GCC学习笔记

目录

[GCC 3](#_Toc477249919)

[GCC 预编译－编译－链接－加载 3](#_Toc477249920)

[GCC Plugin 20](#_Toc477249921)

[v4.8.4：23 插件 20](#_Toc477249922)

[23.1 插件加载 20](#_Toc477249923)

[23.2 插件 API 21](#_Toc477249924)

[23.2.1 Plugin license check 21](#_Toc477249925)

[23.2.2 Plugin initialization 21](#_Toc477249926)

[23.2.3 Plugin callbacks 22](#_Toc477249927)

[23.3 Interacting with the pass manager 24](#_Toc477249928)

[23.4 Interacting with the GCC Garbage Collector 25](#_Toc477249929)

[23.5 Giving information about a plugin 25](#_Toc477249930)

[23.6 Registering custom attributes or pragmas 25](#_Toc477249931)

[23.7 Recording information about pass execution 26](#_Toc477249932)

[23.8 Controlling which passes are being run 27](#_Toc477249933)

[23.9 Keeping track of available passes 27](#_Toc477249934)

[23.10 Building GCC plugins 27](#_Toc477249935)

[GCC GENERIC 28](#_Toc477249936)

[v4.8.4： GENERIC 28](#_Toc477249937)

[1 概要 28](#_Toc477249938)

[2 类型（types） 29](#_Toc477249939)

[3 声明 32](#_Toc477249940)

[3.1使用声明：在声明上工作的宏和函数。 32](#_Toc477249941)

[3.2 内部结构：如何表示声明节点。 34](#_Toc477249942)

[4 树中的属性 35](#_Toc477249943)

[5 表达式 36](#_Toc477249944)

[5.1常量表达式 36](#_Toc477249945)

[5.2引用存储 37](#_Toc477249946)

[5.4 一元和二进制表达式 38](#_Toc477249947)

[5.5 向量 43](#_Toc477249948)

[6 语句 45](#_Toc477249949)

[6.1 基本语句 45](#_Toc477249950)

[6.2 块（Blocks） 46](#_Toc477249951)

[6.3 语句序列 46](#_Toc477249952)

[6.4 空语句 46](#_Toc477249953)

[6.5 跳转 47](#_Toc477249954)

[6.6 清除 47](#_Toc477249955)

[6.7 OpenMP 47](#_Toc477249956)

[7 函数 49](#_Toc477249957)

[7.1 函数基础 49](#_Toc477249958)

[7.2 函数属性 50](#_Toc477249959)

[8 语言依赖树 51](#_Toc477249960)

[9 C 和 C++ 树 52](#_Toc477249961)

[9.1 C ++的类型 52](#_Toc477249962)

[9.2 52](#_Toc477249963)

# GCC

GCC（GNU Compiler Collection，GNU编译器套装），是一套由 GNU 开发的编程语言编译器。它是一套以 GPL 及 LGPL 许可证所发行的自由软件，也是 GNU计划 的关键部分，亦是自由的 类Unix 及苹果计算机 Mac OS X 操作系统的标准编译器。GCC（特别是其中的C语言编译器）也常被认为是跨平台编译器的事实标准。

GCC 原名为 GNU C 语言编译器（GNU C Compiler），因为它原本只能处理 C语言。GCC 很快地扩展，变得可处理 C++。之后也变得可处理 Fortran、Pascal、Objective-C、Java, 以及 Ada 与其他语言。

gcc在后台实际上经历了预处理，汇编，编译，链接这几个过程，我们可以通过-v参数查看它的编译细节，如果想看某个具体的编译过程，则可以分别使用-E,-S,-c和 -O，对应的后台工具则分别为cpp,cc1/gcc/g++,as,ld。

符号可以分为强符号和弱符号。比如初始化了的全局变量就是强符号，未初始化的全局变量为弱符号。针对强弱符号，有如下规则：

-> 不允许强符号重定义，及不同的目标文件中不能有同名的强符号。Multiple definition错误就是违反了这种规定。

-> 如果一个符号在某个文件中是强符号，在另一个文件中是弱符号，那么选择强符号。

-> 如果一个弱符号出现在多个目标文件中，则选择其中占用空间最大的那个。

强弱符号都是针对变量定义，对于引用则无效，当然针对应用，也有强引用与弱引用。对于强引用来说，在链接成可执行文件时，如果找不到对应变量的定义则会报错。对于弱引用，在这种情况下，不会报错，链接器会采用一个默认值。

弱符号和弱引用的存在允许用户定义自己的实现去覆盖现有的实现。这样就可以更灵活的对程序进行扩充或裁减。

**Undefined reference to ：**链接时符号未定义。每个目标文件中，都可能存在一些，undefined类型的符号，这种类型的符号需要在链接时，能够在链接产生的全局符号表中找到其定义，如果找不到，链接器就会产生该错误信息。产生的原因有很多，比如少链接了某个库，符号写错了等等。

GCC 预编译－编译－链接－加载**（http://blog.csdn.net/wangbaochu/article/details/38112177）**

**1. 概述**

广义的代码编译过程，实际上应该细分为：预处理，编译，汇编，链接。

预处理过程，负责头文件展开，宏替换，条件编译的选择，删除注释等工作。gcc –E表示进行预处理。

编译过程，负载将预处理生成的文件，经过词法分析，语法分析，语义分析及优化后生成汇编文件。gcc –S表示进行编译。

汇编，是将汇编代码转换为机器可执行指令的过程。通过使用gcc –C或者as命令完成。

链接，负载根据目标文件及所需的库文件产生最终的可执行文件。链接主要解决了模块间的相互引用的问题，分为地址和空间分配，符号解析和重定位几个步骤。实际上在编译阶段生成目标文件时，会暂时搁置那些外部引用，而这些外部引用就是在链接时进行确定的。链接器在链接时，会根据符号名称去相应模块中寻找对应符号。待符号确定之后，链接器会重写之前那些未确定的符号的地址，这个过程就是重定位。

**2. 相关参数选项**

-WL：这个选项可以将指定的参数传递给链接器。

比如使用”-Wl,-soname,my-soname”，GCC会将-soname,my-soname传递给链接器，用来指定输出共享库的SO-NAME。

-shared：表示产生共享对象，产生的代码会在装载时进行重定位。但是无法做到让一份指令由多个进程共享。因为单纯的装载时重定位会对程序中所有指令和数据中的绝对地址进行修改。要做到让多个进程共享，还需要加上-fPIC。

-fPIC：地址无关代码，是为了能让多个进程共享一份指令。基本思想就是将指令中需要进行修改的那部分分离出来，跟数据放到一块。这样指令部分就可以保持不变，而需要变化的那部分则与数据一块，每个进程都有自己的一份副本。

-export-dynamic：默认情况下，链接器在产生可执行文件时，为了减少符号表大只会将那些被其他模块引用到的符号放到动态符号表。也就是说，在共享模块引用主模块时，只有那些在链接时被共享模块引用到的符号才会被导出。当程序使用dlopen()加载某个共享模块时，如果该共享模块反向引用了主模块的符号，而该符号可能在链接时因为未被其他模块引用而未被导出到动态符号表，这样反向引用就会失败。这个参数就是用来解决这个问题的。它表示，链接器在产生可执行文件时，将所有全局符号导出动态符号表。

-soname：指定输出共享库的SO-NAME。

-I：(大写的i) 指定头文件搜索路径。

-l：(小写的L) 指定链接某个库。指定链接的比如是libxxx.so.x.y.z的一个库，只需要写-lxxx即可，编译器根据当前环境，在相关路径中查找名为xxx的库。xxx又称为共享库的链接名(link name)。不同的库可能具有同样的链接名，比如动态和静态版本，libxxx.a libxxx.so。如果链接时采用-lxxx，那么链接器会根据输出文件的情况(动态/静态)选择合适的版本。比如如果ld采用了-static参数，就会使用静态版本，如果使用了-Bdynamic(这也是默认情况)，就会使用动态版本。

-L：指定链接时查找路径，多个路径用逗号分隔

-rpath：这种方式可以指定产生的目标程序的共享库查找路径。还有一个类似选项-rpath-link，与-rpath选项的区别在于，-rpath选项指定的目录被硬编码到可执行文件中，-rpath-link选项指定的目录只在链接阶段生效。这两个选项都是链接器ld的选项。更多链接器选项可以通过man ld查看。

**3. 编译与链接**

.so和.a的生成，可执行文件的生成。.a的生成只需要编译阶段，而可执行文件的生成还需要进行链接。静态库文件的生成很简单，主要就是分两步，第一步将源文件生成目标文件，可以使用gcc –c，第二步就是将目标文件打包，可以通过ar实现。所以该过程只要求源文件能够通过gcc –c这个命令即可。

共享库的生成要复杂一些。可以有三种方法生成：

$ld -G

$gcc -shared

$libtool

用ld最复杂，用gcc -shared就简单的多，但是-shared并非在任何平台都可以使用。-shared 该选项指定生成动态连接库（让连接器生成T类型的导出符号表，有时候也生成弱连接W类型的导出符号），不用该标志外部程序无法连接。GNU提供了一个更好的工具libtool，专门用来在各种平台上生成各种库。在编译生成某个.so文件时，比如liba.so，虽然它里面可能用到了libb.so的东西，但是在生成a.so时是可以不加-lb的，因为so的生成不会进行符号解析和重定位。

以GCC为例，它在编译静态库/动态库时到底使用了什么命令？比如：gcc –v -shared hello.c -o libhello.so。ld –G用来产生.so文件，也是gcc链接时实际调用的命令。

生成可执行文件时，如果链接的是静态库，那么链接器会按照静态链接规则，将对应的符号引用进行重定位。而如果是动态库，链接器会将这个符号标记为动态链接的符号，不进行重定位，而是在装载时再进行。所以，尽管是动态链接，如果是已经进入到了链接阶段，那么也需要能在相应的.so中找到某符号的定义，否则也会引发Undefined reference to的链接错误。因为链接器只有通过.so文件，才能判断某符号是个动态链接符号，所以也需要读取这些.so文件，找到相应符号的定义。

**3.1. 头文件查找**

#include有两种写法形式，分别是：

#include <> ： 直接到系统指定的某些目录中去找某些头文件。

#include “” ： 先到源文件所在文件夹去找，然后再到系统指定的某些目录中去找某些头文件。

gcc寻找头文件的路径(按照1->2->3的顺序)：

(1). 在gcc编译源文件的时候，通过参数-I指定头文件的搜索路径，如果指定路径有多个路径时，则按照指定路径的顺序搜索头文件。命令形式如：“gcc -I /path/where/theheadfile/in sourcefile.c“，这里源文件的路径可以是绝对路径，也可以是相对路径。比如设当前路径为/root/test，include\_test.c如果要包含头文件“include/include\_test.h“，有两种方法：

-> include\_test.c中#include “include/include\_test.h”或者#include "/root/test/include/include\_test.h"，然后gcc include\_test.c即可。

-> include\_test.c中#include <include\_test.h>或者#include <include\_test.h>，然后gcc –I include include\_test.c也可

(2).通过查找gcc的环境变量来搜索头文件位置，分别是:

CPATH/C\_INCLUDE\_PATH/CPLUS\_INCLUDE\_PATH/OBJC\_INCLUDE\_PATH。

(3). 再在缺省目录下搜索，分别是：

/usr/include

/usr/local/include

/usr/lib/gcc-lib/i386-linux/2.95.2/include

最后一行是gcc程序的库文件地址，各个用户的系统上可能不一样。gcc在默认情况下，都会指定到/usr/include文件夹寻找头文件。 gcc还有一个参数：-nostdinc，它使编译器不再系统缺省的头文件目录里面找头文件，一般和-I联合使用，明确限定头文件的位置。在编译驱动模块时，由于特殊的需求必须强制GCC不搜索系统默认路径，也就是不搜索/usr/include，要用参数-nostdinc，还要自己用-I参数来指定内核头文件路径，这个时候必须在Makefile中指定。

当#include使用相对路径的时候，gcc最终会根据上面这些路径，来最终构建出头文件的位置。如#include <sys/types.h>就是包含文件/usr/include/sys/types.h

**3.2. 链接时库书写顺序**

Q: 有几个库文件A.a、B.a、common.a，前两者用到了定义在后者中的例程，如果把 common.a放在前面，链接器报告存在无法解析的符号名，放在最后则无问题。

A: Floyd Davidson <floyd@ptialaska.net>

链接器按照命令行上指定顺序搜索库文件和目标文件(.a .o)，二者之间的区别在 于.o文件被全部链接进来，而只从库文件中析取所需模块，仅当某个模块可以解析当前尚未成功解析的符号时，该模块被析取后链接进来。如果库文件无法解析任何当前尚未成功解析的符号，不从中析取也不发生链接。

Unix编程新手的常见问题是数学函数并不在标准C库中，而是在libm.a中

cc -lm foo.c

这里foo.c用到了数学库中的符号，但是链接器无法正确解析。当搜索到libm.a时， 来自foo.c的数学函数符号尚未出现，因此不需要析取libm.a的任何模块。接下来 foo.o链接进来，增加了一批尚未成功解析的符号，但已经没有libm.a可供使用了， 因此数学库必须在foo.o之后被搜索到。

cc foo.c –lm

在你的问题中，如果common.a首先被搜索到，因为不匹配尚未成功解析的符号，而被丢弃。结果A.a和B.a真正链接进来的时候，已经没有库可以解析符号了。

**3.3. 链接时库文件查找**

链接时需要告诉链接器，在哪里找到库文件？以静态还是动态的方式链接库文件？默认情况下使用动态方式链接，这要求存在对应的.so动态库文件，如果不存在，则寻找相应的.a静态库文件。若在编译时向gcc传入-static选项，则使用静态方式链接，这要求所有库文件都必须有对应的\*.a静态库。

1.gcc会去找-L

2.再找gcc的环境变量LIBRARY\_PATH

3.再找内定目录 /lib /usr/lib /usr/local/lib，这是当初编译gcc时，写在文件里的

需要澄清一下：

-> ldconfig做的事情都与运行程序时有关，跟编译时一点关系都没有

-> 需要注意的是LD\_LIBRARY\_PATH通常只是在程序运行时告诉loader该去哪查找共享库，比如gcc编译时的链接器可能就不会去查找LD\_LIBRARY\_PATH。

-> 查找时如果找到了同名的动态库和静态库如何处理？当我们指定一个路径下的库文件名时，假如此时同时存在xxx.a和xxx.so的两个库形式，那么优先选择.so链接(共享库优先)。如果使用了-static，找到了.so是否会使用？如果使用了-Bdynamic，那找到了.a会不会使用？.a的生成能否依赖.so？它所依赖的这些.so能否不加入-l参数列表？

->系统中链接器与加载器的区别？编译时的链接器是ld，加载器则位于ld-linux.so。加载器是否去/etc/ld.so.conf中目录下去寻找所需的.so，还是依赖于ldconfig去更新ld.so.cache文件，？答案是依赖于ldconfig去更新cache。

-> 系统中的ld与gcc采用的链接器的区别？gcc –v查看gcc调用 as/ld 之类程序的时候传给它们的参数。通过gcc命令进行链接，与直接使用ld的区别？库查找路径是否不同？gcc的LIBRARY\_PATH，应该是gcc本身用的环境变量。

-> GCC，gcc与g++？GCC在预处理阶段会调用cpp，在编译阶段调用gcc或g++，汇编阶段调用as，最后链接阶段比较复杂，在GCC内部的这一步调用如下：

$ ld -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o

/usr/lib/crti.o /usr/lib/gcc-lib/i686/3.3.1/crtbegin.o

-L/usr/lib/gcc-lib/i686/3.3.1 hello.o -lgcc -lgcc\_eh

-lc -lgcc -lgcc\_eh /usr/lib/gcc-lib/i686/3.3.1/crtend.o

/usr/lib/crtn.o

**4. 加载**

**4.1. 可执行文件装载过程**

Linux系统通过execve()系统调用来执行程序，系统会为相应格式的文件查找合适的装载处理函数。elf格式文件的加载主要通过load\_elf\_binary()完成，有如下过程：

-> 检查文件格式有效性，比如magic number，文件头

-> 寻找动态链接的.interp段，设置动态链接器路径

-> 根据elf文件描述，对elf文件进行映射，比如代码，数据

-> 初始化elf进程环境，比如进程启动时EDX寄存器地址应该是DT\_FINI地址

-> 将系统调用的返回地址，修改成elf可执行文件入口点，该入口点取决于程序的链接方式，对于静态链接的可执行文件，这个入口点就是elf文件中e\_entry所指向的地址，对于动态链接的elf，程序入口点就是动态链接器。{如何判断elf采用的是静态链接还是动态链接的？可以通过对它执行ldd命令，如果是静态链接，会输出：statically linked}

load\_elf\_binary()执行完毕后，可执行文件就被装入了内存，接下来就可以执行了。

**4.2. 动态库加载**

如上，对于动态链接的可执行文件在真正执行前，实际上它的很多外部符号还处于无效状态，还未与实际的so文件联系起来，因此还有一个动态链接过程。操作系统会首先加载动态链接器ld.so(/lib/ld-linux.so.2)，加载完这个so后，系统就会把控制权交给它，然后它会进行一些初始化操作，根据当前环境参数对可执行文件进行动态链接工作。动态链接器会寻找所需要的.so文件并进行装载，然后进行符号查找及重定位。如果找不到所需要的符号定义就会产生“undefined symbol”错误。

