Buildroot 用户手册（中文版）

前言

Buildroot 是一个简单高效、易于使用的可通过交叉编译来构建嵌入式 Linux 系统的工具。它能够处理交叉编译工具链、根文件系统生成、内核镜像编译和引导加载程序编译等，还支持数千种软件包，例如 Gtk3、QT5、GStreamer、Webkit 以及大量与网络相关、系统相关的实用工具。Buildroot 采用类似于 linux 内核的 menuconfig、gconfig 和 xconfig 配置界面，因此使用 Buildroot 构建一个基本的系统是非常轻松的，这通常会花费 15-30 分钟。同时，Buildroot 是一个开源项目，开发者可以对它做出贡献，来让 Buildroot 变得更加完善。

在正点原子 Linux 开发板相关教程中，详细讲解了如何使用 Buildroot 来构建一个基本根文件系统，配置添加其他实用工具，并在板子上运行。为了让大家更加了解 Buildroot 使用方法、Buildroot 工作机制以及 Buildroot 本身的开发方式等，正点原子 linux 团队对 Buildroot 官方用户手册进行中文翻译，形成 Buildroot 中文用户手册，以供用户参阅。

本手册适用人员范围：嵌入式开发人员、任何想要了解 Buildroot 的爱好者或贡献者

如发现本手册需勘误之处，请发送邮件至 marc07@qq.com。

修订历史

版本	版本更新说明	负责人	校审	发布日期
V1.0	初稿	正点原子 linux 团队	正点原子 linux 团队	2021.06

目录

Buildroot 2020.02.6 手册生成于 2020-09-05 19:12:42 UTC，对应 git 版本为 b120226e0e。

第一部分 入门

1 关于 Buildroot

Buildroot 是一个工具，它使用交叉编译，可以简化和自动化为嵌入式系统构建一个完整 Linux 系统的过程。

为了实现这一目标， Buildroot 能够为您的目标对象生成交叉编译工具链、根文件系统、Linux 内核镜像和 bootloader 引导加载程序。Buildroot 可以独立应用于这些选项的任意组合（例如，您可以使用现有的交叉编译工具链，通过 Buildroot 来单独构建根文件系统）。

Buildroot 主要对使用嵌入式系统的用户有用。嵌入式系统通常使用的处理器并不是那些每个人在 PC 电脑上使用的常规 x86 处理器。它们可以是 PowerPC 处理器、MIPS 处理器、ARM 处理器等。

Buildroot 支持多种处理器及其变体；它还为一些现成板子提供默认配置。除此之外，许多第三方的项目都是基于 Buildroot 或者在 Buildroot 之上开发其 BSP[1] 或 SDK[2] 。

[1]BSP：板级支持包
[2]SDK：软件开发工具包

2 系统要求

Buildroot 被设计为在 Linux 系统上运行。

尽管 Buildroot 会自行构建用于编译的大多数宿主机软件包，但一些标准 Linux 工具需要提前在宿主机系统上安装好。下面，您将看到强制软件包和可选软件包的概述（请注意，软件包名称在不同的发行版之间可能会有所不同）。

2.1 强制软件包

- 构建工具
  - Which
  - sed
  - make (version 3.81 or any later)
  - binutils
  - build-essential (only for Debian based systems)
  - gcc (version 4.8 or any later)
  - g++ (version 4.8 or any later)
  - bash
  - patch
  - gzip
  - bzip2
  - perl (version 5.8.7 or any later)
  - tar
  - cpio

- unzip
- rsync
- file (must be in /usr/bin/file)
- bc

• 源码获取工具
- wget

2.2 可选软件包

• 推荐的依赖项

Buildroot 中一些功能或实用程序（例如 legal-info 或图形生成工具）具有附加依赖项。尽管这对于简单的构建并不是必需的，但仍强烈建议安装：

- Python(2.7 或更高版本)

• 配置界面依赖项：

对于这些库，您需要同时安装 runtime 和 development 数据，在许多发行版中，这些是单独打包的。development 软件包通常具有-dev 或-devel 后缀。

- ncurses5 使用 menuconfig 界面
- qt5 使用 xconfig 界面
- glib2，gtk2 和 glade2 使用 gconfig 界面

• 源码获取工具

在官方源码树中，大多数软件包源码是使用 wget 工具从 ftp、http 或 https 位置检索获取的。只有少数软件包是从版本控制系统获取。此外，Buildroot 能够通过其他工具下载源码，例如 rsync 或 scp（更多详细信息，请参阅第 19 章）。如果使用其中任何一种方法启用软件包，则需要在宿主机系统上安装相应的工具：

- bazaar
- cvs
- git
- mercurial
- rsync
- scp
- subversion

• 与 Java 有关的软件包，如果需要为目标系统构建 Java Classpath：
- The **javac** compiler
- The **jar** tool

• 文档生成工具：
- asciidoc, version 8.6.3 or higher
- w3m
- **python** with the **argparse** module(automatically present in 2.7+ and 3.2+)
- dblatex (required for the pdf manual only)

• 图表生成工具：
- **graphviz** to use graph-depends and <pkg>-graph-depends
- **python-matplotlib** to use graph-build

3 获取 Buildroot

Buildroot 每 3 个月发布一次，分别在 2 月、5 月、8 月、11 月发布。发布版本号格式为 YYYY.MM，例如 2013.02、2014.08。

buildroot 源码包可以在 http://buildroot.org/downloads/获取。

为了您的方便，可以在 Buildroot 源码树的 support/misc/Vagrantfile 中找到工具 Vagrantfile，通过它来快速设置具有所需依赖关系的虚拟机，进行开始使用。

如果要在 Linux 或 Mac Os X 上设置隔离的 buildroot 环境，请将此行粘贴到终端上：

```
curl -O https://buildroot.org/downloads/Vagrantfile; vagrant up
```

如果您使用的是 Windows，请将其粘贴到您的 Powershell 中：

```
(new-object System.Net.WebClient).DownloadFile(
"https://buildroot.org/downloads/Vagrantfile","Vagrantfile");
vagrant up
```

如果您想要关注 buildroot 开发，则可以使用它的每日快照或克隆 Git 存储库。可访问 Buildroot 网站查阅 Downloadpage 以获取更多详细信息。

4 Buildroot 快速入门

重要：您可以并且应该以普通用户身份来构建所有内容。不需要使用 root 用户去配置和使用 Buildroot。以常规用户身份去运行所有命令，能够保护系统免受在编译和安装过程中表现异常的软件包的侵害。使用 Buildroot 的第一步是创建配置。Buildroot 有一个不错的配置工具，类似于您可以在 Linux 内核或在 BusyBox 找得到的配置工具。

在 buildroot 目录中，运行

```
$ make menuconfig
```

用于原始的 curses-based 的配置，或者运行

```
$ make nconfig
```

用于新的 curses-based 的配置，或者运行

```
$ make xconfig
```

用于 Qt-based 的配置，或者运行

```
$ make gconfig
```

用于 GTK-based 的配置。

所有这些“make”命令都需要构建一个配置实用程序（包括界面），因此您可能需要为配置实用程序所使用的相关库安装“development”软件包。更多详细信息请参阅第 2 章，特别是获取您所喜欢的界面的依赖项可选要求。

对于配置工具中的每个菜单条目，您都可以找得到描述条目用途的相关帮助。有关一些特定配置方面的详细信息，请参阅第 6 章。

完成所有配置后，配置工具将生成一个包含全部配置的.config 文件。该文件将被顶级 Makefile 文件读取。

要开始构建过程，只需运行：

```
$ make
```

默认情况下，Buildroot 不支持顶级并行构建，因此无需运行 make -jN。但是，有针对顶级并行构建的实验性支持，请参阅第 8.11 节。

make 命令通常将执行以下步骤：

- 下载源文件（根据需要）；
- 配置，构建和安装交叉编译工具链，或者仅导入外部工具链；
- 配置，构建和安装选定的目标软件包；
- 构建内核镜像（如果选择）；
- 构建引导加载程序镜像（如果选择）；
- 以选定的格式创建一个根文件系统。

Buildroot 编译后的输出保存在单个目录 output/中。该 output 目录下包含几个子目录：

- images/ 存储着所有镜像（包括内核镜像，引导加载程序和根文件系统镜像）。这些是您需要放在目标系统上的文件。
- build/ 构建所有组件（这包括 Buildroot 在宿主机上所需的工具以及为目标编译的软件包）。该目录包含着每个组件的一个子目录。
- host/ 包含为宿主机构建的工具，以及目标工具链的 sysroot。前者是宿主机正确执行 Buildroot 时所需安装的工具，包括交叉编译工具链。后者是一种类似于根文件系统层次结构的目录结构。它包含所有用户空间软件包的头文件和库，这些软件包可提供并安装由其他软件包使用的库。但是，该目录并不旨在成为目标的根文件系统：它包含许多开发文件、未剥离的二进制文件和库，这些文件对于嵌入式系统而言太大了。这些开发文件用于为依赖于其他库的目标去编译库和应用程序。
- staging/ 是一个指向 host/中目标工具链 sysroot 的符号链接，为了向后兼容而存在。
- target/ 包含指定目标的几乎完整根文件系统：除了/dev/中的设备文件，所有需要的东西都存在（Buildroot 无法创建设备文件，因为 Buildroot 不能以 root 身份运行，也不想以 root 身份运行）。另外，它没有正确的权限（例如 busybox 中的 setuid 权限）。因此，该 target 目录不应在您的目标上使用。相反，您应该使用 images/目录下编译生成的其中一个镜像。如果您需要提取根文件系统镜像来通过 NFS 进行引导，请使用 images/中生成的压缩包镜像并将其提取为 root。与 staging/相比，target/仅包含运行目标应用程序所需的文件和库：不存在开发文件（例如 headers 等等），剥离了二进制文件。

