有在代码中使用别名。可选项/Oa和/Ow之间的区别是，用/Oa表示你将不会使用别名（这样编译程序便可执行重要的优化，否则是不可能的）且函数调用是安全的。/Ow选择项类似于/Oa，例外是，函数调用之后，指针变量必须从存贮器重新装入。

下面的程序例子使我们对/Oa或/Ow优化可选项的使用略见一斑：

```
int g;

void main (void)
{
     add_em (&g) ;
}

int add_em (int *p)
```

```
    *p=2;          /* Assign a value to an alias for g. */
    g=3;           /* Assign a value directly to g. */
}
```

在函数add_em中，g和*p均指示同一存贮器位置，该位置先赋以值2，后值3。*p（g的别名）所指之值然后加上g，结果返回主程序。如果你不使用/Oa命令行可选项，编译程序便假定对*p的引用可以和g一样对同一存贮器位置引用，于是就不会用寄存器存放两者的值。不过，假如你说明了/Oa可选项，编译程序假定g和*p引用不同存贮器位置，于是将两者存放在不同寄存器中。在return语句处，g与*p具有不同值，尽管两个别名实际上应该包含同样值。

注意，编译程序是在有限时间内将值保存在寄存器中的。如果一个存贮器位置的不同别名发生在不同函数，它们就不会带来预料不到的结果，没有把握时就不要使用别名。

包含别名的错误难于发现，别名错误最常见的错误是数据的破坏。如果发现全局或局部变量被指定了似乎随机的值，那么采取下列步骤确定是否有优化和别名方面的问题：

- 用/Od（非使能优化）编译程序。
- 如果用/Od编译后程序能工作，对于/Oa可选项检查正常的编译可选项。
- 若使用了/Oa可选项，修改编译可选项使/Oa没有说明。

注意，用带a或w可选项的optimize杂注，你可以指示编译程序非使能带有别名使用的不安全代码的优化。

1.5.4 进行循环优化（/Ol）

任选项/Ol允许对循环进行优化。由于循环部分被多次重复，所以它们是优化的主要目标。这些优化都涉及到移动或重写代码以便执行起来更快。

循环优化可用任选项/Ol打开，也可以用Loop_opt编译指示控制。下列语句允许循环优化：

```
#pragma loop_opt (on)
```

而下列行则关闭循环优化：

```
#pragma loop_opt (off)
```

优化的循环中只能包括每次循环都改变的表达式。如果在循环中某个子表达式保持不变，就应该先计算它们。不幸的是，有些子表达式难以发现。优化程序的目的就是在编译时把这些子表达式移出循环体，

例如：

```
i=-100;
while (i<0)
{
    i+=x+y;
}
```

上例中表达式x+y在循环体中不改变。循环优化程序将这个子表达式从循环体中移出，使它只计算一次，这些优化包括移动代码或重写代码，使它们执行得更快。最后得到的代码如

下：

```
i=-100;
t=x+y;
while (i<0)
{
    i+=t;
}
```

当编译程序假定无别名使用时，循环优化更加有效。当然可以在循环优化时不用/Oa或/Ow，当使用了它，会得到更高的效率。

下面的代码段可能有别名使用的问题：

```
i=-100;
while (i<0)
{
    i+=x+y;
    *p=i;
}
```

如果没有用/Oa任选项，编译程序只好假定x或y可能被*p所修改。这时子表达式x+y就不是常量。如果你使用了任选项/Oa，编译程序就会假定*p不影响x或y，作为常量的子表达式就可以提出循环体之外。

注意：用/Ol任选项或Loop_opt说明一切循环优化都是安全的。要进一步循环优化，必须使用猜测式优化（/Oz），/Ol和/Oz组合可以满足大部分程序的要求，但并不总是安全的。