在elf文件中有一个.interp段，该段指定了所要采用的动态链接器路径。操作系统会根据该段内容，选择加载的动态链接器。可以通过objdump –s a.out查看该段内容，也可以如下命令来查看：readelf –l a.out|grep interpreter。

.dynamic段，保存了动态链接器所需的基本信息，比如依赖于哪些共享对象，动态链接符号表的位置，动态链接重定位表的位置，共享对象初始化代码的位置。可以通过readelf –d a.out来查看该段内容，其中比较重要的有如下几个：

DT\_RPATH：.so搜索路径

DT\_INIT：初始化代码地址

DT\_FINIT：结束代码地址

DT\_NEED：依赖的.so

.dynsym段，动态符号表段。可以通过readelf –sD \*.so来查看如何确定所需要的加载库、如何解析符号、重名的文件如何处理。

**4.2.1. 加载过程**

由于动态链接器本身也是一个.so，因此首先要完成自己的重定位过程，称为“自举(bootstrap)”。

完成自举之后，动态链接器会将可执行文件和它本身的符号表合并到到一个全局符号表中。然后链接器开始查找所需要的.so，在前面提到的.dynamic段中有一个DT\_NEEDED，它指出了可执行文件所依赖的.so，据此链接器就可以找到所需要的.so集合，然后开始装载该.so，完成后再从它需要的.so集合中取出一个.so，如果在此时找不到某个.so文件就会产生“cannot open shared object file”错误。如果找到相应文件，就会打开它，读取相应的elf文件头和.dynamic段，然后将其代码段和数据段进行映射。如果该.so还依赖于其他.so，就会把其他.so加入到待装载集合中，实际上是一个图的遍历问题，上面的过程实际上就是广度优先遍历。

当新对象被加载之后，它的符号表会被合并到全局符号表中，所以当所有对象加载之后，全局符号表中应该包含了进程进行动态链接所需要的所有符号。这个过程里面存在一个称为“全局符号介入(global symbol interpose)”问题，假设a.out依赖了c.so和d.so，而c.so依赖于a.so，d.so依赖于b.so，但是a.so和b.so存在一个同名的符号，此时编译链接无法发现这个同名的符号，因为编译a.out时只需要链接c.so和d.so即可，这样链接器无法发现在a.so和b.so中存在同名的符号，这样只能加载时解决，而动态链接器有这样的一个规则：当一个符号需要加入全局符号表时，如果相同的符号名已经存在，那么后加入的符号会被忽略。所以，对于这种情况要格外注意。

上面步骤完成后，就会开始进行符号解析和重定位，链接器会开始遍历可执行文件和每个.so的重定位表，将它们的GOT/PLT中每个需要重定位的位置进行修正。此时，如果某些符号还是无法解析，就会报出“undefined symbol”错误。

重定位完成之后，如果.so中有“.init”段，动态链接器会执行这里的代码，以实现初始化过程，比如静态/全局对象的初始化。相应的也会有“.finit”来进行退出操作。可执行文件本身的.init，不会由动态链接器执行，而是通过程序自身的初始化代码完成。

重定位和初始化完成后，动态链接器所要做的事情就完成了，之后就会把控制权交给可执行程序的入口，开始执行程序。

**4.3. 显式运行时链接**

上面的加载过程是在程序启动时由系统完成的，对于程序本身是透明的。如果程序想在运行时自行加载某个动态库，实现类似插件之类的机制，则需要使用如下几个函数。

**4.3.1. dlopen()**

用于打开一个动态库，将其加载到进程的地址空间，完成初始化过程。C语言原形是：

void \* dlopen(const char \*filename, int flag);

如果文件名filename是以“/”开头，也就是使用绝对路径，那么dlopne就直接使用它，而不去查找某些环境变量或者系统设置的函数库所在的目录了。否则dlopen()就会按照下面的次序查找函数库文件：

1. 环境变量LD\_LIBRARY指明的路径。

2. /etc/ld.so.cache中的函数库列表。

3./lib目录，然后/usr/lib。不过一些很老的a.out的loader则是采用相反的次序，也就是先查 /usr/lib，然后是/lib。

同时如果filename值设为NULL的话，那么dlopen将返回全局符号表的句柄。也就是说可以在运行时找到全局符号表里的任何一个符号，并可以执行它们。全局符号表中包括了可执行程序本身，被动态链接器加载到进程中的所有动态库以及通过dlopen打开并使用了RTLD\_GLOBAL方式的模块中的符号。

dlopen()函数中，参数flag的值必须是RTLD\_LAZY或者RTLD\_NOW，RTLD\_LAZY的意思是resolve undefined symbols as code from the dynamic library is executed，而RTLD\_NOW的含义是resolve all undefined symbols before dlopen() returns and fail if this cannot be done'。如果有任何未定义的符号引用的绑定工作无法完成，那么dlopen就会返回错误，这两种方式必须二选一。此外还有RTLD\_GLOBAL，可以与上面两种方式之一一起使用，表示将被加载的模块的全局符号合并到进程的全局符号表中，这样以后加载的模块就可以使用这些符号。

如果有好几个函数库，它们之间有一些依赖关系的话，例如X依赖Y，那么你就要先加载那些被依赖的函数。例如先加载Y，然后加载X。dlopen()函数的返回值是一个句柄，然后后面的函数就通过使用这个句柄来做进一步的操作。如果打开失败dlopen()就返回一个NULL。如果一个函数库被多次打开，它会返回同样的句柄。

同时dlopen还会执行被加载模块的.init段来进行模块的初始化操作。

**4.3.2. dlsym()**

这个函数在一个已经打开的函数库里面查找给定的符号。这个函数如下定义：

void \* dlsym(void \*handle, char \*symbol);

函数中的参数handle就是由dlopen打开后返回的句柄，symbol是一个以’\0’结尾的字符串。如果dlsym()函数没有找到需要查找的symbol，则返回NULL。dlsym的返回值对于不同的符号类型具有不同的意义，如果符号是个函数，则返回的就是函数地址，如果是变量，就是变量的地址，如果是常量就是该常量的值。如果该常量值刚好是NULL，那如何区分它是没有找到该常量，还是该常量就是NULL呢？这就需要看dlerror()，如果符号找到了dlerror()就会返回NULL，否则会返回相应的错误信息。

符号优先级，前面已经提过，在加载过程中如果发现符号名冲突，先载入的符号会优先，这种优先级方式成为装载序列(load ordering)。进程通过dlopen载入对象时，动态链接器在进行符号解析和重定位时，都是采用装载序列。

但是在使用dlsym进行符号地址查找时，如果是在全局符号表中进行查找，即在dlopen时，参数filename为NULL，由于全局符号表采用的是装载序列，因此dlsym使用的也是装载序列。但是，如果是通过某个dlopen打开的共享对象进行符号查找的话，采用的是一种称为依赖序列(dependency ordering)的优先级方式。即它会以被打开的那个共享对象为根节点，进行广度优先遍历，直到找到该符号。{!因为如果dlopen调用时，未采用RTLD\_GLOBAL方式的话，那么被打开对象的符号表不会加入全局符号表，这样它和它依赖的模块可能就会都有自己的符号表，而没有一个全局的符号表}

**4.3.3. dlerror()**

通过调用dlerror()函数，我们可以获得最后一次调用dlopen()，dlsym()，或者dlclose()的错误信息。 它的返回值类型是char \*，如果返回NULL，表示上一次调用成功，如果不是NULL，则代表相应的错误信息。

**4.3.4. dlclose()**

dlopen()函数的反过程就是dlclose()，dlclose()用于将已经加载的模块卸载。系统会维护一个加载引用计数器，当调用dlclose的时候，就把这个计数器的计数减一，如果计数器为0，则真正的释放掉。真正释放的时候，如果函数库里面有\_fini()这个函数，则自动调用\_fini()这个函数，做一些必要的处理。dlclose()返回0表示成功，非0值表示错误。

**4.4. 共享库版本命名规则**

**4.4.1. Libname.x.y.z**

X代表主版本号，y代表次版本号，z代表发布版本号。主版本号代表库的重大升级，不同主版本号的库之间是不兼容的。次版本号代表了库的增量升级，即增加一些新的符号接口，且保持原来的符号不变，具有向后兼容性，即依赖于旧版本库的程序在新版本库上也可以运行。发布版本号，代表了库的一些bug fix和性能优化，不添加任何新的接口，也不对现有接口进行修改。

**4.4.2. So-Name**

由上可知主版本号和此版本号实际上决定了程序的接口。通常程序中都会包含了所依赖的库的名称和主版本号，系统是通过一种称为SO-NAME的命名机制来记录共享库的依赖关系。每个共享库都有一个对应的SO-NAME，即名称和主版本号。在linux系统中，会为每个共享库在它所在的目录创建一个跟SO-NAME相同的并且指向它的软连接。比如系统中有一个/lib/liba.1.1.2那么系统的共享库管理程序就会创建一个/lib/liba.1的软连接。

SO-NAME会指向系统中相同主版本号的最新版的那个库，使用SO-NAME的目的在让所有依赖于某个共享库的模块，在编译，链接和运行时，都使用SO-NAME，而不使用详细的版本号。前面提过，在.dynamic段中的DT\_NEEDED描述了模块所依赖的共享库，为了保证兼容性，采用SO-NAME进行.dynamic段的依赖描述。通过readelf –d liba.so，可以查看到elf文件的DT\_NEEDED部分。

**4.5. 系统配置**

ldconfig，当系统安装或更新一个共享库就需要运行这个工具，它会遍历所有的默认共享库目录，比如/lib /usr/lib，然后更新所有的软连接，使它们指向最新版的共享库；如果是新安装的共享库，则会为它创建软连接。

**4.6. 运行时库查找过程**

动态链接器对保存在.dynamic段的DT\_NEEDED的共享库的查找有一定的规则，如果DT\_NEEDED保存的是绝对路径，那么动态链接器就按这个路径去查找，如果是相对路径，那么动态链接器会在/lib /usr/lib和/etc/ld.so.conf配置文件指定的目录中进行查找。

ld.so.conf是一个配置文件，可能包含其他的配置文件，存放了路径信息。为了避免每次查找共享库都去遍历这些目录，linux系统提供了ldconfig。ldconfig除了负责每个共享库目录下的共享库SO-NAME的创建删除和更新外，它还会将这些SO-NAME收集起来存放到/etc/ld.so.cache中。当查找共享库时，直接去/etc/ld.so.cache里找即可，而它的结构则是专为查找优化过的。如果在这里没有找到，它会继续查找/lib和/usr/lib，如果还未找到就宣告失败。

所以，如果在系统指定的共享库目录下，添加删除或更新任何一个共享库，或者更改了/etc/ld.so.conf的配置，都应该运行ldconfig这个程序。以便调整SO-NAME和/etc/ld.so.cache，而很多安装软件包实际在往系统安装共享库之后都会调用ldconfig。

此外，还可以通过LD\_LIBRARY\_PATH来改变库搜索路径。如果为进程设置了LD\_LIBRARY\_PATH，那么在启动时，动态链接器会首先在LD\_LIBRARY\_PATH指定的目录下查找。但是，有不少声音主张要避免使用 LD\_LIBRARY\_PATH 变量，尤其是作为全局变量。这些声音是：

\* LD\_LIBRARY\_PATH is not the answer - http://prefetch.net/articles/linkers.badldlibrary.html

\* Why LD\_LIBRARY\_PATH is bad - http://xahlee.org/UnixResource\_dir/\_/ldpath.html

\* LD\_LIBRARY\_PATH - just say no - http://blogs.sun.com/rie/date/20040710

综上，动态链接器会按照如下顺序加载或查找共享库：

-> 链接时由-rpath选项指定的目录(已被硬编码到可执行文件中)

-> LD\_LIBRARY\_PATH指定的目录

-> 路径缓存文件/etc/ld.so.cache指定的路径

-> 默认共享库目录，先/usr/lib，后/lib

LD\_PRELOAD，可以预先装载一些共享库。由LD\_PRELOAD指定的文件，会在动态链接库按指定规则搜索共享库之前加载。比LD\_LIBRARY\_PATH指定的还要优先，同时无论程序是否依赖于它，LD\_PRELOAD指定的共享库都会被装载。由于全局符号介入这一机制，LD\_PRELOAD指定的库的符号会覆盖后面加载的库的同名符号。这就使得我们可以修改标准c库的某些或某几个函数，而不影响其他使用。比如我们可以在实验程序执行时通过设置LD\_PRELOAD，来让系统优先加载我们修改后的库。

**4.7. 常见错误**

**4.7.1. undefined symbol**

我们将函数和变量称为符号，函数名和变量名就是符号名。每个目标文件都有一个符号表。表中记录了所用到的所有符号。通常，链接过程只关注全局性的符号，即那些本模块引用自别处的符号，及可能被别处引用的在本模块定义的那些。

可以使用readelf，objdump，nm来查看符号。此外使用c++filt工具可以查看被编译器改名(为支持重载及名字空间等机制)后的符号对应的原始名称。

动态加载时，如果找不到某个符号引用的定义，就会产生该错误。通常是该符号所在的动态库未被加载，也就是说DT\_NEEED缺少了某个.so。解决方式就是在链接程序时，使用-l指定所需要的库。

**4.7.2. Double free or corruption**

原因可能有多种：比如malloc了一段内存空间，但是写的时候越界了，这实际上会导致corruption；链接了同一个库的动态和静态两个版本，且该库内具有全局或静态变量；不同的库内含有相同的全局或静态变量。这里主要关注下由于链接多个具有同名变量的库的情况。

在通常情况下，共享库都是通过使用附加选项 -fpic 或 -fPIC 进行编译，从目标代码产生位置无关的代码（Position Independent Code，PIC），使用 -shared选项将目标代码放进共享目标库中。位置无关代码需要能够被加载到不同进程的不同地址，并且能得以正确的执行，故其代码要经过特别的编译处理：位置无关代码（PIC）对常量和函数入口地址的操作都是采用基于基寄存器（base register）BASE+ 偏移量的相对地址的寻址方式。即使程序被装载到内存中的不同地址，即 BASE 值不同，而偏移量是不变的，所以程序仍然可以找到正确的入口地址或者常量。

然而，当应用程序链接了多个共享库，如果在这些共享库中，存在相同作用域范围的同名静态成员变量或者同名 ( 非静态 ) 全局变量，那么当程序访问完静态成员变量或全局变量结束析构时，由于某内存块的 double free 会导致 core dump，这是由于 Linux 编译器的缺陷造成的。

链接时符号查找原理如下：

1、应用程序在链接阶段时，会顺序生成符号表。也就是说，在应用程序中涉及到的符号，会在链接文件中逐个顺次查找

2、一旦查找到符号，就停止本符号的查找工作，转向第二个符号的查找

3、如果没有用到.a里的符号，即查找的过程中没有涉及到该.a，则不会在程序中链接该.a

4、对于.so，无论是否涉及到符号查找，均会进行加载

5、so的加载和卸载会涉及到自身内存分配和释放，而.a不会（.a相当于.o的集合，.o直接静态编译到应用程序，成为程序一部分）

6、.a和.o有不同，.a是.o的集合，但是，.o必定会加载，.a不一定会加载（只加载符号表相关的.o）

这样，对于有不同库的同名全局变量，只会产生一个符号，但是由于.so本身在卸载的时候会对全局变量进行析构，同时如果多个共享库，或者程序本身具有该全局变量，这样就会出现重复free的情况，导致double free错误。

解决方法：使用选项-fpie或-fPIE，此时生成的共享库不会为静态成员变量或全局变量在GOT中创建对应的条目（通过objdump或readelf命令可以查看），从而避免了由于静态对象“在同一地址构造两次，析构两次”而对同一内存区域释放两次引起的程序 core dump(重复构造没有问题，但是重复析构会导致double free，但是如果构造函数有内存分配动作，是否会导致内存泄露？)。

选项-fpie和-fPIE与-fpic及-fPIC的用法很相似，区别在于前者总是将生成的位置无关代码看作是属于程序本身，并直接链接进该可执行程序，而非存入全局偏移表GOT中；这样，对于同名的静态或全局对象的访问，其构造与析构操作将保持一一对应。

还有如果具有同名变量的库一个是.so和一个是.a，将.so放到前面，也可以避免这个问题，根据上面的链接规则，当在.so中找到该符号时，那么.a中的内容就不会被链接了。更详细内容可以参考这两篇文章：库冲突 技巧：多共享动态库中同名对象重复析构问题的解决方法。

