

G C C 编译优化指南

作者：[金步国](#)

版权声明

本文作者是一位自由软件爱好者，所以本文虽然不是软件，但是本着 G P L 的精神发布。任何人都可以自由使用、转载、复制和再分发，但必须保留作者署名，亦不得对声明中的任何条款作任何形式的修改，也不得附加任何其它条件。您可 以自由链接、下载、传播此文档，但前提是必须保证全文完整转载，包括完整的版权信息和作译者声明。

其他作品

本文作者十分愿意与他人共享劳动成果，如果你对我的其他翻译作品或者技术文章有兴趣，可以在如下位置查看现有作品的列表：

- [金步国作品列表](#)

BUG 报告，切磋与探讨

由于作者水平有限，因此不能保证作品内容准确无误，请在阅读中自行鉴别。如果你发现了作品中的错误，请您来信指出，哪怕是错别字也好，任何提高作品 质量的建议我都将虚心接纳。如果你愿意就作品中的相关内容与我进行进一步切磋与探讨，也欢迎你与我联系。联系方式：Em ail: csfrank@ citiz.net ； Q Q : 70171448 ； M S N : csfrank122@ hotm ail.com

前言

网上关于编译优化的文章很多，但大多零零散散，不成体系，本文试图给出一个完整和清晰的优化思路，同时提供在实践中如何进行优化的详尽参考。但是， 在介绍所有优化知识之前首先引用 LFS-Book 中的一句忠告：使用编译器优化得到的小幅度性能提升，与它带来的风险相比微不足道”。你还要进行优化 吗？

% @ & # = ^ % ~ * # ...
O K, crazy guy! Let's G o !!

在继续之前，作者还是奉劝各位：如果追求极致的优化，那么它将是一件既耗时又麻烦的事情，你会陷入无止尽的测试、测试、再测试……另外 G entoo w iki 上有这么一句话：“G C C has w e l l o v e r a h u n d r e d i n d i v i d u a l o p t i m i z a t i o n f l a g s a n d i t w o u l d b e i n s a n e t o t r y a n d d e s c r i b e t h e m a l l ”所以本文不会涉及全部 G C C 优化选项。最后作者还是再罗唆一句：优化应当适可而止为好，将精力留出来做一些其它事情会更有意义！

先决条件

本文的主要读者是 LFS/G entoo 的玩家，基本上比较 crazy 的玩家都接触过，如果你之前从未使用过 LFS/G entoo ，请先按照《[Linux From Scratch 6.2 中文版](#)》做一遍 LFS ，然后再来阅读此文将会更有意义。另外，本文是建立在《[深入理解软件包的配置、编译与安装](#)》一文基础之上的，在开始阅读本文之前，请先阅读它。

基本原理

我们首先从三个方面来看与优化相关的内容：

1. 从运行时的依赖关系来看，对性能有较大影响的组件有 k e m e l 和 g l b c ，虽然这严格说来这不属于本文的话题，但是经过精心选择、精心配置、精心编译的内核与 C 库将对提高系统的运行速度起着基础性的作用。
2. 从被编译的软件包来看，每个软件包的 c o n f i g u r e 脚本都提供了许多配置选项，其中有许多选项是与性能息息相关的。比如，对于 A p a c h e - 2 . 2 . 6 而言，你可以使用 `--enable-M O D U L E = s t a t i c` 将模块静态编译进核心，使用 `--disab le-M O D U L E` 禁用不需要的模块，使用 `--w ith-m p m = M P M` 选择一个高效的多路处理模块，在不需要 I P v 6 的情况下使用 `--disab le- i p v 6` 禁用 I P v 6 支持，在不使用线程化的 M P M 时使用 `--disab le-th re ad s` 禁用线程支持，等等……这部分内容显然不可能在本文中进行完整的讲述，本文只能讲述与优化相关的通用选项。针对特定的软件包，请在编译前使用 `configure --he lp` 查看所有选项，并精心选择。
3. 从编译过程自身来看，将源代码编译为二进制文件是在 M a k e f i l e 文件的指导下，由 m a k e 程序调用一条条编译命令完成的。而将源代码编译为二进制文件又需要经过以下四个步骤：预处理(cpp) → 编译(gcc 或 g++) → 汇编(as) → 连接(ld)；括号中表示每个阶段所使用的程序，它们分别属于 G C C 和 B i n u t i l s 软件包。显然的，优化应当从编译工具自身的选择以及控制编译工具的行为入手。