这些命令 make menuconfig | nconfig | gconfig | xconfig 和 make 是基本的命令，它们可以轻松、快速地生成符合您需要的镜像，并带有您启用的所有功能和应用程序。

有关“make”命令用法的更多详细信息，请参阅第 8.1 节。

5 社区资源

像任何开源项目一样，Buildroot 能够通过不同的方式在社区内部和外部来共享信息。

如果您想了解 Buildroot 或者想为项目做贡献，而正在寻找帮助，那么下面每种方式都可能使您感兴趣。

Mailing List

Buildroot 有一个邮件列表，用于讨论和开发。它是 Buildroot 用户和开发人员进行交互的主要方式。只有 Buildroot 邮件列表的订阅者才可以发布到此列表。您可以通过邮件列表信息页进行订阅。

发送到邮件列表的邮件也可以在邮件列表存档中获取，和通过 Gmane 在 gmane.comp.lib.uclibc.buildroot（*译者注：已失效*）中获取。在提出问题之前，请先搜索邮件列表，因为很可能其他人在以前也问过同样的问题。

IRC

Buildroot IRC 频道“#buildroot”托管于 Freenode。这是一个提出简易问题或讨论某些主题

的有用地方。

在 IRC 上寻求帮助时，请使用代码共享网站共享相关日志或代码段，例如 http://code.bulix.org（*译者注：已失效*）。请注意，对于某些问题，将其发布到邮件列表可能会更好，因为它将吸引更多的开发人员和用户。

Bug tracker

Buildroot 中的 Bugs 可以通过邮件列表或通过 Buildroot bugtracker 来上报。在创建一个 bug 报告之前，请参阅第 21.6 节。

Wiki

Buildroot Wiki 页面托管于 eLinux wiki。它包含一些有用的链接、对过去和即将发生的事件的概述以及 TODO 列表。

Patchwork

Patchwork 是基于 Web 的补丁跟踪系统，旨在促进对开源项目的贡献和管理。已发送到邮件列表的补丁会被该系统“捕获”，并显示在网页上。对引用该补丁的任何评论也将被附加到补丁页面。有关 Patchwork 的更多信息，请参阅 http://jk.ozlabs.org/projects/patchwork/。

Buildroot 的 Patchwork 网站主要供 Buildroot 的维护者使用，以确保不会丢失补丁。Buildroot 补丁审核者也使用它（请参阅第 21.3.1 节）。同时，由于该网站 Web 界面公开了补丁情况及相应的审查评论，因此它对所有 Buildroot 开发人员都非常有用。

Buildroot 补丁管理界面位于：http://patchwork.buildroot.org。

第二部分 用户指南

6 Buildroot 配置

make *config 中的所有配置选项都有一个帮助文本，它能够提供有关该选项的详细信息。

make *config 命令还提供了搜索工具。在不同的前台菜单中来查看帮助消息，以了解如何使用它：

- 在 menuconfig 中，按“/”调用搜索工具。
- 在 xconfig 中，按 Ctrl+f 调用搜索工具。

搜索结果将显示匹配目标的帮助信息。在 menuconfig 中，左栏中的数字提供了相应条目的快捷方式。只需输入该数字即可直接跳转至该条目，或者在因缺少依赖而无法选择该条目的情况下跳转至包含该条目的菜单。

尽管条目的菜单结构和帮助文本应该充分解释清楚自身，但一些主题难以在帮助文本中完全覆盖到，而需要进行额外的说明，因此将在下面各节中对它们进行介绍。

6.1 交叉编译工具链

编译工具链是一组允许您为目标系统编译代码的工具。它由一个编译器（在我们环境下为 gcc）、二进制工具（比如汇编器和链接器，在我们环境下为 binutils）以及一个 C 标准库（如 GNU Libc、uClibc-ng)组成。

安装在开发工作站上的系统肯定已有一个编译工具链，可用它来编译在系统上运行的应用程序。如果使用的是 PC，则该编译工具链可在 x86 处理器上运行，并为 x86 处理器生成代码。在大多数 Linux 系统中，编译工具链使用 GNU libc（glibc）作为 C 标准库。这种编译工具链称为“宿主机编译工具链”。工具链在上面运行并且您在其上工作的计算机称为“宿主机系统”[1]。

编译工具链由您的发行版提供，而 Buildroot 与它无关（除了使用它来构建交叉编译工具链和在开发主机上运行的其他工具）。

如上所述，系统随附的编译工具链可在宿主机上运行并为宿主机系统中的处理器生成代码。由于您的嵌入式系统具有不同的处理器，因此您需要交叉编译工具链-一种能在您的主机系统上运行但为目标系统（和目标处理器）生成代码的编译工具链。例如，如果您的主机系统使用 x86，而目标系统使用 ARM，则主机上的常规编译工具链是在 x86 上运行并为 x86 生成代码，而交叉编译工具链是在 x86 上运行并为 ARM 生成代码。

Buildroot 为交叉编译工具链提供两种解决方案：

- 内部工具链后端，在配置界面中调用“Buildroot toolchain”。
- 外部工具链后端，在配置界面中调用“External toolchain”。

可以在“Toolchain”菜单下“Toolchain Type”选项中进行这两种方案的选择。选择一种方案后，将出现许多配置选项，以下各节将详细介绍它们。

[1] 此术语不同于 GNU configure 所使用的术语，宿主机是指能够在上面运行应用程序的机器（通常与目标计算机相同）

6.1.1 内部工具链后端

内部工具链后端是 Buildroot 在构建目标嵌入式系统用户应用程序和库之前，自行构建一个交叉编译工具链。

该后端支持几个 C 库：uClibc-ng，glibc 和 musl。

选择该后端后，将显示许多配置选项。其中最重要的是允许：

• 更改用于构建工具链的 Linux 内核头文件的版本。这个选项简要说明下。在构建交叉编译工具链的过程中，C 库也会被构建。该库提供了用户空间应用程序和 Linux 内核之间的接口。为了知道如何与 Linux 内核进行“对话”，C 库需要访问 Linux 内核头文件（即来自内核的.h 文件），这些头文件定义了用户空间和内核之间的接口（系统调用，数据结构等）。由于此接口是向后兼容的，因此用于构建工具链的 Linux 内核头文件的版本不需要与打算在目标嵌入式系统上运行的 Linux 内核的版本完全匹配。它们只需与要运行的 Linux 内核具有相同或更低的版本即可。如果所使用的内核头文件比在嵌入式系统上运行的 Linux 内核的头文件更新，则 C 库可能正在使用 Linux 内核尚未提供的接口。

• 更改 GCC 编译器、binutils 和 C 库三者的版本。

• 选择一些工具链选项（仅限 uClibc）：工具链是否应该具有 RPC 支持（主要用于 NFS）、宽字符支持、语言环境支持（用于国际化）、C++支持或者线程支持。根据您所选择的选项，Buildroot 菜单中可见的用户空间应用程序和库的数量将会发生变化：许多应用程序和库需要开启某些工具链选项。当需要指定某个工具链选项才能启用那些软件包时，大多数软件包会显示提示。如果需要，您可以通过运行“make uclibc-menuconfig”来进一步优化 uClibc 配置。请注意，尽管已针对 Buildroot 中附带的 uClibc 默认配置所涉及的所有软件包进行了测试，但如果通过从 uClibc 中删除功能而使之偏离该默认配置，则某些软件包可能不再构建。

值得注意的是，只要修改这些选项中的其中一个，就必须重新构建整个工具链和系统。请参阅第 8.2 节。

该后端的优点：

• 与 Buildroot 良好集成；

• 快速，只编译必要的；

此后端的缺点：

• 进行“make clean”清理时需要重新构建工具链，这需要时间。如果您想减少构建时间，请考虑使用外部工具链后端。

6.1.2 外部工具链后端

外部工具链后端允许使用现有的预先构建好的交叉编译工具链。Buildroot 集成许多著名的交叉编译工具链，如 Linaro for ARM、Sourcery CodeBenc for ARM、x86-64、PowerPC 和 MIPS，并且能够自动下载它们，也可以指定可供下载的或本地已安装的自定义工具链。

然后，您有三种方法来使用外部工具链：

• 使用预定义的外部工具链配置文件，并让 Buildroot 下载、提取和安装工具链。Buildroot 已经集成了一些 CodeSourcery 和 Linaro 工具链，只需在“Toolchain”可用选项中选择工具链配置文件。这绝对是最简单的解决方案。

• 使用预定义的外部工具链配置文件，告诉 Buildroot 您的工具链在系统中的安装位置，而不是让 Buildroot 去下载并提取工具链。只需在“Toolchain”可用选项中选择工具链配置文件，不需要选择“Download toolchain automatically”，然后在“Toolchain path”文本条目中填入交叉编译工具链的路径即可。

