automake 中文手册收藏

GNU Automake

For version 1.3, 3 April 1998

David MacKenzie and Tom Tromey

目录

·其它派生对象
o 可执行的脚本
o 头文件
o 与体系结构无关（Architecture-independent）的数据文件
o 已创建的源代码
·其它 GNU 工具
oEmacs Lisp
oGettext
oGuile
oLibtool
oJava
·创建文档
oTexinfo
oMan 手册
·安装了些什么
·清除了些什么
·需要发布哪些文件
·对测试套件（test suites）的支持
·改变 Automake 的行为
·其它规则
o 与 etags 之间的界面
o 处理新的文件扩展名
·条件（Conditionals）
·--gnuand --gnits 的效果
·--cygnus 的效果
·什么时候 Automake 不够用
·发布`Makefile.in'
·未来的某些想法
·索引

@dircategory GNU admin @direntry * automake: (automake). Making Makefile.in's @dircategory Individual utilities @direntry * aclocal: (automake)Invoking aclocal. Generating aclocal.m4

这是 GNU Automake 文档的第一版，

并且是针对 GNU Automake 1.3 的。

本文档由王立翻译。 1999.12.17
译者在此声明：不对任何由译文错误或者对译文的误解承担任何责任。

介绍

Automake 是一个从文件`Makefile.am'自动生成`Makefile.in'的工具。每个`Makefile.am'基本上是一系列 make 的宏定义（make 规则也会偶尔出现）。生成的`Makefile.in's 服从 GNU Makefile 标准。

GNU Makefile 标准文档（参见 GNU 编码标准中的‘Makefile 惯例’节）长、复杂，而且会发生改变。Automake 的目的就是解除个人 GNU 维护者维护 Makefile 的负担（并且让 Automake 的维护者来承担这个负担）。

典型的 Automake 输入文件是一系列简单的宏定义。处理所有这样的文件以创建 `Makefile.in'。在一个项目（project）的每个目录中通常包含一个 `Makefile.am'。

Automake 在几个方面对一个项目做了限制；例如它假定项目使用 Autoconf （参见 Autoconf 手册），并且对`configure.in'的内容施加了某些限制。

为生成`Makefile.in'，Automake 需要 perl。但是由 Automake 创建的发布完全服从 GNU 标准，并且在创建中不需要 perl。

请把关于 Automake 的建议和 bug 发送到 automake-bugs@gnu.org。

通用性概念

一些基本概念将有助于理解 Automake 是如何工作的。

通用操作

Automake 读入`Makefile.am'并且生成`Makefile.in'。在`Makefile.am'中定义的一些宏和目标（targets）指挥 automake 生成更多特定的代码；例如一个`bin_PROGRAMS'宏定义将生成一个需要被编译、连接的目标。

`Makefile.am'中的宏定义和目标被复制到生成的文件中。这使得你可以把任何代码添加到生成的`Makefile.in'文件中。例如，Automake 的发布包含了非标准的 cvs-dist 目标，Automake 的维护者用它从他的版本控制系统中创建发布版本。

Automake 不能识别 GNU 对 make 的扩展。在`Makefile.am'中使用这些扩展将导致错误或者令人不解的行为。

Automake 试图明智地把注释和相邻的目标（或者变量定义）关联起来。

在`Makefile.am'中定义的目标通常覆盖了所有由 automake 自动生成的拥有相似名字的目标。虽然 Automake 提供了这一功能，但最好避免使用它，因为有些时候生成的规则将是十分特别的。

类似地，在`Makefile.am'中定义的变量将覆盖任何通常由 automake 创建的变量定义。该功能比覆盖目标定义的功能要常用得多。需要警告的是许多由 automake 生成的变量都被认为是内部使用的，并且它们的名字可能在未来的版本中改变。

在检验变量定义的时候，Automake 将递归地检验定义中的变量引用。例如，如果 Automake 在如下片断中搜索`foo_SOURCES'的内容。

```
xs = a.c b.c
foo_SOURCES = c.c $(xs)
```

它将把文件`a.c'、 `b.c'和 `c.c'作为`foo_SOURCES'的内容。

Automake 还允许给出不被复制到输出中的注释；所有以`##'开头的行将被 Automake 彻底忽略。

作为惯例，`Makefile.am'的第一行是：

```
## Process this file with automake to produce Makefile.in
```

深度

automake 支持三种目录层次： “flat” 、 “shallow” 和 “deep” 。

一个 flat（平）包指的是所有文件都在一个目录中的包。为这类包提供的`Makefile.am'缺少宏 SUBDIRS。这类包的一个例子是 termutils。

一个 deep（深）包指的是所有的源代码都被储存在子目录中的包；顶层目录主要包含配置信息。GNU cpio 是这类包的一个很好的例子，GNU tar 也是。deep 包的顶层`Makefile.am'将包括宏 SUBDIRS，但没有其它定义需要创建的对象的宏。

一个 shallow（浅）包指的是主要的源代码储存在顶层目录中，而各个部分（典型的是库）则储存在子

目录中的包。Automake 本身就是这类包（GNU make 也是如此，它现在已经不使用 automake）。

严格性

Automake 的目的是用于维护 GNU 包，它为适应那些希望使用它的人做出了一些努力，但并不指望应用所有的 GNU 惯例。

按照这个目标，Automake 支持三级严格性---严格性指的是 Automake 将如何检查包所服从的标准。

可用的严格性级别有：

`foreign'（外来）

Automake 将仅仅检查那些为保证正确操作所必需的事项。例如，尽管 GNU 标准指出文件`NEWS'必须存在，在本方式下，并不需要它。该模式名来自于 Automake 是被设计成用于 GNU 程序的事实的；它放松了标准模式的操作规则。

`gnu'

Automake 将尽可能地检查包是否服从 GNU 标准。这是缺省设置。

`gnits'

Automake 将按照还没有完成的 Gnits 标准进行检查。它们是基于 GNU 标准的，但更加详尽。除非你是 Gnits 标准的参与奉献者，我们建议您在 Gnits 标准正式出版之前不要使用这一选项。

关于严格性级别的精确含义的详细说明，参见--gnu 和--gnits 的效果

统一命名机制

Automake 变量通常服从统一的命名机制，以易于确定如何创建和安装程序（和其它派生对象）。这个机制还支持在运行 configure 的时候确定应该创建那些对象。

在运行 make 时，某些变量被用于确定应该创建那些对象。这些变量被称为主（primary）变量。例如，主变量 PROGRAMS 保存了需要被编译和连接的程序的列表。

另一组变量用于确定应该把创建了的对象安装在哪里。这些变量在主变量之后命名，但是含有一个前缀以指出那个标准目录将作为安装目录。标准目录名在 GNU 标准中给出（参见 GNU 编码标准中的`为 Directory Variables'节）。 Automake 用 pkglibdir、pkgincludedir 和 pkgdatadir 扩展了这个列表；除了把`@PACKAGE@'附加其后之外，与非`pkg'版本是相同的。例如，pkglibdir 被定义为$(datadir)/@PACKAGE@.

对于每个主变量，还有一个附加的变量，它的名字是在主变量名之前加一个`EXTRA_'。该变量用于储存根据 configure 的运行结果，可能创建、也可能不创建的对象列表。引入该变量是因为 Automake 必须静态地知道需要创建的对象的完整列表以创建在所有情况下都能够工作的`Makefile.in'。

例如，在配置时刻 cpio 确定创建哪些程序。一部分程序被安装在 bindir，还有一部分程序被安装在 sbindir：

```
EXTRA_PROGRAMS = mt rmt
bin_PROGRAMS = cpio pax
sbin_PROGRAMS = @PROGRAMS@
```

定义没有前缀的主变量（比如说 PROGRAMS）是错误的。

在构造变量名的时候，通常省略后缀`dir'；因此我们使用 `bin_PROGRAMS'而不是 `bindir_PROGRAMS'.

不是每种对象都可以安装在任何目录中。Automake 将记录它们以试图找出错误。Automake 还将诊断目录名中明显的拼写错误。

有时标准目录--即使在 Automake 扩展之后---是不够的。特别在有些时候，为了清晰起见，把对象安装到预定义目录的子目录中是十分有用的。为此，Automake 允许你扩展可能的安装目录列表。如果定义了一个添加了后缀`dir'的变量（比如说`zardir'），则给定的前缀（比如`zar'）就是合法的。

例如，在 HTML 支持成为 Automake 的一部分之前，你可以使用它安装原始的 HTML 文档。

```
htmldir = $(prefix)/html
html_DATA = automake.html
```

特殊前缀`noinst'表示根本不会安装这些有问题的对象。

特殊前缀`check'表示仅仅在运行 make check 命令的时候才创建这些有问题的对象。

可能的主变量名有`PROGRAMS'、`LIBRARIES'、 `LISP'、`SCRIPTS'、`DATA'、 `HEADERS'、`MANS'和`TEXINFOS'。

派生变量是如何命名的

有时 Makefile 变量名是从用户提供的某些文本中派生而来的。例如程序名被重写到 Makefile 宏名中。Automake 把这些文本规范化， 以使它可以不必服从 Makefile 的变量名规则。在名字中除了字母、数字和下划线之外的所有字符都将用下划线代替。例如，如果你的程序被命名为 sniff-glue，那么派生出的变量名将是 sniff_glue_SOURCES，而不是 sniff-glue_SOURCES。

一些实例软件包

一个简单的例子，从起点到终点

让我们假定你刚刚写完 zardoz，一个是你的头从一个漩涡漂流到另一个漩涡的程序。你已经使用了 autoconf 以提供一个可移植的框架，但你的`Makefile.in'还未完成，所以你需要 automake。