大体上编译优化就这“三板斧”(其实是“三脚猫”)了，本文接下来的内容将讨论这只猫的后两只脚。

编译工具的选择

对于编译工具自身的选择，在假定使用 B i n u t i l s 和 G C C 以及 M a k e 的前提下，没什么好说的，基本上新版本都能带来性能提升，同

时比老版本对新硬件的支持更好，所以应当尽量选用新版本。不过追新也可能带来系统的不稳定，这就要针对实际情况进行权衡了。本文以 Binutils-2.18 和 GCC-4.2.2/GCC-4.3.0 以及 Make-3.81 为例进行说明。

configure 选项

这里我们只讲解通用的“体系结构选项”，由于“特性选项”在每个软件包之间千差万别，所以不可能在此处进行讲解。

这部分内容很简单，并且其含义也是不言而喻的，下面只列出常用的值：

- i586-pc-linux-gnu
- i686-pc-linux-gnu
- x86_64-pc-linux-gnu
- powerpc-unknown-linux-gnu
- powerpc64-unknown-linux-gnu

如果你实在不知道应当使用哪一个，那么就干脆不使用这几个选项，让 config.guess 脚本自己去猜吧，反正也挺准的。

编译选项

让我们先看看 Makefile 规则中的编译命令通常是怎么写的。

大多数软件包遵守如下约定俗成的规范：

```
#1,首先从源代码生成目标文件(预处理,编译,汇编), "-c"选项表示不执行链接步骤。
$(CC) $(CPPFLAGS) $(CFLAGS) example.c -c -o example.o
#2,然后将目标文件连接为最终的结果(连接), "-o"选项用于指定输出文件的名字。
$(CC) $(LDFLAGS) example.o -o example

# 有一些软件包一次完成四个步骤:
$(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) example.c -o example
```

当然也有少数软件包不遵守这些约定俗成的规范，比如：

```
#1,有些在命令行中漏掉应有的 Makefile 变量(注意：有些遗漏是故意的)
$(CC) $(CFLAGS) example.c -c -o example.o
$(CC) $(CPPFLAGS) example.c -c -o example.o
$(CC) example.o -o example
$(CC) example.c -o example
#2,有些在命令行中增加了不必要的 Makefile 变量
$(CC) $(CFLAGS) $(LDFLAGS) example.o -o example
$(CC) $(CPPFLAGS) $(CFLAGS) $(LDFLAGS) example.c -c -o example.o
```

当然还有极个别软件包完全是“胡来”：乱用变量(增加不必要的又漏掉了应有的)者有之，不用\$(CC)者有之，不一而足.....

尽管将源代码编译为二进制文件的四个步骤由不同的程序(cpp,gcc/g++,as,ld)完成，但是事实上 cpp, as, ld 都是由 gcc/g++ 进行间接调用的。换句话说，控制了 gcc/g++ 就等于控制了所有四个步骤。从 Makefile 规则中的编译命令可以看出，编译工具的行为全靠 CC/CXX CPPFLAGS CFLAGS/CXXFLAGS LDFLAGS 这几个变量在控制。当然理论上控制编译工具行为的还应当有 AS ASFLAGS ARFLAGS 等变量，但是实践中基本上没有软件包使用它们。

那么我们如何控制这些变量呢？一种简易的做法是首先设置与这些 Makefile 变量同名的环境变量并将它们 export 为全局，然后运行 configure 脚本，大多数 configure 脚本会使用这同名的环境变量代替 Makefile 中的值。但是少数 configure 脚本并不这样做(比如 GCC-3.4.6 和 Binutils-2.16.1 的脚本就不传递 LDFLAGS)，你必须手动编辑生成的 Makefile 文件，在其中寻找这些变量并修改它们的值，许多源码包在每个子文件夹中都有 Makefile 文件，真是一件很累人的事！