**4.7.3. LD\_DEBUG**

LD\_DEBUG这个环境变量可以打开动态库的调试功能，输出很多信息，对于开发调试动态库很有帮助。比如设置LD\_DEBUG=files，就可以看到整个装载过程。此外它还可以设置为：

-> bindings：显示动态链接的符号绑定过程

-> libs：显示共享库的查找过程

-> versions：显示符号的版本依赖关系

-> reloc：显示重定位过程

-> symbols：显示符号表查找过程

-> statistics：显示动态链接过程中的各种统计信息

-> all：显示以上所有信息

-> help：显示帮助信息

**5. 相关工具**

**5.1. file**

file程序是用来判断文件类型的，比如可以用命令file xxx.tar.tar看一下文件类型，然后用tar加适当的参数解压。在这里主要是可以通过file来查看某文件是否链接了动态库。

$ file a.out

a.out: ELF 32-bit LSB executable, Intel 80386,

version 1 (SYSV), dynamically linked (uses shared

libs), not stripped

最后一个not stripped表示该文件含符号表(可以利用命令strip去除符号表)

**5.2. nm**

用于列出目标文件的符号。可以通过-C选项，来显示符号的可读形式，即未被mangle的形式。-D可以用来显示动态符号。其中U类型的符号，代表该目标文件中未定义的那些symbol，而这些symbol通常都是定义在其他文件中，T表示该symbol在此文件中有定义。所以，nm最常用的地方在于，查看这个文件中是否包含某函数的定义，包含哪些未定义符号，因此在产生链接问题时，通常需要对U和T类型格外关注

**5.3. c++filt**

c++filt可以将mangle的符号进行还原。

**5.4. ldd**

ldd可以用来查看.so所依赖的共享库文件列表及未找到的.so，ldd –r还会报告缺少的目标对象和函数。

ldd实际上是个脚本，能够显示可执行模块的dependency，其原理是通过设置一系列的环境变量，如下：LD\_TRACE\_LOADED\_OBJECTS、LD\_WARN、LD\_BIND\_NOW、LD\_LIBRARY\_VERSION、LD\_VERBOSE等。当LD\_TRACE\_LOADED\_OBJECTS环境变量不为空时，任何可执行程序在运行时，它都会只显示模块的dependency，而程序并不真正执行。

ldd显示可执行模块的dependency的工作原理，其实质是通过ld-linux.so(elf动态库的装载器)来实现的。我们知道，ld-linux.so模块会先于executable模块程序工作，并获得控制权，因此当上述的那些环境变量被设置时，ld-linux.so选择了显示可执行模块的dependency。

**5.5. readelf**

它是专门针对 ELF 文件格式的解析器，但是它并不提供反汇编功能，可以通过objdump进行反汇编。readelf可以用来查看头信息，符号信息，动态重定位信息等elf内部各个部分。

**5.6. ldconfig**

ldconfig是一个动态链接库管理命令。为了让动态链接库为系统所共享,还需运行动态链接库的管理命令—ldconfig ldconfig命令的用途,主要是在默认搜寻目录(/lib和/usr/lib)以及动态库配置文件/etc/ld.so.conf内所列的目录下,搜索出可共享的动态链接库(格式如前介绍,lib\*.so\*),进而创建出动态装入程序(ld.so)所需的连接和缓存文件.缓存文件默认为/etc/ld.so.cache,此文件保存已排好序的动态链接库名字列表。

**5.7. strip**

可以用来清理共享库和可执行文件中的符号信息和调试信息。也可以通过ld –s或ld –S，二者区别是：-S消除调试符号信息，-s消除所有符号信息。也可以通过gcc参数”-Wl,-s”或”-Wl,-S”来向ld传递这两个参数。

**6. 问题分析**

**6.1. 分析步骤**

无论是编译，链接还是加载，本质上都需要去寻找某些符号的定义。当找不到对应的符号时通常都会产生某些错误，类似于上面提到的那些。

实际上基本上都需要确定以下问题。需要哪些符号的定义？去哪里寻找这些符号的定义？找到的符号定义是否满足要求？

具体来说，比如动态库加载时，需要确定需要加载哪些.so？去哪找到这些.so？这些.so中是否包含了所有的未定义符号？加载哪些.so，是由可执行文件及各个.so的.dynamic段中的DT\_NEEDED决定的。去哪寻找这些.so，则是由各种路径确定的。如果出现.so文件找不到或符号未定义或版本不正确之类的动态库相关问题时，通常的分析步骤如下：

-> 查看-l参数，确定可执行程序已经链接了所需的库文件

-> 使用find，在当前系统中查找，看是否能够找到该库文件

-> 确定系统当前的库搜索路径

-> 如果可以找到该库文件，则需要确定它所在目录是否已经在库搜索路径中，如果不在则将它加入合适路径

-> 如果它刚好也在库搜索路径中，需要确定路径中是否还有同名的库文件

-> 如果它是唯一的存在于库搜索路径中的库文件，则详细分析该文件：使用ldd查看该.so的依赖性, 使用nm查看其符号表，找到未定义符号，nm –a \*.so|grep U

->如果还找不到问题所在，设置LD\_DEBUG环境变量，进行调试

此外，如果是诡异的编译问题，还可以利用gcc的-E,-S,-c和 –O分阶段进行，查看每个阶段的输出是否正常。同时为每个命令加上-v，查看实际执行的命令，同时还可以看到头文件及库文件的搜索路径。比如可能因为宏替换，导致某些用户定义的变量被其他的一些宏定义替换，或者因为头文件搜索路径顺序，导致找到了一个错误的头文件。

**6.2. 实例分析**

最近碰到的一个问题是：调用dlopen时产生undefined symbol，采用了RTLD\_NOW方式。对于undefined symbol：

类似上面的步骤，使用ldd查看它现在依赖的.so，使用nm找到那个未定义的symbol，如果是c++程序，使用c++filt将它转换为可读形式，确认该symbol定义所在的库，然后判断该库是否已经在加载列表中，或者库的版本是否正确。

最终确定原因是：我们的可执行程序，本身会调用dlopen去在运行时加载某些.so，但是这些.so文件在生成时的-l参数，没有加入那些该.so所依赖的其他.so。这样该.so文件的DT\_NEEDED列表就是不完整的，这样加载器就不会去加载它们，这样某些符号就找不到它们的定义。

解决方案有两个：一个是保持现有.so不变，我们在编译可执行文件的-l参数中加入这些必需的库的。另一个是修改现有.so，在-l中加入它所依赖的那些库。第一个方案，只需要修改可执行文件即可，但是这种方式不是一种彻底的解决方案，因为比如该.so文件被其他程序使用时，仍然会产生这种问题，因此根本的解决方案是，.so文件本身在编译时就要在-l参数中写入它所依赖的那些库。

**6.3. 编译链接和加载实验**

**6.3.1. 源代码**

//test.h

void test();

//test.c

#include <iostream>

using namespace std;

void test()

{

cout << “test” << endl;

}

//main.c

#include “test.h”

int main()

{

test();

return 0;

}

**6.3.2. 创建静态库**

gcc –v -c test.c; ar r libtest.a test.o

[admin@clu01-gala16.dev.sd.aliyun.com]$gcc -v -c test.cpp

Using built-in specs.

Target: x86\_64-redhat-linux

Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --enable-\_\_cxa\_atexit --disable-libunwind-exceptions --enable-libgcj-multifile --enable-languages=c,c++,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --disable-dssi --enable-plugin --with-java-home=/usr/lib/jvm/java-1.4.2-gcj-1.4.2.0/jre --with-cpu=generic --host=x86\_64-redhat-linux

可以看到加上-v之后，就能看到各个步骤的具体命令，还能看到头文件搜索路径。

**6.3.3. 创建动态库**

gcc -shared test.cpp -o libtest.so –fPIC -v

[admin@clu01-gala16.dev.sd.aliyun.com]$gcc -shared test.cpp -o libtest.so -fPIC -v

**6.3.4. 创建可执行文件**

LIBRARY\_PATH=.;export LIBRARY\_PATH

g++ main.cpp -ltest

gcc头文件搜索路径

-iquote用于搜索"#include "file""形式的头文件

-I

C\_PATH：类似于-I但优先级在-I之后，可以用于任何类型语言的预处理(比如c,c++)

-isystem

C\_INCLUDE\_PATH：c语言的，类似于-isystem，但优先级在-isystem之后

CPLUS\_INCLUDE\_PATH：c++的

OBJC\_INCLUDE\_PATH：Objective-C的

gcc库文件搜索路径

-L

LIBRARY\_PATH：可以用来指定库文件搜索路径，但是优先级在-L之后

LD\_LIBRARY\_PATH：对于gcc编译不起作用，只与加载有关

测试-static，如果无.a，有.so是否可以？答案是必须是.a的库，否则不行

测试-Bdynamic，如果无.so，有.a是否可以？答案是可以

**6.3.5. 加载测试**

测试-rpath：g++ main.cpp -ltest -Wl,-rpath=. ，-rpath只对加载起作用，对链接无作用，通过它可以把运行时需要的动态库绝对路径写在可执行文件里

测试LD\_PRELOAD：在加载阶段器作用，无论可执行文件链不链接，加载器都会加载它

测试LD\_LIBRARY\_PATH：在加载阶段起作用

测试ldconfig：可以直接对当前路径应用ldconfig，这样也可以将其加入

测试/etc/ld.so.conf：直接只将路径添加到该文件，不起作用，必须执行ldconfig

测试/etc/ld.so.cache：加载时会直接从该处查找

**参考资料**

[GCC编译的背后( 预处理和编译 汇编和链接 )](http://techbbs.zol.com.cn/1/10_2633.html)

[An Introduction to GCC 学习笔记](http://www.yuanma.org/data/2006/0311/article_113.htm)

[LINUX下如何用GCC编译动态库](http://www.linuxdiyf.com/viewarticle.php?id=181828)

[gcc生成静态库和动态库](http://www.uplook.cn/kbase-Index-show-view3113.html)

[GCC编译优化指南](http://lamp.linux.gov.cn/Linux/optimize_guide.html)

[深入理解软件包的配置、编译与安装](http://www.linuxsir.org/bbs/showthread.php?t=281337)

http://www.gentoo.org/proj/en/base/amd64/howtos/fpic.xml

http://blog.csdn[.NET](http://lib.csdn.net/base/dotnet)/stephen\_yin/article/details/7762069

【用于代码复杂度评估的[CYC\_COMPLEXITY](http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/diff/scripts/gcc-plugins/cyc_complexity_plugin.c?id=f716a85cd6045c994011268223706642cff7e485)和用于追踪代码调用的[SANCOV](http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/diff/scripts/gcc-plugins/sancov_plugin.c?id=f716a85cd6045c994011268223706642cff7e485)】

# GCC Plugin

参考链接 <https://gcc.gnu.org/wiki/plugins/>，此页面有gcc内部文档的链接以及其他参考链接

## v4.8.4：23 插件

GCC插件是可为编译器提供额外功能的可加载模块。像GCC本身一样，它们可以以源代码和二进制形式分布。

GCC插件为开发人员提供了GCC API的一个丰富的子集，以允许他们扩展GCC。无论是编写额外的优化过程，转换代码还是分析信息，插件都非常有用。

* [Plugins loading](https://gcc.gnu.org/onlinedocs/gccint/Plugins-loading.html#Plugins-loading)：我们如何加载插件。
* [Plugin API](https://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API)：插件的API。
* [Plugins pass](https://gcc.gnu.org/onlinedocs/gccint/Plugins-pass.html#Plugins-pass)：插件如何与pass管理器交互。
* [Plugins GC](https://gcc.gnu.org/onlinedocs/gccint/Plugins-GC.html#Plugins-GC)：插件如何与GCC垃圾收集器交互。
* [Plugins description](https://gcc.gnu.org/onlinedocs/gccint/Plugins-description.html#Plugins-description)：提供有关插件本身的信息。
* [Plugins attr](https://gcc.gnu.org/onlinedocs/gccint/Plugins-attr.html#Plugins-attr)：注册自定义属性或编译指示。
* [Plugins recording](https://gcc.gnu.org/onlinedocs/gccint/Plugins-recording.html#Plugins-recording)：记录关于pass执行的信息。
* [Plugins gate](https://gcc.gnu.org/onlinedocs/gccint/Plugins-gate.html#Plugins-gate)：控制正在运行的通过。
* [Plugins tracking](https://gcc.gnu.org/onlinedocs/gccint/Plugins-tracking.html#Plugins-tracking)：跟踪可用通行证。
* [Plugins building](https://gcc.gnu.org/onlinedocs/gccint/Plugins-building.html#Plugins-building)：我们如何构建一个插件。

### 23.1 插件加载

支持-ldl -rdynamic的平台支持插件。它们由编译器使用dlopen加载，并在编译过程中的预定位置被调用。

插件加载命令：

-fplugin=/path/to/name.so -fplugin-arg-name-key1[=value1]

插件参数由GCC解析，并作为键值被传递到相应的插件。可以通过指定多个-fplugin参数来调用多个插件。

一个插件可以简单地给出它的短名称（没有点或斜杠）。当简单地传递-fplugin = name时，插件从plugin目录加载，所以-fplugin = name与-fplugin =`gcc -print-file-name = plugin` / name.so相同，使用反引号shell语法 查询plugin目录。

### 23.2 插件 API

插件由编译器在gcc-plugin.h中定义的特定事件中激活。对于每个感兴趣的事件，插件应调用register\_callback，指定事件的名称和处理该事件的回调函数的地址。

gcc-plugin.h头文件必须是要包含的第一个gcc头文件。

#### 23.2.1 Plugin license check

每个插件都应该定义全局符号plugin\_is\_GPL\_compatible来声明它已经根据GPL兼容许可证获得许可。如果此符号不存在，编译器将发出致命错误并退出并显示错误消息：

fatal error: plugin name is not licensed under a GPL-compatible license

name: undefined symbol: plugin\_is\_GPL\_compatible

compilation terminated

符号的声明类型应该是int，以匹配gcc-plugin.h中抑制C ++整理的向前声明。但它不需要在任何分配的部分，编译器仅仅断言该符号存在于全局范围中。这样的事情就像这样就足够了：

int plugin\_is\_GPL\_compatible;

#### 23.2.2 Plugin initialization

每个插件应该导出一个称为plugin\_init的函数，在插件加载后调用。此函数负责注册插件所需的所有回调，并执行任何其他必需的初始化。

此函数在调用解析器之前从compile\_file调用。plugin\_init的参数是：

* plugin\_info: 插件调用信息。
* version: GCC 版本。

plugin\_info结构体定义如下：

struct plugin\_name\_args

{

char \*base\_name; /\* Short name of the plugin

(filename without .so suffix). \*/

const char \*full\_name; /\* Path to the plugin as specified with

-fplugin=. \*/

int argc; /\* Number of arguments specified with

-fplugin-arg-.... \*/

struct plugin\_argument \*argv; /\* Array of ARGC key-value pairs. \*/

const char \*version; /\* Version string provided by plugin. \*/

const char \*help; /\* Help string provided by plugin. \*/

}

如果初始化失败，plugin\_init必须返回非零值。否则，它应该返回0。

加载插件的GCC编译器的版本由以下结构描述：

struct plugin\_gcc\_version

{

const char \*basever;

const char \*datestamp;

const char \*devphase;

const char \*revision;

const char \*configuration\_arguments;

};

函数plugin\_default\_version\_check使用两个指针获取这个结构体，并逐个字段进行比较。它可以由插件的plugin\_init函数使用。

用于编译插件的GCC版本可以在标题plugin-version.h中定义的符号gcc\_version中找到。 推荐的版本检查执行看起来像

#include "plugin-version.h"

...

int plugin\_init (struct plugin\_name\_args \*plugin\_info,

struct plugin\_gcc\_version \*version)

{

if (!plugin\_default\_version\_check (version, &gcc\_version))

return 1;

}

但如果希望执行较不严格的检查，也可以检查单个字段。

#### 23.2.3 Plugin callbacks

回调函数有以下原型：

/\* 插件回调函数的原型。

gcc\_data - 由GCC提供的事件特定数据。

user\_data - 插件提供的插件特定数据。 \*/

typedef void (\*plugin\_callback\_func)(void \*gcc\_data, void \*user\_data);