第一步是更新你的`configure.in'以包含 automake 需要的命令。完成这一步的最简单方式是在 AC_INIT 之后添加 AM_INIT_AUTOMAKE：

```
AM_INIT_AUTOMAKE(zardoz, 1.0)
```

因为你的程序不含有任何复杂性的因素（例如，它不使用 gettext，它不需要共享库），你已经完成了这一步工作。很容易吧！

现在你必须重新生成`configure'。但为此，你需要告诉 autoconf 如何找到你使用的新宏。完成该任务的最简单的方式是使用 aclocal 程序为你生成你的`aclocal.m4'。但是等等...你已经有了一个`aclocal.m4'，这是因为你必须为你的程序写一些宏。aclocal 允许你把你自己的宏放到

`acinclude.m4'中去，所以简单地改名并且运行：
mv aclocal.m4 acinclude.m4
aclocal
autoconf

现在是你为 zardoz 写的`Makefile.am'的时候了。 zardoz 是一个用户程序，所以你需要把它安装到其它用户程序安装的地方去。 zardoz 还有一些 Texinfo 文档。你的`configure.in'脚本使用 AC_REPLACE_FUNCS，因此你需要与`@LIBOBJS@'连接。所以这里你写：
bin_PROGRAMS = zardoz
zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c
zardoz_LDADD = @LIBOBJS@

info_TEXINFOS = zardoz.texi

现在你运行 automake --add-missing 以生成你的`Makefile.in'并且得到任何你可能需要的附加文件，现在你完成了你的任务！
一个经典的程序
hello 因为它经典的简单性和多用性而出名。本节展示 Automake 将被如何用于 Hello 包。下面的例子来自于最新的 GNU Hello，但剔除了所有仅为维护者使用的代码和所有的版权注释。
当然，GNU Hello 比您的传统的两行的代码具有更多的特征。GNU Hello 是国际化的，进行选项处理，并且含有一个手册和一个测试套件。GNU Hello 是一个 deep 包。
这里是来自于 GNU Hello 的`configure.in'：
dnl Process this file with autoconf to produce a configure script.
AC_INIT(src/hello.c)
AM_INIT_AUTOMAKE(hello, 1.3.11)
AM_CONFIG_HEADER(config.h)

dnl Set of available languages.
ALL_LINGUAS="de fr es ko nl no pl pt sl sv"

dnl Checks for programs.
AC_PROG_CC
AC_ISC_POSIX

dnl Checks for libraries.

```
dnl Checks for header files.
AC_STDC_HEADERS
AC_HAVE_HEADERS(string.h fcntl.h sys/file.h sys/param.h)

dnl Checks for library functions.
AC_FUNC_ALLOCA

dnl Check for st_blksize in struct stat
AC_ST_BLKSIZE

dnl internationalization macros
AM_GNU_GETTEXT
AC_OUTPUT([Makefile doc/Makefile intl/Makefile po/Makefile.in \
        src/Makefile tests/Makefile tests/hello],
  [chmod +x tests/hello])
```

宏`AM_'由 Automake（或者 Gettext 库）提供；其它的是标准 Autoconf 宏。
顶层`Makefile.am'：

```
EXTRA_DIST = BUGS ChangeLog.O
SUBDIRS = doc intl po src tests
```

就像你所见到的，这里的所有工作实际上都是在子目录中完成的。
`po'和`intl'目录是 gettextize 自动生成的；在这里我们不讨论它们。
在`doc/Makefile.am'中我们看到：

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = gpl.texi
```

它足以创建、安装并且发布 Hello 手册。
这里是`tests/Makefile.am':

```
TESTS = hello
EXTRA_DIST = hello.in testdata
```

脚本`hello'是由 configure 生成的，并且仅仅在测试时才生成。 make check 将运行这个测试。
最后我们有`src/Makefile.am'，所有实际工作在此完成：

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
hello_LDADD = @INTLLIBS@ @ALLOCA@
localedir = $(datadir)/locale
INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"
```

创建 etags 和 ctags

这里是另一个复杂一些的例子。它展示了如何从同一个源文件（`etags.c'）生成两个程序（ctags 和 etags）。困难的部分是对 `etags.c'的每个编译需要不同的 cpp 选项。

```
bin_PROGRAMS = etags ctags
ctags_SOURCES =
ctags_LDADD = ctags.o

etags.o: etags.c
	$(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
	$(COMPILE) -DCTAGS -o ctags.o -c etags.c
```

其中 ctags_SOURCES 被定义为空--这种方式表明没有替换隐含的值然而，隐含的值被用于从 `etags.o'生成 etags。

ctags_LDADD 用于把`ctags.o'添加到连接行中。 ctags_DEPENDENCIES 由 Automake 生成。

如果你的编译器不接受`-c'和`-o'，那么上述规则将不能工作。对此，最简单的修正是引入伪依赖（bogus dependency）（以避免由并行 make 所导致的问题）：

```
etags.o: etags.c ctags.o
	$(COMPILE) -DETAGS_REGEXPS -c etags.c

ctags.o: etags.c
	$(COMPILE) -DCTAGS -c etags.c && mv etags.o ctags.o
```

同样，如果使用了 de-ANSI-fication 的特征，这些显式规则将不能工作；支持它需要一些更多的工作：

```
etags._o: etags._c ctags.o
	$(COMPILE) -DETAGS_REGEXPS -c etags.c
```

```
ctags._o: etags._c
	$(COMPILE) -DCTAGS -c etags.c && mv etags._o ctags.o
```

创建`Makefile.in'

为了为一个包创建所有的`Makefile.in'，在顶层目录不带任何参数地运行 automake。 automake 将自动地寻找每个合适的`Makefile.am'（通过扫描 `configure.in'；参见扫描`configure.in'）并生成相应的`Makefile.in'。automake 认为包的组成是相当简单的；它假定包仅仅在顶层目录含有一个 `configure.in'。如果你的包含有多个 `configure.in'，那么你必须在每个含有`configure.in'的目录中运行 automake。

你偶尔可能需要给 automake 参数；`.am'被添加到参数之后并且其结果将作为输入文件名。该特征通常仅仅用于自动重新创建一个过时的 `Makefile.in'。automake 必须总是在项目的最顶层目录中运行，即使用于重新生成某些子目录中的`Makefile.in'也是如此。这是因为 automake 必须扫描 `configure.in'，并且因为在某些情况下，automake 根据`Makefile.in'在子目录中这一情况确定它的行为。

automake 接受以下选项：

-a

--add-missing

Automake 要求一些通用文件在特定的位置存在。例如如果`configure.in'运行了 AC_CANONICAL_HOST，就需要`config.guess'。 Automake 与几个这样的文件一同发布；只要可能，该选项将把缺少的文件自动添加到包中。通常如果 Automake 告诉你缺少文件，试一下本选项。

--amdir=dir

在 dir 中而不是安装目录中，寻找 Automake 数据文件，它通常用于调试。

--build-dir=dir

告诉 Automake 创建目录在那里。本选项在把依赖性添加到由 make dist 生成的 `Makefile.in'中的时候使用；在其它情况下不应该使用它。

--cygnus

按照 Cygnus 规则，而不是 GNU 或者 Gnits 规则，生成`Makefile.in'，详情请参见--cygnus 的效果。

--foreign

把全局严格性设置成`foreign'。详情请参见严格性。

--gnits

把全局严格性设置成`gnits'。详情请参见 --gnu 和--gnits 的效果

--gnu

把全局严格性设置成`gnu'。详情请参见 --gnu 和--gnits 的效果。这是缺省严格性。

--help
打印命令行选项的概述并且退出。
-i
--include-deps
包含生成的`Makefile.in'中所有自动生成的依赖信息（参见自动的依赖性跟踪）。通常在制作发布版本时使用；参见需要发布哪些文件。
--generate-deps
生成一个连接了所有自动生成的依赖信息的文件（参见 自动的依赖性跟踪）文件，`.dep_segment'。通常在制作发布版本时使用；参见需要发布哪些文件。在维护为其它平台所制作的`SMakefile'或者makefile（`Makefile.DOS'，等等，）时是有用的。它只能与--include-deps、--srcdir-name和--build-dir 一同使用。如果给出了本选项，不会实行任何其他处理。
--no-force
通常 automake 创建在`configure.in'中提到的所有 `Makefile.in'。本选项仅仅更新那些按照它们的依赖性过时了的 `Makefile.in'。
-o dir
--output-dir=dir
把生成的`Makefile.in'放到目录 dir 中。通常每个`Makefile.in'在对应的`Makefile.am'所在的目录中创建。本选项被用于创建发布版本。
--srcdir-name=dir
告诉 Automake 与当前任务相关的源代码目录名。本选项在把依赖性引入由 make dist 生成的`Makefile.in'中时使用；它不应被用于其它情况。
-v
--verbose
让 Automake 打印关于被读入或创建的文件的信息。
--version
打印 Automake 的版本号并且退出。
扫描`configure.in'
Automake 扫描包的`configure.in'以确定关于包的一些信息。 Automake 需要一些 autoconf 宏并且一些变量必须在 `configure.in'中定义。Automake 还用来自`configure.in'的信息以进一步修整它的输出。
为了简化维护，Automake 还支持一些 autoconf 宏。通过使用程序 aclocal，可以自动地把这些宏附加到你的`aclocal.m4'中。
配置需求
达到 Automake 要求的最简单方式就是使用宏 AM_INIT_AUTOMAKE（参见由 Automake 支持的 Autoconf 宏）。但是如果你愿意，你可以手工完成所需的各个步骤：