1.5.5 禁止不安全的循环优化（/On）

关闭不安全的循环优化（/On）是一个已废弃的任选项。保留它的目的是与旧版本的兼容。缺省情况下的循环优化是安全优的。/On作为缺省值与/Oz作用相反。

1.5.6 允许猜测式优化（/Oz）

编译程序可进行极主动的优化，这种优化在速度和长短方面均产生高效的代码，然而某些程序不适合使用这种技术。对于它们就不能用任选项/Oz，但你仍然可以使用其它优化方法。

由于猜测式优化/Oz采用的方法非常富有进攻性，所以它们不属于最大优化（/Ox）的一部分。

下面用实例说明/Oz项的效果：

· 循环优化（/Ol），循环优化使用的方法是分析程序流程，并在循环中提取不变表达式。当使用（/Oz）时，编译器在提取不变的表达式后可能会出错。这种错误常常导致循环体中的if语句所保护的条件出错。不变的表达式被提出循环体外，导致保护它的if语句无法检测错误。下面是两个引起这类问题的例子：

```
for (i=0; i 100; ++i)
    if (float_val !=0.0F)
```

```
/*protect against divide-by-zero.*/
    float_result=pi/float_val;


while (condition)
    if (ptr_val != NULL)
      /*protect pointer dereference.*/
       char_var=*ptr_val;
```

· 全局寄存器分配（/Oe）。允许猜测式优化时所采用的寄存器的分配策略可能会导致寄存器中存放错误的段。虽然这个问题源于DOS，但在OS/2中却会引起保护错。

1.5.7 删除堆栈检查（/Gs）

每次调用一个函数时，都要为函数的参数和局部变量申请栈空间。有一个小的汇编函数用来检查是否出现栈溢出。堆栈上的溢出会导致无限循环或无限递归。这类错误常出现在特别多的参数和大块局部变量函数中。

在开发程序的过程中，堆栈检查非常重要。堆栈上溢错误可以提醒你代码中有问题，但是程序经过调试之后，堆栈检查就没有必要了。编译程序允许你用任选项/Gs或checkstack编译指示删除栈检查。删除后的程序会更小，也会运行得更快。

1.5.8 允许全局寄存器分配（/Oe）

全局寄存器分配任选项（/Oe）指示编译程序对你的程序进行分析，从而尽可能有效地分配CPU寄存器。在不使用这种优化的情况下，编译器按下面的方法使用寄存器：

1） 存放变量的临时副本

2） 存放用register关键字说明的变量

3） 将参数传递给_fastcall关键字说明的函数（或者用/Gr命令行任选项编译过的程序中的函数）

当使用了全局寄存器分配时，编译程序则忽略register关键字并将寄存器分配给变量（也可能会给公共子表达式）。编译程序是按照使用频率将寄存器分配给变量或子表达式的。由于物理寄存器数目有限，所以有时将寄存器中的变量放回到主存中，以便腾出寄存器用在它处。下面的C程序例子示范了编译程序是怎样改写你的代码来完成这项任务的：

```
/Original program*/

func ( )
{
    int i, j;
    char*pc;


    for (i=0; i<1000; ++i)
    {
        j=i/3;
        *pc++= (char) i;
```

```
    }

    for (j=0, --pc; j<1000;
        ++j, --pc)
        *pc--;
}
/* Example of how the compiler might optimize the
 * code to move i and j in and out of registers*/

func ( )
{
    int i, j;
    char *pc;

    {
    register int i;     /* i is in a register for this block.*/
        for (i=0; i<1000; ++i)
        {
            j=i/3;
            *pc++ = (char) j;
        }
    }

    {
    register int j; /* j is in a register for this block.*/
        ++j, --pc)
        *pc--;
    }
}
```