回调可以在以下预定事件中调用：

enum plugin\_event

{

PLUGIN\_START\_PARSE\_FUNCTION, /\* 在解析函数体之前调用。\*/

PLUGIN\_FINISH\_PARSE\_FUNCTION, /\* 完成解析一个函数后。\*/

PLUGIN\_PASS\_MANAGER\_SETUP, /\* 挂接到过程（pass）管理。\*/

PLUGIN\_FINISH\_TYPE, /\* 完成解析类型后。\*/

PLUGIN\_FINISH\_DECL, /\* 完成解析声明后。\*/

PLUGIN\_FINISH\_UNIT, /\* 用于汇总处理。 \*/

PLUGIN\_PRE\_GENERICIZE, /\*允许在C和C ++前端看到低级别的AST\*/

PLUGIN\_FINISH, /\* 在GCC退出前调用。\*/

PLUGIN\_INFO, /\* 有关插件的信息。\*/

PLUGIN\_GGC\_START, /\* 在GCC垃圾收集开始时调用。\*/

PLUGIN\_GGC\_MARKING, /\* 扩展GGC标记。\*/

PLUGIN\_GGC\_END, /\* 在GGC结束时调用。\*/

PLUGIN\_REGISTER\_GGC\_ROOTS, /\* 注册一个额外的GGC根表。 \*/

PLUGIN\_ATTRIBUTES, /\* 在属性注册期间调用 \*/

PLUGIN\_START\_UNIT, /\* 在处理翻译单元之前调用。\*/

PLUGIN\_PRAGMAS, /\* 在pragma注册期间调用。\*/

PLUGIN\_ALL\_PASSES\_START, /\* 在从all\_passes首次pass之前调用。\*/

PLUGIN\_ALL\_PASSES\_END, /\* 最后一次pass后从all\_passes调用。\*/

PLUGIN\_ALL\_IPA\_PASSES\_START, /\* 在第一次ipa通过之前调用。\*/

PLUGIN\_ALL\_IPA\_PASSES\_END, /\* 最后ipa通过后调用。\*/

PLUGIN\_OVERRIDE\_GATE, /\* 允许覆盖current\_pass的pass gate决定。\*/

PLUGIN\_PASS\_EXECUTION, /\* 在执行pass之前调用。\*/

/\* 在execute\_ipa\_pass\_list中执行GIMPLE\_PASS的子遍之前调用。 \*/

PLUGIN\_EARLY\_GIMPLE\_PASSES\_START,

/\* 在execute\_ipa\_pass\_list中执行GIMPLE\_PASS的子遍之后调用。 \*/

PLUGIN\_EARLY\_GIMPLE\_PASSES\_END,

PLUGIN\_NEW\_PASS, /\* 首次实例化pass时调用。\*/

/\* 当文件是＃include-d或通过#line指令给出时调用。这可能发生多次。事件数据是包含的文件路径，作为一个const char \*指针。\*/

PLUGIN\_INCLUDE\_FILE,

PLUGIN\_EVENT\_FIRST\_DYNAMIC /\* 用于索引回调数组的虚拟事件。\*/

};

此外，插件还可以通过调用函数get\_named\_event\_id来查找命名事件的枚举器和 / 或动态生成新事件。

要注册一个回调，插件调用register\_callback包括参数：

* char \*name: 插件名称
* int event: 事件代码。
* plugin\_callback\_func callback: 处理事件的函数。
* void \*user\_data: 指向插件特定数据的指针。

对于PLUGIN\_PASS\_MANAGER\_SETUP，PLUGIN\_INFO和PLUGIN\_REGISTER\_GGC\_ROOTS伪事件，回调应为null，user\_data是特定的。

当触发PLUGIN\_PRAGMAS事件（使用空指针作为来自GCC的数据）时，插件可以注册它们自己的编译指示。 请注意，lto1不提供编译指示，因此在链接时优化期间使用-flto选项的GCC的插件不能使用pragmas，甚至看不到像c\_register\_pragma或pragma\_lex这样的函数。

将触发PLUGIN\_INCLUDE\_FILE事件（将const char \*文件路径作为GCC数据），以处理#include或#line指令。

PLUGIN\_FINISH事件是插件可以调用GCC函数的最后一次，特别是发出具有警告，错误等的诊断。

### 23.3 Interacting with the pass manager

需要有一种方法来动态添加/重新排序/删除pass。这对于分析插件（在某个pass后插入，例如CFG或IPApass）和优化插件都很有用。

提供了插入新pass或替换现有pass的基本支持。一个插件通过调用带有PLUGIN\_PASS\_MANAGER\_SETUP事件的register\_callback和一个如下定义的struct register\_pass\_info对象的指针来注册与GCC的新pass

enum pass\_positioning\_ops

{

PASS\_POS\_INSERT\_AFTER, // 在引用pass之后插入。

PASS\_POS\_INSERT\_BEFORE, // 在引用pass之前插入。

PASS\_POS\_REPLACE // 替换引用pass。

};

struct register\_pass\_info

{

struct opt\_pass \*pass; /\* 插件提供的新pass。 \*/

const char \*reference\_pass\_name; /\*挂钩新pass的参考pass名称\*/

int ref\_pass\_instance\_number; /\*在指定的参考pass的实例号处插入pass\*/

/\* 如果某个实例是0就做此操作。\*/

enum pass\_positioning\_ops pos\_op; /\* 如何插入新的pass。 \*/

};

/\* 注册新pass的示例插件代码。\*/

int plugin\_init (struct plugin\_name\_args \*plugin\_info,

struct plugin\_gcc\_version \*version)

{

struct register\_pass\_info pass\_info;

...

/\* 用新pass信息填充pass\_info对象的代码。 \*/

...

/\* 注册新pass。 \*/

register\_callback (plugin\_info->base\_name, PLUGIN\_PASS\_MANAGER\_SETUP, NULL, &pass\_info);

...

}

### 23.4 Interacting with the GCC Garbage Collector

一些插件可能希望在GGC（GCC垃圾收集器）运行时被通知。它们可以注册PLUGIN\_GGC\_START和PLUGIN\_GGC\_END事件的回调（为此，使用空gcc\_data调用回调），以通知GCC垃圾回收的开始或结束。

一些插件可能需要有GGC标记额外的数据。这可以通过为PLUGIN\_GGC\_MARKING事件注册回调（使用空gcc\_data调用）来完成。这样的回调可以调用ggc\_set\_mark例程，最好通过ggc\_mark宏（相反，这些例程通常不能在PLUGIN\_GGC\_MARKING事件之外的插件中使用）。希望保存对gc数据的弱引用的插件也可以在对象即将被收集时使用此事件删除弱引用。ggc\_marked\_p函数可用于判断对象是否被标记或要被收集。一些类型定义的gt\_clear\_cache重载也可能用于管理弱引用。

某些插件可能需要添加额外的GGC根表，例如处理自己的GTY编辑数据。这可以通过带有null回调的PLUGIN\_REGISTER\_GGC\_ROOTS伪事件和作为user\_data的额外根表（类型为struct ggc\_root\_tab \*）完成。运行gengtype -p source-dir文件列表插件\* .c ...实用程序生成这些额外的根表。

在使用PLUGIN\_GGC\_MARKING或PLUGIN\_REGISTER\_GGC\_ROOTS之前，应该了解GCC内部内存管理的详细信息。

### 23.5 Giving information about a plugin

插件应该向用户提供关于其自身的一些信息。这使用以下结构：

struct plugin\_info

{

const char \*version;

const char \*help;

};

这样的结构作为user\_data由插件的init例程使用register\_callback与PLUGIN\_INFO伪事件和一个null回调传递。

### 23.6 Registering custom attributes or pragmas

注册自定义属性或编译指示，对于分析（或其他）目的，能够添加自定义属性或编译指示是有用的。

在属性注册期间调用PLUGIN\_ATTRIBUTES回调。使用register\_attribute函数注册自定义属性。

/\* 属性处理程序回调 \*/

static tree handle\_user\_attribute (tree \*node, tree name, tree args,

int flags, bool \*no\_add\_attrs)

{

return NULL\_TREE;

}

/\* 属性定义 \*/

static struct attribute\_spec user\_attr =

{ "user", 1, 1, false, false, false, handle\_user\_attribute, false };

/\* 在属性注册期间调用Plugin回调。 使用register\_callback注册（plugin\_name，PLUGIN\_ATTRIBUTES，register\_attributes，NULL）\*/

static void

register\_attributes (void \*event\_data, void \*data)

{

warning (0, G\_("Callback to register attributes"));

register\_attribute (&user\_attr);

}

在pragma注册期间调用PLUGIN\_PRAGMAS回调一次。使用c\_register\_pragma，c\_register\_pragma\_with\_data，c\_register\_pragma\_with\_expansion，c\_register\_pragma\_with\_expansion\_and\_data函数从c-family / c-pragma.h注册自定义pragmas及其处理程序（通常要调用pragma\_lex）。

/\* 在pragmas注册期间调用的插件回调。注册register\_callback（plugin\_name，PLUGIN\_PRAGMAS，register\_my\_pragma，NULL）; \*/

static void

register\_my\_pragma (void \*event\_data, void \*data)

{

warning (0, G\_("Callback to register pragmas"));

c\_register\_pragma ("GCCPLUGIN", "sayhello", handle\_pragma\_sayhello);

}

建议将“GCCPLUGIN”（或标识您的插件的短名称）作为您的编译指示的“空格”参数。

使用c\_register\_pragma\_with\_expansion或c\_register\_pragma\_with\_expansion\_and\_data注册的pragmas支持预处理器扩展。例如：

#define NUMBER 10

#pragma GCCPLUGIN foothreshold (NUMBER)

### 23.7 Recording information about pass execution

事件PLUGIN\_PASS\_EXECUTION将指向执行的pass（与current\_pass相同）的指针作为gcc\_data传递给回调。您还可以检查cfun以了解执行此pass的函数。请注意，只有在门检查（如果适用，由PLUGIN\_OVERRIDE\_GATE修改）成功时，才会调用此事件。您可以使用其他钩子，例如PLUGIN\_ALL\_PASSES\_START，PLUGIN\_ALL\_PASSES\_END，PLUGIN\_ALL\_IPA\_PASSES\_START，PLUGIN\_ALL\_IPA\_PASSES\_END，PLUGIN\_EARLY\_GIMPLE\_PASSES\_START和/或PLUGIN\_EARLY\_GIMPLE\_PASSES\_END，以操作插件中的全局状态，以获取传递执行的上下文。

### 23.8 Controlling which passes are being run

在调用传递的原始门功能之后，其结果（门状态）存储为整数。然后调用事件PLUGIN\_OVERRIDE\_GATE，指向回调函数的gcc\_data参数中的门状态的指针。门状态的非零值意味着要执行通过。您可以通过传递的指针读取和写入门状态。

### 23.9 Keeping track of available passes

当插件加载时，您可以检查各种传递列表以确定可用的传递。但是，其他插件可能会添加新的passes。此外，对GCC的未来更改可能导致在插件加载后添加通用passes。当一个pass首先被添加到其中一个pass列表时，事件PLUGIN\_NEW\_PASS被调用，回调参数gcc\_data指向新的pass。

### 23.10 Building GCC plugins

如果启用了插件，GCC将安装构建插件所需的标题（在安装树中的某个位置，例如在/usr/local下）。 特别是安装了一个plugin / include目录，包含构建插件所需的所有头文件。

在大多数系统上，您可以通过调用gcc -print-file-name = plugin（如果需要，用适当的程序路径替换gcc）查询此插件目录。

内部插件，这个插件目录名可以通过调用default\_plugin\_dir\_name（）查询。

插件可能知道，当他们编译时，GCC版本为其提供plugin-version.h。 常量宏GCCPLUGIN\_VERSION\_MAJOR，GCCPLUGIN\_VERSION\_MINOR，GCCPLUGIN\_VERSION\_PATCHLEVEL，GCCPLUGIN\_VERSION是整数，因此插件可以确保它是为GCC 4.7构建的

#if GCCPLUGIN\_VERSION != 4007

#error this GCC plugin is for GCC 4.7

#endif

下面的GNU Makefile摘录显示了如何构建一个简单的插件：

HOST\_GCC=g++

TARGET\_GCC=gcc

PLUGIN\_SOURCE\_FILES= plugin1.c plugin2.cc

GCCPLUGINS\_DIR:= $(shell $(TARGET\_GCC) -print-file-name=plugin)

CXXFLAGS+= -I$(GCCPLUGINS\_DIR)/include -fPIC -fno-rtti -O2

plugin.so: $(PLUGIN\_SOURCE\_FILES)

$(HOST\_GCC) -shared $(CXXFLAGS) $^ -o $@

单个源文件插件可以使用g++ -I`gcc -print-file-name = plugin`/include -fPIC -shared -fno-rtti -O2 plugin.c -o plugin.so来构建，使用反引号shell语法查询插件目录。

当插件需要使用gengtype时，请确保gengtype和gtype.state具有与构建插件的GCC相同的版本。

# GCC GENERIC

## v4.8.4： [GENERIC](file:///C:\Users\Reyan\Desktop\gccint\index.html#toc_GENERIC)

### 1 概要

**1.1 Tree的大致表现形式**

TREE\_CHAIN是一个指针，可以用作其他树的单链表。

TREE\_TYPE。 许多树在此字段中存储表达式或声明的类型。

【基本形式是'code，type，[operands]'。 code是TREE\_CODE，type是表示TREE\_TYPE的树。 这些操作后面跟着操作数，每个操作数也是一棵树。】

**1.2 标识符 Identifiers**

IDENTIFIER\_NODE表示一个稍微更一般的概念，即标准C或C ++标识符的概念。 特别地，IDENTIFIER\_NODE可以包含“$”或其他非常字符。

没有两个不同的IDENTIFIER\_NODE表示相同的标识符。 因此，可以使用指针等式来比较IDENTIFIER\_NODEs，而不是使用像strcmp这样的例程。 使用get\_identifier获取提供的字符串的唯一IDENTIFIER\_NODE。

可以使用以下宏访问标识符：

**IDENTIFIER\_POINTER**

标识符表示的字符串，表示为char \*。 此字符串始终为NUL终止，并且不包含嵌入的NUL字符。

**IDENTIFIER\_LENGTH**

由IDENTIFIER\_POINTER返回的字符串的长度，不包括尾随的NUL。 IDENTIFIER\_LENGTH（x）的值总是与strlen（IDENTIFIER\_POINTER（x））相同。

**IDENTIFIER\_OPNAME\_P**

如果标识符表示重载的运算符的名称，则此谓词成立。 在这种情况下，不应该依赖于IDENTIFIER\_POINTER或IDENTIFIER\_LENGTH的内容。

**IDENTIFIER\_TYPENAME\_P**

如果标识符表示用户定义的转换运算符的名称，则此谓词成立。 在这种情况下，IDENTIFIER\_NODE的TREE\_TYPE保存转换运算符转换的类型。

**1.3 tree中的容器**

两个公共容器数据结构可以直接用树节点表示。 **TREE\_LIST**是包含每个节点两个树的单链表。 这些是每个节点的**TREE\_PURPOSE**和**TREE\_VALUE**。 （通常，TREE\_PURPOSE包含某种标签或附加信息，而TREE\_VALUE包含大部分有效负载。在其他情况下，TREE\_PURPOSE只是NULL\_TREE，而在其他情况下，TREE\_PURPOSE和TREE\_VALUE的性质是相同的。 给定一个TREE\_LIST节点，通过跟随TREE\_CHAIN找到下一个节点。 如果TREE\_CHAIN为NULL\_TREE，那么您已到达列表的末尾。

**TREE\_VEC**是一个简单的向量。 TREE\_VEC\_LENGTH是给出向量中节点数量的整数（不是树）。 节点本身是使用TREE\_VEC\_ELT宏访问的，它需要两个参数。 第一个是所讨论的TREE\_VEC; 第二个是指示向量中期望哪个元素的整数。 元素从零开始索引。

### 2 类型（types）

所有类型都有相应的树节点。经常有多个节点对应于相同类型。

在大多数情况下，不同种类的类型具有不同的树码。 （例如，指针类型使用POINTER\_TYPE代码，而数组使用ARRAY\_TYPE代码。）但是，成员函数的指针使用RECORD\_TYPE代码。 因此，当编写一个依赖于与特定类型相关的代码的switch语句时，应该注意在RECORD\_TYPE大小写标签下处理指向成员函数的指针。

以下函数和宏处理cv-qualification的类型：

TYPE\_MAIN\_VARIANT

此宏返回类型的未限定版本。 它可以应用于非限定类型，但是在这种情况下它不总是身份函数。

一些其他宏和函数可用于所有类型：

**TYPE\_SIZE**

表示类型所需的位数，表示为INTEGER\_CST。 对于不完整的类型，TYPE\_SIZE将为NULL\_TREE。

**TYPE\_ALIGN**

类型的对齐，以位表示，表示为int。

**TYPE\_NAME**

此宏返回类型的声明（以TYPE\_DECL的形式）。 （注意这个宏不会返回一个IDENTIFIER\_NODE，正如你所期望的，给定它的名字！）你可以看看TYPE\_DECL的DECL\_NAME，以获得类型的实际名称。 对于不是内置类型的类型，typedef的结果或命名类类型，TYPE\_NAME将为NULL\_TREE。

**TYPE\_CANONICAL**

此宏返回给定类型节点的“规范”类型。规范类型用于提高C ++和Objective-C ++前端的性能，允许在same\_type\_p中的两个类型节点之间进行有效比较：如果类型的TYPE\_CANONICAL值相等，则类型是等价的;否则，类型不等效。规范类型的等价概念与语言本身中类型等价的概念相同。例如，

