第 3 章

GCC 总体结构

GCC 是一个复杂的软件系统，例如 gcc-4.4.0.tar.gz 软件包中包含了成千上万个文件。本章主要对 GCC 的代码结构和目录结构进行介绍，阐明 GCC 的主要模块及其相互关系，并给出 GCC 源代码编译的主要步骤和关键问题。

3.1 GCC 的目录结构

GCC 的源代码可以从 GCC 的官网（https://gcc.gnu.org）上获得。该源代码包主要包括 bz2 和 gz 两种压缩形式的 tar 包，以 gcc-4.4.0 为例，分别为 gcc-4.4.0.tar.bz2 及 gcc-4.4.0.tar.gz。

可以通过如下的命令获取 gcc-4.4.0.tar.bz2 代码，进行源代码包的解压，并查看其主要的目录结构。

```
    [GCC@host2 gcc-4.4.0]$ wget -c http://www.netgull.com/gcc/releases/gcc-4.4.0/gcc-
4.4.0.tar.bz2
    [GCC@host2 gcc-4.4.0]$ tar -xjvf gcc-4.4.0.tar.bz2
    [GCC@host2 gcc-4.4.0]$ cd gcc-4.4.0; ls
    ABOUT-NLS              COPYING.LIB      libgfortran      MAINTAINERS
    boehm-gc               COPYING.RUNTIME  libgomp          maintainer-scripts
    ChangeLog              depcomp          libiberty        Makefile.def
    ChangeLog.tree-ssa     fixincludes      libjava          Makefile.in
    compile                gcc              libmudflap       Makefile.tpl
    config                 gnattools        libobjc          MD5SUMS
    config.guess           include          libssp           missing
    config-ml.in           INSTALL          libstdc++-v3     mkdep
    config.rpath           install-sh       libtool-ldflags  mkinstalldirs
    config.sub             intl             libtool.m4       move-if-change
    configure              LAST_UPDATED     ltgcc.m4         NEWS
    configure.ac           libada           ltmain.sh        README
    contrib                libcpp           lt~obsolete.m4   symlink-tree
    COPYING                libdecnumber     ltoptions.m4     tags
    COPYING3               libffi           ltsugar.m4       ylwrap
    COPYING3.LIB           libgcc           ltversion.m4     zlib
```

该源代码目录中的主要内容包括：

（1）与 GCC 编译配置有关的 config* 文件。

（2）lib* 目录：各种各样的库文件，既包括一些通用的库文件，也包含一些与语言相关的库文件，例如 libcpp 中包含与 C++ 语言相关的代码库文件，libada 中包含与 ADA 语言相关的代码库文件。

（3）gcc 目录中包含 GCC 的核心代码，包括了与各种编程语言相关的词法、语法等前端分析程序，与各种目标机器相关的机器描述文件，以及与前端语言无关且与机器无关的核心处理代码等。

使用如下 shell 命令可以列出 gcc 目录中的所有子目录，其中包含如下的一些子目录：

```
[GCC@host2 gcc-4.4.0]$ ls -l gcc | grep ^d
drwxrwxr-x.  3 GCC GCC   69632 Apr 21  2009 ada
drwxrwxr-x. 37 GCC GCC    4096 Apr 21  2009 config
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 cp
drwxrwxr-x.  3 GCC GCC    4096 Apr 21  2009 doc
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 fortran
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 ginclude
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 java
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 objc
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 objcp
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 po
drwxrwxr-x. 18 GCC GCC    4096 Apr 21  2009 testsuite
```

gcc 目录下的 gcc/cp、gcc/fortran、gcc/java、gcc/objc、gcc/objcp 等子目录就是与各种编程语言相关的处理部分，这几个目录分别处理编程语言 C++、Fortran、Java、Object C、Object C++ 等，C 语言的处理则是 GCC 默认的处理前端语言，其部分处理代码在 gcc/ 目录中。