·用 AC_SUBST 定义变量 PACKAGE 和 VERSION。 PACKAGE 应该是在捆绑发布的时候使用的包的名称。例如，Automake 把 PACKAGE 定义成`automake'。 VERSION 应该是被开发的版本的版本号。我们建议你仅仅在你的包中定义版本号的地方创建`configure.in'；这使得发布简单化了。 除非在`Gnits'模式（参见--gnu 和--gnits 的效果），Automake 不会对 PACKAGE 或者 VERSION 进行任何解释。
·如果要安装一个程序或者一个脚本，使用宏 AC_ARG_PROGRAM。
·如果包不是平（flat）的，使用宏 AC_PROG_MAKE_SET。
·使用宏 AM_SANITY_CHECK 以确认创建环境的完整性。
·如果包安装了任何脚本（参见可执行的脚本），使用宏 AM_PROG_INSTALL。否则，使用 AC_PROG_INSTALL。
·使用 AM_MISSING_PROG 以确认在创建环境中，程序 aclocal、 autoconf、automake、autoheader 和 makeinfo 是否存在。下面是如何完成这个任务：
·missing_dir=`cd $ac_aux_dir && pwd`
AM_MISSING_PROG(ACLOCAL, aclocal, $missing_dir)
AM_MISSING_PROG(AUTOCONF, autoconf, $missing_dir)
AM_MISSING_PROG(AUTOMAKE, automake, $missing_dir)
AM_MISSING_PROG(AUTOHEADER, autoheader, $missing_dir)
AM_MISSING_PROG(MAKEINFO, makeinfo, $missing_dir)

这里是 Automake 需要的，但是没有被 AM_INIT_AUTOMAKE 运行的其它宏：
AC_OUTPUT
Automake 用它确定创建那个文件。列出的名为 Makefile 的文件作为`Makefile'处理。对其它列出的文件进行不同的处理。目前唯一的不同是`Makefile'将被 make distclean 删除，而其它的文件将被 make clean 删除。
Automake 能够识别的其它事情
Automake 还将能够识别某些宏的使用并且适当地修整生成的`Makefile.in'。目前能够识别的宏以及它们的效果是：
AC_CONFIG_HEADER
Automake 要求使用 AM_CONFIG_HEADER，它类似于 AC_CONFIG_HEADER 而且完成一些有用的 Automake 特定的工作。
AC_CONFIG_AUX_DIR
Automake 将在调用本宏时命名的目录中寻找各种求助脚本，例如`mkinstalldirs'。如果没找到，将在其它标准的位置（顶层目录中，或者在对应与当前`Makefile.am'的源代码目录，任何一个都是合适的）中寻找脚本。 请帮助我：以给出寻找该目录的完整列表。
AC_PATH_XTRA

Automake 将把由 AC_PATH_XTRA 定义的变量的定义插入每个创建 C 程序或者库的 `Makefile.in'中。

AC_CANONICAL_HOST

AC_CHECK_TOOL

Automake 将确认`config.guess'和`config.sub'的存在。并且将引入`Makefile'变量 `host_alias'和 `host_triplet'。

AC_CANONICAL_SYSTEM

它类似于 AC_CANONICAL_HOST，此外还定义了`Makefile'变量`build_alias'和 `target_alias'。

AC_FUNC_ALLOCA

AC_FUNC_GETLOADAVG

AC_FUNC_MEMCMP

AC_STRUCT_ST_BLOCKS

AC_FUNC_FNMATCH

AM_FUNC_STRTOD

AC_REPLACE_FUNCS

AC_REPLACE_GNU_GETOPT

AM_WITH_REGEX

Automake 将确认为对应于这些宏的对象生成了适当的依赖关系。此外，Automake 将验证适当的源文件成为发布的一部分。使用这些宏，Automake 并不需要任何 C 源代码，所以 automake -a 将不会安装源代码。详情请参见创建一个库

LIBOBJS

Automake 将检测把`.o'文件添加到 LIBOBJS 中的语句，并且按照与在 AC_REPLACE_FUNCS 中发现的文件相同的方式处理这些附加的文件。

AC_PROG_RANLIB

如果在包中创建了任何库，就需要它。

AC_PROG_CXX

如果包含了任何 C++源代码，就需要它。

AM_PROG_LIBTOOL

Automake 将启动为 libtool 所做的处理（参见 Libtool 手册）。

AC_PROG_YACC

如果找到了 Yacc 源文件，那么你必须使用这个宏或者在`configure.in'中定义变量`YACC'。前者更好一些。

AC_DECL_YYTEXT

如果在包中有 Lex 源代码，需要使用这个宏。

AC_PROG_LEX

如果找到了 Lex 源代码，那么必须使用本宏。

ALL_LINGUAS

如果 Automake 发现在`configure.in'中设置了该变量，它将检查目录`po'以确认所有命名了的`.po'文件都是存在的，并且所有存在的`.po'文件都被命了名。

AM_C_PROTOTYPES

在使用自动 de-ANSI-fication 时，需要它。参见自动 de-ANSI-fication。

AM_GNU_GETTEXT

使用了 GNU gettext 的包需要使用本宏。（参见 Gettext）。它将与 gettext 一起发布。如果 Automake 看到这个宏，Automake 将确认包是否符合 gettext 的某些要求。

AM_MAINTAINER_MODE

该宏为 configure 添加一个`--enable-maintainer-mode'选项。如果使用了本宏，automake 将关闭在生成的 `Makefile.in'中缺省的“maintainer-only”规则。在`Gnits'模式中，不允许使用本宏。（参见--gnu 和--gnits 的效果）。

AC_SUBST

AC_CHECK_TOOL

AC_CHECK_PROG

AC_CHECK_PROGS

AC_PATH_PROG

AC_PATH_PROGS

上述任意一个宏的第一个参数将在每个生成的`Makefile.in'中自动地被定义为一个变量。

自动生成 aclocal.m4

Automake 包含了许多可以你的包中使用的的 Autoconf 宏；其中一些实际上是 Automake 在某些情况下需要的。你必须在你的`aclocal.m4'中定义这些宏；否则 autoconf 将不能找到它们。

程序 aclocal 将基于`configure.in'的内容自动生成文件 `aclocal.m4'。它提供了一个不必四处寻找而获得 Automake 提供的宏的便利方式。此外，aclocal 机制对使用它的其它包来说，是可以扩展的。

在启动时，aclocal 扫描所有它能够找到的`.m4'文件，以寻找宏定义。而后它扫描`configure.in'。任何在第一步中提到的宏，以及它所需要的宏，将被放到`aclocal.m4'中。

如果`acinclude.m4'存在，它的内容将被自动包含在 `aclocal.m4'中。这对于把本地宏合并到`configure'是有用的。

aclocal 接受如下选项：

--acdir=dir

在目录 dir 中，而不是在安装目录中，寻找宏文件。这通常用于调试。

--help

打印命令行选项的概述并且退出。

-I dir

把目录 dir 添加到搜索`.m4'的目录列表中。

--output=file

把输出储存到文件 file 中，而不是`aclocal.m4'中。

--print-ac-dir

打印 aclocal 将搜索`m4'文件的目录名。当给出本选项的时候，不实施通常的处理。包可以用本选项确定应该把宏文件安装到哪里。

--verbose

打印它所检测的文件名。

--version

打印 Automake 的版本号并且退出。

由 Automake 支持的 Autoconf 宏

AM_CONFIG_HEADER

Automake 将生成规则以自动地重新生成 config 头文件。如果你使用本宏，你必须在你的源代码目录中创建文件`stamp-h.in'。它可以为空。

AM_CYGWIN32

检查本 configure 是否是在`Cygwin32'环境中运行。（FIXME xref）。如果是，把输出变量 EXEEXT 定义为 `.exe'；否则，把它定义为空字符串。Automake 识别该宏并且用它生成在`Cygwin32'中可以自动工作的`Makefile.in'。在`Cygwin32'环境中，即使在命令行中没有明确指出，gcc 将生成文件名以`.exe'结尾的可执行文件。Automake 向 `Makefile.in'添加特定的代码以适当地处理它。

AM_FUNC_STRTOD

如果不能使用函数 strtod，或者不能正确地工作（例如在 SunOS 5.4 上），就把`strtod.o'添加到输出变量 LIBOBJS 中。

AM_FUNC_ERROR_AT_LINE

如果没有找到 error_at_line 函数，就把`error.o'添加到 LIBOBJS 中。

AM_FUNC_MKTIME

检查函数 mktime。如果没有找到，就把`mktime.o'添加到`LIBOBJS'中。

AM_FUNC_OBSTACK

检查 GNU obstacks 代码；如果没有找到，就把`obstack.o'添加到`LIBOBJS'中。

AM_C_PROTOTYPES

检查编译器是否可以识别函数原型。如果可以识别，就定义`PROTOTYPES'并且把输出变量`U'和`ANSI2KNR'设置为空。否则，把`U'设置成`_'，并且把`ANSI2KNR'设置成`./ansi2knr'。Automake 使用这些值以实现自动 de-ANSI-fication。

AM_HEADER_TIOCGWINSZ_NEEDS_SYS_IOCTL

如果使用 TIOCGWINSZ 需要`<sys/ioctl.h>'，那么定义 GWINSZ_IN_SYS_IOCTL。否则，TIOCGWINSZ 可以在`<termios.h>'中发现。