当TYPE\_CANONICAL为NULL\_TREE时，给定类型节点没有规范类型。在这种情况下，此类型和任何其他类型之间的比较需要编译器执行深层“结构”比较，以查看两个类型节点是否具有相同的形式和属性。

节点的规范类型始终是类型的等价类中最基本的类型。例如，int是它自己的规范类型。 int的typedef I将有int作为它的规范类型。类似地，I \*和typedef IP（定义为I \*）将具有int \*作为其规范类型。在构建新类型节点时，请务必将TYPE\_CANONICAL设置为适当的规范类型。如果新类型是复合类型（从其他类型构建），并且任何其他类型都需要结构相等，请使用SET\_TYPE\_STRUCTURAL\_EQUALITY来确保新类型还需要结构相等性。最后，如果由于某种原因不能保证TYPE\_CANONICAL将指向规范类型，那么使用SET\_TYPE\_STRUCTURAL\_EQUALITY来确保新类型以及基于它构造的任何类型都需要结构相等。如果怀疑规范类型系统是比较类型，则将--param verify-canonical-types = 1传递给编译器，或者使用--enable-checking进行配置，以强制编译器针对结构比较验证其规范类型比较;如果规范类型不匹配，编译器将打印任何警告。

**TYPE\_STRUCTURAL\_EQUALITY\_P**

当节点需要结构相等性检查时，例如TYPE\_CANONICAL为NULL\_TREE时，此谓词成立。

**SET\_TYPE\_STRUCTURAL\_EQUALITY**

这个宏声明给定的类型节点需要结构性等同性检查，例如，它将TYPE CANONICAL设置为NULL TREE。

**same\_type\_p**

此谓词使用两种类型作为输入，如果它们是相同类型，则保留。 例如，如果一个类型是另一个类型的typedef，或者两者都是同一类型的typedef。 如果作为输入给出的两个树仅仅是彼此的副本，则该谓词也成立; 即在源级别上它们之间没有差别，但是由于任何原因，在表示中已经做出了副本。 你不应该使用==（指针相等）来比较类型; 总是使用same\_type\_p。

下面详细介绍了各种类型，以及可用于访问它们的宏。 虽然其他类型的类型在G ++的其他地方使用，但这里描述的类型是您在检查中间表示时将会遇到的类型。

**VOID\_TYPE**

用于表示void类型。

**INTEGER\_TYPE**

用于表示各种整数类型，包括char，short，int，long和long long。此代码不用于枚举类型，也不用于bool类型。 TYPE\_PRECISION是表示中使用的位数，表示为unsigned int。 （注意，在一般情况下，这不是与TYPE\_SIZE相同的值;假设有一个24位整数类型，但ABI的对齐要求需要32位对齐，那么TYPE\_SIZE将是32的INTEGER\_CST，而TYPE\_PRECISION将为24.）如果TYPE\_UNSIGNED成立，整数类型为无符号;否则，它被签名。

TYPE\_MIN\_VALUE是可以由此类型表示的最小整数的INTEGER\_CST。类似地，TYPE\_MAX\_VALUE是可以由该类型表示的最大整数的INTEGER\_CST。

**REAL\_TYPE**

用于表示float，double和long double类型。浮点表示中的位数由TYPE\_PRECISION给出，如在INTEGER\_TYPE的情况。

**FIXED\_POINT\_TYPE**

用于表示short \_Fract，\_Fract，long \_Fract，long long \_Fract，short \_Accum，\_Accum，long \_Accum和long long \_Accum类型。 定点表示中的位数由TYPE\_PRECISION给出，如在INTEGER\_TYPE的情况。 可以存在填充比特，分数比特和整数比特。 分数位的数量由TYPE\_FBIT给出，并且整数位的数量由TYPE\_IBIT给出。 如果TYPE\_UNSIGNED成立，定点类型是无符号的; 否则，它被签名。 如果TYPE\_SATURATING成立，定点类型是饱和; 否则，它不饱和。

**COMPLEX\_TYPE**

用于表示GCC内置的\_\_complex\_\_数据类型。 TREE\_TYPE是实部和虚部的类型。

**ENUMERAL\_TYPE**

用于表示枚举类型。 TYPE\_PRECISION给出（作为int）用于表示类型的位数。 如果没有负枚举常数，TYPE\_UNSIGNED将保持。 最小和最大枚举常数可以分别用TYPE\_MIN\_VALUE和TYPE\_MAX\_VALUE获得; 每个这些宏返回一个INTEGER\_CST。

实际枚举常量本身可以通过查看TYPE\_VALUES来获得。 此宏将返回一个TREE\_LIST，其中包含常量。 每个节点的TREE\_PURPOSE将是一个IDENTIFIER\_NODE，给出常量的名称; TREE\_VALUE将是一个INTEGER\_CST赋值给该常量的值。 这些常量将按照声明的顺序显示。 这些常量的每一个的TREE\_TYPE将是枚举类型本身的类型。

**BOOLEAN\_TYPE**

用于表示bool类型。

**POINTER\_TYPE**

用于表示指针类型，以及指向数据成员类型的指针。 TREE\_TYPE给出了该类型所指向的类型。

**REFERENCE\_TYPE**

用于表示引用类型。 TREE\_TYPE给出此类型所指的类型。

**FUNCTION\_TYPE**

用于表示非成员函数和静态成员函数的类型。 TREE\_TYPE给出函数的返回类型。 TYPE\_ARG\_TYPES是参数类型的TREE\_LIST。 该列表中每个节点的TREE\_VALUE是相应参数的类型; TREE\_PURPOSE是缺省参数值的表达式（如果有）。 如果列表中的最后一个节点是void\_list\_node（TREE\_VALUE是void\_type\_node的TREE\_LIST节点），则此类型的函数不接受可变参数。 否则，它们采用可变数量的参数。

注意，在C中（但不是在C ++中）一个声明为void f（）的函数是一个未经过原型验证的函数，其参数数量可变; 这样的函数的TYPE\_ARG\_TYPES将为NULL。

**METHOD\_TYPE**

用于表示非静态成员函数的类型。与FUNCTION\_TYPE类似，返回类型由TREE\_TYPE给出。 \* this的类型，即该类型的函数是其成员的类，由TYPE\_METHOD\_BASETYPE给出。 TYPE\_ARG\_TYPES是参数列表，与FUNCTION\_TYPE一样，包括此参数。

**ARRAY\_TYPE**

用于表示数组类型。 TREE\_TYPE给出了数组中元素的类型。如果类型中存在数组边界，则TYPE\_DOMAIN是INTEGER\_TYPE，其TYPE\_MIN\_VALUE和TYPE\_MAX\_VALUE将分别是数组的下限和上限。对于零，TYPE\_MIN\_VALUE将始终是INTEGER\_CST，而TYPE\_MAX\_VALUE将比阵列中的元素的数量少一个，即，可以用于索引数组中的元素的最高值。

**RECORD\_TYPE**

用于表示结构和类类型，以及指向成员函数和其他语言的类似结构的指针。 TYPE\_FIELDS包含此类型中包含的项，每个项可以是FIELD\_DECL，VAR\_DECL，CONST\_DECL或TYPE\_DECL。您不能对类型中的字段的排序做任何假设，或者它们中的一个或多个重叠。

**UNION\_TYPE**

用于表示联合类型。类似于RECORD\_TYPE，除了TYPE\_FIELD中的所有FIELD\_DECL节点都从位位置零开始。

**QUAL\_UNION\_TYPE**

用于表示Ada中变体记录的一部分。与UNION\_TYPE类似，但每个FIELD\_DECL都有一个DECL\_QUALIFIER字段，该字段包含一个布尔表达式，用于指示该字段是否存在于对象中。该类型将只有一个字段，因此每个字段的DECL\_QUALIFIER仅在TYPE\_FIELDS中的前面字段中的表达式都不为零时才进行计算。通常这些表达式将使用PLACEHOLDER\_EXPR引用外部对象中的字段。

**LANG\_TYPE**

此节点用于表示特定于语言的类型。前端必须处理它。

**OFFSET\_TYPE**

此节点用于表示指向数据成员的指针。对于数据成员X :: m，TYPE\_OFFSET\_BASETYPE是X，TREE\_TYPE是m的类型。

有一些变量的值代表一些基本类型。 这些包括：

**void\_type\_node**

void的节点。

**integer\_type\_node**

int的节点。

**unsigned\_type\_node。**

unsigned int的节点。

**char\_type\_node。**

char的节点。

有时可能有用的是使用same\_type\_p将这些变量之一与手中的类型进行比较。

### 3 声明

本节涵盖内部表示中出现的各种类型的声明，函数声明（由FUNCTION\_DECL节点表示）除外，这些声明在函数中描述。

#### 3.1使用声明：在声明上工作的宏和函数。

一些宏可以与任何类型的声明一起使用。 这些包括：

**DECL\_NAME**

此宏返回一个IDENTIFIER\_NODE，给出实体的名称。

**TREE\_TYPE**

此宏返回所声明的实体的类型。

**EXPR\_FILENAME**

此宏返回实体被声明的文件的名称，作为char \*。 对于由编译器隐式声明的实体（如\_\_builtin\_memcpy），这将是字符串“<internal>”。

**EXPR\_LINENO**

此宏返回实体声明时的行号，为int。

**DECL\_ARTIFICIAL**

如果声明由编译器隐式生成，则此谓词成立。 例如，此谓词将保存隐式声明的成员函数，或者为类类型隐式生成的TYPE\_DECL。 回想一下在C ++代码中：

           struct S {};

大致相当于C代码如：

           struct S {};

           typedef struct S S;

隐式生成的typedef声明由DECL\_ARTIFICIAL保存的TYPE\_DECL表示。

各种声明包括：

**LABEL\_DECL**

这些节点用于表示函数体中的标签。有关详细信息，请参阅函数。这些节点仅显示在块作用域中。

**CONST\_DECL**

这些节点用于表示枚举常量。常数的值由DECL\_INITIAL给出，它将是与CONST\_DECL的TREE\_TYPE具有相同类型的INTEGER\_CST，即ENUMERAL\_TYPE。

**RESULT\_DECL**

这些节点表示函数返回的值。当一个值分配给RESULT\_DECL时，表示该值应通过该函数按位复制返回。您可以在RESULT\_DECL上使用DECL\_SIZE和DECL\_ALIGN，就像使用VAR\_DECL一样。

**TYPE\_DECL**

这些节点表示typedef声明。 TREE\_TYPE是声明为具有由DECL\_NAME给定的名称的类型。在某些情况下，没有关联的名称。

**VAR\_DECL**

**这些节点表示具有命名空间或块范围的变量，以及静态数据成员。** DECL\_SIZE和DECL\_ALIGN类似于TYPE\_SIZE和TYPE\_ALIGN。对于声明，您应该始终使用DECL\_SIZE和DECL\_ALIGN，而不是由TREE\_TYPE给定的TYPE\_SIZE和TYPE\_ALIGN，因为特殊属性可能已应用于变量以赋予其特定大小和对齐。您可以使用谓词DECL\_THIS\_STATIC或DECL\_THIS\_EXTERN来测试存储类说明符static或extern是否用于声明变量。

如果初始化该变量（但不需要构造函数），则DECL\_INITIAL将是初始化器的表达式。应该评估初始化程序，并对执行的变量执行逐位复制。如果DECL\_INITIAL是error\_mark\_node，则有一个初始化器，但它在代码后面的一个显式语句中给出;不需要按位复制。

GCC提供了一个扩展，允许将自动变量或全局变量放置在特定的寄存器中。如果DECL\_REGISTER适用于VAR\_DECL，并且如果DECL\_ASSEMBLER\_NAME不等于DECL\_NAME，则此扩展用于特定的VAR\_DECL。在这种情况下，DECL\_ASSEMBLER\_NAME是将放置变量的寄存器的名称。

**PARM\_DECL**

用于表示函数的参数。这些节点与VAR\_DECL节点类似。这些节点仅出现在FUNCTION\_DECL的DECL\_ARGUMENTS中。

PARM\_DECL的DECL\_ARG\_TYPE是将值传递给此函数时实际使用的类型。它可能是比参数的TREE\_TYPE更宽的类型;例如，普通类型可能很短，而DECL\_ARG\_TYPE是int。

**DEBUG\_EXPR\_DECL**

用于表示一个匿名调试信息临时创建以保持表达式被优化，以便其值可以在调试绑定语句中引用。

**FIELD\_DECL**

这些节点表示非静态数据成员。 DECL\_SIZE和DECL\_ALIGN的行为与VAR\_DECL节点相同。父记录中的字段的位置由三个属性的组合指定。 DECL\_FIELD\_OFFSET是包含最接近结构开始的字段的位的DECL\_OFFSET\_ALIGN位大小的字的位置，以字节计数。 DECL\_FIELD\_BIT\_OFFSET是该字中的字段的第一位的位偏移;即使对于不是位字段的字段，这也可能是非零的，因为DECL\_OFFSET\_ALIGN可能大于字段类型的自然对齐。

如果DECL\_C\_BIT\_FIELD成立，则该字段是位字段。在位字段中，DECL\_BIT\_FIELD\_TYPE还包含最初为其指定的类型，而DECL\_TYPE可以是根据位字段的大小具有较小精度的修改类型。

**NAMESPACE\_DECL**

命名空间为其他声明提供名称层次结构。它们出现在其他\_DECL节点的DECL\_CONTEXT中。

#### 3.2 内部结构：如何表示声明节点。

DECL节点在内部表示为结构的层次结构。

**当前结构层次：当前DECL节点结构层次。**

struct tree\_decl\_minimal

这是继承的最小结构，以便普通DECL宏可以工作。它包含的字段是唯一的ID，源位置，上下文和名称。

struct tree\_decl\_common

此结构继承struct tree\_decl\_minimal。它包含大多数DECL节点需要的字段，例如存储对齐，机器模式，大小和属性的字段。

struct tree\_field\_decl

这个结构继承于struct tree\_decl\_common。它用于表示FIELD\_DECL。

struct tree\_label\_decl

这个结构继承于struct tree\_decl\_common。它用于表示LABEL\_DECL。

struct tree\_translation\_unit\_decl

这个结构继承于struct tree\_decl\_common。它用于表示TRANSLATION\_UNIT\_DECL。

struct tree\_decl\_with\_rtl

这个结构继承于struct tree\_decl\_common。它包含用于存储与DECL节点相关联的低级RTL的字段。

struct tree\_result\_decl

这个结构继承于struct tree\_decl\_with\_rtl。它用于表示RESULT\_DECL。

struct tree\_const\_decl

这个结构继承于struct tree\_decl\_with\_rtl。它用于表示CONST\_DECL。

struct tree\_parm\_decl

这个结构继承于struct tree\_decl\_with\_rtl。它用于表示PARM\_DECL。

struct tree\_decl\_with\_vis

这个结构继承于struct tree\_decl\_with\_rtl。它包含存储可见性信息所需的字段，以及节名称和汇编器名称。

struct tree\_var\_decl

此结构继承于struct tree\_decl\_with\_vis。它用于表示VAR\_DECL。

struct tree\_function\_decl

此结构继承于struct tree\_decl\_with\_vis。它用于表示FUNCTION\_DECL。

**添加新的DECK节点类型：如何将新的DECL节点添加到前端。**

添加新的DECL树由以下步骤组成

为DECL节点添加新的树代码

对于语言特定的DECL节点，在每个前端目录中都有一个.def文件，其中应该添加树代码。 对于作为中端一部分的DECL节点，代码应该添加到tree.def。

为DECL节点创建一个新的结构类型

这些结构应该通过使用该结构作为第一成员从语言层次结构中的现有结构之一继承。

           struct tree\_foo\_decl

           {

              struct tree\_decl\_with\_vis common;

           }}

将创建一个从struct tree\_decl\_with\_vis继承的结构名tree\_foo\_decl。

对于语言特定的DECL节点，这个新的结构类型应该放在适当的.h文件中。 对于作为中端的一部分的DECL节点，结构类型应该在tree.h中。

将一个成员添加到该节点的树结构枚举器

对于垃圾回收和动态检查目的，每个DECL节点结构类型都需要有一个由它指定的唯一枚举值。对于语言特定的DECL节点，这个新的枚举值应该放在适当的.def文件中。对于作为中端的一部分的DECL节点，枚举器值在treestruct.def中指定。

更新联合tree\_node

为了使您的新结构类型可用，必须将其添加到union tree\_node。对于语言特定的DECL节点，应将一个新条目添加到表单的相应.h文件中

struct tree\_foo\_decl GTY（（tag（“TS\_VAR\_DECL”）））foo\_decl;

对于作为中端一部分的DECL节点，附加成员直接进入tree.h中的union tree\_node。

更新动态检查信息

为了能够检查访问union tree\_node的命名部分是否合法，以及某个DECL节点是否包含层次结构中列举的DECL节点结构之一，使用简单的查找表。此查找表需要与树结构层次结构保持最新，否则检查和遏制宏将不适当地失败。