上例中有些块（用花括弧括起）的唯一目的就是确定变量应该放在寄存器中在代码中的范围。

注释：可以用带e任选项的optimize来允许或关闭以函数为基础的全局寄存器分配。

1.5.8 允许公共子表达式提取（/Oc和/Og）

如使用了任选项/Og（允许全局公共子表达式优化），则编译程序将从整个函数中搜索公共子表达式。任选项/Oc（缺省的公共子表达式优化）只从短小的代码部分寻找公共子表达式。可以用任选项/Od来关闭缺省的公共子表达式优化。有关公共子表达式优点的更详尽内容，参见1.4节，“缺省优化”。

注释：可以用带C任选项的optimize来允许或关闭在函数基础上的块范围公共子表达式优化。可以用带g任选项的optimize来允许或关闭在函数基础上的全局公共子表达式优化。

1.5.10 获取一致的浮点结果（/Op）

浮点数在存贮器中占据32、64或80位，分别依次对应于类型float、double或long double。80x87家族的协处理器对所有运算使用80位寄存器进行操作。如果在进行若干个运算期间，float或double类型的值是保存在这些寄存器中而不是在存贮器之间来回传送，那么就会更加精确。

鉴于浮点数在存贮器与寄存器中的表示方法具有精度差别，所以存放在内存贮器中的值不是永远同80x87寄存器中的值相同。

精度差别主要影响严格相等或严格不等测试（==和!=）。但是，如果协处理器能够保持高位数字而存贮器变量不能的话，那么数量的关系测试（>、>=、<=和<）就可能出错。

用/Op任选项能够避免精度差别，这个任选项强迫浮点值在运算期间写入内存贮器。虽然将值存入存贮器中会降低浮点表达式的精度，但它却保证了不管其它代码如何，这些表达式都会产生一致的结果。

可以用带p任选项的optimize来改变在函数基础上的浮点处理结果。

注释：如果浮点寄存器不可用来存放中间结果，那么就用/Op可选项取消其它优化。取消这些优化后，用/Op任选项编译的代码比没用这个任选项编译的代码执行速度慢。通过精心地设计代码（尤其在严格相等和不等的测试中），还是可以避免这个任选项的。

1.5.11 使用80186、80188或80286处理器（/G0、/G1、/G2）

若不采取特殊步骤，编译程序就生成8086目标代码（/G0）。因为新的处理器（80186、80188和80286）与8086指令集兼容，所以采用这个指令集就保证了用所有基于80x86的计算机兼容。在获得与80x86处理器整个家族的兼容性的同时，也失去了新处理器的某些功能更强指令的优点。

如果已知程序只在80186、80188或者80286上运行，就可以让编译程序生成这些处理器专用的指令。这些指令加快了程序的速度，但却失去了与使用80x86家族较早处理器的机器的兼容性。表1.1中列举了用于专用处理器代码生成的任选项：

表1.1 处理器的兼容性

Command-Line Option ①	Compatible Processors ②
/G0	8088, 8086, 80188, 80186, 80285, 80388, 80486
/G1	80188, 80186, 80286, 80386, 80486
/G2	80286, 80386, 80486

①命令行任选项
②可兼容的处理器

注释：开发如果仅仅是为了OS/2，那么总是用/G2任选项，因为OS/2不可在8086、8088、80186或80188上运行。如果使用传统API，就不要使用/G2任选项。

1.5.12 最大限度优化（/Ox）

/Ox任选项将若干不同优化组合在一起：

1） 允许全局寄存器分配（/Oe） 2)允许全局公共子表达式优化（/Og） 3）允许块范围公共子表达式优化（/Oc） 4）生成内部函数（/Oi） 5）执行循环优化（/Oi） 6）速度优化（/Ot） 7）删除堆栈检查（/Gs）

用/Oaax /Gr可获得运行速度最快的程序。/Ox任选项中不包括以下几个改进代码效率的优化：/Oa（假定无别名使用）、/Oz（猜测式优化）和/Gr（调用约定）。为了使用这些优化，应该先阅读描述/Oa和/z任选项的章节并了解调用的约定，再确定它们是否适用于你的程序。