• 使用完全自定义的外部工具链。这对于使用 crosstool-NG 或 Buildroot 本身生成的工具链特别有用。为此，请在“Toolchain”列表中选择“Custom toolchain”条目。您需要填写“Toolchain path”、“Toolchain prefix”和“External toolchain C library”选项。然后，您必须告诉 Buildroot 您的外部工具链支持什么。如果您的外部工具链使用 glibc 库，则只需告诉 Buildroot 您

工具链是否支持 C++，以及是否具有内置的 RPC 支持。如果您的外部工具链使用 uClibc 库，那么您必须告诉 Buildroot 它是否支持 RPC、宽字符、语言环境、程序调用、线程和 C++。在执行开始时，如果所选的选项与工具链配置不匹配，则 Buildroot 会告诉您。

我们外部工具链支持 CodeSourcery 和 Linaro 的工具链、由 crosstool-NG 生成的工具链以及由 Buildroot 本身生成的工具链。通常来说，所有支持 sysroot 功能的工具链都应该正常工作。如果没有，请与 Buildroot 开发人员联系。

我们不支持由 OpenEmbedded 或 Yocto 生成的工具链或 SDK，因为它们不是纯粹的工具链（即仅编译器、binutils、C 和 C++库）。相反，这些工具链带有大量预编译的库和程序。因此，Buildroot 不能导入这些工具链的 sysroot，因为它将包含数百兆字节预编译库会被 Buildroot 正常构建。

我们也不支持使用发行版工具链（例如发行版中安装的 gcc/binutils/C 库）作为工具链来构建目标软件。这是因为发行版工具链不是“纯”工具链（即仅使用 C/C++库），因此我们无法将其正确导入到 Buildroot 构建环境中。因此，即使您正在为 x86 或 x86_64 目标构建系统，您务必使用 Buildroot 或 crosstool-NG 来生成交叉编译工具链。

如果您想为自己项目生成一个能够在 Buildroot 中用作外部工具链的自定义工具链，我们建议是使用 Buildroot 本身来构建（请参阅第 6.1.3 节）或使用 crosstool-NG 来构建。

该后端的优点：

- 允许使用众所周知的并且经过充分测试的交叉编译工具链。
- 避免了交叉编译工具链的构建时间，这通常在嵌入式 Linux 系统的整体构建时间中非常重要。

该后端的缺点：

- 如果您预先构建的外部工具链存在 bug，则这可能很难从工具链供应商处获得修复，除非您自己使用 Buildroot 或 Crosstool-NG 来构建外部工具链。

6.1.3 使用 Buildroot 构建外部工具链

Buildroot 内部工具链选项可用于创建外部工具链。以下是构建一个内部工具链并将其打包以供 Buildroot 本身（或其他项目）重复使用的一系列步骤。

使用以下详细信息来创建一个新的 Buildroot 配置：

- 为您的目标 CPU 架构选择适当的“Target options”；
- 在“Toolchain”菜单中，“Toolchain type”选项保留默认的“Buildroot toolchain”，并根据需要配置工具链；
- 在“System configuration”菜单中，“Init system”选项选择“None”，“/bin/sh”选项选择“none”；
- 在“Target packages”菜单中，禁用“BusyBox”；
- 在“Filesystem images”菜单中，禁用“tar the root filesystem”；

然后，我们可以启动构建，让 Buildroot 生成 SDK。以下命令将方便地为我们生成一个包含工具链的压缩包：

```
make sdk
```

这会在$(O)/images 目录下生成 SDK 压缩包，其名称类似于 arm-buildroot-linux-uclibcgnueabi_sdk-buildrot.tar.gz。请保存此压缩包，因为您可以在其他 Buildroot 项目中将它用作外部工具链。

在那些其他的 Buildroot 项目中，需在“Toolchain”菜单中设置：

- 将“Toolchain type”设置为“External toolchain”；

- 将“Toolchain”设置为“Custom toolchain”；
- 将“Toolchain origin”设置为“Toolchain to be downloaded and installed”；
- 将“Toolchain URL”设置为“file:///path/to/your/sdk/tarball.tar.gz”。

6.1.3.1 外部工具链包装器

使用外部工具链时，Buildroot 会生成一个包装程序，该程序会将合适的选项设置（根据配置）透传给外部工具链程序。如果您需要调试此包装器以检查传递了哪些参数，可以将环境变量 BR2_DEBUG_WRAPPER 设置为以下任意一个：

- 0，为空或未设置：无调试
- 1，在单独一行上跟踪所有参数
- 2，每行跟踪一个参数

6.2 /dev 管理

在 Linux 系统上，/dev 目录包含特殊文件，即设备文件，这些文件允许用户空间应用程序访问由 Linux 内核管理的硬件设备。没有这些设备文件，即使 Linux 内核正确识别了硬件设备，您的用户空间应用程序也将无法使用它们。

在“System configuration”选项下的“/dev management”，Buildroot 提供了四种方式来处理/dev 目录：

- 第一种方式是“Static using device table”。这是 Linux 处理设备文件的传统方法。使用这种方法，设备文件会被持久存储在根文件系统中（即重新启动后它们仍然存在），并且在系统添加或者移除硬件设备时，不能自动创建和删除这些设备文件。因此，Buildroot 使用设备表来创建一组标准的设备文件，默认设备表存储在 Buildroot 源代码的 system/device_table_dev.txt 文件中。Buildroot 在生成最终根文件系统镜像时才会处理该设备表文件，因此设备文件在 output/target 目录中不可见。BR2_ROOTFS_STATIC_DEVICE_TABLE 选项允许更改 Buildroot 默认使用的设备表，或者添加其他设备表，以便 Buildroot 在构建过程中可以创建其他设备文件。因此，如果您使用此方法，并且系统中缺少设备文件，则可以创建一个包含其他设备文件描述的 board/<yourcompany>/<yourproject>/device_table_dev.txt 文件，然后将 BR2_ROOTFS_STATIC_DEVICE_TABLE 设置为“system/device_table_dev.txt board/<yourcompany>/<yourproject>/device_table_dev.txt”。关于设备表文件格式的更多详细信息，请参阅第 24 章。
- 第二种方式是“Dynamic using devtmpfs only”。devtmpfs 是 Linux 内核中的一个虚拟文件系统，已在内核 2.6.32 中引入（如果使用较旧的内核，则无法使用此选项）。在挂载到/dev 后，此虚拟文件系统将在系统添加或者移除硬件设备时自动使设备文件显示或消失。该文件系统在重新启动后并不持久：它是由内核动态填充的。使用 devtmpfs 时需要启用以下内核配置选项：CONFIG_DEVTMPFS 和 CONFIG_DEVTMPFS_MOUNT。当 Buildroot 负责为您的嵌入式设备构建 Linux 内核时，请确保启用了这两个选项。但是如果您是在 Buildroot 之外构建 Linux 内核，则您有责任去启用这两个选项（如果不这么做，则 Buildroot 构建的系统将不会启动）。
- 第三种方式是“Dynamic using devtmpfs+mdev”。该方法依赖于上面介绍的 devtmpfs 虚拟文件系统（因此同样需要在内核配置中启用 CONFIG_DEVTMPFS 和 CONFIG_DEVTMPFS_MOUNT），但在上面添加了 mdev 用户空间程序。mdev 是 BusyBox 里面的程序部分，每次添加或移除设备时，内核都会调用 mdev。由于使用/etc/mdev.conf 配置文件，因此 mdev 可以进行相关配置，例如给设备文件设置特定的权限或所有权、在设备出现或消失时调用脚本或应用程序等等。基本上，它允许用户空间对设备添加和删除事件做出反应。例如，当设备出现在系统上时，mdev 可用于自动加载内核模块。如果您的设备需要固件，则 mdev 也很重要，因为它会

负责将固件内容推送到内核。mdev 是 udev 的轻量级实现（功能较少）。关于 mdev 及其配置文件语法的更多详细信息，请参阅 http://git.busybox.net/busybox/tree/docs/mdev.txt。

• 第四种方式是“Dynamic using devtmpfs+eudev”。此方法同样依赖于上面介绍的 devtmpfs 虚拟文件系统，但在上面添加了 eudev 用户空间守护程序。eudev 是后台运行的守护程序，当系统添加或者移除设备时，内核将会调用它。与 mdev 相比，它是重量级的解决方案，但是具有更高的灵活性。eudev 是 udev 的独立版本，udev 是大多数桌面 Linux 发行版中使用的原始用户空间守护程序，现已成为 Systemd 的一部分。更多详细信息，请参阅 http://en.wikipedia.org/wiki/Udev。

Buildroot 开发人员建议是从“Dynamic using devtmpfs only”方式开始，直到您需要在添加或者删除设备时通知用户空间。如果需要固件，则“Dynamic using devtmpfs+mdev”通常是个不错的解决方案。

请注意，如果选择 systemd 作为初始化系统，则/dev 管理将由 systemd 提供的 udev 程序执行。

6.3 初始化系统

init 程序是由内核启动的第一个用户空间程序（带有 PID 号 1），它负责启动用户空间服务和程序（例如：Web 服务器、图形应用程序、其他网络服务器等）。

Buildroot 允许使用三种不同类型的初始化系统，可以在“System configuration”选项下的“Init system”进行选择：

• 第一种是“BusyBox”。BusyBox 实现了基本的 init 程序，对于大多数嵌入式系统而言，这已经足够了。启用“BR2_INIT_BUSYBOX”将确保 BusyBox 会生成并安装其 init 程序。这是 Buildroot 的默认解决方案。BusyBox init 程序会在启动时去读取/etc/inittab 文件，以了解需要处理的内容。inittab 文件的语法介绍位于：http://git.busybox.net/busybox/tree/examples/inittab（请注意，BusyBox inittab 语法很特殊，请勿使用 Internet 上随意的 inittab 文档来了解 BusyBox inittab）。Buildroot 默认的 inittab 文件存储在 system/skeleton/etc/inittab（*译者注：经验证，本文 Buildroot 版本此路径没有该文件，可采用 package/busybox/inittab*）。除了挂载一些重要的文件系统之外，默认的 inittab 的主要工作是启动/etc/init.d/rcS shell 脚本，并启动一个 getty 程序（提供登录提示）。