CC 与 CXX

这是 C 与 C++ 编译器命令。默认值一般是“gcc”与“g++”。这个变量本来与优化没有关系，但是有些人因为担心软件包不遵守那些约定俗成的规范，害怕自己苦心设置的 CFLAGS/CXXFLAGS/LDFLAGS 之类的变量被忽略了，而索性将原本应当放置在其它变量中的选项一股老儿塞到 CC 或 CXX 中，比如：CC=“gcc -m arch=k8 -O2 -s”。这是一种怪异的用法，本文不提倡这种做法，而是提倡按照变量本来的含义使用变量。

CPPFLAGS

这是用于预处理阶段的选项。不过能够用于此变量的选项，看不出有哪个与优化相关。如果你实在想设一个，那就使用下面这两个吧：

```
-DNDEBUG
    “NDEBUG”是一个标准的 ANSI 宏，表示不进行调试编译。
-D_FILE_OFFSET_BITS=64
    大多数包使用这个来提供大文件(◇ 2G)支持。
```

CFLAGS 与 CXXFLAGS

CFLAGS 表示用于 C 编译器的选项，CXXFLAGS 表示用于 C++ 编译器的选项。这两个变量实际上涵盖了编译和汇编两个步骤。大多数程序和库在编译时默认的优化级别是“2”(使用“-O2”选项)并且带有调试符号来编译，也就是 CFLAGS=“-O2 -g”, CXXFLAGS = \$CFLAGS 。事实上，“-O2”已经启用绝大多数安全的优化选项了。另一方面，由于大部分选项可以同时用于这两个变量，所以仅在最后讲述只能用于其中一个变量的选项。[提醒]下面所列选项皆为非默认选项，你只要按需添加即可。

先说说“-O3”在“-O2”基础上增加的几项：

- finline-functions 允许编译器选择某些简单的函数在其被调用处展开，比较安全的选项，特别是在 CPU 二级缓存较大时建议使用。
- funswitch-loops 将循环体中不改变值的变量移动到循环体之外。
- fgcse-after-reload 为了清除多余的溢出，在重载之后执行一个额外的载入消除步骤。

另外：

- fomit-frame-pointer 对于不需要栈指针的函数就不在寄存器中保存指针，因此可以忽略存储和检索地址的代码，同时对许多函数提供一个额外的寄存器。所有“-O”级别都打开它，但仅在调试器可以不依靠栈指针运行时才有效。在 AMD64 平台上此选项默认打开，但是在 x86 平台上则默认关闭。建议显式的设置它。
 - falign-functions=N
 - falign-jumps=N
 - falign-loops=N
 - falign-labels=N
- 这四个对齐选项在“-O2”中打开，其中的根据不同的平台 N 使用不同的默认值。如果你想指定不同于默认值的 N，也可以单独指定。比如，对于 L2-cache>=1M 的 cpu 而言，指定 -falign-functions=64 可能会获得更好的性能。建议在指定了 -march 的时候不明确指定这里的值。

调试选项：

- fprofile-arcs 在使用这一选项编译程序并运行它以创建包含每个代码块的执行次数的文件后，程序可以再次使用 -fbranch-probabilities 编译，文件中的信息可以用来优化那些经常选取的分支。如果没有这些信息，gcc 将猜测哪个分支将被经常运行以进行优化。这类优化信息将会存放在一个以源文件为名字的并以“.da”为后缀的文件中。

全局选项：

- pipe 在编译过程的不同阶段之间使用管道而非临时文件进行通信，可以加快编译速度。建议使用。

目录选项：

- sysroot=dir 将 dir 作为逻辑根目录。比如编译器通常会在 /usr/include 和 /usr/lib 中搜索头文件和库，使用这个选项后将在 dir /usr/include 和 dir/usr/lib 目录中搜索。如果使用这个选项的同时又使用了 -isysroot 选项，则此选项仅作用于库文件的搜索路径，而 -isysroot 选项将作用于头文件的搜索路径。这个选项与优化无关，但是在 CLFS 中有着神奇的作用。