采用optimize优化可缩短代码的长度。如果与执行时间相比，你更关心可执行文件的长度，使用/Ox和/Gs任选项和optimize优化，如下：

```
#pragma optimize ( "t" , off)
```

这组任选项在执行某些速度优化的同时，产生尽可能短的代码。

1.6 控制优化的连接器（LINK）任选项

大部分代码优化是在生成目标代码之前进行的。还有四种优化是在连接时实现，以加快程序速度和减少程序尺寸。

1.6.1 允许far调用优化（/FARCALLTRANSLATION）

函数调用有两种方式，在far调用时，既使用函数的段地址，也使用偏移量，这样程序调用的函数可以超出64K。在near调用时，调用语句必须和函数处于同一个段内。仅用偏移量来存取函数，而段地址是隐含的。对同一段内的例程，只能用near调用。

处理器的结构决定了near调用执行速度比far调用快。把函数说明为near还是far的往往是在选择存贮器模型时决定的。因为要确定连接程序会将给定函数放在存贮器的哪处是困难的，所以实际上由程序员选择程序调用的方式。

对中大规模的程序采用/FARCALLTRANSLATION，这个任选项允许far调用优化。若采用了这个任选项，任何与被调用函数同处一段的函数调用均被转换成near调用。如果选取的是小程序，那么这个优化就没有效果，因为所有调用已经是near调用了。

/FARCALLTRANSLATION任选项的缩写是/F。

1.6.1.1 /FARCALLTRANSLATION如何影响你的程序

连接程序可以使用一种后期优化（在大部分实际代码生成之后的优化）把far转变成near调用。这种优化允许在一个程序中，既可被far调用又可被near调用。为了进行这种转变，连接程序取一段目标代码，如：

```
CALL    FAR    _func
```

其中func在当前段中定义，连接程序把它替换成：

```
PUSH    CS
CALL    NEAR    func
NOP
```

这种替换是成立的，因为连接程序插入了一句PUSH CS，将far返回地址放在堆栈上。

/FARCALLTRANSLATION任选项与/PACKCODE任选项连用是最有效的。PACKCODE任选项将在1.6.2中讨论。/PACKCODE任选项将段之间的far调用转变成段内的far调用。而/FARCALLTRANSLATION则把段内的far调用变为near调用。

1.6.1.2 /FARCALLTRANSLATION的好处

/FARCALLTRANSLATION任选项对保护方式下的程序带来很大的好处。表1.2说明了原因。

表1.2 调用序列的处理器时钟周期

	Cycles (Real Mode)		Cycles (ProtectedMode)	
Instructions	286	386	286	386
Far Function Call				
CALL FAR PTR_func	13	17	26	34
Total	13	17	26	34
Near Function Call				
PUSH CS	3	2	3	2
CALL NEAR PTR_func	7	7	7	7
NOP	3	3	3	3
Total	13	12	13	12
Savings	0	5	13	22

① 指令　② far函数调用　③ 总和　④ near函数调用　⑤ 节约
⑥ 周期实方式　⑦ 周期保护方式

1.6.2 压缩代码（/PACKCODE）

/PACKCODE任选项将相邻的代码段聚集在一起。当与/F任选项一起使用时，/PACKCODE任选项大大增加函数的near调用数目。这个任选项后可接一界限值(以字节为单位)。在达到这个界限值时停止压缩，并开始新的分组。下面是/PACKCODE任选项的语法：