• 第二种是“systemV”。该解决方案使用传统的 sysvinit 程序，位于 Buildroot 目录 package/sysvinit，这是大多数桌面 Linux 发行版使用的解决方案，直到它们被切换到更新的替代版本（例如 Upstart 或 Systemd）。Sysvinit 同样使用 inittab 文件（其语法与 BusyBox 中的语法略有不同）。与此 init 解决方案一起安装的默认 inittab 位于 package/sysvinit/inittab。

• 第三种是“systemd”。systemd 是用于 Linux 的新一代 init 系统。它的功能远远超过传统的 init 程序：强大的并行处理能力、使用 socket 和 D-Bus 激活启动服务、按需启动守护程序、使用 Linux 控制组跟踪进程、支持对系统状态进行快照和还原等等。systemd 在相对复杂的嵌入式系统上很有用，例如需要 D-Bus 和服务之间相互通信的系统。值得注意的是 systemd 带来了大量的大型依赖项：dbus、udev 等。有关 systemd 的更多详细信息，请参阅 http://www.freedesktop.org/wiki/Software/systemd。

Buildroot 开发人员推荐的解决方案是使用“BusyBox”作为初始化系统，因为它对于大多数嵌入式系统来说已经够用了。“systemd”可用于更复杂的情况。

7 其他组件的配置

在尝试修改下面任何组件之前，请确保您已经配置过 Buildroot 本身，并启用相应的软件包。

- **BusyBox**

如果您已经有一个 BusyBox 配置文件，则可以在 Buildroot 配置中使用“BR2_PACKAGE_BUSYBOX_CONFIG”直接指定该文件。否则，Buildroot 将使用默认的 BusyBox 配置文件。如果要对配置进行后续更改，请使用“make busybox-menuconfig”打开 BusyBox 配置编辑器。也可以通过环境变量指定 BusyBox 配置文件，尽管不建议这样做。更多详细信息，请参阅第 8.6 节。

- **uClibc**

uClibc 的配置方式与 BusyBox 相同。用于指定现有配置文件的配置变量是 BR2_UCLIBC_CONFIG。进行后续更改的命令是“make uclibc-menuconfig”。

- **Linux kernel**

如果已经有了内核配置文件，则可以在 Buildroot 配置中使用“BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG”变量直接指定该文件。

如果还没有内核配置文件，则可以在 Buildroot 配置中使用“BR2_LINUX_KERNEL_USE_DEFCONFIG”指定一个 defconfig 文件，或者创建一个空文件并使用“BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG”将其指定为自定义配置文件。

要对配置进行后续更改，请使用“make linux-menuconfig”打开 Linux 配置编辑器。

- **Barebox**

Barebox 的配置方式与 Linux kernel 配置相同。相应的配置变量是“BR2_TARGET_BAREBOX_USE_CUSTOM_CONFIG”和“BR2_TARGET_BAREBOX_USE_DEFCONFIG”。使用“make barebox-menuconfig”打开配置编辑器。

- **U-Boot**

U-Boot（版本 2015.04 或更高版本）的配置方式与 Linux kernel 配置相同。相应的配置变量是“BR2_TARGET_UBOOT_USE_CUSTOM_CONFIG”和“BR2_TARGET_UBOOT_USE_DEFCONFIG”。使用“make uboot-menuconfig”打开配置编辑器。

8 Buildroot 的一般用法

8.1 make 技巧

这是一系列技巧，可帮助您充分利用 Buildroot。

显示 make 执行的所有命令：

```
$ make V=1 <target>
```

显示带有 defconfig 的板级列表：

```
$ make list-defconfigs
```

显示所有可用的目标：

```
$ make help
```

并非所有目标都始终可用，“.config”文件中的某些设置可能会隐藏一些目标：

- busybox-menuconfig 仅在启用 busybox 时有效；
- linux-menuconfig 和 linux-savedefconfig 仅在启用 linux 时有效；
- uclibc-menuconfig 仅在内部工具链后端选择 uClibc C 库时可用；
- beadbox-menuconfig 和 beardbox-savedefconfig 仅在启用 barebox 引导加载程序时有效；
- uboot-menuconfig 和 uboot-savedefconfig 仅在启用 U-Boot 引导加载程序时有效；

清理：当更改体系架构或工具链任何配置选项时，需要进行显式清理。要删除所有构建时生成的产物（包括 build、host、staging and target trees、images 和 toolchain）：

```
$ make clean
```

生成手册：当前的手册位于 docs/manual 目录中。要生成手册：

```
$ make manual-clean
$ make manual
```

将在 output/docs/manual 中生成手册。

请注意：

- 需要一些工具来构建文档（请参阅 2.2 节）。

为新目标重置 Buildroot：删除所有构建时的产物以及相关配置：

```
$ make distclean
```

请注意，如果启用了“ccache”，则运行“make clean”或 distclean 不会清空 Buildroot 所使用的编译器缓存。要删除它，请参阅第 8.13.3 节。

转储内部 make 变量：可以转储 make 已知的变量，以及它们的值：

```
$ make -s printvars VARS='VARIABLE1 VARIABLE2'
VARIABLE1=value_of_variable
VARIABLE2=value_of_variable
```

可以使用一些变量来调整输出：

- VARS，会限制列表中的变量名称与指定的 make-patterns 相匹配，必须设置此项，否则不会打印任何内容；
- QUOTED_VARS，如果设置为 YES，则将给变量值赋上单引号；
- RAW_VARS，如果设置为 YES，则将打印未扩展的值。

例如：

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES
BUSYBOX_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_ALL_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_PATCH_DEPENDENCIES=
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES QUOTED_VARS=YES
BUSYBOX_DEPENDENCIES='skeleton toolchain'
BUSYBOX_FINAL_ALL_DEPENDENCIES='skeleton toolchain'
BUSYBOX_FINAL_DEPENDENCIES='skeleton toolchain'
BUSYBOX_FINAL_PATCH_DEPENDENCIES=''
BUSYBOX_RDEPENDENCIES='ncurses util-linux'
```

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES RAW_VARS=YES
BUSYBOX_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_ALL_DEPENDENCIES=$(sort $(BUSYBOX_FINAL_DEPENDENCIES) $(BUSYBOX_FINAL_PATCH_DEPENDENCIES))
BUSYBOX_FINAL_DEPENDENCIES=$(sort $(BUSYBOX_DEPENDENCIES))
BUSYBOX_FINAL_PATCH_DEPENDENCIES=$(sort $(BUSYBOX_PATCH_DEPENDENC
```

```
IES))
    BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

输出结果为带引号的变量可以在 shell 脚本中重复使用，例如：

```
$ eval $(make -s printvars VARS=BUSYBOX_DEPENDENCIES QUOTED_VARS=YES)
$ echo $BUSYBOX_DEPENDENCIES
skeleton toolchain
```

8.2 了解何时需要完全重新构建

当通过“make menuconfig”、“make xconfig”或其他配置工具更改系统配置时，Buildroot 不会尝试检测应重建系统的哪些部分。在某些情况下，Buildroot 应当重建整个系统，而在某些情况下，仅应重建软件包的特定部分。要以完全可靠的方式去检测是非常困难，因此 Buildroot 开发人员已决定不尝试这样做。

相反，用户有责任知道 Buildroot 何时需要完全重建。作为提示，以下有一些经验法则可以帮助您了解如何使用 Buildroot：

- 更改目标体系架构配置时，需要完全重建。例如，更改体系架构变体、二进制格式或浮点策略等，这些更改会影响整个系统。
- 更改工具链配置时，通常需要完全重建。更改工具链配置通常涉及更改编译器版本、C 库的类型或其配置、其他一些基本配置项等，这些更改会影响整个系统。
- 在配置中新增软件包时，不一定需要完全重建。Buildroot 将检测到此软件包从未构建过，并对它进行构建。但是，如果此软件包是由已构建过的软件包可选择性使用的库时，则 Buildroot 将不会自动重建它们。除非您知道应该重建哪些软件包，进而手动重建它们，否则应该进行完全重建。例如，假设您构建了一个具有 ctorrent 软件包的系统，但没有 openssl。系统工作后，您意识到您想要 ctorrent 提供 SSL 支持，因此您在 Buildroot 配置中启用了 openssl 软件包并重新进行构建。Buildroot 将检测到 openssl 应该被构建，并进行构建，但是它不会检测到应该重新构建 ctorrent 以受益于 openssl 软件包来增加 OpenSSL 支持。您务必要进行完全重建，或者重建 ctorrent 本身。
- 从配置中删除软件包时，Buildroot 不会执行任何特殊操作。它不会从目标根文件系统或工具链 sysroot 中删除该软件包所安装的文件。需要完全重建才能移除此软件包。但是，通常您不必立即删除此软件包：您可以等待下一个午餐休息时间以从头开始构建。
- 更改软件包子选项时，不会自动重建软件包。进行此类更改后，仅重建该软件包通常就足够了，除非启用子选项后向该软件包中添加了一些对已构建的另一个软件包很有用的功能。同样，Buildroot 不会跟踪何时应该重新构建软件包：一旦构建了软件包，就不会对其进行重新构建，除非明确要求这样做。
- 更改根文件系统框架（skeleton）时，需要完全重建。但是，在更改根文件系统 overlay 层、post-buid 脚本或 post-image 脚本时，不需要完全重建：简单的 make 调用会将它们考虑在内。
- 重建或删除由“FOO_DEPENDENCIES”列出的软件包时，软件包 foo 不会自动重建。例如，如果软件包 bar 在“FOO_DEPENDENCIES”中通过“FOO_DEPENDENCIES = bar”配置列出，并且 bar 软件包的配置已更改，则该配置更改将不会引起软件包 foo 自动重建。在这种情况下，您可能需要重新构建在其“DEBENDENCIES”中引用 bar 的任何软件包，或者执行完全重建以确保任何依赖于 bar 的软件包都是最新的。*（译者注：“DEBENDENCIES”是 Buildroot 用于指定各个软件包依赖关系的配置变量，通常命名为**XXX**_DEBENDENCIES）*

一般来说，当您遇到构建错误并且不确定所做的配置更改可能带来的后果时，请进行完全重建。如果您遇到相同的构建错误且确定该错误与软件包的部分重建无关，并且如果此错误发

生在官方 Buildroot 的软件包中，请立即报告该问题！随着您经验的增多，您将逐步了解何时才真正需要完全重建 Buildroot，并且可以节省越来越多的时间。

作为参考，可以通过运行以下命令来进行完全重建：*（译者注：以下命令是清除全部配置）*