AM_INIT_AUTOMAKE

运行大部分`configure.in'需要的多个宏。本宏有两个参数，包名称和版本号。缺省情况下，本宏用 AC_DEFINE 定义`PACKAGE'和 `VERSION'。可以通过添加非空的第三个参数以避免这一行为。

AM_PATH_LISPDIR

搜索程序 emacs，并且，如果找到了，把输出变量 lispdir 设置为到 Emac 的 site-lisp 目录的完整路径。

AM_PROG_CC_STDC

如果 C 编译器的缺省状态不是标准 C（ANSI C），试图把一个选项添加到输出变量 CC 中以使得 C 编译器这样做。本宏尝试在各种系统中选择标准 C 的各种选项。如果编译器正确地处理函数原型，它就认为编译器处于标准 C 模式。 如果你使用本宏，你应该在调用它之后检查 C 编译器是否被设置成接受标准 C；如果不是，shell 变量 am_cv_prog_cc_stdc 被设置成`no'。如果你按照标准 C 写你的源代码，你可以利用 ansi2knr 选项创建它的非标准 C 版本。

AM_PROG_INSTALL

类似与 AC_PROG_INSTALL，但还定义了 INSTALL_SCRIPT。

AM_PROG_LEX

类似与带有 AC_DECL_YYTEXT 的 AC_PROG_LEX，但在没有 lex 的系统上使用脚本 missing。`HP-UX 10'是一个这样的系统。

AM_SANITY_CHECK

它检查并确保在创建目录中被创建的文件比源代码目录中的文件要新。在时钟设置不正确的系统中它可能失败。本宏在 AM_INIT_AUTOMAKE 中自动运行。

AM_SYS_POSIX_TERMIOS

检查系统中，是否可以使用 POSIX termios 头文件和函数。如果可以，就把 shell 变量 am_cv_sys_posix_termios 设置为`yes'。如果不能使用，就把 am_cv_sys_posix_termios 设置为`no'。

AM_TYPE_PTRDIFF_T

如果类型`ptrdiff_t'是在`<stddef.h>'中定义的，就定义`HAVE_PTRDIFF_T'。

AM_WITH_DMALLOC

增加 dmalloc 包支持。如果用户用`--with-dmalloc'进行配置，那么定义 WITH_DMALLOC 并且把`-ldmalloc'添加到 LIBS。包 dmalloc 可以在 ftp://ftp.letters.com/src/dmalloc/dmalloc.tar.gz 找到。

AM_WITH_REGEX

把`--with-regex'添加到 configure 的命令行中。如果给出`--with-regex'（缺省设置），那么使用

`regex'常规表达式库，`regex.o'被添加到`LIBOBJS'中，并且定义`WITH_REGEX'。如果给出`--without-regex'，那么使用`rx'常规表达式库，`rx.o'被添加到`LIBOBJS'中。

编写你自己的 aclocal 宏

aclocal 不含有任何宏的任何内置信息，所以扩展你自己的宏是十分容易的。

它通常被用于那些需要为使用它的其它程序提供它们自己的 Autoconf 宏的库。例如 gettext 库支持宏 AM_GNU_GETTEXT，该宏将被任何使用 gettext 的任何包所使用。在安装库的时候，它安装宏以便 aclocal 可以找到它。

一个宏文件应该是一系列 AC_DEFUN'。aclocal 还懂得 AC_REQUIRE，所以把每个宏储存在一个单独的文件中是安全的。

一个宏文件的文件名应该以`.m4'结尾。这类文件都应该安装在`$(datadir)/aclocal'中。

顶层`Makefile.am'

在非平（non-flat）包中，顶层`Makefile.am'必须告诉 Automake 应该在那个子目录中进行创建。这通过变量 SUBDIRS 来完成。

宏 SUBDIRS 保存了需要进行各种创建的子目录列表。在生成的`Makefile'中的许多目标（例如，all）即需要在本目录下运行，也需要在所有特定的子目录下运行。需要指出，出现在 SUBDIRS 中的子目录并不一定含有`Makefile.am'；只要在配置（运行 configuration）之后含有`Makefile'就行了。这使得你可以从不使用 Automake 的软件包（例如，gettext）中引入库。在 SUBDIRS 中提到的目录必须是当前目录的直接子目录。例如，你可以把`src/subdir'添加到 SUBDIRS 中。

在一个深（deep）包中，顶层`Makefile.am'通常十分简短。例如，下面是 Hello 发布版中的`Makefile.am'：

```
EXTRA_DIST = BUGS ChangeLog.O README-alpha
SUBDIRS = doc intl po src tests
```

就像在 GNU Inetutils 中的情况一样，如果你只需要创建整个包的一个子集，你可以覆盖变量 SUBDIRS。在你的`Makefile.am'中包含：

```
SUBDIRS = @SUBDIRS@
```

而后在你的`configure.in'中，你可以指明：

```
SUBDIRS = "src doc lib po"
AC_SUBST(SUBDIRS)
```

以上修改的结果是：automake 将创建包以获得 subdirs，但实际上在运行 configure 之前并没有把目录列表包括进来。

SUBDIRS 可以包含配置替换（例如，`@DIRS@'）；Automake 本身并不实际检查这个变量的内容。

如果定义了 SUBDIRS，那么你的`configure.in'必须包含 AC_PROG_MAKE_SET。
对 SUBDIRS 的使用并不限于顶层目录中的`Makefile.am'。 Automake 可以用于构造任意深度的包。

创建程序和库
Automake 的大部分功能的目的是使创建 C 程序和库变得容易些。
创建一个程序
在一个含有将被创建成一个程序（而不是创建成一个库）的源代码的目录中，要使用主变量`PROGRAMS'。程序可以安装到`bindir'、 `sbindir'、`libexecdir'、`pkglibdir'中，或者根本不安装（`noinst'）。
例如：

```
bin_PROGRAMS = hello
```

在这种简单的情况下，最终的`Makefile.in'将含有代码以生成名为 hello 的一个程序。变量 hello_SOURCES 用于确定哪些源代码应该被创建到可执行文件中去：

```
hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h
```

它将导致所有提及的`.c'文件被编译成对应的`.o'文件。而后被一同连接以生成`hello'。
如果需要`prog_SOURCES'，但并未定义它，那么它被缺省地设置成单个文件`prog.c'。在上面的例子中，对 hello_SOURCES 的定义实际上是多余的。
可以把多个程序创建到一个目录中。多个程序可以共享一个源代码文件。源代码文件必须在需要它的每个`_SOURCES'中列出。
出现在`_SOURCES'中的头文件将被包含在发布版本中，其它的头文件将被忽略。因为它并不明显，你不应该把由`configure'生成的头文件包含在变量`_SOURCES'中；不应该发布这个文件。也可以列出 lex（`.l'）和 yacc（`.y'）文件；参见对 Yacc 和 Lex 的支持。
即使不是所有的文件在所有的情况下都需要被创建，Automake 也必须知道所有可能被连接到程序中的源文件。所有仅仅是可能被创建的文件应该被适当地添加到变量`EXTRA_'中。例如，如果`hello-linux.c'是有条件地被包含在 hello 中，`Makefile.am'应该包括：

```
EXTRA_hello_SOURCES = hello-linux.c
```

类似地，有时候需要在配置的时刻确定创建那些程序。例如 GNU cpio 仅仅在特殊情况下才创建 mt 和 rmt。
在这种情况下，你必须把可能创建的所有程序都告诉 automake，但同时使生成的`Makefile.in'使用由 configure 指明的程序。这可以通过在 EXTRA_PROGRAMS 中列出所有可能创建的程序的同时，把 configure 提供的值的替换到每个`_PROGRAMS'变量的定义中，来完成。
如果你需要和 configure 没有找到的库相连接，你可以使用 LDADD 来完成。该变量实际上可以用于把任何选项添加到连接器的命令行中。

有时，要在一个目录中创建多个程序，但并不共享相同的连接时刻需求（link-time requirement）。在这种情况下，你可以使用变量`prog_LDADD'（其中 prog 是出现在某些`_PROGRAMS'变量中的程序名）来覆盖全局的 LDADD。（对于某个程序来说，如果存在这个变量，那么那个程序的连接就不使用 LDADD。）
例如，在 GNU cpio 中，pax、cpio 和 mt 需要和库`libcpio.a'连接。然而，rmt 在同一个目录中创建，并且不需要与`libcpio.a'连接。此外，mt 和 rmt 也仅仅在这种结构下创建。这里是 cpio 的`src/Makefile.am'内容（有删节）：

```
bin_PROGRAMS = cpio pax @MT@
libexec_PROGRAMS = @RMT@
EXTRA_PROGRAMS = mt rmt

LDADD = ../lib/libcpio.a @INTLLIBS@
rmt_LDADD =

cpio_SOURCES = ...
pax_SOURCES = ...
mt_SOURCES = ...
rmt_SOURCES = ...
```

变量`prog_LDADD'并不适用与传递程序特定的连接器选项（除了`-l'和`-L'之外）。所以，为此请使用变量`prog_LDFLAGS'。
有时候，是否创建一个程序依赖于不属于那个程序的某些其它目标。通过使用变量`prog_DEPENDENCIES'可以实现该功能。每个程序都依赖于这个变量的内容，但是不对它进行进一步的解释。
如果没有给出`prog_DEPENDENCIES'，它就由 Automake 来计算。自动获取的值是进行了大部分配置替换后的`prog_LDADD'内容，即删除了`-l'和`-L'选项。没有进行的配置替换仅仅是`@LIBOBJS@'and `@ALLOCA@'；没有进行这些替换是因为它们不会使生成的`prog_DEPENDENCIES'含有非法的值。
创建一个库
创建库与创建程序十分类似。在这种情况下，主变量的名字是`LIBRARIES'。库可以安装到 libdir 或 pkglibdir 之中。
关于如何使用 Libtool 和主变量`LTLIBRARIES'创建共享库的详情，请参见创建共享库。
每个`_LIBRARIES'变量都是需要被创建的库的列表。例如创建一个名为`libcpio.a'的库，但并不安装它，你可以写：

```
noinst_LIBRARIES = libcpio.a
```