对于语言特定的DECL节点，它们是在适当的.c文件中的init\_ts函数，它初始化查找表。应该在那里添加为新的DECL节点设置表的代码。对于表示继承层次结构成员的每个DECL树代码和枚举器值，如果该树代码从该成员继承（直接或间接），则该表应包含1。因此，从struct decl\_with\_rtl和枚举值TS\_FOO\_DECL导出的FOO\_DECL节点将被设置如下

tree\_contains\_struct [FOO\_DECL] [TS\_FOO\_DECL] = 1;

tree\_contains\_struct [FOO\_DECL] [TS\_DECL\_WRTL] = 1;

tree\_contains\_struct [FOO\_DECL] [TS\_DECL\_COMMON] = 1;

tree\_contains\_struct [FOO\_DECL] [TS\_DECL\_MINIMAL] = 1;

对于作为中端的一部分的DECL节点，设置代码进入tree.c。

添加宏以访问任何新字段和标志

每个添加的字段或标志应该具有用于访问它的宏，该宏执行适当的检查以确保只有正确类型的DECL节点访问该字段。

这些宏通常采用以下形式

#define FOO\_DECL\_FIELDNAME（NODE）FOO\_DECL\_CHECK（NODE） - > foo\_decl.fieldname

然而，如果结构仅仅是用于其他结构的基类，则应该使用类似下面的结构

#define BASE\_STRUCT\_CHECK（T）CONTAINS\_STRUCT\_CHECK（T，TS\_BASE\_STRUCT）

#define BASE\_STRUCT\_FIELDNAME（NODE）\

 （BASE\_STRUCT\_CHECK（NODE） - > base\_struct.fieldname

### 4 树中的属性

使用\_\_attribute\_\_关键字指定的属性在内部表示为TREE\_LIST。 TREE\_PURPOSE是属性的名称，作为IDENTIFIER\_NODE。 TREE\_VALUE是属性的参数（如果有）的TREE\_LIST，如果没有参数，则为NULL\_TREE; 参数被存储为列表中的连续条目的TREE\_VALUE，并且可以是标识符或表达式。 属性的TREE\_CHAIN是应用于同一声明或类型的属性列表中的下一个属性，如果列表中没有其他属性，则为NULL\_TREE。

属性可以附加到声明和类型; 这些属性可以使用以下宏访问。 所有属性以这种方式存储，并且许多还会导致对声明或类型或其他内部编译器数据结构的其他更改。

- 树宏：树DECL\_ATTRIBUTES（tree decl）

此宏返回声明decl的属性。

- 树宏：树TYPE\_ATTRIBUTES（tree type）

此宏返回type类型上的属性。

### 5 表达式

表达式的内部表示在很大程度上是相当直接的。 然而，有一些事实，人们必须铭记。 特别地，表达式“树”实际上是有向无环图。 （例如，在整个源程序中可能有许多对整数常数零的引用;其中许多将由相同的表达式节点表示。）你不应该依赖某些类型的节点被共享，也不应该依赖于某些类型的节点未共享。

以下宏可与所有表达式节点一起使用：

**TREE\_TYPE**

返回表达式的类型。 该值可能不是与原始程序中的表达式相同的类型。

在下面的内容中，一些可能期望总是具有类型bool的节点被记录为具有整数类型或布尔类型。 在将来的某个时候，C前端也可以利用这个相同的中间表示，并且在这一点上，这些节点将肯定具有整数类型。 前面的句子并不意味着C++前端不会或不会给这些节点整数类型。

下面，我们列出了各种表达式节点。 除非另有说明，表达式的操作数使用TREE\_OPERAND宏访问。 例如，要访问第一个操作数为二进制加表达式expr，请使用：

      TREE\_OPERAND（expr，0）

如本例所示，操作数为零索引。

#### 5.1常量表达式

下面的表从常数开始，移动到一元表达式，然后进入二进制表达式，并结束与各种其他种类的表达式：

**INTEGER\_CST**

这些节点表示整数常量。请注意，这些常量的类型是使用TREE\_TYPE获取的;它们不总是int类型。特别地，char常量用INTEGER\_CST节点表示。整数常数e的值由下式给出

          （（TREE\_INT\_CST\_HIGH（e）<< HOST\_BITS\_PER\_WIDE\_INT）

          + TREE\_INST\_CST\_LOW（e））

HOST\_BITS\_PER\_WIDE\_INT在所有平台上至少有32个。 TREE\_INT\_CST\_HIGH和TREE\_INT\_CST\_LOW都返回HOST\_WIDE\_INT。根据常量的类型，INTEGER\_CST的值被解释为有符号或无符号数。一般来说，上面给出的表达式将溢出，因此它不应该用于计算常量的值。

变量integer\_zero\_node是值为零的整数常量。类似地，integer\_one\_node是值为1的整数常量。 size\_zero\_node和size\_one\_node变量类似，但类型为size\_t，而不是int。

函数tree\_int\_cst\_lt是一个谓词，如果它的第一个参数小于它的第二个参数。两个常数都假定具有相同的符号（即，或者两者都应该是有符号的，或者两者都是无符号的。）当进行比较时使用常数的全宽度;关于促销和转化的常见规则将被忽略。类似地，如果两个常量相等，则tree\_int\_cst\_equal成立。 tree\_int\_cst\_sgn函数返回常量的符号。根据常数是大于，等于还是小于零，该值为1,0或-1。同样，考虑常数类型的正负号;无符号常量从不小于零，无论其位模式如何。

**REAL\_CST**

**FIXME**：谈论如何获得这个常量的表示，进行比较，等等。

**FIXED\_CST**

这些节点表示定点常数。 这些常量的类型是使用TREE\_TYPE获取的。 TREE\_FIXED\_CST\_PTR指向struct fixed\_value; TREE\_FIXED\_CST返回结构本身。 struct fixed\_value包含大小为两个HOST\_BITS\_PER\_WIDE\_INT的数据和模式作为数据的关联定点机器模式。

**COMPLEX\_CST**

这些节点用于表示复数常数，即\_\_complex\_\_，它的部分是常数节点。 TREE\_REALPART和TREE\_IMAGPART分别返回实部和虚部。

**VECTOR\_CST**

这些节点用于表示向量常数，其部分是常数节点。 每个单独的常数节点是整数或双常数节点。 第一个操作数是常数节点的TREE\_LIST，通过TREE\_VECTOR\_CST\_ELTS访问。

**STRING\_CST**

这些节点表示字符串常量。 TREE\_STRING\_LENGTH返回字符串的长度，为int。 TREE\_STRING\_POINTER是一个包含字符串本身的char \*。 字符串可能不是NUL终止的，它可能包含嵌入的NUL字符。 因此，如果TREE\_STRING\_LENGTH存在，则TREE\_STRING\_LENGTH包括尾随NUL。

对于宽字符串常量，TREE\_STRING\_LENGTH是字符串中的字节数，TREE\_STRING\_POINTER指向字符串的字节数组，如目标系统上所表示的（即，作为目标字节序中的整数）。 宽字符串和非宽字符串常量仅由STRING\_CST的TREE\_TYPE区分。

**FIXME**：当目标系统字节与主机系统字节宽度不同时，字符串常量的格式不能很好定义。

#### 5.2引用存储

**ARRAY\_REF**

这些节点表示数组访问。第一个操作数是数组;第二是索引。要计算所访问的内存的地址，必须按照数组元素类型的大小来缩放索引。这些表达式的类型必须是数组的组件的类型。第三和第四操作数在gimplification之后用于表示下限和组件大小，但不应直接使用;调用array\_ref\_low\_bound和array\_ref\_element\_size。

**ARRAY\_RANGE\_REF**

这些节点表示对数组的范围（或“切片”）的访问。操作数与ARRAY\_REF的操作数相同，含义相同。这些表达式的类型必须是其组件类型与第一个操作数的组件类型相同的数组。该数组类型的范围决定了这些表达式访问的数据量。

**TARGET\_MEM\_REF**

这些节点表示其地址直接映射到目标体系结构的寻址模式的存储器访问。第一个参数是TMR\_SYMBOL，并且必须是具有固定地址的对象的VAR\_DECL。第二个参数是TMR\_BASE，第三个参数是TMR\_INDEX。第四个参数是TMR\_STEP，并且必须是INTEGER\_CST。第五个参数是TMR\_OFFSET，并且必须是INTEGER\_CST。如果适当的组件未出现在地址中，则任何参数可以为NULL。 TARGET\_MEM\_REF的地址以下列方式确定。

＆TMR\_SYMBOL + TMR\_BASE + TMR\_INDEX \* TMR\_STEP + TMR\_OFFSET

第六个参数是对原始存储器访问的引用，为了RTL别名分析的目的保留它。第七个参数是表示树级别别名分析的结果的标签。

**ADDR\_EXPR**

这些节点用于表示对象的地址。 （这些表达式将始终具有指针或引用类型。）操作数可以是另一个表达式，或者它可以是一个声明。

作为扩展，GCC允许用户获取标签的地址。在这种情况下，ADDR\_EXPR的操作数将是LABEL\_DECL。这样的表达式的类型是void \*。

如果所寻址的对象不是一个左值，则创建一个临时，并使用临时的地址。

**INDIRECT\_REF**

这些节点用于表示指针指向的对象。操作数是被取消引用的指针;它将始终具有指针或引用类型。

**MEM\_REF**

这些节点用于表示由常数偏移的指针指向的对象。第一个操作数是被取消引用的指针;它将始终具有指针或引用类型。第二个操作数是一个指针常量。其类型指定要用于基于类型的别名分析的类型。

**COMPONENT\_REF**

这些节点表示非静态数据成员访问。第一个操作数是对象（而不是指向它的指针）;第二个操作数是数据成员的FIELD\_DECL。第三个操作数表示字段的字节偏移，但不应直接使用;调用component\_ref\_field\_offset。

#### 5.4 一元和二进制表达式

**NEGATE\_EXPR**

这些节点表示整数和浮点类型的单个操作数的一元否定。否定的类型可以通过查看表达式的类型来确定。

此操作对带符号算术溢出的行为由flag\_wrapv和flag\_trapv变量控制。

**ABS\_EXPR**

这些节点表示整数和浮点类型的单个操作数的绝对值。这通常用于实现整数类型的abs，labs和llabs内置函数，以及浮点类型的fabs，fabsf和fabsl内置函数。 abs操作的类型可以通过查看表达式的类型来确定。

此节点不用于复杂类型。要表示复数值的模数或复数abs，请使用BUILT\_IN\_CABS，BUILT\_IN\_CABSF或BUILT\_IN\_CABSL内置函数，用于实现C99 cabs，cabsf和cabsl内置函数。

**BIT\_NOT\_EXPR**

这些节点表示按位互补，并且将始终具有整数类型。唯一的操作数是要补充的值。

**TRUTH\_NOT\_EXPR**

这些节点表示逻辑否定，并且将始终具有整数（或布尔）类型。操作数是被否定的值。操作数和结果的类型始终为BOOLEAN\_TYPE或INTEGER\_TYPE。

**PREDECREMENT\_EXPR**

**PREINCREMENT\_EXPR**

**POSTDECREMENT\_EXPR**

**POSTINCREMENT\_EXPR**

这些节点表示增量和减量表达式。计算单个操作数的值，并且操作数递增或递减。在PREDECREMENT\_EXPR和PREINCREMENT\_EXPR的情况下，表达式的值是在增量或减量后产生的值;在POSTDECREMENT\_EXPR的情况下，POSTINCREMENT\_EXPR是在增量或减量发生之前的值。操作数的类型，如结果的类型，将是整数，布尔或浮点。

**FIX\_TRUNC\_EXPR**

这些节点表示将浮点值转换为整数。单个操作数将具有浮点类型，而完整表达式将具有整数（或布尔）类型。操作数向零舍入。

**FLOAT\_EXPR**

这些节点表示将整数（或布尔）值转换为浮点值。单个操作数将具有整数类型，而完整表达式将具有浮点类型。

FIXME：操作数应该如何舍入？这是依赖于-mieee吗？

**COMPLEX\_EXPR**

这些节点用于表示由相同（整数或实数）类型的两个表达式构成的复数。第一个操作数是实部，第二个操作数是虚部。

**CONJ\_EXPR**

这些节点表示它们的操作数的共轭。

**REALPART\_EXPR**

**IMAGPART\_EXPR**

这些节点分别表示复数的实部和虚部（它们的唯一参数）。

**NON\_LVALUE\_EXPR**

这些节点指示它们的唯一操作数不是左值。后端可以将这些相同地对待单个操作数。

**NOP\_EXPR**

这些节点用于表示不需要任何代码生成的转换。例如，将char \*转换为int \*不需要生成任何代码;这样的转换由NOP\_EXPR表示。单个操作数是要转换的表达式。从指针到引用的转换也用NOP\_EXPR表示。

**CONVERT\_EXPR**

这些节点类似于NOP\_EXPR，但是在需要生成代码的情况下使用。例如，如果将int \*转换为int代码，则可能需要在某些平台上生成。这些节点从不用于C++特定的转换，如在继承层次结构中指向不同类的指针之间的转换。在这种情况下需要进行的任何调整总是明确指出。类似地，用户定义的转换从不由CONVERT\_EXPR表示;相反，函数调用是明确的。

**FIXED\_CONVERT\_EXPR**

这些节点用于表示涉及定点值的转换。例如，从定点值到另一个定点值，从整数到定点值，从定点值到整数，从浮点值到定点值，或者从定点值到浮点值。

**LSHIFT\_EXPR**

**RSHIFT\_EXPR**

这些节点分别表示左移和右移。第一个操作数是要移位的值;它将永远是整体型。第二操作数是用于移位的位数的表达式。右移应当被视为算术，即当表达式具有无符号类型时，高阶位应该是零填充，并且当表达式具有有符号类型时，应该用符号位填充。请注意，如果第二个操作数大于或等于第一个操作数的类型大小，则结果未定义。与大多数节点不同，这些可以具有作为第一操作数的向量和作为第二操作数的标量。

**BIT\_IOR\_EXPR**

**BIT\_XOR\_EXPR**

**BIT\_AND\_EXPR**

这些节点分别表示逐位包含或逐位异或，和逐位和。两个操作数将始终具有整数类型。

**TRUTH\_ANDIF\_EXPR**

**TRUTH\_ORIF\_EXPR**

这些节点分别表示逻辑“和”和逻辑“或”。这些运营商不严格;即，只有当表达式的值不是由第一操作数的评估确定时，才评估第二操作数。操作数的类型和结果的类型始终为BOOLEAN\_TYPE或INTEGER\_TYPE。

**TRUTH\_AND\_EXPR**

**TRUTH\_OR\_EXPR**

**TRUTH\_XOR\_EXPR**

这些节点表示逻辑和、逻辑或、逻辑异或。他们是严格的;两个参数总是被评估。在C或C ++中没有相应的运算符，但是前端有时会生成这些表达式，无论如何，如果它可以告诉严格性无关紧要。操作数的类型和结果的类型始终为BOOLEAN\_TYPE或INTEGER\_TYPE。

**POINTER\_PLUS\_EXPR**

此节点表示指针运算。第一个操作数始终是指针/引用类型。第二个操作数总是与sizetype兼容的无符号整数类型。这是可以对指针类型进行操作的唯一二进制算术操作数。

**PLUS\_EXPR**

**MINUS\_EXPR**

**MULT\_EXPR**

这些节点表示各种二进制算术运算。分别地，这些操作是加法，减法（从第一个的第二个操作数）和乘法。它们的操作数可以具有整数类型或浮点类型，但是不会有一个操作数是浮点类型而另一个是整数类型的情况。

这些操作对带符号算术溢出的行为由flag\_wrapv和flag\_trapv变量控制。

**MULT\_HIGHPART\_EXPR**

该节点表示加宽乘法的“高部分”。对于具有b位精度的积分类型，结果是完整2b乘积的最高有效位b。

**RDIV\_EXPR**

该节点表示浮点除法运算。

**TRUNC\_DIV\_EXPR**

**FLOOR\_DIV\_EXPR**

**CEIL\_DIV\_EXPR**

**ROUND\_DIV\_EXPR**

这些节点表示返回整数结果的整数除法运算。 TRUNC\_DIV\_EXPR向零舍入，FLOOR\_DIV\_EXPR向负无穷大舍入，CEIL\_DIV\_EXPR向正无穷大舍入，ROUND\_DIV\_EXPR舍入为最接近的整数。 C和C ++中的整数除法是截断，即TRUNC\_DIV\_EXPR。

当将最小有符号整数除以1时，这些对带符号算术溢出的操作的行为由flag\_wrapv和flag\_trapv变量控制。

**TRUNC\_MOD\_EXPR**

**FLOOR\_MOD\_EXPR**

**CEIL\_MOD\_EXPR**

**ROUND\_MOD\_EXPR**

这些节点表示整数余数或模运算。两个操作数a和b的整数模量被定义为a - （a / b）\* b，其中使用相应的除法运算符计算出除法。因此，对于TRUNC\_MOD\_EXPR，该定义假定使用向零舍去，即TRUNC\_DIV\_EXPR。 C和C ++中的整数余数使用截断除法，即TRUNC\_MOD\_EXPR。