```
$ make clean all
```

8.3 了解如何重建软件包

Buildroot 用户提出的最常见问题之一是在未从头开始重新构建所有内容的情况下如何重建给定的软件包或如何删除软件包。

Buildroot 在尚未从头开始重新构建的情况下不支持删除软件包。这是因为 Buildroot 无法跟踪哪个软件包在 output/staging 和 output/target 目录中安装了哪些文件，或者哪个软件包将根据另一个软件包的用途进行不同的编译。

从头开始重建单个软件包的最简单方法是在 output/build 中删除其构建目录。然后，Buildroot 将从头开始重新提取、重新配置、重新编译和重新安装此软件包。您可以要求 Buildroot 使用“make <package>-dirclean”命令来执行此操作。

另一方面，如果您只想从 build 构建步骤重新启动软件包的构建过程，则可以运行“make <package>-rebuild”。它将重新启动软件包的编译和安装，但不会从头开始：它只是重新执行软件包内部的“make”和“make install”，因此它仅重建已更改的文件。

如果您想从 configuration 配置步骤重新启动软件包的构建过程，可以运行“make <package>-reconfigure”。它将重新启动软件包的配置、编译和安装。

虽然<package>-rebuild 包含<package>-reinstall，<package>-reconfigure 包含<package>-rebuil，但这些命令以及<package>只作用于指定的软件包，并不会触发创建新的根文件系统镜像。如果有必要重新创建根文件系统，则应另外运行“make”或“make all”。

在内部，Buildroot 通过创建 stamp 文件来跟踪每个软件包完成了哪些构建步骤。它们保存在软件包构建目录中，位于 output/build/<package>-<version>/，并命名为.stamp_<step-name>。上面详细介绍的命令仅仅操作这些 stamp 文件即可强制 Buildroot 来重新执行软件包构建过程中的特定步骤。

有关软件包特定 make 目标的更多详细信息，请参阅第 8.13.5 节。

8.4 离线构建

如果您打算进行离线构建，只想下载之前在配置菜单（menuconfig，nconfig，xconfig 或 gconfig）中选择过的所有源程序，则执行：

```
$ make source
```

此时您可以断开连接，或者可以将 dl 目录中的内容复制到执行构建的主机。

8.5 目录树外构建

默认情况下，Buildroot 构建后生成的所有内容都保存在 Buildroot 目录树的 output 目录中。

Buildroot 还支持使用类似 Linux 内核中的语法，从目录树外进行构建。要使用它，请将“O=<directory>”添加到 make 命令行：

```
$ make O=/tmp/build
```

或者：

```
$ cd /tmp/build; make O=$PWD -C path/to/buildroot
```

所有输出的文件将位于/tmp/build 下。如果“O”指定的路径目录不存在，则 Buildroot 将创

建它。

注意：“O”指定的路径可以是绝对路径，也可以是相对路径，但如果是相对路径，则需要注意，它是相对于 Buildroot 的主目录而不是当前工作目录。

使用目录树外构建时，Buildroot 的.config 文件和临时文件也保存在输出目录中（*译者注：需执行“make menuconfig O=<directory>”*）。这意味着只要指定唯一的输出目录，就可以在同一源码目录树下安全且并行地执行多个构建。

为了易于使用，Buildroot 在输出目录中生成一个 Makefile 包装器。因此，在第一次运行后，您不再需要传递 O=<...>和-C <...>，只需运行（在输出目录中）：

```
$ make <target>
```

8.6 环境变量

Buildroot 支持一些环境变量，可传递给 make 或者用于设置环境：

- HOSTCXX，宿主机 C++编译器需要使用；
- HOSTCC，宿主机 C 编译器需要使用；
- UCLIBC_CONFIG_FILE=<path/to/.config>，这是 uClibc 配置文件的路径，在构建内部工具链时用于编译 uClibc；

注意，也可以通过 Buildroot .config 文件从配置界面设置 uClibc 配置文件；这是建议的设置方式。

- BUSYBOX_CONFIG_FILE=<path/to/.config>，这是 BusyBox 配置文件的路径；

注意，也可以通过 Buildroot .config 文件从配置界面设置 BusyBox 配置文件；这是建议的设置方式。

- BR2_CCACHE_DIR，用于覆盖 Buildroot 在使用 ccache 时存储 cache 文件的目录；
- BR2_DL_DIR，用于覆盖 Buildroot 存储/检索已下载文件的目录；

请注意，也可以通过 Buildroot .config 文件从配置界面设置 Buildroot 下载目录。有关如何设置下载目录的更多详细信息，请参阅 8.13.4 节。

- BR2_GRAPH_ALT，如果已设置且为非空值，则在 build-time 图表中使用交替颜色方案；
- BR2_GRAPH_OUT，设置生成图表的文件类型，可以是 pdf（默认值）或 png；
- BR2_GRAPH_DEPS_OPTS，将额外的选项传递给依赖关系图；有关可接受的选项，请参阅第 8.8 节。
- BR2_GRAPH_DOT_OPTS，作为选项逐个传递给 dot 程序以绘制依赖关系图；
- BR2_GRAPH_SIZE_OPTS 将额外的选项传递给 size 图；有关可接受的选项，请参阅第 8.10 节。

使用位于顶层目录和$HOME 中的配置文件的示例：

```
$ make UCLIBC_CONFIG_FILE=uClibc.config BUSYBOX_CONFIG_FILE=$HOME/bb.config
```

如果要在主机上使用除了默认 gcc 或 g++以外的其他编译器来构建 helper-linaries，请执行：

```
$ make HOSTCXX=g++-4.3-HEAD HOSTCC=gcc-4.3-HEAD
```

8.7 有效处理文件系统镜像

文件系统镜像可能会变得非常大，具体取决于您所选择的文件系统、软件包数量以及是否配置可用空间…但是，文件系统镜像中的某些位置可能是空的（例如长时间为零）；这样的文件称为稀疏文件（*译者注：原文为 sparse file*）。

大多数工具可以有效地处理稀疏文件，并且只会存储或写入稀疏文件中不为空的部分。例如：

- tar 接受-S 选项，以告诉它仅存储稀疏文件的非零块：
  - tar cf archive.tar -S [files ...]将有效地将稀疏文件存储在压缩包中；
  - tar xf archive.tar -S 将有效地从压缩包中提取稀疏文件；
- cp 接受--sparse=WHEN 选项（WHEN 可选 auto，never 或 always）：
  - cp --sparse=always source.file dest.file 如果 source.file 长时间为零，则生成 dest.file 稀疏文件；

其他工具可能具有类似的选项，请查阅对应的手册页。

如果您需要存储文件系统镜像（例如从一台计算机传输到另一台计算机），或者需要将其发送出去（例如发送给 Q&A 团队），则可以使用稀疏文件。

但是请注意，当使用 dd 命令的稀疏模式来将文件系统镜像烧写到设备时，可能会导致文件系统受损（例如，ext2 文件系统的块位图可能会受损；或者，如果文件系统中存在稀疏文件，则这部分有可能读回时不是全零）。您仅应在执行构建的计算机上处理文件时使用稀疏文件，而不是在将它们传输到目标实际设备时使用。

8.8 绘制软件包之间的依赖关系图

Buildroot 的工作之一就是了解软件包之间的依赖关系，并确保它们以正确的顺序构建。这些依赖关系有时可能相当复杂，对于给定的系统，通常不容易理解为什么 Buildroot 会将这样的软件包引入到构建过程。

为了帮助理解依赖关系，从而更好地理解嵌入式 Linux 系统中不同组件的作用，Buildroot 能够生成依赖关系图。

要生成已编译过的整个系统的依赖关系图，只需运行：