代码生成选项：

- fno-bounds-check 关闭所有对数组访问的边界检查。该选项将提高数组索引的性能，但当超出数组边界时，可能会造成不可接受的行为。
- freg-struct-return 如果 struct 和 union 足够小就通过寄存器返回，这将提高较小结构的效率。如果不够小，无法容纳在一个寄存器中，将使用内存返回。建议仅在完全使用 GCC 编译的系统上才使用。
- fpic 生成可用于共享库的位置独立代码。所有的内部寻址均通过全局偏移表完成。要确定一个地址，需要将代码自身的内存位置作为表中一项插入。该选项产生可以在共享库中存放并从中加载的目标模块。
- fstack-check 为防止程序栈溢出而进行必要的检测，仅在多线程环境中运行时才可能需要它。
- fvisibility=hidden 设置默认的 ELF 镜像中符号的可见性为隐藏。使用这个特性可以非常充分的提高连接和加载共享库的性能，生成更加优化的代码，提供近乎完美的 API 输出和防止符号碰撞。我们强烈建议你在编译任何共享库的时候使用该选项。参见 -fvisibility-inlines-hidden 选项。

硬件体系结构相关选项[仅仅针对 x86 与 x86_64]：

- march=cpu-type 为特定的 cpu-type 编译二进制代码(不能在更低级别的 cpu 上运行)。Intel 可以用：pentium 2, pentium 3 (= pentium 3m), pentium 4 (= pentium 4m), pentium -m , prescott, nocona, core2 (GCC-4.3 新增) 。AMD 可以用：k6-2 (= k6-3), athlon (= athlon-tbird), athlon-xp (= athlon-m p), k8 (= opteron = athlon64 = athlon-fx)
- mfpmath=sse P3 和 athlon-xp 级别及以上的 cpu 支持“sse”标量浮点指令。仅建议在 P4 和 K8 以上级别的处理器上使用该选项。

`-m align-double`
将 `double`, `long double`, `long long` 对齐于双字节边界上；有助于生成更高速的代码，但是程序的尺寸会变大，并且不能与未使用该选项编译的程序一起工作。

`-m 128bit-long-double`
指定 `long double` 为 128 位，`pentium` 以上的 `cpu` 更喜欢这种标准，并且符合 `x86-64` 的 ABI 标准，但是却不附合 `i386` 的 ABI 标准。

`-m regparm = N`
指定用于传递整数参数的寄存器数目 (默认不使用寄存器)。 $0 < N \leq 3$ ；注意：当 $N > 0$ 时你必须使用同一参数重新构建所有的模块，包括所有的库。

`-m sseregparm`
使用 SSE 寄存器传递 `float` 和 `double` 参数和返回值。注意：当你使用了这个选项以后，你必须使用同一参数重新构建所有的模块，包括所有的库。

`-m mmx`
`-m sse`
`-m sse2`
`-m sse3`
`-m 3dnow`
`-m ssse3` (没写错!GCC-4.3 新增)
`-m sse4.1` (GCC-4.3 新增)
`-m sse4.2` (GCC-4.3 新增)
`-m sse4` (含 4.1 和 4.2,GCC-4.3 新增)
是否使用相应的扩展指令集以及内置函数，按照自己的 `cpu` 选择吧！

`-m accumulate-outgoing-args`
指定在函数引导段中计算输出参数所需最大空间，这在大部分现代 `cpu` 中是较快的方法；缺点是会明显增加二进制文件尺寸。

`-m threads`
支持 `Mingw32` 的线程安全异常处理。对于依赖于线程安全异常处理的程序，必须启用这个选项。使用这个选项时会定义“`-D_MT`”，它将包含使用选项“`-lm ingw thrd`”连接的一个特殊的线程辅助库，用于为每个线程清理异常处理数据。