/PACKCODE：number

其中number为可选择的数字，表示压缩限度。该数字的基数（八进制、十进制、十六进制或二进制）用C语言的方式给出。

基数	规则
八进制	用0开头，只能用数字0～7。例如，07777。
十进制	没有引导零的数，例如，65530。
十六进制	以ox开头的数。例如，ox3FFF

如果省略了该数字，缺省值为65,530。/PACKCODE任选项简写为/PACKC。

1.6.3 压缩数据（/PACKDATA）

/PACKDATA任选项把相邻的数据段组合在一起。这个任选项对起过Os/2的255段的

1.5.12 最大限度优化（/Ox）

/Ox任选项将若干不同优化组合在一起：

1）允许全局寄存器分配（/Oe） 2)允许全局公共子表达式优化（/Og） 3）允许块范围公共子表达式优化（/Oc） 4）生成内部函数（/Oi） 5）执行循环优化（/Oi） 6）速度优化（/Ot） 7）删除堆栈检查（/Gs）

用/Oax /Gr可获得运行速度最快的程序。/Ox任选项中不包括以下几个改进代码效率的优化：/Oa（假定无别名使用）、/Oz（猜测式优化）和/Gr（调用约定）。为了使用这些优化，应该先阅读描述/Oa和/z任选项的章节并了解调用的约定，再确定它们是否适用于你的程序。

采用optimize优化可缩短代码的长度。如果与执行时间相比，你更关心可执行文件的长度，使用/Ox和/Gs任选项和optimize优化，如下：

```
#pragma optimize ( "t" , off)
```

这组任选项在执行某些速度优化的同时，产生尽可能短的代码。

1.6 控制优化的连接器（LINK）任选项

大部分代码优化是在生成目标代码之前进行的。还有四种优化是在连接时实现，以加快程序速度和减少程序尺寸。。

1.6.1 允许far调用优化（/FARCALLTRANSLATION）

函数调用有两种方式，在far调用时，既使用函数的段地址，也使用偏移量。这样程序调用的函数可以超出64K。在near调用时，调用语句必须和函数处于同一个段内。仅用偏移量来存取函数，而段地址是隐含的。对同一段内的例程，只能用near调用。

处理器的结构决定了near调用执行速度比far调用快。把函数说明为near还是far的往往是在选择存贮器模型时决定的。因为要确定连接程序会将给定函数放在存贮器的哪处是困难的，所以实际上由程序员选择程序调用的方式。

对中大规模的程序采用/FARCALLTRANSLATION，这个任选项允许far调用优化。若采用了这个任选项，任何与被调用函数同处一段的函数调用均被转换成near调用。如果选取的是小程序，那么这个优化就没有效果，因为所有调用已经是near调用了。

/FARCALLTRANSLATION任选项的缩写是/F。

1.6.1.1 /FARCALLTRANSLATION如何影响你的程序

连接程序可以使用一种后期优化（在大部分实际代码生成之后的优化）把far转变成near调用。这种优化允许在一个程序中，既可被far调用又可被near调用，为了进行这种转变，连接程序取一段目标代码，如：

```
CALL   FAR   _func
```

其中func在当前段中定义，连接程序把它替换成：

```
PUSH   CS
CALL   NEAR   func
NOP
```

例如，如果在编译时用/Gr（寄存器调用约定）任选项后，你还需要add_two函数采用C调用约定，说明如下：

```
int _cdecl add_two (int x, int y) ;
```

1.8.2 FORTRAN/Pascal调用约定（/Gc）

FORTRAN/Pascal调用约定是用关键字_fortran或_Pascal说明的。(目前这两个关键字产生同样结果。）这些函数的参数总是自左至右地压入栈中。由于任何函数均可用FORTRAN/Pascal约定来说明，所以可以在C程序中调用Pascal 或 FORTRAN 过程。同时这种调用还可生成较短较快的程序。

/Gc任选项（生成Pascal类型的函数调用）可以使文件中所有函数都采用FORTRAN/Pascal调用约定。

注意，C语言库函数永远采用C调用约定。它们是在头文件中用_cdecl关键字说明的，在使用库函数的程序中必须嵌入恰当的头文件。

带有可变参数的函数（如Printf）不可以用FORTRAN/Pascal调用约定。

注释：/ML、/MD和/ML任选项都采用FORTRAN/Pascal调用约定说明浮点函数。详见第十六章，“用OS/2进行动态连接”。

1.8.3 寄存器调用约定（/Gr）

如果参数不通过栈而通过寄存器传送，，就可以缩短程序执行时间。用/Gr编译任选项可以对整个文件使用寄存器调用约定。_fastcall关键字以函数为单位，说明寄存器调用。