确定那些源代码应该被创建到库中的方式与创建程序的情况完全相同，是通过变量`_SOURCES'。需要指出的是，库的名字是规范化的（参见派生变量是如何命名的），所以对应与`liblob.a'的`_SOURCES'变量对应的变量名为`liblob_a_SOURCES'，而不是`liblob.a_SOURCES'。
通过使用变量`library_LIBADD'，可以把额外的对象添加到库中。这应该由 configure 确定的对象使用。再看看 cpio：
libcpio_a_LIBADD = @LIBOBJS@ @ALLOCA@

对 LIBOBJS 和 ALLOCA 的特别处理
Automake 显式地识别对@LIBOBJS@和@ALLOCA@的使用，并用该信息，以及从`configure.in'中派生出的 LIBOBJS 文件列表，把适当的源文件自动添加到发布版本中。（参见需要发布哪些文件）。这些源文件还按照依赖跟踪机制进行自动处理，参见自动依赖跟踪。
在任何`_LDADD'或`_LIBADD'变量中， @LIBOBJS@和@ALLOCA@都将被自动识别出来。
创建一个共享库
创建共享库是一件相对复杂的事情。为此，提供了 GNU Libtool 以使我们可以按照与平台无关的方式创建共享库（参见 Libtool 手册）。
Automake 使用 Libtool 来创建在主变量`LTLIBRARIES'中声明的库。每个`_LTLIBRARIES'变量都是一个需要创建的共享库的列表。例如，为了创建一个名为`libgettext.a'的库和它对应的共享库，并且把它安装到`libdir'，可以写：
lib_LTLIBRARIES = libgettext.la

需要指出的是：共享库必须被安装，所以不允许使用 `noinst_LTLIBRARIES'和`check_LTLIBRARIES'。
对于每个库，变量`library_LIBADD'包含了需要被添加到共享库中的额外的 libtool 对象（`.lo'文件）。变量`library_LDFLAGS'包含了所有附加的 libtool 选项，例如`-version-info'或者`-static'。
普通的库可能需要使用@LIBOBJS@，而 libtool 库必须是使用 @LTLIBOBJS@。必须这样做是因为 libtool 所操作的目标文件并不仅仅是`.o'。libtool 手册包含了关于这个问题的细节。
对于安装在某些目录中的库，automake 将自动提供适当的 `-rpath'选项。然而，对于那些在配置时刻才能都确定的库（因而必须在 EXTRA_LTLIBRARIES 中给出），automake 并不知道它们最终安装的目录；对于这类库，你必须把 `-rpath'选项手工地添加到适当的`_LDFLAGS'变量中去。
详情请参见 Libtool 手册。
创建一个程序时使用的变量
有时有必要知道那个`Makefile'变量被 Automake 用于编译；例如在某些特殊情况下，你可能需要完成你自己的编译任务。
有些变量是从 Autoconf 中继承而来的；它们是 CC、 CFLAGS、CPPFLAGS、DEFS、 LDFLAGS

和 LIBS。
还有一些附加的变量是 Automake 自行定义的：
INCLUDES
一个`-I'选项的列表。如果你需要包含特殊的目录，你可以在你的`Makefile.am'中设置它。automake 已经自动地提供了一些`-I'选项。特别地，它生成`-I$(srcdir)'和一个指向保存了`config.h'的目录的`-I'选项（如果你已经使用了 AC_CONFIG_HEADER 或者 AM_CONFIG_HEADER）。 除了`-I'以外，INCLUDES 实际上还可以用于添加任何 cpp 选项。例如，有时用它把任意的`-D'选项传递给编译器。
COMPILE
实际用于编译 C 源文件的命令。文件名被添加到它的后面以形成完整的命令行。
LINK
实际用于连接 C 程序的命令。
对 Yacc 和 Lex 的支持
Automake 对 Yacc 和 Lex 有一些特殊的支持。
Automake 假定由 yacc（或 lex）生成的`.c'文件是以输入文件名为基础命名的。就是说，对于 yacc 源文件`foo.y'， automake 将认为生成的中间文件是`foo.c'（而不是更加传统的 `y.tab.c'）。
yacc 源文件的扩展名被用于确定生成的`C'或`C++'文件的扩展名。使用扩展名`.y'的文件将被转化成`.c'文件；类似地，扩展名`.yy'转化成 `.cc'；`.y++'转化成`c++'； `.yxx'转化成`.cxx'；类似地，Lex 源文件可以用于生成`C'或者`C++'；扩展名 `.l'、`.ll'、`.l++'和 `.lxx'都可以被识别。
你不应该在任何`SOURCES'变量中明确地提及中间的（`C'或者`C++'）文件；只要列出源文件就可以了。
由 yacc（或 lex）生成的中间文件将被包含在由它创建的任何发布版本中。这样用户就不需要拥有 yacc 或 lex 了。
如果出现了 yacc 源文件，那么你的`configure.in'必须定义变量`YACC'。完成这个任务最容易的方式是使用宏`AC_PROG_YACC'。
相似地，如果出现了 lex 源文件，那么你的`configure.in'必须定义变量`LEX'。你可以用宏 `AC_PROG_LEX'来完成这个工作。Automake 对 lex 的支持还要求你使用宏 `AC_DECL_YYTEXT'--automake 需要知道 `LEX_OUTPUT_ROOT'的值。
Automake 允许在一个程序中使用多个 yacc （或 lex）源文件。 Automake 使用一个称为 ylwrap 的小程序在子目录中运行 yacc（或者 lex）。必须这样做是因为 yacc 的输出文件名被修改了，并且并行的 make 可以同时地处理多于一个的 yacc 实例。ylwrap 和 automake 一同发布。它应该出现在由`AC_CONFIG_AUX_DIR'给出的目录，如果没有在`configure.in'中给出这个宏，它就应该出现在当前目录中。
对于 yacc，仅仅管理锁是不够的。yacc 的输出还总是在内部使用相同的符号名，所以不可能把两个 yacc 解析器（parser）连接到同一个可执行文件中。

我们建议使用如下在 gdb 中应用的改名方式：

```
#define     yymaxdepth c_maxdepth
#define     yyparse     c_parse
#define     yylex    c_lex
#define     yyerror c_error
#define     yylval   c_lval
#define     yychar c_char
#define     yydebug    c_debug
#define     yypact c_pact
#define     yyr1     c_r1
#define     yyr2     c_r2
#define     yydef   c_def
#define     yychk   c_chk
#define     yypgo  c_pgo
#define     yyact   c_act
#define     yyexca c_exca
#define yyerrflag c_errflag
#define yynerrs     c_nerrs
#define     yyps     c_ps
#define     yypv     c_pv
#define     yys c_s
#define     yy_yys  c_yys
#define     yystate c_state
#define     yytmp  c_tmp
#define     yyv c_v
#define     yy_yyv  c_yyv
#define     yyval    c_val
#define     yylloc   c_lloc
#define yyreds      c_reds
#define yytoks      c_toks
#define yylhs   c_yylhs
#define yylen  c_yylen
#define yydefred c_yydefred
#define yydgoto   c_yydgoto
#define yysindex c_yysindex
```

#define yyrindex c_yyrindex
#define yygindex c_yygindex
#define yytable c_yytable
#define yycheck c_yycheck

对于每一个 define，用任何你喜欢的东西替换前缀`c_'。这些定义可以为 bison、byacc 和传统的 yacc 工作。如果你发现某个解析器使用了这里所没有提到的符号，请告诉我们以便把它添加到上述列表中。

C++和其它语言

Automake 对 C++提供了完整的支持，对其它语言也有一些不完善的支持。对其它语言的支持将根据实际需要被添加进来。

任何包括了 C++代码的包都必须在`configure.in'中定义输出变量`CXX'；完成这一任务的最简单方式就是使用宏 AC_PROG_CXX。

在出现 C++源文件的时候需要定义几个附加的变量：

CXX

C++编译器的名称。

CXXFLAGS

传递给 C++编译器的任何选项。

CXXCOMPILE

实际用于编译 C++源文件的命令。文件名将被添加到它的后面以构成完整的命令行。

CXXLINK

实际用于连接 C++程序的命令。

自动 de-ANSI-fication

虽然 GNU 标准允许使用标准 C，使用标准 C 可能使包难以移植到一些旧的编译器上（典型的是 SunOS）。

在实际的编译发生之前，Automake 允许你通过对每个源文件实施"de-ANSI-fying" 以在这一类机器上进行工作。

如果`Makefile.am'变量 AUTOMAKE_OPTIONS（参见改变 Automake 的行为）包括了选项 ansi2knr， 那么处理 de-ANSI-fication 的代码就被插入到生成的`Makefile.in'中。

这使得目录中的每个 C 源文件都被看作标准 C。如果有标准 C 编译器可以使用，就使用它。如果没有标准 C 编译器可用，就用程序 ansi2knr 把文件转换成 K&R C，而后再被编译。