`-m inline-all-stringops`
默认时 GCC 只将确定目的地会被对齐在至少 4 字节边界的字符串操作内联进程序代码。该选项启用更多的内联并且增加二进制文件的体积，但是可以提升依赖于高速 `memcpy`, `strlen`, `memset` 操作的程序的性能。

`-m inline-stringops-dynamically`
GCC-4.3 新增。对未知尺寸字符串的小块操作使用内联代码，而对大块操作仍然调用库函数，这是比“`-m inline-all-stringops`”更聪明的策略。决定策略的算法可以通过“`-m stringop-strategy`”控制。

`-m omit-leaf-frame-pointer`
不为叶子函数在寄存器中保存栈指针，这样可以节省寄存器，但是将会使调试变的困难。注意：不要与 `-fomit-frame-pointer` 同时使用，因为会造成代码效率低下。

`-m 64`
生成专门运行于 64 位环境的代码，不能运行于 32 位环境，仅用于 `x86_64` [含 `EMT64`] 环境。

`-m code-model=small`
[默认值]程序和它的符号必须位于 2GB 以下的地址空间。指针仍然是 64 位。程序可以静态连接也可以动态连接。仅用于 `x86_64` [含 `EMT64`] 环境。

`-m code-model=kernel`
内核运行于 2GB 地址空间之外。在编译 `linux` 内核时必须使用该选项！仅用于 `x86_64` [含 `EMT64`] 环境。

`-m code-model=medium`
程序必须位于 2GB 以下的地址空间，但是它的符号可以位于任何地址空间。程序可以静态连接也可以动态连接。注意：共享库不能使用这个选项编译！仅用于 `x86_64` [含 `EMT64`] 环境。

其它优化选项：

`-fforce-addr`
必须将地址复制到寄存器中才能对他们进行运算。由于所需地址通常在前面已经加载到寄存器中了，所以这个选项可以改进代码。

`-finline-limit=n`
对伪指令数超过 `n` 的函数，编译程序将不进行内联展开，默认为 600。增大此值将增加编译时间和编译内存用量并且生成的二进制文件体积也会变大，此值不宜太大。

`-fmerge-all-constants`
试图将跨编译单元的所有常量值和数组合并在一个副本中。但是标准 C/C++ 要求每个变量都必须有不同的存储位置，所以该选项可能会导致某些不兼容的行为。

`-fgcse-sm`
在全局公共子表达式消除之后运行存储移动，以试图将存储移出循环。`gcc-3.4` 中曾属于“-O2”级别的选项。

`-fgcse-las`
在全局公共子表达式消除之后消除多余的在存储到同一存储区域之后的加载操作。`gcc-3.4` 中曾属于“-O2”级别的选项。

`-floop-optimize`
已废除 (GCC-4.1 曾包含在“-O1”中)。

`-floop-optimize2`
使用改进版本的循环优化器代替原来“-floop-optimize”。该优化器将使用不同的选项 (`-funroll-loops`, `-fpeel-loops`, `-funswitch-loops`, `-ftree-loop-in`) 分别控制循环优化的不同方面。目前这个新版本的优化器尚在开发中，并且生成的代码质量并不比以前的版本高。已废除，仅存在于 GCC-4.1 之前的版本中。

`-funsafe-loop-optimizations`
假定循环不会溢出，并且循环的退出条件不是无穷。这将可以在一个比较广的范围内进行循环优化，即使优化器自己也不能断定这样做是否正确。

`-fsched-spec-load`
允许一些装载指令执行一些投机性的动作。

`-ftree-loop-linear`
在 `trees` 上进行线型循环转换。它能够改进缓冲性能并且允许进行更进一步的循环优化。

`-fivopts`
在 `trees` 上执行归纳变量优化。

`-ftree-vectorize`
在 `trees` 上执行循环向量化。

`-ftracer`
执行尾部复制以扩大超级块的尺寸，它简化了函数控制流，从而允许其它的优化措施做的更好。据说挺有效。

`-funroll-loops`
仅对循环次数能够在编译时或运行时确定的循环进行展开，生成的代码尺寸将变大，执行速度可能变快也可能变慢。

`-fpre-fetch-loop-arrays`
生成数组预读取指令，对于使用巨大数组的程序可以加快代码执行速度，适合数据库相关的大型软件等。具体效果如何取决于代码。