进一步查看 gcc/config 目录中所包含的子目录：

```
[GCC@host2 gcc-4.4.0]$ ls -l gcc/config | grep ^d
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 alpha
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 arc
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 arm
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 avr
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 bfin
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 cris
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 crx
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 fr30
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 frv
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 h8300
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 i386
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 ia64
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 iq2000
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 m32c
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 m32r
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 m68hc11
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 m68k
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 mcore
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 mips
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 mmix
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 mn10300
```

```
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 pa
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 pdp11
drwxrwxr-x. 3 GCC GCC  4096 Apr 21  2009 picochip
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 rs6000
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 s390
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 score
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 sh
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 soft-fp
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 sparc
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 spu
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 stormy16
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 v850
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 vax
drwxrwxr-x. 2 GCC GCC  4096 Apr 21  2009 xtensa
```

从目录的名称上就可以看出来，这些目录分别对应了各种不同的目标机器名称。目录中包含的内容就是针对不同目标机器的机器描述文件，包括 md 文件及相应的 c 文件和 h 文件等。例如 i386 目录中包含了 Intel x86 处理器的机器描述文件等，arm 目录中则包含了 ARM 处理器的机器描述文件等。

完整的目录结构说明请查阅 GCC 相关说明文档。也可以参考 Uday Khedker 的《GCC Source Code: An Internal View》(http://www.cse.iitb.ac.in/grc/)。