程序 ansi2knr 相当简单。它假定源代码按照特定的方式进行排版；详情请参见 ansi2knr 的 man。

支持 de-ANSI-fication 需要把源文件`ansi2knr.c'和 `ansi2knr.1'与标准 C 源代码放在同一个包中；这些文件与 Automake 一同发布。此外，包的`configure.in'必须调用宏 AM_C_PROTOTYPES。

Automake 还负责在当前包的其它目录中寻找 ansi2knr 的支持文件。这通过把到达适当目录的相对路

径添加到选项 ansi2knr 之前来完成。例如，假定一个包的标准 C 代码储存在子目录`src'和`lib'中。文件`ansi2knr.c'和`ansi2knr.1'出现在`lib'中。那么下述内容应该在`src/Makefile.am'中出现：
AUTOMAKE_OPTIONS = ../lib/ansi2knr

如果没有给出前缀，就假定文件在当前目录中。
自动依赖性跟踪
作为开发者，经常痛苦地在每次项目的 include 文件的依赖关系发生变化的时候对`Makefile.in'进行更新。 automake 提供了一种方式以自动跟踪依赖关系的变化，并且在生成的`Makefile.in'中给出这些依赖关系。
当前这种支持需要使用 GNU make 和 gcc。如果有足够的必要性，将来可能提供对不同的依赖性生成程序的支持。在此之前，如果当前目录中定义了任何 C 程序或者库，在缺省状态下就启动这种模式，因而你可能从非 GNU make 那里得到一个`Must be a separator'错误。
当你决定创建一个发布版本时，目标 dist 将用`--include-deps'和其它选项重新运行 automake。这将是的以前生成的依赖关系被插入到生成的`Makefile.in'中，并因而被插入到发布之中。这一步骤还不会把依赖性生成代码包括进来，所以那些下载你的发布版本的人就不必使用 GNU make 和 gcc，而且不会导致错误。
在添加到`Makefile.in'的时候，所有系统特定的依赖性都被自动删除了。这可以通过把文件罗列在变量`OMIT_DEPENDENCIES'中来实现。例如：automake 将删除所有对系统头文件的引用。有时有必要指明应该删除哪个头文件。例如，如果你的 `configure.in'使用了`AM_WITH_REGEX'，那么就应该删除任何与`rx.h'或者 `regex.h'有关的依赖性，这是因为在用户配置包之前还不知道应该使用那个头文件。
实际上，automake 足够地聪明以至于可以处理诸如常规表达式头文件的特殊情况。如果使用了`AM_GNU_GETTEXT'，它还将自动忽略 `libintl.h'。
自动依赖性跟踪功能可以通过在变量 AUTOMAKE_OPTIONS 中设置 no-dependencies 来关闭。
如果你打开由 make dist 创建的发布版本，而且你希望重新添加依赖性跟踪的代码，只要重新运行 automake 就行了。
实际的依赖性文件被储存在创建目录下的一个名为`.deps'的子目录中。这些依赖性是机器特有的。如果你愿意，删除它们是安全的；它们将在下次创建的时候自动重建。
其它派生对象
Automake 可以处理不是 C 程序的其它派生对象。有时对实际创建这类对象的支持必须显式地给出，但 Automake 仍然会自动地处理安装和发布。
可执行脚本
定义和安装脚本程序是可能的。这种程序被罗列在主变量`SCRIPTS'中。 automake 没有为脚本定义任何依赖性关系； `Makefile.am'应该包含正确的规则。
automake 并不假定脚本是派生的对象；这些对象必须被手工地删除；详情请参见清除了些什么。
automake 本身就是在配置时刻从`automake.in'中生成的脚本。下面给出了如何处理它：

bin_SCRIPTS = automake

因为 automake 出现在宏 AC_OUTPUT 中，自动地生成了一个关于它的目标。
脚本对象可以安装在 bindir、sbindir、 libexecdir 或者 pkgdatadir 中。
头文件
头文件由`HEADERS'变量族所指定。通常是不安装头文件的，所以变量 noinst_HEADERS 是最常用的。
所有的头文件都必须在某些地方列出；没有列出的头文件将不会出现在发布版本中。通常最清楚的方式是和程序的其它源代码一起列出不会被安装的头文件。参见创建一个程序。在变量`_SOURCES'中列出的头文件不需要在任何 `_HEADERS'变量中再次列出。
头文件可以安装到 includedir、oldincludedir 或者 pkgincludedir 中。
与体系结构无关的数据文件
Automake 使用`DATA'族变量来支持对各种数据文件的安装。
这些数据可以安装在目录 datadir、 sysconfdir、sharedstatedir、 localstatedir 或者 pkgdatadir 中。
在缺省状态下，数据文件不会被包含在发布版本中。
下面是 automake 如何安装它的附加数据文件：
pkgdata_DATA = clean-kr.am clean.am ...

已创建的源代码
有时候，一个可以被称作“源文件”的文件（例如一个 C `.h'文件）实际上是从其它文件中派生出来的。这类文件应该被罗列在变量 BUILT_SOURCES 中。
在缺省状态下，不会编译已创建的源文件。你必须在一些其它的`_SOURCES'变量中明确地给出它们，以便对其进行编译。
需要说明的是，在某些情况下，BUILT_SOURCES 将以令人惊讶的方式工作。为了获得已创建的源文件以进行自动依赖性跟踪，`Makefile'必须依赖于 $(BUILT_SOURCES)。这导致这些源文件可能在某些可笑的时候被重新创建。
其它的 GNU 工具
因为 Automake 被设计成为 GNU 程序自动生成`Makefile.in'，它为与其它 GNU 工具进行互操作做出了努力。
Emacs Lisp
Automake 为 Emacs Lisp 提供了一些支持。主变量`LISP'被用于保存一个`.el'文件的列表。该主变量的可能前缀有：`lisp_'和`noinst_'。如果定义了 lisp_LISP，那么 `configure.in'就必须运行 AM_PATH_LISPDIR（参见 Automake 支持的 Autoconf 宏）.
在缺省状态下，Automake 将使用通过 AM_PATH_LISPDIR 找到的 Emacs 按字节编译（byte-compile）所有 Emacs Lisp 源文件。如果你不希望使用字节编译，请把变量`ELCFILES'定义为空。

字 节编译的 Emacs Lisp 文件并不能在所有版本的 Emacs 间移植，所以如果你希望在同一个地方安装多于一个版本的 Emacs，你可以关闭它。进一步，许多包并没有从字节 编译中获得实际的好处。我们仍然建议你不要改变缺省设置。对站点进行特殊的配置以适用于它们自身，可能比为其它所有人创建不和谐的安装要更好一些。

Gettext

如果在`configure.in'中出现了 AM_GNU_GETTEXT，那么 Automake 就开启对 GNU gettext 的支持，一个支持国际化的消息编目系统（参见 GNU gettext 工具中的`GNU Gettext'节）。

在 Automake 中对 gettext 的支持需要把两个子目录附加到包中，`intl'和`po'。Automake 确认这些目录的存在并且在 SUBDIRS 中被给出。

更进一步，Automake 检查在`configure.in'中包含了与所有合法的`.po'文件相对应的 `ALL_LINGUAS'定义，并且没有多余的定义。

Guile

Automake 为构造 Guile 模块提供了一些自动支持。如果宏 AM_INIT_GUILE_MODULE 出现在 `configure.in'中，Automake 将开启对 Guile 的支持。

现在对 Guile 的支持仅仅表示宏 AM_INIT_GUILE_MODULE：

·运行了 AM_INIT_AUTOMAKE。

·带路径`..'运行了 AC_CONFIG_AUX_DIR。

随着 Guile 模块代码的成熟，无疑 Automake 的支持也将会更好。

Libtool

Automake 通过主变量`LTLIBRARIES'提供了对 GNU Libtool 的支持（参见 Libtool 手册）参见创建一个共享库。

Java

通过主变量`JAVA'，Automake 为 Java 的编译提供了少量支持。

任何在变量`_JAVA'列举的`.java'文件在创建时刻将用 JAVAC 进行编译。在缺省状态，`.class'文件不会被包含在发布版本中。

目前 Automake 正试图限制只能在`Makefile.am'中使用一个`_JAVA'主变量。引入这个限制是因为，通常是不可能知道哪个`.class'文件是从哪个`.java'文件中生成的 -- 因此不可能知道哪个文件应该被安装到哪里。

创建文档

目前 Automake 提供了对 Texinfo 和 man 的支持。

Texinfo

如果当前目录中含有 Texinfo 源文件，你必须在主变量`TEXINFOS'中给出声明。通常 Texinfo 文件被转换成 info，因此经常在这里使用宏 info_TEXINFOS。需要指出的是任何 Texinfo 源文件的文件名的扩展名都必须是`.texi'或者`.texinfo'。

如果`.texi'文件@includes `version.texi'，就将自动生成那个文件。 `version.texi'定义了三个你可以引用的 Texinfo 宏：EDITION、VERSION 和 UPDATED。前两个宏保存了你的包的版本号（为清

晰起见而分别保存）；最后一个宏是最后一次修改主文件的日期。对`version.texi'的支持需要程序 mdate-sh；这个程序由 Automake 提供。
有时，一个 info 文件实际上依赖于多个`.texi'文件。例如，在 GNU Hello 中，`hello.texi'包括了文件`gpl.texi'。你可以通过使用变量`texi_TEXINFOS'告诉 Automake 这一依赖性。下面就是 Hello 处理它的代码：

```
info_TEXINFOS = hello.texi
hello_TEXINFOS = gpl.texi
```