`-fweb`
建立经常使用的缓存器网络，提供更佳的缓存器使用率。`gcc-3.4` 中曾属于“-O3”级别的选项。

`-ffast-math`
违反 IEEE/ANSI 标准以提高浮点数计算速度，是个危险的选项，仅在编译不需要严格遵守 IEEE 规范且浮点计算密集的程序考虑采用。

`-fsingle-precision-constant`
将浮点常量作为单精度常量对待，而不是隐式地将其转换为双精度。

`-fb ranch-probabilities`
在使用 `-fprofile-arcs` 选项编译程序并执行它来创建包含每个代码块执行次数的文件之后，程序可以利用这一选项再次编译，文件中所产生的信息将被用来优化那些经常发生的分支代码。如果没有这些信息，`gcc` 将猜测那一支可能经常发生并进行优化。这类优化信息将会存放在一个以源文件为名字的并以“.da”为后缀的文件中。

`-frename-registers`
试图驱除代码中的假依赖关系，这个选项对具有大量寄存器的机器很有效。`gcc-3.4` 中曾属于“-O3”级别的选项。

`-fb ranch-target-load-optimize`
`-fb ranch-target-load-optimize2`
在执行序启动以及结尾之前执行分支目标缓存器加载最佳化。

`-fstack-protector`
在关键函数的堆栈中设置保护值。在返回地址和返回值之前，都将验证这个保护值。如果出现了缓冲区溢出，保护值不再匹配，程序就会退出。程序每次运行，保护值都是随机的，因此不会被远程猜出。

`-fstack-protector-all`
同上，但是在所有函数的堆栈中设置保护值。

`--param max-gcse-memory=xxM`
执行 GCSE 优化使用的最大内存量(xxM)，太小将使该优化无法进行，默认为 50M。

`--param max-gcse-passes=n`
执行 GCSE 优化的最大迭代次数，默认为 1。

传递给汇编器的选项：

`-W a,options`
`options` 是一个或多个由逗号分隔的可以传递给汇编器的选项列表。其中的每一个均可作为命令行选项传递给汇编器。

`-W a,--strip-local-absolute`
从输出符号表中移除局部绝对符号。

`-W a,-R`
合并数据段和正文段，因为不必在数据段和代码段之间转移，所以它可能会产生更短的地址移动。

`-W a,--64`
设置字长为 64bit，仅用于 x86_64，并且仅对 ELF 格式的目标文件有效。此外，还需要使用“-enable-64-bit-bfd”选项编译的 BFD 支持。

`-W a,-m arch=CPU`
按照特定的 CPU 进行优化：`pentium iii`, `pentium 4`, `prescott`, `nocona`, `core`, `core2`; `athlon`, `sledgehammer`, `opteron`, `k8`。

仅可用于 CFLAGS 的选项：

`-fhosted`
按宿主环境编译，其中需要有完整的标准库，入口必须是 `main()` 函数且具有 `int` 型的返回值。内核以外几乎所有的程序都是如此。该选项隐含设置了 `-fbuiltin`，且与 `-fno-freestanding` 等价。

`-ffreestanding`
按独立环境编译，该环境可以没有标准库，且对 `main()` 函数没有要求。最典型的例子就是操作系统内核。该选项隐含设置了 `-fno-builtin`，且与 `-fno-hosted` 等价。

仅可用于 CXXFLAGS 的选项：

`-fno-enforce-eh-specs`
C++ 标准要求强制检查异常违例，但是该选项可以关闭违例检查，从而减小生成代码的体积。该选项类似于定义了“NDEBUG”宏。