3.2 GCC 的逻辑结构

GCC 的源代码文件数量庞大，目录结构复杂，总体结构理解有一定的难度，但从代码功能和逻辑结构上来讲，这些代码大致可以分为如图 3-1 所示的几个部分。

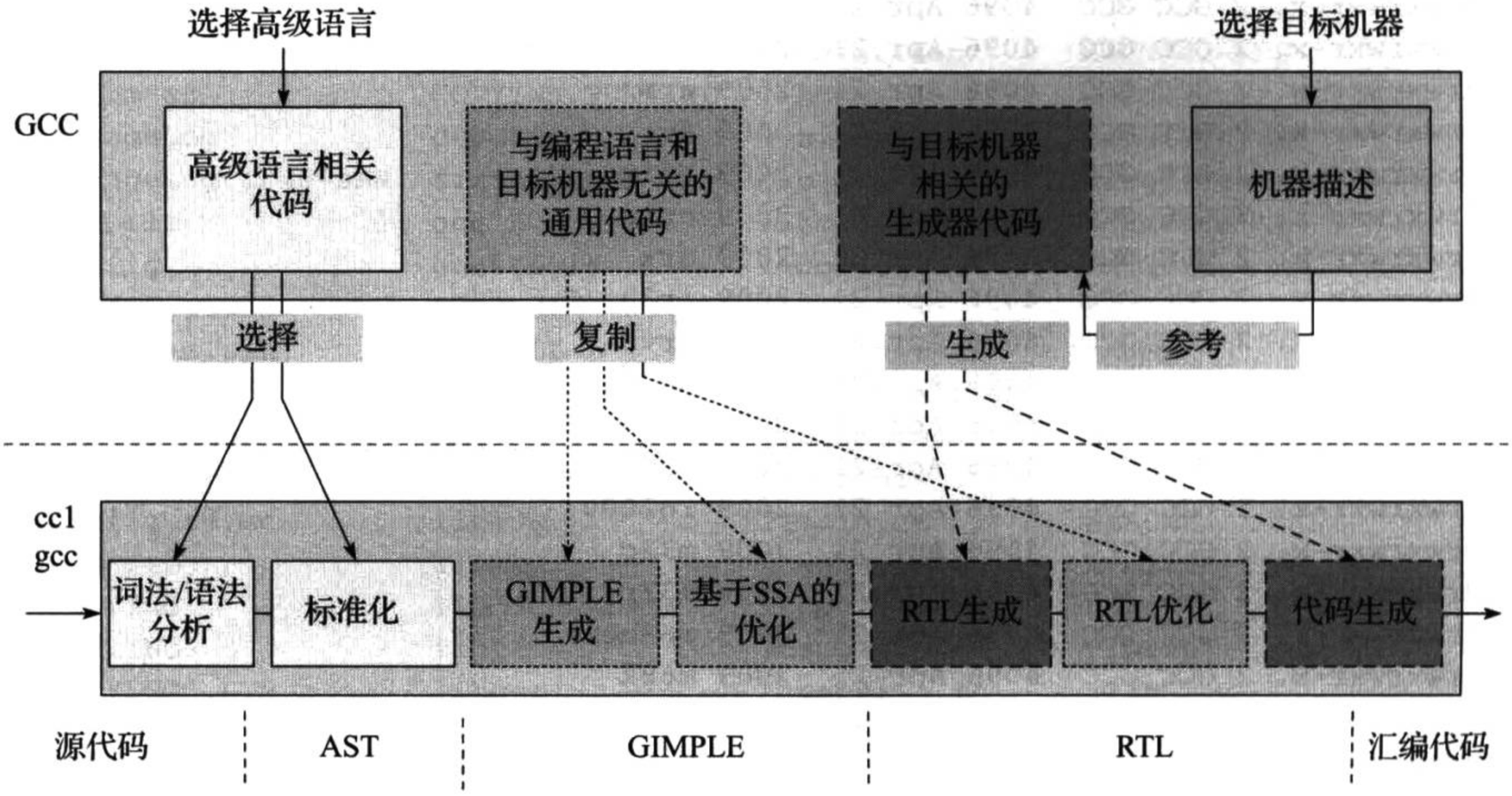

图 3-1 GCC 与 gcc 的逻辑结构

图 3-1 分为上下两个部分，上半部分使用 GCC 表示 GCC 4.4.0 的源代码内容，下半部分使用 gcc/cc1 表示使用 GCC 源代码编译生成的编译器程序。

图 3-1 的上半部分根据源代码的功能将 GCC 源代码分为 4 大部分：

（1）高级语言相关代码（High-Level-Language Specific Code）。在 GCC 的源代码中，对于 GCC 能够编译的每一种编程语言都有其相应的处理代码，这些代码主要集中在 ${GCC_SOURCE}/${Language} 目录下。其中 ${Language} 代表了编程语言的名称，这部分代码主要完成高级编程语言的词法、语法分析等功能，从而生成该语言对应的抽象语法树（AST，Abstract Syntax TREE），并完成其规范化（Genericize）操作。

（2）与编程语言和目标机器无关的通用代码（Language & Machine Independent Generic Code）。这部分代码主要包括 ${GCC_SOURCE}/ 目录下的代码，用于完成 GIMPLE 和 RTL 的生成，以及数量庞大的基于 GIMPLE 和 RTL 的处理及编译优化工作。

（3）机器描述（Machine Descriptions）代码。一般来说，对于 GCC 支持的每一种名称为 ${target} 的目标机器，在 GCC 的代码中均有一个名称为 ${GCC_SOURCE}/config/${target} 的子目录，用来存放与该目标机器相关的机器描述代码及其相应的头文件和 c 文件等。

（4）与目标机器相关的生成器代码（Machine Dependent Generator Code）。这部分代码比较难以理解，读者可以试着这样来考虑。为了生成目标机器上编译器程序 cc1，GCC 提供的源代码在设计阶段是不完整的，其中缺少的部分主要包括目标机器相关的 RTL 构造及目标代码生成等部分的源代码。由于这一部分源代码是与目标机器相关的，在 GCC 设计源代码时是难以确定的，因此，GCC 采用了这样一种解决的思路，就是通过一些生成器（Generator）代码，这些代码能够根据目标机器的机器描述文件，提取目标机器的信息，从而自动地生成关于目标机器上 RTL 构造及目标代码生成的源代码，并将这些源代码与 GCC 原有的其他代码结合在一起编译，从而生成与目标机器相关的编译器程序。与目标机器相关的生成器代码的文件名称一般为 ${GCC_SOURCE}/gen*.[ch]，其主要的功能就是根据机器描述文件生成与目标机器相关的部分源代码。