**EXACT\_DIV\_EXPR**

EXACT\_DIV\_EXPR代码用于表示整数除法，其中分子已知为分母的精确倍数。这允许后端在当前目标的TRUNC\_DIV\_EXPR，CEIL\_DIV\_EXPR和FLOOR\_DIV\_EXPR的较快之间进行选择。

**LT\_EXPR**

**LE\_EXPR**

**GT\_EXPR**

**GE\_EXPR**

**EQ\_EXPR**

**NE\_EXPR**

这些节点表示小于，小于或等于，大于，大于或等于，等于和不等于比较运算符。第一和第二操作数将是整数类型，浮动类型或向量类型两者。这些表达式的结果类型将始终为整数，布尔或有符号整数向量类型。这些操作返回结果类型的零值为false，结果类型的一个值为true，向量的元素为零（false）或减一（true）的向量。

对于浮点比较，如果我们荣誉IEEE NaN并且任一操作数是NaN，则NE\_EXPR总是返回真，而其余的运算符总是返回假。在一些目标上，除了等式和不等式之外，与IEEE NaN的比较可能生成浮点异常。

**ORDERED\_EXPR**

**UNORDERED\_EXPR**

这些节点表示非陷阱有序和无序的比较运算符。这些操作采用两个浮点操作数，并确定它们是相对于彼此是有序的还是无序的。如果任一操作数是IEEE NaN，则它们的比较被定义为无序，否则比较被定义为有序的。这些表达式的结果类型将始终为整数或布尔类型。这些操作返回结果类型的零值为false，结果类型的一个值为true。

**UNLT\_EXPR**

**UNLE\_EXPR**

**UNGT\_EXPR**

**UNGE\_EXPR**

**UNEQ\_EXPR**

**LTGT\_EXPR**

这些节点表示无序的比较运算符。这些操作采用两个浮点操作数，并确定操作数是无序的还是小于，小于或等于，大于，大于或等于或等于。例如，如果任一操作数是IEEE NaN或第一个操作数小于第二个操作数，则UNLT\_EXPR返回true。由于LTGT\_EXPR可能异常，所有这些操作都保证不会生成浮点异常。这些表达式的结果类型将始终为整数或布尔类型。这些操作返回结果类型的零值为false，结果类型的一个值为true。

**MODIFY\_EXPR**

这些节点表示分配。左侧是第一个操作数;右侧是第二个操作数。左侧将是VAR\_DECL，INDIRECT\_REF，COMPONENT\_REF或其他左值。

这些节点不仅用于表示赋值'='，还用于表示复合赋值（如'+ ='），通过减少到'='赋值。换句话说，'i + = 3'的表示看起来就像'i = i + 3'。

**INIT\_EXPR**

这些节点与MODIFY\_EXPR类似，但仅在变量初始化时使用，而不是随后分配。这意味着我们可以假设初始化的目标不用于计算其自己的值;在计算rhs中对lhs的任何引用是未定义的。

**COMPOUND\_EXPR**

这些节点表示逗号表达式。第一个操作数是一个表达式，其值在第二个操作数的计算之前被计算和丢弃。整个表达式的值是第二个操作数的值。

**COND\_EXPR**

这些节点表示？：表达式。第一个操作数是布尔型或整数型。如果它的值为非零值，则应计算第二个操作数，并作为表达式的值返回。否则，将评估第三个操作数，并将其作为表达式的值返回。

第二个操作数必须具有与整个表达式相同的类型，除非它无条件地抛出异常或调用noreturn函数，在这种情况下，它应该具有void类型。相同的约束适用于第三操作数。这允许阵列边界检查被方便地表示为（i> = 0 && i <10）？ i：abort（）。

作为GNU扩展，C语言前端允许在源中省略？：操作符的第二个操作数。例如，x？ ：3相当于x？ x：3，假设x是没有副作用的表达式。然而，在树表示中，第二操作数总是存在，可能由SAVE\_EXPR保护，如果第一参数确实引起副作用。

**CALL\_EXPR**

这些节点用于表示对函数的调用，包括非静态成员函数。 CALL\_EXPRs被实现为具有可变数目的操作数的表达式节点。而不是使用TREE\_OPERAND来提取它们，最好使用专门在CALL\_EXPR节点上操作的专用访问器宏和函数。

CALL\_EXPR\_FN返回一个指向要调用的函数的指针;它总是一个类型为POINTER\_TYPE的表达式。

调用的参数数由call\_expr\_nargs返回，而参数本身可以使用CALL\_EXPR\_ARG宏访问。参数是零索引和从左到右编号。可以使用FOR\_EACH\_CALL\_EXPR\_ARG对参数进行迭代，如：

tree call, arg;

call\_expr\_arg\_iterator iter;

FOR\_EACH\_CALL\_EXPR\_ARG（arg，iter，call）

/ \* arg被绑定到连续的调用参数。 \* /

...;

对于非静态成员函数，将有一个对应于this指针的操作数。将始终存在与所有参数相对应的表达式，即使函数使用默认参数声明并且在调用站点未显式提供某些参数。

CALL\_EXPRs还有一个用于实现嵌套函数的CALL\_EXPR\_STATIC\_CHAIN操作数。否则此操作数为null。

**CLEANUP\_POINT\_EXPR**

这些节点表示完全表达式。单个操作数是要评估的表达式。应在评估表达式后立即执行在评估该表达式期间通过创建临时表达式产生的任何析构函数调用。

**CONSTRUCTOR**

这些节点表示结构或数组的括号括起来的初始化器。第一个操作数保留供后端使用。第二个操作数是TREE\_LIST。如果CONSTRUCTOR的TREE\_TYPE是RECORD\_TYPE或UNION\_TYPE，则TREE\_LIST中每个节点的TREE\_PURPOSE将是FIELD\_DECL，并且每个节点的TREE\_VALUE将是用于初始化该字段的表达式。

如果CONSTRUCTOR的TREE\_TYPE是ARRAY\_TYPE，则TREE\_LIST中每个元素的TREE\_PURPOSE将是两个INTEGER\_CST的INTEGER\_CST或RANGE\_EXPR。单个INTEGER\_CST指示要分配给数组的哪个元素（从零开始索引）。 RANGE\_EXPR表示要初始化的元素的包含范围。在这两种情况下，TREE\_VALUE是相应的初始化程序。它对RANGE\_EXPR的每个元素重新求值。如果TREE\_PURPOSE为NULL\_TREE，则初始化器用于下一个可用的数组元素。

在前端，您不应该依赖以任何特定顺序显示的字段。但是，在中间，字段必须以声明顺序显示。您不应该假设所有字段都将被表示。未显示的字段将设置为零。

**COMPOUND\_LITERAL\_EXPR**

这些节点表示ISO C99复合字面量。 COMPOUND\_LITERAL\_EXPR\_DECL\_EXPR是一个DECL\_EXPR，包含由复合字面量表示的未命名对象的匿名VAR\_DECL;该VAR\_DECL的DECL\_INITIAL是一个CONSTRUCTOR，表示复合字面值中初始值的大括号列表。匿名VAR\_DECL也可以通过COMPOUND\_LITERAL\_EXPR\_DECL宏直接访问。

**SAVE\_EXPR**

SAVE\_EXPR表示多次使用的表达式（可能涉及副作用）。副作用应仅在第一次评估表达式时发生。后续使用应该只重用计算的值。 SAVE\_EXPR的第一个操作数是要求值的表达式。副作用应该在首先在表达式树的深度优先预先遍历中首次遇到SAVE\_EXPR的地方执行。

**TARGET\_EXPR**

TARGET\_EXPR表示临时对象。第一个操作数是临时变量的VAR\_DECL。第二个操作数是临时的初始化程序。初始化器被评估，并且如果非空，将（按位）复制到临时中。如果初始化程序是void，那意味着它将执行初始化本身。

通常，TARGET\_EXPR出现在赋值的右侧，或者作为逗号表达式的第二个操作数，它本身是赋值语句的右侧等。在这种情况下，我们说TARGET\_EXPR是“正常”;否则，我们说它是“孤儿”。对于正常的TARGET\_EXPR，临时变量应该被视为赋值左侧的别名，而不是作为一个新的临时变量。

TARGET\_EXPR的第三个操作数（如果存在）是临时的清除表达式（即析构函数调用）。如果此表达式是孤立的，则当包含此表达式的语句完成时，必须执行此表达式。这些清理必须始终以与遇到它们相反的顺序执行。注意，如果在条件运算符的一个分支（即，在COND\_EXPR的第二或第三操作数中）创建了临时，则只有当该分支被实际执行时，才必须运行清除。

**VA\_ARG\_EXPR**

此节点用于实现对C/C ++变量参数列表机制的支持。它表示类似va\_arg（ap，type）的表达式。它的TREE\_TYPE产生类型的树表示，其唯一的参数产生ap的表示。

#### 5.5 向量

**VEC\_LSHIFT\_EXPR**

**VEC\_RSHIFT\_EXPR**

这些节点分别表示整个向量左和右移位。第一个操作数是要移位的向量;它将永远是矢量类型。第二操作数是用于移位的位数的表达式。请注意，如果第二个操作数大于或等于第一个操作数的类型大小，则结果未定义。

**VEC\_WIDEN\_MULT\_HI\_EXPR**

**VEC\_WIDEN\_MULT\_LO\_EXPR**

这些节点分别表示两个输入向量的高部分和低部分的加宽向量乘法。它们的操作数是包含相同整数类型的相同数量的元素（N）的向量。结果是包含一半数量的元素的矢量，其是尺寸是两倍宽的整数类型。在VEC\_WIDEN\_MULT\_HI\_EXPR的情况下，两个向量的高N / 2个元素相乘以产生N / 2个乘积的向量。在VEC\_WIDEN\_MULT\_LO\_EXPR的情况下，两个向量的低N / 2个元素相乘以产生N / 2个乘积的向量。

**VEC\_UNPACK\_HI\_EXPR**

**VEC\_UNPACK\_LO\_EXPR**

这些节点分别表示输入向量的高部分和低部分的拆包。单个操作数是包含相同积分或浮点类型的N个元素的向量。结果是一个包含一半数量的元素的向量，它是一个整数或浮点类型，其大小是宽度的两倍。在VEC\_UNPACK\_HI\_EXPR的情况下，提取和扩展（提升）矢量的高N / 2个元素。在VEC\_UNPACK\_LO\_EXPR的情况下，提取和扩展（提升）向量的低N / 2个元素。

**VEC\_UNPACK\_FLOAT\_HI\_EXPR**

**VEC\_UNPACK\_FLOAT\_LO\_EXPR**

这些节点表示输入向量的高部分和低部分的解包，其中值从固定点转换为浮点。单个操作数是包含相同整数类型的N个元素的向量。结果是一个矢量，其中包含一半的浮点类型的元素，其大小是宽度的两倍。在VEC\_UNPACK\_HI\_EXPR的情况下，向量的高N / 2元素被提取，转换和加宽。在VEC\_UNPACK\_LO\_EXPR的情况下，提取，转换和加宽向量的低N / 2元素。

**VEC\_PACK\_TRUNC\_EXPR**

该节点表示将两个输入向量的截断元素打包到输出向量中。输入操作数是包含相同数量的相同积分或浮点类型的元素的向量。结果是一个矢量，其中包含两倍的整数或浮点类型的元素，其大小是宽度的一半。两个向量的元素被降级和合并（级联）以形成输出向量。

**VEC\_PACK\_SAT\_EXPR**

该节点表示使用饱和将两个输入向量的元素打包到输出向量中。输入操作数是包含相同数量的具有相同积分类型的元素的向量。结果是一个矢量，其中包含两倍的整数类型的元素，其大小是宽度的一半。两个向量的元素被降级和合并（级联）以形成输出向量。

**VEC\_PACK\_FIX\_TRUNC\_EXPR**

该节点表示将两个输入向量的元素打包到输出向量中，其中值从浮点转换到固定点。输入操作数是包含相同数量的浮点类型的元素的向量。结果是一个矢量，其中包含两倍的整数类型的元素，其大小是宽度的一半。两个向量的元素被合并（连接）以形成输出向量。

**VEC\_COND\_EXPR**

这些节点表示？：表达式。三个操作数必须是相同大小和数量的元素的向量。第二和第三操作数必须具有与整个表达式相同的类型。第一个操作数是带符号的整数向量类型。如果第一个操作数的元素计算为零值，则结果的相应元素取自第三个操作数。如果它求值为一个值，则从第二个操作数取值。它从来不应该评估为任何其他值，但优化不应该依赖于该属性。与COND\_EXPR相反，所有操作数总是被评估。

### 6 语句

GIMPLE中的大多数语句是赋值语句，由GIMPLE\_ASSIGN表示。 在语句级别不能显示其他C表达式; 对volatile对象的引用将转换为GIMPLE\_ASSIGN。  
还有几种不同的复杂语句。

#### 6.1 基本语句

**ASM\_EXPR**

用于表示内联汇编语句。对于内联汇编语句如：

  asm（“mov x，y”）;

ASM\_STRING宏将为“mov x，y”返回一个STRING\_CST节点。如果原始语句使用扩展汇编语法，则ASM\_OUTPUTS，ASM\_INPUTS和ASM\_CLOBBERS将是该语句的输出，输入和clobbers，表示为STRING\_CST节点。扩展汇编语法如下所示：

  asm（“fsinx％1，％0”：“= f”（result）：“f”（angle）

第一个字符串是ASM\_STRING，包含指令模板。接下来的两个字符串分别是输出和输入;这个语句没有破绽。如本例所示，“plain”汇编语句仅仅是扩展汇编语句的一种特殊情况;他们没有cv限定符，输出，输入或破坏。所有的字符串将被NUL终止，并且将不包含嵌入的NUL字符。

如果汇编语句被声明为volatile，或者该语句不是扩展汇编语句，因此是隐式易变的，则谓词ASM\_VOLATILE\_P将保存ASM\_EXPR。

**DECL\_EXPR**

用于表示本地声明。 DECL\_EXPR\_DECL宏可以用于获取声明的实体。此声明可以是LABEL\_DECL，表示声明的标签是本地标签。 （作为扩展，GCC允许声明具有范围的标签。）在C中，该声明可以是FUNCTION\_DECL，指示使用GCC嵌套函数扩展。有关详细信息，请参阅函数。

**LABEL\_EXPR**

用于表示标签。此语句声明的LABEL\_DECL可以使用LABEL\_EXPR\_LABEL宏获取。提供标签名称的IDENTIFIER\_NODE可以从带有DECL\_NAME的LABEL\_DECL中获取。

**GOTO\_EXPR**

用于表示goto语句。 GOTO\_DESTINATION通常为LABEL\_DECL。但是，如果使用了“computed goto”扩展名，GOTO\_DESTINATION将是一个表示目的地的任意表达式。此表达式将始终具有指针类型。

**RETURN\_EXPR**

用于表示返回语句。操作数0表示要返回的值。它应该是包含函数的RESULT\_DECL，或者是设置函数的RESULT\_DECL的MODIFY\_EXPR或INIT\_EXPR。如果语句是just，它将是NULL\_TREE

   return;

**LOOP\_EXPR**

这些节点表示“无限”循环。 LOOP\_EXPR\_BODY表示循环的主体。它应该永远执行，除非遇到EXIT\_EXPR。

**EXIT\_EXPR**

这些节点表示从最近的包围LOOP\_EXPR的条件退出。单个操作数是条件;如果它是非零，那么应该退出循环。 EXIT\_EXPR只会出现在LOOP\_EXPR中。

**SWITCH\_STMT**

用于表示switch语句。 SWITCH\_STMT\_COND是发生切换的表达式。有关用于条件的表示的更多信息，请参阅IF\_STMT的文档。 SWITCH\_STMT\_BODY是switch语句的主体。在任何编译器转换之前，SWITCH\_STMT\_TYPE是源中给出的开关表达式的原始类型。

**CASE\_LABEL\_EXPR**

用于表示案例标签，案例标签范围或默认标签。如果CASE\_LOW为NULL\_TREE，那么这是一个默认标签。否则，如果CASE\_HIGH为NULL\_TREE，那么这是一个普通的案例标签。在这种情况下，CASE\_LOW是给出标签值的表达式。 CASE\_LOW和CASE\_HIGH都是INTEGER\_CST节点。这些值将与switch语句中的条件表达式具有相同的类型。

否则，如果定义了CASE\_LOW和CASE\_HIGH，则语句是一系列案例标签。这样的语句起源于允许用户编写以下形式的扩展：

  case 2 ... 5:

第一个值为CASE\_LOW，第二个值为CASE\_HIGH。

#### 6.2 块（Blocks）

块范围及其在GENERIC中声明的变量使用BIND\_EXPR代码表示，BIND\_EXPR代码在先前版本的GCC中主要用于C语句表达式扩展。

块中的变量通过其TREE\_CHAIN字段以声明顺序收集到BIND\_EXPR\_VARS中。任何运行时初始化都从DECL\_INITIAL中移出并进入受控块中的语句。当从C或C ++ Gimplifying时，此初始化将替换DECL\_STMT。这些变量永远不需要清理。这些变量的范围只是主体。可变长度数组（VLA）使这个过程复杂化，因为它们的大小通常指的是在块中较早初始化的变量。为了处理这个，我们当前分割块，并将VLA移动到一个新的内部BIND\_EXPR。这个策略可能在将来改变。

C++程序通常比源代码中的句法块包含更多的BIND\_EXPR，因为几个C++结构具有与它们相关联的隐式作用域。另一方面，尽管C++前端使用伪scope来处理具有析构函数的对象的清除，但是这些不会转换为GIMPLE形式;在同一级别的多个声明使用相同的BIND\_EXPR。

#### 6.3 语句序列

同一嵌套级别的多个语句被收集到STATEMENT\_LIST中。 语句列表使用'tree-iterator.h'中的接口修改和遍历。

#### 6.4 空语句

只要可能，就会丢弃没有效果的语句。 但是如果它们嵌套在另一个由于某种原因不能被丢弃的构造中，则它们将被一个由语句build\_empty\_stmt生成的空语句替换。 最初，所有空语句都是共享的，后面的模式是Java前端，但这在实践中造成了很多麻烦。

空语句表示为（void）0。

#### 6.5 跳转

其他跳转由GOTO\_EXPR或RETURN\_EXPR表示。

GOTO\_EXPR的操作数必须是标签或包含要跳转到的地址的变量。

RETURN\_EXPR的操作数为NULL\_TREE，RESULT\_DECL或设置返回值的MODIFY\_EXPR。 将MODIFY\_EXPR移动到单独的语句是很好的，但是expand\_return中的特殊返回语义使得这很困难。 它可能仍然发生在未来，也许通过将大部分逻辑转移到expand\_assignment。

#### 6.6 清除

用于本地C ++对象和类似动态清除的析构函数在GIMPLE中由TRY\_FINALLY\_EXPR表示。 TRY\_FINALLY\_EXPR有两个操作数，两者都是要执行的语句序列。执行第一个序列。当完成时，执行第二序列。

第一序列可以以下列方式完成：

1.执行序列中的最后一条语句，并从末尾开始。

2.对序列外部的普通标签执行goto语句（GOTO\_EXPR）。

3.执行return语句（RETURN\_EXPR）。

4.抛出异常。这在GIMPLE中目前没有明确表示。

如果第一个序列通过调用setjmp或exit或任何其他不返回的函数完成，则不会执行第二个序列。如果第一序列通过非局部goto或计算的goto完成，则第二序列也不被执行（一般来说，编译器不知道这样的goto语句是否退出第一序列，因此我们假定它不是）。

在第二个序列被执行后，如果它通过掉落结束而正常完成，则执行继续到第一序列将继续的任何地方，通过脱落结束或执行goto等。

TRY\_FINALLY\_EXPR使流程图复杂化，因为清除需要出现在受控块的每个边上;这减少了跨越这些边缘移动代码的自由。因此，在大多数优化遍之前运行的EH降低遍次通过明确地将清除添加到每个边缘来消除这些表达式。 Rethrowing异常使用RESX\_EXPR表示。

#### 6.7 OpenMP

所有以OMP\_开头的语句表示OpenMP API http://www.openmp.org/使用的指令和子句。

**OMP\_PARALLEL**

表示#pragma omp parallel [clause1 ... clauseN]。它有四个操作数：

操作数OMP\_PARALLEL\_BODY在GENERIC和高GIMPLE表单中有效。它包含要由所有线程执行的代码体。在GIMPLE降低期间，此操作数变为NULL，并且在OMP\_PARALLEL之后线性发出正文。

操作数OMP\_PARALLEL\_CLAUSES是与指令相关联的子句的列表。

操作数OMP\_PARALLEL\_FN由pass\_lower\_omp创建，它包含将包含并行区域主体的函数的FUNCTION\_DECL。

操作数OMP\_PARALLEL\_DATA\_ARG也由pass\_lower\_omp创建。如果有要传递给子线程的共享变量，则此操作数将包含包含所有共享值和变量的VAR\_DECL。

**OMP\_FOR**

表示[clause1 ... clauseN]的#pragma omp。它有5个操作数：

操作数OMP\_FOR\_BODY包含循环体。

操作数OMP\_FOR\_CLAUSES是与指令相关联的子句的列表。

操作数OMP\_FOR\_INIT是形式为VAR = N1的循环初始化代码。

操作数OMP\_FOR\_COND是形式为VAR {<，>，<=，> =} N2的循环条件表达式。

操作数OMP\_FOR\_INCR是形式为VAR {+ =， - =} INCR的循环索引增量。

操作数OMP\_FOR\_PRE\_BODY包含操作数OMP\_FOR\_INIT，OMP\_FOR\_COND和OMP\_FOR\_INC的副作用代码。这些副作用是OMP\_FOR块的一部分，但必须在循环体开始之前进行评估。

循环索引变量VAR必须是有符号整数变量，它是每个线程的隐式私有变量。边界N1和N2以及增量表达式INCR必须是在不进行任何同步的情况下求值的循环不变整数表达式。标准没有规定评价顺序，评价频率和副作用。

**OMP\_SECTIONS**

表示#pragma omp sections [clause1 ... clauseN]。

操作数OMP\_SECTIONS\_BODY包含段主体，它又包含由#pragma omp段分隔的每个并发段的一组OMP\_SECTION节点。

操作数OMP\_SECTIONS\_CLAUSES是与指令相关联的子句的列表。

**OMP\_SECTION**

OMP\_SECTIONS的段定界符。

**OMP\_SINGLE**

表示#pragma omp单。

操作数OMP\_SINGLE\_BODY包含要由单个线程执行的代码体。

操作数OMP\_SINGLE\_CLAUSES是与指令相关联的子句的列表。

**OMP\_MASTER**

表示#pragma omp主控。

操作数OMP\_MASTER\_BODY包含要由主线程执行的代码主体。

**OMP\_ORDERED**

表示#pragma omp有序。

操作数OMP\_ORDERED\_BODY包含要按循环索引变量指示的顺序执行的代码体。

**OMP\_CRITICAL**

表示#pragma omp critical [name]。

操作数OMP\_CRITICAL\_BODY是关键部分。

操作数OMP\_CRITICAL\_NAME是用于标记关键部分的可选标识符。

**OMP\_RETURN**

这不表示任何OpenMP指令，它是一个人工标记来指示OpenMP的主体的结束。它由流图（tree-cfg.c）和OpenMP区域构建代码（omp-low.c）使用。

**OMP\_CONTINUE**

类似地，此指令不表示OpenMP指令，它由OMP\_FOR和OMP\_SECTIONS用来标记代码需要循环到下一次迭代的位置（在OMP\_FOR的情况下）或下一个部分（在OMP\_SECTIONS的情况下） 。

在某些情况下，OMP\_CONTINUE位于OMP\_RETURN之前。但是如果有清除需要发生在循环体之后，它将在OMP\_CONTINUE和OMP\_RETURN之间发射。

**OMP\_ATOMIC**

表示#pragma omp atomic。

操作数0是要执行原子操作的地址。

操作数1是要评估的表达式。 gimplifier尝试三个替代代码生成策略。只要可能，就使用原子更新内置。如果失败，尝试比较和交换循环。如果这也失败，使用表达式周围的常规临界区。

**OMP\_CLAUSE**

表示与某个OMP\_指令相关联的子句。子句由tree.h中定义的单独的子代码表示。第代码可以是之一：OMP\_CLAUSE\_PRIVATE，OMP\_CLAUSE\_SHARED，OMP\_CLAUSE\_FIRSTPRIVATE，OMP\_CLAUSE\_LASTPRIVATE，OMP\_CLAUSE\_COPYIN，OMP\_CLAUSE\_COPYPRIVATE，OMP\_CLAUSE\_IF，OMP\_CLAUSE\_NUM\_THREADS，OMP\_CLAUSE\_SCHEDULE，OMP\_CLAUSE\_NOWAIT，OMP\_CLAUSE\_ORDERED，OMP\_CLAUSE\_DEFAULT，OMP\_CLAUSE\_REDUCTION，OMP\_CLAUSE\_COLLAPSE，OMP\_CLAUSE\_UNTIED，OMP\_CLAUSE\_FINAL和OMP\_CLAUSE\_MERGEABLE。每个代码表示相应的OpenMP子句。

与相同指令相关的条目通过OMP\_CLAUSE\_CHAIN链接在一起。那些接受变量列表的子句被限制为一个，通过OMP\_CLAUSE\_VAR访问。因此，同一子句C下的多个变量需要表示为多个C子句链接在一起。这有助于在编译期间添加新子句。

### 7 函数

函数由FUNCTION\_DECL节点表示。 它存储函数的基本部分，如主体，参数和返回类型，以及关于周围上下文，可见性和链接的信息。

函数基础：函数名，主体和参数。

功能属性：上下文，链接等

#### 7.1 函数基础

函数有四个核心部分：名称，参数，返回值和函数体（body）。以下宏和函数访问FUNCTION\_DECL的这些部分以及其他基本功能：

**DECL\_NAME**

此宏返回函数的非限定名称，如IDENTIFIER\_NODE。对于函数模板的实例化，DECL\_NAME是模板的非限定名称，不是f <int>。当用于构造函数，析构函数，重载运算符或类型转换运算符或由编译器隐式生成的任何函数时，DECL\_NAME的值未定义。请参阅下面的宏，可用于区分这些情况。

**DECL\_ASSEMBLER\_NAME**

此宏返回函数的标记名称，也是IDENTIFIER\_NODE。此名称在包含所有标识符下划线的系统上不包含前导下划线。在所有平台上以相同的方式计算该标记名称;如果需要特殊处理来处理在特定平台上使用的目标文件格式，则后端负责执行那些修改。 （当然，后端不应该修改DECL\_ASSEMBLER\_NAME本身。）

使用DECL\_ASSEMBLER\_NAME将导致分配额外的内存（对于实体的受损名称），因此只应在发出汇编代码时使用它。不应该在优化器中使用它来确定两个声明是否相同，即使一些现有的优化器确实以这种方式使用它。这些用途将随时间推移而消除。

**DECL\_ARGUMENTS**

此宏返回函数的第一个参数的PARM\_DECL。可以通过跟随TREE\_CHAIN链接获得后续PARM\_DECL节点。

**DECL\_RESULT**

此宏返回函数的RESULT\_DECL。

**DECL\_SAVED\_TREE**

此宏返回函数的完整主体。

**TREE\_TYPE**

此宏返回函数的FUNCTION\_TYPE或METHOD\_TYPE。

**DECL\_INITIAL**

在当前翻译单元中具有定义的函数将具有非NULL DECL\_INITIAL。但是，后端不应该使用DECL\_INITIAL给出的特定值。

它应该包含一个BLOCK节点树，它反映变量在函数中绑定的范围。每个块包含在基本块中声明的decls列表，指向下一个较低范围级别的块链的指针，然后是指向同一级别的下一个块的指针，以及到父BLOCK或FUNCTION\_DECL的backpointer。所以给定一个函数如下：

void foo()

{

int a;

{

int b;

}

int c;

}

你会得到以下：

tree foo = FUNCTION\_DECL;

tree decl\_a = VAR\_DECL;

tree decl\_b = VAR\_DECL;

tree decl\_c = VAR\_DECL;

tree block\_a = BLOCK;

tree block\_b = BLOCK;

tree block\_c = BLOCK;

BLOCK\_VARS(block\_a) = decl\_a;

BLOCK\_SUBBLOCKS(block\_a) = block\_b;

BLOCK\_CHAIN(block\_a) = block\_c;

BLOCK\_SUPERCONTEXT(block\_a) = foo;

BLOCK\_VARS(block\_b) = decl\_b;

BLOCK\_SUPERCONTEXT(block\_b) = block\_a;

BLOCK\_VARS(block\_c) = decl\_c;

BLOCK\_SUPERCONTEXT(block\_c) = foo;

DECL\_INITIAL(foo) = block\_a;

#### 7.2 函数属性

要确定函数的范围，可以使用DECL\_CONTEXT宏。此宏将返回函数为其成员的类（RECORD\_TYPE或UNION\_TYPE）或命名空间（NAMESPACE\_DECL）。对于虚函数，此宏返回实际定义函数的类，而不是返回虚函数声明的基类。

在C中，函数的DECL\_CONTEXT可能是另一个函数。此表示表示正在使用GNU嵌套函数扩展。有关嵌套函数的语义的详细信息，请参阅GCC手册。嵌套函数可以引用其包含函数中的局部变量。这样的引用没有在树结构中被明确标记;后端必须查看引用的VAR\_DECL的DECL\_CONTEXT。如果引用的VAR\_DECL的DECL\_CONTEXT与当前正在处理的函数不同，并且既不是DECL\_EXTERNAL也不是TREE\_STATIC保持，则引用是包含函数中的局部变量，后端必须采取适当的操作。

**DECL\_EXTERNAL**

如果函数未定义，此谓词成立。

**TREE\_PUBLIC**

如果函数具有外部链接，则此谓词成立。

**TREE\_STATIC**

如果函数已定义，则此谓词成立。

**TREE\_THIS\_VOLATILE**

如果函数未正常返回，则此谓词成立。

**TREE\_READONLY**

如果函数只能读取其参数，则此谓词成立。

**DECL\_PURE\_P**

如果函数只能读取其参数，但也可以读取全局内存，则此谓词成立。

**DECL\_VIRTUAL\_P**

如果函数是虚函数，则此谓词成立。

**DECL\_ARTIFICIAL**

如果函数由编译器隐式生成，而不是显式声明，则此宏成立。除了隐式生成的类成员函数之外，此宏还适用于为实现静态初始化和销毁​​而创建的特殊函数，用于计算运行时类型信息等。

**DECL\_FUNCTION\_SPECIFIC\_TARGET**

此宏返回一个树节点，该节点保存用于编译此特定函数的目标选项，如果要使用命令行中指定的目标选项编译该函数，则返回NULL\_TREE。

**DECL\_FUNCTION\_SPECIFIC\_OPTIMIZATION**

此宏返回一个树节点，该节点保存用于编译此特定函数或NULL\_TREE的优化选项，如果要使用命令行中指定的优化选项编译该函数。

### 8 语言依赖树

前端可能希望在解析时保持与各种GENERIC树相关联的某种状态。 为了支持这一点，树提供了可以由前端使用的一组标志。 它们使用TREE\_LANG\_FLAG\_n访问，其中“n”当前为0到6。

如果有必要，前端可以在其GENERIC表示中使用一些语言相关的树代码，只要它提供了一个钩子将它们转换为GIMPLE，并且不期望它们与在转换之前运行的任何（假设的）优化器一起工作 到GIMPLE。 在解析C和C ++时使用的中间表示看起来很像GENERIC，但是C和C++ gimplifier钩子非常乐意将其作为输入并吐出GIMPLE。

### 9 C 和 C++ 树

本节记录了GCC用来表示C和C ++源程序的内部表示。 当呈现C或C ++源程序时，GCC解析程序，执行语义分析（包括错误消息的生成），然后产生这里描述的内部表示。 该表示包含作为前端的输入提供的整个翻译单元的完整表示。 然后，这种表示通常由代码生成器处理以便产生机器代码，但是也可以用于创建源浏览器，智能编辑器，自动文档生成器，解释器以及需要处理C的能力的任何其他程序 C ++代码。

本节介绍内部表示。特别地，它记录了C和C ++源结构的内部表示，以及可以用于访问这些结构的宏，函数和变量。 C ++表示主要是C前端中使用的表示的超集。在C中只有一个不在C ++前端中使用的构造，它是GNU的“嵌套函数”扩展。这里记载的许多宏不适用于C，因为相应的语言结构不出现在C.

C和C ++前端生成一个GENERIC树和C和C ++的混合树。这些特定于语言的树是比GENERIC中的更高级的结构，以使解析器的工作更容易。本节描述不属于GENERIC的那些树以及以特定语言方式处理的GENERIC树的各个方面。

如果你正在开发一个“后端”，无论是代码生成器还是其他工具，使用这种表示，你可能偶尔发现，你需要提出问题不容易回答这里提供的功能和宏。如果发生这种情况，很可能GCC已经支持你想要的功能，但是接口在这里没有记录。在这种情况下，您应该请GCC维护者（通过邮件gcc@gcc.gnu.org）记录您需要的功能。同样，如果你发现自己写的函数不直接处理你的后端，而是可能对使用GCC前端的其他人有用，你应该提交补丁包括在GCC。

C ++的类型：基本类型和聚合类型。

命名空间：命名空间。

类：类。

C ++的函数：C ++的重载和访问器。

C ++的语句：C和C ++特有的语句。

C ++表达式：从typeid到throw。

#### 9.1 C ++的类型

#### 9.2