```
make graph-depends
```

您可以在 output/graphs/graph-depends.pdf 中找到生成的关系图。

如果您的系统相当大，则依赖关系图可能过于复杂且难以阅读。因此，可以只为给定的软件包生成依赖关系图：

```
make <pkg>-graph-depends
```

您可以在 output/graph/<pkg>-graph-depends.pdf 中找到生成的关系图。

请注意，依赖关系图是使用 Graphviz 项目中的 dot 工具生成的，您必须已将其安装在系统上才能使用此功能。在大多数发行版中，可以通过“graphviz”软件包获取。

默认情况下，依赖关系图以 PDF 格式生成。但是，通过传递 BR2_GRAPH_OUT 环境变量，可以切换到其他输出格式，例如 PNG、PostScript 或 SVG。支持 dot 工具中-T 选项支持的所有格式。

```
BR2_GRAPH_OUT=svg make graph-depends
```

可以通过 BR2_GRAPH_DEPS_OPTS 环境变量设置选项来控制“graph-depends”的行为。可接受的选项是：

- --depth N，-d N，将依赖深度限制为 N 级别。默认值为 0，表示没有限制。
- --stop-on PKG，-s PKG，结束软件包 PKG 的依赖绘图。PKG 可以是实际的软件包名称、全局名称、关键字 virtual（虚拟软件包）或关键字 host（宿主机软件包）。该软件包依然存在于图上，但没有依赖关系。*（译者注：应用示例 BR2_GRAPH_DEPS_OPTS='--stop-on toolchain' make graph-depends）*
- --exclude PKG，-x PKG，如同--stop-on，但在关系图中将忽略 PKG。
- --transitive，--no-transitive，绘制（或不绘制）传递依赖关系。默认不绘制传递依赖关系。
- --colors R，T，H，以逗号分隔的颜色列表，用于绘制 root 软件包（R）、target 软件包（T）

和 host 软件包（H）。默认为：lightblue，grey，gainsboro。

```
BR2_GRAPH_DEPS_OPTS='-d 3 --no-transitive --colors=red,green,blue' make graph-depends
```

8.9 绘制构建持续时间图

当系统构建耗时很长时，若能够了解哪些软件包构建耗时最长，将有助于采取措施来加快构建速度。为了帮助进行此类构建时间分析，Buildroot 收集了每个软件包每个步骤的构建时间，并允许使用该数据来生成图表。

要在构建后生成构建时间图，请运行：

```
make graph-build
```

这将在 output/graphs 中生成一系列文件：

- build.hist-build.pdf，每个软件包的构建时间直方图，按构建顺序排序；
- build.hist-duration.pdf，每个软件包的构建时间直方图，按持续时间排序（最长的一个）；
- build.hist-name.pdf，每个软件包的构建时间直方图，按软件包名称排序；
- build.pie-packages.pdf，每个软件包的构建时间饼图；
- build.pie-steps.pdf，软件包构建过程中每个步骤所耗费的全部时间的饼图。

该“graph-build”目标需要安装 Python Matplotlib 和 Numpy 库（在大多数发行版中安装包为 python-matplotlib 和 python-numpy），如果您使用的 Python 版本早于 2.7，则还需要安装 argparse 模块（在大多数发行版中安装包为 python-argparse）。

默认情况下，图表输出的格式为 PDF，但可以使用 BR2_GRAPH_OUT 环境变量来选择其他格式。支持的唯一其他格式是 PNG：

```
BR2_GRAPH_OUT=png make graph-build
```

8.10 绘制软件包对文件系统大小贡献图

当目标系统逐渐增大时，了解 Buildroot 每个软件包对整个根文件系统大小的贡献有时会很有用。为了帮助进行此类分析，Buildroot 收集有关每个软件包已安装文件的数据，并使用此数据生成图表和 CSV 文件，以详细说明不同软件包的大小贡献。

要在构建后生成这些数据，请运行：

```
make graph-size
```

这将生成：

- output/graphs/graph-size.pdf，每个软件包对整个根文件系统大小贡献的饼图；
- output/graphs/package-size-stats.csv，CSV 文件，每个软件包对整个根文件系统大小的贡献。
- output/graphs/file-size-stats.csv，CSV 文件，每个已安装文件对其所属软件包大小以及对整个文件系统大小的贡献。

此“graph-size”目标需要安装 Python Matplotlib 库（在大多数发行版中安装包为 python-matplotlib），如果您使用的 Python 版本早于 2.7，则还需要安装 argparse 模块（在大多数发行版中安装包为 python-argparse）。

就像持续时间图一样，支持使用 BR2_GRAPH_OUT 环境变量来调整输出文件格式。有关此环境变量的详细信息，请参阅第 8.8 节。

另外，可以设置环境变量 BR2_GRAPH_SIZE_OPTS 以进一步控制生成的图形。可接受的选项有：

- --size-limit X，-l X，会将单个贡献低于 X%的所有软件包分组到该图中标记为“Others”的单个条目。默认情况下，X=0.01，这意味着每个贡献少于 1%的软件包都归类为“Others”。

可接受的值在[0.0..1.0]范围内。

• --iec，--binary，--si，--decimal，使用 IEC（二进制，幂为 1024）或 SI（十进制，幂为 1000；默认值）前缀。

• --biggest-first，按降序而不是升序对软件包大小进行排序。

请注意，仅在完全清理之后重建，收集的文件系统大小数据才有意义。在使用“make graph-size”之前请务必运行“make clean all”。

要比较两个不同 Buildroot 编译的根文件系统大小，例如在调整配置后或切换到另一个 Buildroot 发布版本时，请使用 size-stats-compare 脚本。它需要两个 file-size-stats.csv 文件（由“make graph-size”生成）作为输入。有关更多详细信息，请参阅此脚本的帮助文本：

```
utils/size-stats-compare -h
```

8.11 顶级并行构建

注意，本节介绍一个具有实验性的功能，该功能在某些非常规情况下会被破坏。使用时风险自负。

Buildroot 一直能够在每个软件包基础上使用并行构建：每个软件包都是由 Buildroot 使用 make -jN 进行构建（或者是 non-make-based 构建系统的等效调用）。默认情况下，并行级别为 CPU 数量+1，但可以使用 BR2_JLEVEL 配置选项进行调整。

但是，直到 2020.02 为止，Buildroot 都是以串行方式构建软件包：每个软件包都是一个接一个构建的，而没有并行化软件包之间的构建。从 2020.02 开始，Buildroot 对顶级并行构建（top-level parallel build）提供了实验性的支持，通过并行化构建不具有依赖关系的软件包，可以节省大量构建时间。但是，该功能被标记为实验性功能，在某些情况下不起作用。

为了使用顶级并行构建，必须执行以下操作：

1. 在 Buildroot 配置中启用选项 BR2_PER_PACKAGE_DIRECTORIES；
2. 启动 Buildroot 构建时使用 make -jN；

在内部，BR2_PER_PACKAGE_DIRECTORIES 选项会启用一种称为“per-package directories”的机制，这将产生以下影响：

• 代替所有软件包全局通用的“target”目录和“host”目录，将使用每个软件包各自的“target”目录和“host”目录，分别位于$(O)/per-package/<pkg>/target/和$(O)/per-package/<pkg>/host/。在<pkg>构建开始时，将从软件包依赖项的相应文件夹中获取文件来填充这些文件夹。因此，编译器和所有其他工具将只能查看和访问由<pkg>明确列出的依赖项安装的文件。

• 在构建结束时，将填充全局“target”目录和“host”目录，分别位于$(O)/target 和$(O)/host。这意味着在构建期间，这些文件夹将为空，并且仅在构建的最后才填充它们。

8.12 与 Eclipse 集成

部分嵌入式 Linux 开发人员喜爱使用 Vim 或 Emacs 等经典文本编辑器和基于命令行的界面，而多数其他嵌入式 Linux 开发人员则喜爱使用更丰富的图形界面来完成他们的开发工作。Eclipse 是最受欢迎的集成开发环境之一，Buildroot 与 Eclipse 集成在一起可以简化 Eclipse 用户的开发工作。

与 Eclipse 的集成简化了（在 Buildroot 系统上构建的）应用程序和库的编译、远程执行和远程调试。它不集成 Buildroot 配置，也不使用 Eclipse 构建自身的进程。因此，我们 Eclipse 集成的典型使用模型是：

• 使用“make menuconfig”，“make xconfig”或 Buildroot 提供的任何其他配置界面来配置 Buildroot 系统。

- 通过运行“make”来构建 Buildroot 系统。
- 启动 Eclipse 以开发、执行和调试您自己定制的应用程序和库，它们将依赖于由 Buildroot 所构建和安装的库。

有关 Buildroot Eclipse 集成安装过程和用法的详细信息，请查阅 https://github.com/mbats/eclipse-buildroot-bundle/wiki。

8.13 高级用法

8.13.1 在 Buildroot 外部使用生成的工具链

您可能想要针对自己的目标，编译自己的程序或者其他未打包在 Buildroot 中的软件。为此，您可以使用 Buildroot 生成的工具链。

默认情况下，Buildroot 生成的工具链位于 output/host/中。最简单的使用方法是添加 output/host/bin 到 PATH 环境变量，然后使用 ARCH-linux-gcc，ARCH-linux-objdump，ARCH-linux-ld 等。

Buildroot 还可以通过运行“make sdk”命令，将工具链和所有选定的软件包的开发文件导出为 SDK。这将利用宿主机目录 output/host/中的内容来生成名为<TARGET-TUPLE>_sdk-buildroot.tar.gz 的压缩包（前缀可以通过设置环境变量 BR2_SDK_PREFIX 进行覆盖），并输出到目录 output/images 中。

然后，当他们要开发尚未打包为 Buildroot 软件包的应用程序时，可以将该工具链压缩包分发给应用程序开发人员。

提取 SDK 压缩文件后，用户必须运行脚本 relocate-sdk.sh（位于 SDK 的顶层目录中），以确保使用新位置更新所有路径。

另外，如果您只想准备 SDK 而又不生成压缩包（例如，您只是移动 host 目录，或者自己生成压缩包），则 Buildroot 允许使用“make prepare-sdk”命令准备 SDK 而不实际生成压缩包。

8.13.2 在 Buildroot 中使用 gdb

Buildroot 允许进行交叉调试，其中调试器在构建主机上运行，并与目标机器上的 gdbserver 进行通信，以控制程序的执行。

为了实现这个：

- 如果使用内部工具链（由 Buildroot 构建），则必须启用 BR2_PACKAGE_HOST_GDB、BR2_PACKAGE_GDB 和 BR2_PACKAGE_GDB_SERVER。这样可以确保交叉 gdb 和 gdbserver 都已构建，并且将 gdbserver 安装至目标系统中。
- 如果使用外部工具链，则应启用 BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY，它将把外部工具链附带的 gdbserver 复制到目标系统。如果您的外部工具链没有交叉 gdb 或 gdbserver 程序，也可以通过启用与内部工具链后端（internal toolchain backend）相同的选项，让 Buildroot 来构建它们。

现在，要开始调试一个名为 foo 的程序，您应该在目标机器上运行：