因此，最终参与编译，生成目标机器编译器的源代码主要包括了语言相关的代码、语言及机器无关的通用代码，以及根据机器描述文件由机器相关代码生成器所生成的代码等三部分。

图 3-1 的下半部分给出了根据上述 GCC 的源代码所生成的目标机器上编译器 cc1（gcc 程序所调用的编译器）的主要工作流程。从整体上看，目标机器上编译器 cc1 的功能就是将用户输入的高级程序代码最终编译成目标机器上的汇编代码，其中经历了前端的词法分析、语法分析、语义分析，中间的 GIMPLE 生成、GIMPLE 优化，以及后端的 RTL 生成、RTL 优化、代码生成等几个步骤。在这些处理过程中，GCC 也分别使用几种不同的中间表示（Intermediate Representation，IR）形式，包括 AST、GIMPLE、RTL 等。这些处理步骤与上半部分的代码具有一定的对应关系，例如词法、语法分析以及 AST 的规范化过程对应上半部分的“高级语言相关代码”；GIMPLE 生成、GIMPLE 优化及 RTL 优化部分则对应上半部分

的“与编程语言和目标机器无关的代码”；RTL 生成以及最终的汇编代码生成部分则由上半部分的“与目标机器相关的生成器代码”根据上半部分的“机器描述”生成。

对图 3-1 的上半部分和下半部分进行对照，可以看出不同部分的 GCC 源代码在功能上的差异。

本书在分析 GCC 时，也是按照 cc1 的执行流程，围绕各种中间表示的生成和处理进行深入分析，从而帮助读者理解 GCC 设计的关键思路和技术，主要包括：

第 4 章主要以 C 语言为例，介绍 GCC 前端对于高级语言进行词法、语法分析，从而生成其 AST 的过程，重点描述了其中 AST 的表示、存储结构及其操作等。

第 5 章主要描述 GIMPLE 中间表示的生成过程。

第 6 章主要描述基于 GIMPLE 中间表示的各种编译优化，这些优化大多是基于静态单赋值（Static Single Assignment，SSA）形式的 GIMPLE 表示，而且都是与目标机器无关的优化。

第 7 章详细地介绍了 GCC 中 RTL 中间表示的基本概念，并对其类型、存储以及操作做了详细描述。

第 8 章主要介绍 GCC 中机器描述文件 ${target}.md 的指令模板的基本概念及其主要内容，并对机器描述文件中 define_insn、define_expand、define_split、define_peephole 等主要操作进行了详细的描述和实例说明，这些内容对于理解机器描述文件和用户机器描述文件至关重要。

第 9 章主要对 GCC 中机器描述文件的 c 文件和头文件进行了详细描述。这些内容也为第 12 章的 GCC 向新处理器的移植做了充分的准备。另外，9.9 节则重点介绍了与目标机器相关的生成器代码的结构及其作用。

第 10 章主要描述 RTL 中间表示的生成技术。

第 11 章主要描述基于 RTL 中间表示的优化技术，这些优化大部分是与目标机器相关的。

第 12 章重点给出将 GCC 移植到新的处理器的基本过程和实例。

关于 GCC 代码的结构，也可以参考 Abhijat Vichare 的《GCC-conceptual-structure》（http://www.cse.iitb.ac.in/grc/）。

3.3 GCC 源代码编译

在获得了 GCC 的源代码后，为了生成目标机器上的编译器程序，需要对源代码进行编译，一般步骤包括：

（1）使用 configure 脚本完成编译配置，生成 Makefile 文件。

（2）使用 make 工具编译源代码。

（3）使用 make 工具安装生成的编译程序等。

使用的典型脚本为：

```
./configure
```

```
make
make install
```

3.3.1 配置

这个过程一般很简单，可以直接在 ${GCC_SOURCE} 目录下使用命令 ./configure。该脚本会对 GCC 源代码编译、安装的环境进行检查，并根据 configure 脚本的参数对编译环境进行配置，从而最终生成 Makefile 文件，作为后续 make 工具进行编译和安装的依据。