在缺省状态下，Automake 认为`texinfo.tex'出现在 Texinfo 源文件所在的目录中。然而，如果你在`configure.in'中使用了 AC_CONFIG_AUX_DIR 那么将到那个目录中去寻找`texinfo.tex'文件。如果给出了 `--add-missing'，Automake 将自动提供`texinfo.tex'
如果你的包在许多目录中储存了 Texinfo 文件，你可以用变量 TEXINFO_TEX 来告诉 automake 到那里去寻找你的包的规范`texinfo.tex'文件。这个变量的值应该是从当前`Makefile.am'到`texinfo.tex'的相对路径。

```
TEXINFO_TEX = ../doc/texinfo.tex
```

选项`no-texinfo.tex'可以消除对`texinfo.tex'的需要。但使用变量 TEXINFO_TEX 更好些，因为它使得目标 dvi 仍然可以工作。
Automake 生成一个 install-info 目标；一些人显式地使用它。在缺省状态下，info 文档通过运行`make install'来安装。可以通过给出选项 no-installinfo 以阻止执行这一操作。
Man 手册
一个包还可以包含 man 手册。（参见 GNU 关于此事的标准，GNU 编码标准中的`Man 手册'节。）用主变量`MANS'声明 Man 手册。通常使用宏 man_MANS。根据文件的扩展名，Man 手册被自动地安装到 mandir 的正确子目录中。
在缺省状态下，man 手册通过运行`make install'安装。然而，因为 GNU 项目并不需要 man 手册，许多维护者没有花费时间以及时更新 man 手册。在这些情况下，给出选项 no-installman 将阻止对 man 手册的安装。用户仍然可以显式地使用`make install-man'来安装它们。
下面是 GNU cpio 如何处理它的文档（它同时包括了 Texinfo 文档和 man 手册）：

```
info_TEXINFOS = cpio.texi
man_MANS = cpio.1 mt.1
```

Texinfo 源文件和 info 文档都被看作用于创建发布版本的源代码。
Man 手册现在并不被看作源代码，因为 man 手册有时是自动生成的。
安装了些什么
自然地，一旦你的程序被创建了，Automake 实际上处理了安装程序的细节。所有 PROGRAMS、

SCRIPTS、 LIBRARIES、LISP、DATA 和 HEADERS 中列出的文件都被自动地安装在适当的位置。
Automake 还可以处理对 info 文档和 man 手册的安装。
Automake 生成分离的 install-data 和 install-exec 目标，如果安装器（installer）在共享的目录结构中为多个机器进行安装，这些目标允许与机器独立的部分仅仅被安装一次。目标 install 依赖于这两个目标。
Automake 还生成一个 uninstall 目标，一个 installdirs 目标，和一个 install-strip 目标。
通过定义目标 install-exec-local，或者目标 install-data-local 就可以扩展这一机制。如果这些目标存在，它们将在运行`make install'时运行。
使用标准目录前缀`data'、`info'、`man'、 `include'、`oldinclude'、`pkgdata'或者`pkginclude'的变量（例如，`data_DATA'）将由 `install-data'安装。
使用标准目录前缀`bin'、`sbin'、`libexec'、 `sysconf'、`localstate'、`lib'或者 `pkglib'的变量（例如，`bin_PROGRAMS'）将由 `install-exec'安装。
任何使用了含有`exec'的用户定义目录前缀的变量（例如，`myexecbin_PROGRAMS'）将由`install-exec'安装。所有使用其它用户定义的前缀的变量将由`install-data'安装。
Automake 在所有的 install 规则中生成对变量`DESTDIR'的支持；参见 GNU 编码标准中的`Makefile 惯例'节。
清除了些什么
GNU Makefile 标准给出了许多不同的清除规则。通常由 Automake 自动确定可以清除哪些文件。当然，Automake 还能识别一些为指明需要额外地清除的文件而定义的变量。这些变量是 MOSTLYCLEANFILES、CLEANFILES、DISTCLEANFILES 和 MAINTAINERCLEANFILES。
需要发布哪些文件
生成的`Makefile.in'中的 dist 目标可以用于产生用 gzip 压缩了的发布 tar 文件。 tar 文件是根据`PACKAGE'变量和`VERSION'变量命名的；精确地说，被命名为`package-version.tar.gz'。
在大部分情况下，发布版本中的文件由 Automake 自动寻找：所有的源文件被自动地包含在发布版本中，还有所有的 `Makefile.am'和`Makefile.in'。Automake 定义了一些常用的内置文件，如果出现在当前目录中，那么就被自动地包含在发布版本中。可以用`automake --help'打印它们。此外，由 configure 读入的文件（例如，对应于由 AC_OUTPUT 调用指明的文件的源文件）将被自动地包含在发布版本中。
有时，除此而外仍然有一些文件需要被发布而没有包含在自动规则之中。这些文件应该在变量 EXTRA_DIST 中列出。需要指出的是，EXTRA_DIST 只能处理当前目录中的文件；其它目录中的文件将使 make dist 在运行时发生错误。
如果你定义了 SUBDIRS，automake 将递归地把子目录包含在发布版本中。如果是有条件地定义 SUBDIRS（参见条件），通常 automake 将把所有可能出现在 SUBDIRS 中的所有目录包含在发布版本中。如果你需要有条件地给出一组目录，你可以设置变量 DIST_SUBDIRS 以精确地列出需要包含在发布版本中的子目录。
有时在发布版本打包之前修改发布版本是有用的。如果目标 dist-hook 存在，它可以在填充发布目录之

后，创建 tar（或 shar）文件之前运行。使用该功能的一种方式是发布新的`Makefile.am'被删除了的子目录中的文件。

```
dist-hook:
	mkdir $(distdir)/random
	cp -p random/a1 random/a2 $(distdir)/random
```

Automake 还生成一个 distcheck 目标，它有助于确认给定的发布版本实际上是可以工作的。distcheck 首先创建发布版本，而后试图进行一个 VPATH 创建。

对测试套件的支持

Automake 支持两种形式的测试套件。

如果定义了变量 TESTS，它的值被看作为了运行测试而运行的程序列表。程序可以是派生对象或者是源对象；生成的规则将在 srcdir 和`.'中寻找。应该在 srcdir（它可以既是环境变量，又是 make 变量）中寻找程序需要的数据文件，所以它们在一个分离的目录中进行创建的工作（参见 Autoconf 手册中的`创建目录'节），并且是在特别的目标 distcheck 中创建的（参见那些需要被发布）。

在运行的终点将打印失败的次数。如果一个测试程序以状态 77 退出，在最后的计数中它的结果将被忽略。这一特征允许不可移植的测试在对它来说没有意义的环境下被忽略。

变量 TESTS_ENVIRONMENT 可以用于为测试运行而设置环境变量；在该规则中设置环境变量 srcdir。如果你所有的测试程序都是脚本，你还可以把 TESTS_ENVIRONMENT 设置成一个对 shell 的调用（例如`$(SHELL) -x'）；这对调试测试结果来说是有用的。

如果`dejagnu'出现在 AUTOMAKE_OPTIONS 中，那么就假定运行一个基于 dejagnu 的测试套件。变量 DEJATOOL 的值被作为--tool 的参数传递给 runtest；它的缺省值是包的名字。

在缺省状态下，变量 RUNTESTDEFAULTFLAGS 保存了传递给 dejagnu 的--tool 和--srcdir 选项；如果有必要，可以覆盖它。

还可以覆盖变量 EXPECT、RUNTEST 和 RUNTESTFLAGS 以提供项目特定的值。例如，如果你正在测试编译器工具链（toolchain），你就需要这样做。这是因为缺省值并没有把主机名和目标名考虑进去。

在上述两种情况中，测试都是通过`make check'来完成的。

改变 Automake 的行为

Automake 的各种特征可以在`Makefile.am'中用各种选项进行控制。这些选项在一个名为 AUTOMAKE_OPTIONS 的特殊变量中被列出。目前可以理解的选项有：

gnits

gnu

foreign

cygnus

设置适当的严格性。选项 gnits 还隐含了 readme-alpha 和 check-news。

ansi2knr

path/ansi2knr

打开自动 de-ANSI-fication 功能。参见自动 de-ANSI-fication。如果以一个路径开头，那么生成的`Makefile.in'将在特别给定的目录中寻找程序`ansi2knr'。通常，路径应该是到同一个发布版本中的其它目录的相对路径（虽然 Automake 并不进行这项检查）。

check-news

给出该选项后，如果当前的版本号没有出现在`NEWS'文件中的前几行中，将导致 make dist 的失败。

dejagnu

生成 dejagnu 特定的规则。参见对测试套件的支持。

dist-shar

就象普通的 dist 目标那样生成 dist-shar 目标。这个新目标将创建一个发布版本的 shar 包。

dist-zip

就象普通的 dist 目标那样生成 dist-zip 目标。这个新目标将创建一个发布版本的 zip 包。

dist-tarZ

就象普通的 dist 目标那样生成 dist-tarZ 目标。这个新目标将创建一个发布版本的 tar 包；假定使用传统的 tar 和 compress。警告：如果你实际上在使用 GNU tar，那么生成的包可能含有不可移植的结构。

no-dependencies

这与在命令行中使用选项`--include-deps'相类似，但在那些你不希望使用自动依赖性跟踪的场合下更为有用。参见自动依赖性跟踪。在这个情况下将有效地关闭自动依赖性跟踪。

no-installinfo

给出该选项后，生成的`Makefile.in'在缺省的状态下将不会创建或者安装 info 文档。然而，目标 info 和目标 install-info 仍然是可用的安装选项。在`GNU'严格性以及更高的严格性中这个选项是不允许使用的。