`-fno-rtti`
如果没有使用‘`dynamic_cast`’和‘`typeid`’，可以使用这个选项禁止为包含虚方法的类生成运行时表示代码，从而节约空间。此选项对于异常处理无效(仍然按需生成 `rtti` 代码)。

`-ftemplate-depth=n`
将最大模版实例化深度设为‘n’，符合标准的程序不能超过 17，默认值为 500。

`-fno-optional-diags`

禁止输出诊断消息，C++ 标准并不需要这些消息。

`-fno-thread-safe-statics`

GCC 自动在访问 C++ 局部静态变量的代码上加锁，以保证线程安全。如果你不需要线程安全，可以使用这个选项。

`-fvisibility-inlines-hidden`

默认隐藏所有内联函数，从而减小导出符号表的大小，既能缩减文件的大小，还能提高运行性能，我们强烈建议你在编译任何共享库的时候使用该选项。参见 `-fvisibility=hidden` 选项。

LD FLAGS

LD FLAGS 是传递给连接器的选项。这是一个常被忽视的变量，事实上它对优化的影响也是很明显的。

[提示]以下选项是在完整的阅读了 ld-2.18 文档之后挑选出来的选项。http://blog.chinaunix.net/u1/41220/show_art_354602.html 有 2.14 版本的中文手册。

`-s`

删除可执行程序中的所有符号表和所有重定位信息。其结果与运行命令 `strip` 所达到的效果相同，这个选项是比较安全的。

`-Wl,options`

options 是由一个或多个逗号分隔的传递给链接器的选项列表。其中的每一个选项均会作为命令行选项提供给链接器。

`-Wl,-O n`

当 $n > 0$ 时将会优化输出，但是会明显增加连接操作的时间，这个选项是比较安全的。

`-Wl,--exclude-libs=ALL`

不自动导出库中的符号，也就是默认将库中的符号隐藏。

`-Wl,-m <emulation>`

仿真 <emulation> 连接器，当前 ld 所有可用的仿真可以通过“`ld -V`”命令获取。默认值取决于 ld 的编译时配置。

`-Wl,--sort-common`

把全局公共符号按照大小排序后放到适当的输出节，以防止符号间因为排布限制而出现间隙。

`-Wl,-x`

删除所有的本地符号。

`-Wl,-X`

删除所有的临时本地符号。对于大多数目标平台，就是所有的名字以‘L’开头的本地符号。

`-Wl,-zcomberloc`

组合多个重定位节并重新排布它们，以便让动态符号可以被缓存。

`-Wl,--enable-new-dtags`

在 ELF 中创建新式的“dynamic tags”，但在老式的 ELF 系统上无法识别。

`-Wl,--as-needed`

移除不必要的符号引用，仅在实际需要的时候才连接，可以生成更高效的代码。

`-Wl,--no-define-common`

限制对普通符号的地址分配。该选项允许那些从共享库中引用的普通符号只在主程序中被分配地址。这会消除在共享库中的无用的副本的空间，同时也防止了在有多个指定了搜索路径的动态模块在进行运行时符号解析时引起的混乱。

`-Wl,--hash-style=gnu`

使用 gnu 风格的符号散列表格式。它的动态链接性能比传统的 sysv 风格(默认)有较大提升，但是它生成的可执行程序 and 库与旧的 Glibc 以及动态链接器不兼容。

最后说两个与优化无关的系统环境变量，因为会影响 GCC 编译程序的方式，下面两个是咱中国人比较关心的：

LANG

指定编译程序使用的字符集，可用于创建宽字符文件、串文字、注释；默认为英文。[目前只支持日文“C-JIS,C-SJIS,C-EUCJP”，不支持中文]

LC_ALL

指定多字节字符的字符分类，主要用于确定字符串的字符边界以及编译程序使用何种语言发出诊断消息；默认设置与 LANG 相同。中文相关的几项：“zh_CN.GB2312，zh_CN.GB18030，zh_CN.GBK，zh_CN.UTF-8，zh_TW.Big5”。