然而，稍微仔细分析后会发现，这又是一个非常令人烦恼的过程，原因是这个配置命令具有较多的选项。可以使用 ./configure --help 查看这些配置选项及其主要的功能说明。

下面对编译配置中的一些主要选项进行简单的说明。

（1）安装目录（Installation directories）选项：

- --prefix=PREFIX：用来指定目标机器无关代码的安装目录，默认值为 /usr/local。
- --exec-prefix=EPREFIX：用来指定目标机器相关代码的安装目录，一般与 --prefix 选项指定的 PREFIX 值相同。

（2）程序名称（Program names）：

- --program-prefix=PREFIX：设置安装程序名的前缀为 PREFIX。
- --program-suffix=SUFFIX：设置安装程序名的后缀为 SUFFIX。

例如，如果使用如下命令进行配置：

```
[GCC@host1 gcc-4.4.0]$ ./configure --program-prefix=prefix-
```

那么最终生成的 gcc 编译程序的名称将变成：/usr/local/bin/*prefix*-gcc，其中 *prefix*-gcc 里面的“*prefix*-”就是由 --program-prefix=*prefix*- 选项所指定的。

（3）系统类型（System types）：

用来描述生成、运行及目标系统的系统类型，通常由 --build、--host 及 --target 三个选项指定，其中：

- build=BUILD：指定生成（build）编译器程序的机器和操作系统平台信息。
- host=HOST：指定生成的编译器程序所运行的机器和操作系统平台信息，默认值与 BUILD 相同。
- target=TARGET：指定生成的编译器程序生成代码所运行的机器和操作系统平台信息，默认值与 HOST 相同。

关于这几个选项的指定，通常会有如下的几种组合方式：

1）BUILD、HOST、TARGET 三者相同，这一般表示在本机进行 GCC 源代码的编译，生成的编译器程序 GCC 也运行在与本机相同的机器和操作系统平台下，并且生成的编译器编译出的目标程序也将运行在同样的系统环境中。

2）HOST 与 TARGET 相同，但 HOST 与 BUILD 不同，这是一种典型的生成交叉编译工具的情况，即在 BUILD 主机环境上编译 GCC 源代码，生成的编译器程序将运行在 HOST 主

机环境中，并且该编译器程序编译生成的目标文件也将运行在 HOST，即 TARGET 环境中。

3）BUID、HOST 及 TARGET 各不相同，这是一种最复杂的情况，也属于一种生成交叉编译工具的情况，即在 BUILD 主机环境上编译 GCC 源代码，生成的编译器程序将运行在 HOST 主机环境中，并且该编译器程序编译生成的目标文件将运行在另一种 TARGET 机器环境中。

例 3-1　通过 Makefile 文件查看 build、host 及 target 系统的配置值

假设本例运行的主机信息如下：

```
[GCC@host1 gcc-4.4.0]$ uname -a
Linux host1 2.6.32-71.el6.i686 #1 SMP Fri Nov 12 04:17:17 GMT 2010 i686 i686 i386
GNU/Linux
```

第一种情况，在 gcc-4.4.0 源代码目录中直接执行 ./configure，那么在生成 Makefile 文件中包含了如下信息：

```
build=i686-pc-linux-gnu
host=i686-pc-linux-gnu
target=i686-pc-linux-gnu
```

这种情况下，configure 脚本会自动根据系统的信息“猜测”出 build 的值，例如本例中看到的 build 的值为“ i686-pc-linux-gnu”，而 host 的默认值与 build 相同，target 的默认值与 host 相同。此时三者的值都是相同的，这就是上述系统类型 1）中的组合方式。

第二种情况，假如要在本机上对 GCC 源代码进行编译，生成的编译器程序也运行在本机上，但编译器需要为 arm 机器生成运行代码，此时，编译配置可以如下：

```
[GCC@host1 gcc-4.4.0]$ ./configure --target=arm-linux-gnu
```

或者：

```
[GCC@host1 gcc-4.4.0]$ ./configure --build=i686-pc-linux-gnu --target=arm-linux-gnu
```