no-installman

给出这个选项后，生成的`Makefile.in'在缺省状态下将不会安装 man 手册。然而，目标 install-man 仍然是可用的安装选项。在`GNU'严格性以及更高的严格性中这个选项是不允许使用的。

no-texinfo.tex

即使在本目录中含有 texinfo 文件，也不需要`texinfo.tex'

readme-alpha

如果本发布是一个 alpha 版本，并且存在文件`README-alpha'，那么它将被添加到发布版本中。如果给出了该选项，版本号应该是以下两种形式之一。第一种形式是`MAJOR.MINOR.ALPHA'，其中每个元素都是一个数字；最后的点和数字应该被忽略以用于非 alpha 版本。第二种形式是`MAJOR.MINORALPHA'，其中 ALPHA 是一个文字；对于非 alpha 版本来说，应该忽略它。

version

可以给出版本号（例如，`0.30'）。如果 Automake 并不比要求的版本号新，将不会创建`Makefile.in'。

automake 能够诊断不能识别的选项。

其它规则

还有一些不适于放在任何其它地方的几条规则。

与 etags 之间的界面

在某些环境下，automake 将生成规则以产生由 GNU Emacs 使用的`TAGS'文件。

如果出现了任何 C 源代码或者头文件，那么就为该目录创建目标 tags 和目标 TAGS。

在运行的时候，将在多个目录包的顶层目录创建一个 tags 文件，它将生成一个包括了对所有子目录的`TAGS'文件的引用的`TAGS'文件。

此外，如果定义了变量 ETAGS_ARGS，将生成目标 tags。该变量用于包含了能够被标记，但是不能够被 etags 所识别的源文件的目录。

下面是 Automake 如何为它的源代码生成标记和它的 Texinfo 文件中的结点。

```
ETAGS_ARGS = automake.in --lang=none \
 --regex='/^@node[ \t]+\([^,]+\)/\1/' automake.texi
```

如果你把文件名添加到`ETAGS_ARGS'，你将可能还要设置`TAGS_DEPENDENCIES'。该变量的内容将直接添加到目标 tags 的依赖列表中。

Automake 还将生成一个将在源代码上运行 mkid 的 ID 目标。它仅仅可以在一个目录到目录的基础（directory-by-directory basis）上使用。

处理新的文件扩展名

有时，为处理 Automake 不能识别的文件类型而引入新的隐含规则是有用的。如果这样做，你必须把这些新的后缀告诉 GNU Make。可以通过把新后缀的列表添加到变量 SUFFIXES 来完成。

例如，目前 automake 不能对 Java 提供任何支持。如果你写了一个用于从`.java'源文件生成`.class'文件的宏，你还需要把这些后缀添加到列表中：

```
SUFFIXES = .java .class
```

条件

Automake 支持一种简单的条件。

在使用条件之前，你必须在 configure.in 文件中使用 AM_CONDITIONAL 定义它。宏 AM_CONDITIONAL 接受两个参数。

AM_CONDITIONAL 的第一个参数是条件的名字。它应该是一个以字母开头并且仅仅由字母、数字和下划线组成的简单字符串。

AM_CONDITIONAL 的第二个参数是一个适用于 shell 的 if 语句的 shell 条件。该条件将在运行 configure 的时候被求值。

条件典型地依赖于用户提供给 configure 脚本的选项。下面是一个演示如果在用户使用了`--enable-debug'选项的情况下为真的条件的例子。

```
AC_ARG_ENABLE(debug,
[  --enable-debug    Turn on debugging],
[case "${enableval}" in
  yes) debug=true ;;
  no)  debug=false ;;
  *) AC_MSG_ERROR(bad value ${enableval} for --enable-debug) ;;
esac],[debug=false])
AM_CONDITIONAL(DEBUG, test x$debug = xtrue)
```

下面是一个如何在`Makefile.am'中使用条件的例子：

```
if DEBUG
DBG = debug
else
DBG =
endif
noinst_PROGRAMS = $(DBG)
```

这个小例子还可以被处理以用于 EXTRA_PROGRAMS。（参见创建一个程序）。

你只能在 if 语句中测试单个变量。可以忽略 else 语句。条件可以嵌套到任意深度。

需要指出的是 Automake 中的条件与 GNU Make 中的条件是不相同的。Automake 的条件在配置时刻由`configure'脚本完成检查，并且将影响`Makefile.in'到`Makefile'的转换。它们是基于传递给`configure'的选项和`configure'对本地系统的检测结果的。 GNU Make 条件则在 make 时完成检查，并且是基于传递给 make 程序的变量或者是基于在`Makefile'中定义的变量的。

Automake 条件在任何 make 下都可以工作。

--gnu 和--gnits 的效果

选项`--gnu'（或者`AUTOMAKE_OPTIONS'中的`gnu'）将使得 automake 执行如下检查：

·在包的顶层目录中应该含有文件`INSTALL'、`NEWS'、`README'、`COPYING'、`AUTHORS'和`ChangeLog'。

·禁止使用选项`no-installman'和`no-installinfo'。

需要指出的是，该选项将在未来进行扩展以进行更多的检查；熟悉 GNU 标准的精确需求是值得推荐的。此外，`--gnu'可以要求某些非标准 GNU 程序为了各种仅由维护者使用的目标而存在；例如，在将来`make dist'可能会需要 pathchk。

选项`--gnits'进行`--gnu'所做的所有检查，此外还进行如下检查：

·`make dist'将检查以确认文件`NEWS'被更新到当前的版本。
·不允许出现文件`COPYING.LIB'。LGPL 显然是一个失败的试验。
·检查文件`VERSION'以确保它的格式是符合 Gnits 标准的。
·如果`VERSION'表明本版本是 alpha 版本，并且文件`README-alpha'出现在包的顶层目录中，那么它就被包含在发布版本中。因为该模式是唯一对版本号格式实行限制的模式，所以上述操作在`--gnits'模式下实施而不在其它模式下进行，因此，该模式也是 automake 唯一能够自动确定是否把`README-alpha'包含在发布版本中的模式。
·需要出现文件`THANKS'。
--cygnus 的效果
对于如何构造`Makefile.in'，Cygnus 解决方案有一些不同的规则。把`--cygnus'传递给 automake 将使所有生成的`Makefile.in'服从 Cygnus 规则。
下面是`--cygnus'的精确效果：
·总是在创建目录中，而不是在源目录创建 Info 文件。
·如果指明了 Texinfo 源文件，就不需要`texinfo.tex'。其设想是：应该提供该文件，但 automake 不能在适当的位置找到它。这种设想是 Cygnus 包典型的包装方式的产物。
·`make dist'将在创建目录和源目录中寻找文件。为了支持把 info 文件储存在创建目录中而提供这个功能。
·将在创建树和用户的`PATH'中寻找某些工具。这些工具是 runtest、 expect、makeinfo 和 texi2dvi。
·隐含--foreign 选项。
·隐含选项`no-installinfo'和选项`no-dependencies'。
·需要宏`AM_MAINTAINER_MODE'和宏`AM_CYGWIN32'。
·目标 check 并不依赖于目标 all。
建议 GNU 维护者使用`gnu'严格性方式而不是特殊的 Cygnus 模式。
什么时候 Automake 不够用
Automake 的隐含语义意味着许多问题只要通过把一些 make 目标和规则添加到`Makefile.in'中就可以解决了。 automake 将忽略这些添加的目标和规则。
对于这种做法需要提出告诫。虽然你可以覆盖已经被 automake 所使用的目标，但这通常是失策的，在非平包（non-flat）的顶层目录中尤其如此。然而，你可以在你的`Makefile.in'中给出各种带有`-local'的有用目标版本。 Automake 将用那些用户提供的目标补充标准的目标。
支持本地版本的目标有 all、info、dvi、 check、install-data、install-exec、 uninstall 和各种 clean 目标（mostlyclean、clean、distclean 和 maintainer-clean）。需要指出的是没有 uninstall-exec-local 或者 uninstall-data-local 目标；请使用 uninstall-local。仅仅反安装数据或仅仅反安装可执行文件是没有意义的。
例如，下面是把一个文件安装到`/etc'的一种方式：

```
install-data-local:
	$(INSTALL_DATA) $(srcdir)/afile /etc/afile
```

某些目标还可以在完成它的工作之后运行一个称为 hook 的其它目标。 hook 是在源目标名的后面添加`-hook'。允许使用 hook 的目标是 install-data、install-exec、dist 和 distcheck。
例如，下面是如果创建一个到已经安装的程序的硬连接：

```
install-exec-hook:
	ln $(bindir)/program $(bindir)/proglink
```

发布`Makefile.in'
Automake 对于发布生成的`Makefile.in'没有施加任何限制。我们仍然鼓励软件的作者按照诸如 GPL 之类的条款发布它们的作品，但是 Automake 并不要求你这样做。
一些可以通过选项--add-missing 自动安装的文件则受到 GPL 的约束；打开每个文件检查一下。
未来的某些想法
下面是可能在未来发生的一些事情：
·支持 HTML。
·输出将被清理。例如，只有那些确实使用了的变量才会出现在生成的 `Makefile.in'文件中。
·对发布版本提供自动重编码支持。其目的是允许维护者使用对他来说最方便的字符集，但所有的发布都将使用 Unicode 或者带有 UTF-8 编码的 ISO 10646。
·对自动生成包提供支持（例如 Debian 包、RPM 包、Solaris 包等等）。如果有具备创建包的经验的人告诉我怎样做才是有帮助的，那么将会更快地提供这项支持。
·用 Guile 重写。这不会在不久的将来发生，但它终将发生。