```
gdbserver:2345 foo
```

这将引起 gdbserver 在 TCP 端口 2345 上侦听来自交叉 gdb 的连接。然后，在宿主机上应该启动交叉 gdb，使用以下命令行：

```
<buildroot>/output/host/bin/<tuple>-gdb -x <buildroot>/output/staging/usr/share/buildroot/gdbinit foo
```

当然，foo 必须使用调试符号来进行构建，并且在当前目录中处于可用状态。通常，您是从

构建 foo 的目录中启动此命令（而不是从 output/target/中启动，因为该目录中的二进制文件已被移除）。

<buildroot>/output/staging/usr/share/buildroot/gdbinit 文件将告诉交叉 gdb 在哪里找到目标的库。

最后，通过交叉 gdb 连接到目标：

```
(gdb) target remote <target ip address>:2345
```

8.13.3 在 Buildroot 中使用 ccache

ccache 是编译器缓存。它存储每个编译过程所产生的目标文件，并能够通过使用预先存在的目标文件来跳过同一源文件（具有相同的编译器和相同的参数）的进一步编译。当需要多次从头开始进行几乎相同的构建时，这可以很好地加快构建过程。

Buildroot 已经集成 ccache 支持。您只需在“Build options”选项中启用“Enable compiler cache”即可。这将自动构建 ccache 并将其用于每个 host 和 target 的编译。

缓存位于$HOME/.buildroot-ccache 中。它存储在 Buildroot 的 output 目录之外，以便可以由单独的 Buildroot 构建进行共享。如果要移除缓存，只需删除该目录。

可以通过运行“make ccache-stats”来获取有关缓存的统计信息（其大小、命中数、未命中数等）。

make 目标“ccache-options”和 CCACHE_OPTIONS 变量提供对 ccache 更通用的访问。例如：

```
# set cache limit size
make CCACHE_OPTIONS="--max-size=5G" ccache-options
# zero statistics counters
make CCACHE_OPTIONS="--zero-stats" ccache-options
```

ccache 对源文件和编译器选项进行 hash 运算。如果编译器选项不同，则将不使用已缓存的目标文件。但是，许多编译器选项都包含 staging 目录的绝对路径。因此，在不同的输出目录中进行构建将导致许多缓存未命中。

为避免此问题，buildroot 具有使用相对路径的“Use relative paths”选项（BR2_CCACHE_USE_BASEDIR）。这会将所有指向输出目录内部的绝对路径重写为相对路径。因此，更改输出目录不会再导致缓存未命中。

相对路径的一个缺点是它们最终还会成为目标文件中的相对路径。因此，除非您先“cd”到输出目录，否则调试器将不再找到该文件。

有关此绝对路径重写的更多详细信息，请参阅 ccache 手册的“Compiling in different directories”章节。

8.13.4 下载软件包的位置

Buildroot 下载的各种软件压缩包都存储在 BR2_DL_DIR 中，默认情况下为 dl 目录。如果要保留一个完整的 Buildroot 版本，以便与相关的压缩包一起使用，则可以复制此目录。这将允许您使用完全相同的 Buildroot 版本重新生成工具链和目标文件系统。

如果维护多个 Buildroot 源码树，则最好具有共享的下载位置。这可以通过将 BR2_DL_DIR 环境变量指向某个目录来实现。如果设置了该环境变量，则 Buildroot 配置中 BR2_DL_DIR 的值将被覆盖。应将以下行添加到<~/.bashrc>中。

```
export BR2_DL_DIR=<shared download location>
```

也可以使用 BR2_DL_DIR 选项在.config 文件中设置下载位置。与.config 文件中大多数选

项不同，该值会被 BR2_DL_DIR 环境变量覆盖。

8.13.5 软件包特定 make 目标

运行 make <package>将构建并安装该特定软件包及其依赖项。

对于依赖 Buildroot 基础结构的软件包，有许多特殊的 make 目标，可以像下面这样独立调用：

```
make <package>-<target>
```

软件包构建目标 target 是（按照它们执行的顺序）：

命令/目标	描述
source	获取源码（例如下载压缩包、克隆源码仓库等）
depends	构建软件包时构建和安装所有相关的依赖项
extract	将源码提取到软件包构建目录（例如提取压缩包，复制源码等）
patch	应用提供的补丁
configure	运行提供的配置命令
build	运行构建命令
install-staging	target package: 如有必要，在 staging 目录中安装软件包
install-target	target package: 如有必要，在 target 目录中安装软件包
install	target package: 运行前面 2 条安装命令 host package: 在 host 目录中安装软件包

另外，还有一些其他有用的 make 目标 target：

命令/目标	描述
show-depends	显示构建软件包所需的一级依赖项
show-recursive-depends	递归显示构建软件包的依赖项
show-rdepends	显示软件包的一级反向依赖项（例如，软件包直接依赖它）
show-recursive-rdepends	递归显示软件包的反向依赖项（例如，软件包直接或间接依赖它）
graph-depends	在当前 Buildroot 上下文中生成软件包的依赖关系图。关于依赖关系图的更多详细信息，请参阅第 8.8 节。
graph-rdepends	生成该软件包的反向依赖关系图(例如,软件包直接或间接依赖它）
dirclean	删除整个软件包构建目录
reinstall	重新运行安装命令
rebuild	重新运行编译命令-仅在使用 OVERRIDE_SRCDIR 特性或在构建目录中直接修改文件时起作用
reconfigure	重新运行配置命令，然后进行 rebuild-仅在使用 OVERRIDE_SRCDIR 特性或在构建目录中直接修改文件时起作用

8.13.6 在开发期间使用 Buildroot

Buildroot 的正常操作是下载一个压缩包，对它进行解压缩，然后配置、编译和安装该压缩包中的软件组件。源代码会被提取到 output/build/<package>-<version>中，这是一个临时目录：当使用“make clean”时，该目录将被完全删除，并在下一次 make 调用时重新创建。即使将 Git 或 Subversion 存储仓库用作软件包源代码的输入，Buildroot 也会给它创建一个压缩包，然后像对待正常压缩包那样进行工作。

将 Buildroot 主要用作集成工具来构建和集成嵌入式 Linux 系统的所有组件，是非常合适

的。但是，如果是在系统某些组件的开发过程中使用 Buildroot，则以下行为不是很方便：希望对一个软件包的源代码进行少量更改，并且能够使用 Buildroot 快速重建系统。

直接在 output/build/<package>-<version>中进行修改是不合适的，因为在“make clean”时会删除该目录。因此，Buildroot 针对此场景提供了一种特定的机制：<pkg>_OVERRIDE_SRCDIR 机制。Buildroot 会读取 override 文件，而该文件可以让用户告诉 Buildroot 某些软件包的源码位置。

override 文件的默认位置是$(CONFIG_DIR)/local.mk，由 BR2_PACKAGE_OVERRIDE_FILE 配置选项定义。$(CONFIG_DIR)是 Buildroot 中.config 文件的位置，因此默认情况下，local.mk 与.config 文件在同一个目录下，这意味着：

- 在目录树内构建的 Buildroot 源码顶级目录中（即不使用“O=”时）
- 在目录树外构建的“O”参数指定的目录中（即使用“O=”时）

如果需要的位置与默认位置不同，则可以通过 BR2_PACKAGE_OVERRIDE_FILE 配置选项指定该位置。

在该 override 文件中，Buildroot 希望找到以下形式的参数行：

```
<pkg1>_OVERRIDE_SRCDIR = /path/to/pkg1/sources
<pkg2>_OVERRIDE_SRCDIR = /path/to/pkg2/sources
```

例如：