生成的 Makefile 文件都包含了如下内容：

```
build=i686-pc-linux-gnu
host=i686-pc-linux-gnu
target=arm-unknown-linux-gnu
```

此时 build 与 host 相同，即用来编译 GCC 的机器和将来要运行编译器的机器类型是相同的，但是编译器生成的代码却不在 host 主机系统上运行，而是要运行在一种 arm-unknown-linux-gnu 机器上，这就是一种典型的交叉编译。

（4）可选编译特性的运行与禁止。

一般使用 --enable-FEATURE[=ARG] 或者 --disable-FEATURE 表示设置该 FEATURE 的值为 ARG，或者禁止该特性。

（5）编译时可选的一些包的配置。

一般使用 --with-PACKAGE[=ARG] 或者 --without-PACKAGE 来指定包 PACKAGE 的值

或者禁止使用该包。例如使用 --with-mpfr =PATH 指定 mpfr 包的目录。

（6）编译时的环境变量。

这些环境变量可能包括编译、汇编、链接等工具以及这些工具运行时的选项值，例如：

```
./configure CFLAGS="-w -g -O0" --prefix=/opt/paag-gcc --target=paag-linux-gnu --enable-stage1-language=c
```

其中，CFLAGS="-w -g -O0" 就指定了编译标志 CLAGS 的部分值。表 3-1 给出了 configure 脚本中常用环境变量的名称和意义。

表 3-1 configure 中常见的环境变量设置

变量名称	意　义	备　注
CC	编译程序	指定 BUILD 机器上的编译、链接等选项
CPPFLAGS	C/C++ 编译预处理选项	
CFLAGS	C 编译选项	
LDFLAGS	链接器选项	
LD	HOST 上的 ld 程序	指定 HOST 机器上的编译、链接等选项
AR	HOST 上的 ar 程序	
AS	HOST 上的 as 程序	
NM	HOST 上的 nm 程序	
RANLIB	HOST 上的 ranlib 程序	
STRIP	HOST 上的 strip 程序	
OBJCOPY	HOST 上的 objcopy 程序	
CC_FOR_TARGET	TARGET 上的 cc 程序	指定 TARGET 机器上的编译、链接等选项
GCC_FOR_TARGET	TARGET 上的 gcc 程序	
AR_FOR_TARGET	TARGET 上的 ar 程序	
AS_FOR_TARGET	TARGET 上的 as 程序	
LD_FOR_TARGET	TARGET 上的 ld 程序	
NM_FOR_TARGET	TARGET 上的 nm 程序	
OBJDUMP_FOR_TARGET	TARGET 上的 objdump 程序	
RANLIB_FOR_TARGET	TARGET 上的 ranlib 程序	
STRIP_FOR_TARGET	TARGET 上的 strip 程序	

3.3.2 编译

执行 ./configure 命令后，就生成了 gcc-4.4.0 目录下的 Makefile 文件。此时，可以使用 make 工具来进行整个 GCC 源代码的编译过程。这个过程的执行异常复杂，为了了解整个编译的详细过程，建议读者将编译的过程重定向到文件中，并结合 Makefile 文件进行仔细研读。例如，可以采用如下命令形式：

```
[GCC@host1 gcc-4.4.0]$ ./configure CFLAGS="-g -O0" --prefix=/opt/i386 --target=i386-linux-gnu
```

```
[GCC@host1 gcc-4.4.0]$ make > make-process 2&>1
```

GCC 源代码编译生成一个在本机运行的编译器时，通常采用一个叫做 bootstrapping 的技术，即将 GCC 源代码的编译过程分为多个阶段，通常包括 stage1、stage2 及 stage3 三个阶段。这三个阶段的功能分别为：

（1）stage1：使用一个现有的编译器将 GCC 的源代码编译，生成一个新的编译器 new_gcc1；

（2）stage2：使用 new_gcc1 重新编译 GCC 的源代码，生成另外一个新的编译器 new_gcc2；

（3）stage3：使用 new_gcc2 重新编译 GCC 的源代码，生成另外一个新的编译器 new_gcc3，并且对 new_gcc2 和 new_gcc3 进行比较，如果两者相同，则表示 GCC 源代码编译成功，否则表示生成的编译器存在 bug。