因为80x86处理器的寄存器数目有限，所以只有前三个参数使用了寄存器。其余参数用FORTRAN/Pascal调用约定传递。寄存器调用约定能够提高程序执行的速度。

注释：使用寄存器调用约定的参数和register型变量使用的是不同的寄存器。这种调用约定与寄存器变量并无矛盾之处。

在使用寄存器调用约定的函数中编写嵌入汇编语言语句时，汇编语言所使用寄存器可能会与参数所用的寄存器发生冲突。

1.8.4 _fastcall调用约定

本节详细叙述_fastcall调用约定。这些信息是为了对嵌入汇编或用MASM编写_fastcall类型的函数的程序员准备的。用_fastcall说明的函数通过寄存器接收参数，而用_cdecl或_Pascal说明的函数通过栈传送参数。

警言：这里叙述的寄存器用法只适合Microsoft版本6.0，在未来编译程序版本中可能会有改变。

1.8.4.1 参数传递的约定

_fastcall调用约定是“强制类型”寄存器调用约定，这时编译器可以通过寄存器把指定类型的数据用简炼的代码进行传递。由于编译程序依照参数类型而不是严格的线性顺序选择寄存器，故调用程序和调用函数必须在参数型上一致，方能正确传递数据。

对于每种参数类型都有一个候选寄存器。参数被分配到寄存器中，若没有合适的寄存器被使用，就将参数压入自左至右的栈中。每个参数被放入第一个不含参数的候选寄存器。表1.3示意了基本参数类型以及每种参数的候选寄存器。

表1.3 候选寄存器

③Type	①Register Candidates
character	AL, DL, BL
unsigned character	AL, DL, BL
integer	AX, DX, BX
unsigned integer	AX, DX, BX
long integer	DX:AX
unsigned long integer	DX:AX
near Pointer	BX, AX, DX
far or huge pointer	②passed on the stack

①候选寄存器　　②在栈中传递　　③类型

所有的far和huge pointer都通过栈传递，这与structure、union及floating-point类型相同。

返回值约定

_fastcall的返回值取决于返回的类型，但浮点类型例外，所有浮点类型在NDP栈顶返回。关于NDP栈和返回浮点值的详细内容见第四章，“控制浮点算术运算”。下面所列的是示意包括联合(union)和结构(strncture)在内的4字节值或更小值是如何从_fastcall函数返回的。

大小	返回约定
1字节	AL寄存器
2字节	AX寄存器
4字节	DX、AX寄存器（如果是指针，DX为段，Ax为偏移；如果是长实数型，Dx为高位，AX为低位）

注意，这种返回4字节或更短值的约定与说明为_cdecl的函数的规则是一样的。返回大于4字节的结构或联合将由调用程序在最后一个参数后传送一个隐含参数，该参数是与SS寄存器相关的near指针。它存放返回值的缓冲区指针。一个指向SS:hidden-param的far指针必须放在Dx:Ax中返回。这与返回象_pascal这样的结构的约定是一样的。

1.8.4.2 堆栈调整约定

用_fastcall说明的函数与_cdecl说明的函数不同，它自己从栈中清除参数，调用程序在函数返回后不再调整栈。

1.8.4.3 寄存器保护的约定

所有函数一定要保护DS、BP、SI和DI寄存器，因为你的_fastcall函数可能会修改Ax、Bx、Cx、Dx和ES的值。

1.8.4.4 函数命名约定

用_fastcall说明的函数在放入目标文件时加入一个@引导符。对函数名不进行大小写转换。以下的函数：