采用上述过程的目的是充分保证编译器的正确性。详细内容可以参考如下文档：

- http://stackoverflow.com/questions/9429491/how-are-gcc-g-bootstrapped
- https://gcc.gnu.org/install/build.html

可以通过前面生成的 make-process 文件中的部分内容验证上述说法。

```
[GCC@localhost gcc-4.4.0]$ make &> make-process
[GCC@localhost gcc-4.4.0]$ grep stage make-process
[ -f stage_final ] || echo stage3 > stage_final
Configuring stage 1 in host-i686-pc-linux-gnu/intl
Configuring stage 1 in host-i686-pc-linux-gnu/gcc
Configuring stage 1 in host-i686-pc-linux-gnu/libiberty
Configuring stage 1 in host-i686-pc-linux-gnu/zlib
Configuring stage 1 in host-i686-pc-linux-gnu/libcpp
Configuring stage 1 in host-i686-pc-linux-gnu/libdecnumber
Configuring stage 1 in i686-pc-linux-gnu/libgcc
rm -f stage_current
Configuring stage 2 in host-i686-pc-linux-gnu/intl
Configuring stage 2 in host-i686-pc-linux-gnu/gcc
Configuring stage 2 in host-i686-pc-linux-gnu/libiberty
Configuring stage 2 in host-i686-pc-linux-gnu/zlib
Configuring stage 2 in host-i686-pc-linux-gnu/libcpp
Configuring stage 2 in host-i686-pc-linux-gnu/libdecnumber
Configuring stage 2 in i686-pc-linux-gnu/libgcc
rm -f stage_current
Configuring stage 3 in host-i686-pc-linux-gnu/intl
Configuring stage 3 in host-i686-pc-linux-gnu/gcc
Configuring stage 3 in host-i686-pc-linux-gnu/libiberty
Configuring stage 3 in host-i686-pc-linux-gnu/zlib
Configuring stage 3 in host-i686-pc-linux-gnu/libcpp
Configuring stage 3 in host-i686-pc-linux-gnu/libdecnumber
Configuring stage 3 in i686-pc-linux-gnu/libgcc
rm -f stage_current
Comparing stages 2 and 3
```

以下是各个阶段生成的 GCC 文件的对比。

使用现有编译器生成的 xgcc（即 new_gcc1）：

```
[GCC@localhost host-i686-pc-linux-gnu]$ ls stage1-gcc/xgcc -l
-rwxrwxr-x. 1 GCC GCC 499979 Aug 30 15:30 stage1-gcc/xgcc
```

使用 new_gcc1 重新编译 GCC 的源代码，生成另外一个新的编译器 xgcc（即 new_gcc2）：

```
[GCC@localhost host-i686-pc-linux-gnu]$ ls prev-gcc/xgcc -l
-rwxrwxr-x. 1 GCC GCC 453766 Aug 30 15:44 prev-gcc/xgcc
```

使用 new_gcc2 重新编译 GCC 的源代码，生成另外一个新的编译器 xgcc（即 new_gcc3）：

```
[GCC@localhost host-i686-pc-linux-gnu]$ ls gcc/xgcc -l
-rwxrwxr-x. 1 GCC GCC 453766 Aug 30 15:49 gcc/xgcc
```

可以看出，两个阶段生成的编译器 gcc/xgcc 与 prev-gcc/xgcc 的大小相同。

另外，还可以使用 diff 工具对两个编译器目标文件进行对比，从命令的结果可以看出两者是完全相同的，因此，本次 GCC 源代码的编译是成功的。

```
[GCC@localhost host-i686-pc-linux-gnu]$ diff gcc/xgcc prev-gcc/xgcc
```

3.3.3 安装

GCC 源代码成功编译后，生成的可执行程序及文档等的安装可以使用如下命令进行：

```
make install
```