```
LINUX_OVERRIDE_SRCDIR = /home/bob/linux/
BUSYBOX_OVERRIDE_SRCDIR = /home/bob/busybox/
```

当 Buildroot 发现指定软件包已经定义了<pkg>_OVERRIDE_SRCDIR 时，它将不再尝试下载、提取和修补软件包。取而代之的是，它将直接使用指定目录中可用的源代码，并且使用“make clean”时不会删除该目录。这允许将 Buildroot 指向您自己的目录，该目录可以由 Git、Subversion 或任何其他版本控制系统进行管理。为此，Buildroot 将使用 rsync 命令将组件的源代码从指定的<pkg>_OVERRIDE_SRCDIR 复制到 output/build/<package>-custom/。

此机制最好与 make <pkg>-rebuild 和 make <pkg>-reconfigure 目标结合使用。make <pkg>-rebuild all 将使源代码从<pkg>_OVERRIDE_SRCDIR 同步到 output/build/<package>-custom（由于 rsync，它仅复制修改过的文件），然后重新启动该软件包的构建过程。

在上述 linux 软件包示例中，开发人员可以在/home/bob/linux 中更改源代码，然后运行：

```
make linux-rebuild all
```

可以在几秒钟内在 output/images 中获取更新过的 Linux 内核镜像。同样，可以在/home/bob/busybox 中对 BusyBox 源代码进行更改，然后运行：

```
make busybox-rebuild all
```

在 output/images 的根文件系统镜像中会包含更新过的 BusyBox。

大型项目的源码树中通常会包含成百上千个在构建过程中不需要的文件，这些文件会减慢使用 rsync 复制源码的过程。可以选择性地定义<pkg>_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS 以跳过同步源码树中的某些文件。例如，当使用 webkitgtk 软件包时，以下内容将从本地 WebKit 源码树中排除 tests 和 in-tree builds：

```
WEBKITGTK_OVERRIDE_SRCDIR = /home/bob/WebKit
WEBKITGTK_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS = \
--exclude JSTests --exclude ManualTests --exclude PerformanceTests \
--exclude WebDriverTests --exclude WebKitBuild --exclude WebKitLibraries \
--exclude WebKit.xcworkspace --exclude Websites --exclude Examples
```

默认情况下，Buildroot 会跳过 VCS 组件（例如.git 和.svn 目录）的同步。而一些软件包更

偏向于在构建过程中需要使用这些 VCS 目录，例如用于自动确定版本信息的准确提交引用。如果要以较慢的速度作为代价来撤消此内置过滤，则可以重新添加该目录：

```
LINUX_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS = --include .git
```

9 特定项目的定制

对于给定的项目，您可能需要执行的常用操作是：

- 配置 Buildroot（包括构建选项和工具链、引导程序、内核、软件包和文件系统镜像类型选择）；
- 配置其他组件，例如 Linux 内核和 BusyBox；
- 定制生成的目标文件系统：
  - 在目标文件系统上添加或覆盖文件（使用 BR2_ROOTFS_OVERLAY）；
  - 在目标文件系统上修改或删除文件（使用 BR2_ROOTFS_POST_BUILD_SCRIPT）；
  - 在生成文件系统镜像之前运行任意命令（使用 BR2_ROOTFS_POST_BUILD_SCRIPT）；
  - 设置文件的权限和所有权（使用 BR2_ROOTFS_DEVICE_TABLE）；
  - 添加自定义设备节点（使用 BR2_ROOTFS_STATIC_DEVICE_TABLE）；
- 添加自定义用户帐户（使用 BR2_ROOTFS_USERS_TABLES）；
- 在生成文件系统镜像后运行任意命令（使用 BR2_ROOTFS_POST_IMAGE_SCRIPT）；
- 向某些软件包添加特定项目的补丁（使用 BR2_GLOBAL_PATCH_DIR）；
- 添加特定项目的软件包。

有关此类特定项目的自定义项的重要说明：请仔细考虑哪些更改确实是特定项目的，哪些更改对项目外的开发人员也很有用。Buildroot 社区强烈建议并鼓励改进的上游（upstreaming），让更多的软件包和板级硬件支持 Buildroot 官方项目。当然，有时不可能或者不希望进行上游，因为一些更改是高度定制或专有的。*（译者补充：上游（upstreaming）是指提交和合并在正式的开源项目中所做的修改和改进以支持特定的硬件平台或设备，或进行其他类型的改进：错误修复，性能优化，功能添加等。）*

本章介绍了如何在 Buildroot 中进行此类特定项目的定制，以及如何存储它们，能够以一种可重现的方式来构建同一镜像，即使在运行“make clean”之后也是如此。通过以下推荐的策略，您甚至可以使用同一 Buildroot 源码来构建多个不同的项目！

9.1 推荐的目录结构

为项目定制 Buildroot 时，您将创建一个或多个需要存储在某处的特定于项目的文件。虽然这些文件大多数可以放置在任何位置，因为它们的路径可以在 Buildroot 配置中指定，但是 Buildroot 开发人员建议使用本节描述的特定目录结构。

您可以选择将该目录结构放置在何处：在 Buildroot 目录树中还是在其外部（使用 br2-external 目录树）。这两个选项均有效，取决于您的选择。

```
+-- board/
|   +-- <company>/
|       +-- <boardname>/
|           +-- linux.config
|           +-- busybox.config
|           +-- <other configuration files>
|           +-- post_build.sh
```

```
|               +-- post_image.sh
|               +-- rootfs_overlay/
|               |   +-- etc/
|               |   +-- <some file>
|               +-- patches/
|                   +-- foo/
|                   |   +-- <some patch>
|                   +-- libbar/
|                       +-- <some other patches>
|
+-- configs/
|   +-- <boardname>_defconfig
|
+-- package/
|   +-- <company>/
|       +-- Config.in (if not using a br2-external tree)
|       +-- <company>.mk (if not using a br2-external tree)
|       +-- package1/
|       |   +-- Config.in
|       |   +-- package1.mk
|       +-- package2/
|           +-- Config.in
|           +-- package2.mk
|
+-- Config.in (if using a br2-external tree)
+-- external.mk (if using a br2-external tree)
+-- external.desc (if using a br2-external tree)
```

本章会进一步提供上面显示文件的详细信息。

注意：如果您选择将此目录结构放置在 Buildroot 目录树之外但在 br2-external 目录树中，则<company>和<boardname>组件可能都是多余的，可以省略。

9.1.1 实施分层定制

对用户来说，对几个相关项目进行部分相同的定制是很普遍的。我们推荐使用分层定制的方式，而不是为每个项目进行重复定制，如本节所述。

在 Buildroot 中几乎所有可用的定制方法，例如 post-build 脚本和根文件系统 overlays，都可以接受以空格分隔的条目列表。Buildroot 始终按从左到右的顺序处理指定的条目。通过创建多个这样的条目，一个可用于通用的自定义项，另一个可用于真正的特定于项目的自定义项，可以避免不必要的重复。通常由 board/<company>/中的单独目录来呈现每个层次。根据项目需要，您甚至可以引入两个以上的层。

用户具有两个自定义层 common 和 fooboard 的目录结构示例如下：

```
+-- board/
    +-- <company>/
        +-- common/
```

```
|   +-- post_build.sh
|   +-- rootfs_overlay/
|   |   +-- ...
|   +-- patches/
|       +-- ...
|
+-- fooboard/
    +-- linux.config
    +-- busybox.config
    +-- <other configuration files>
    +-- post_build.sh
    +-- rootfs_overlay/
    |   +-- ...
    +-- patches/
        +-- ...
```

例如，如果用户将 BR2_GLOBAL_PATCH_DIR 选项设置为：

```
BR2_GLOBAL_PATCH_DIR="board/<company>/common/patches board/<company>/fooboard/patches"
```

则首先应用来自 common 层的补丁，然后应用来自 fooboard 层的补丁。

9.2 将定制保留在 Buildroot 之外

如 9.1 节所述，可以将特定项目的自定义项放置在两个位置：

- 直接在 Buildroot 目录树中，通常使用版本控制系统中的分支来维护它们，以便轻松升级到较新的 Buildroot 版本。
- 使用 br2-external 机制放在 Buildroot 目录树之外。这种机制允许将软件包配方、板级支持文件和配置文件保留在 Buildroot 目录树之外，同时仍能很好地集成到构建逻辑中。我们将此位置称为 br2-external 目录树。本节将说明如何使用 br2-external 机制以及在 br2-external 目录树中提供了哪些内容。

通过将 BR2_EXTERNAL 设置为要使用的 br2-external 目录树的路径，用户可以告诉 Buildroot 使用一个或多个 br2-external 目录树。可以将它传递给 Buildroot 的任何 make 调用。它会自动保存到输出目录中的.br2-external.mk 隐藏文件。因此，无需每次进行 make 调用时都传递 BR2_EXTERNAL。但是，可以随时传递新值来更改它，也可以通过传递空值来删除它。

请注意，br2-external 目录树路径可以是绝对路径，也可以是相对路径。如果将它作为相对路径传递，则必须注意，它是相对于 Buildroot 的源码主目录，而不是相对于 Buildroot 的输出目录。

注意，如果是使用 Buildroot 2016.11 之前的 br2-external 目录树，则需要对它进行转换，然后才能用于 Buildroot2016.11 及更高版本。请参阅第 26.1 节以获取帮助。

一些示例：

```
buildroot/ $ make BR2_EXTERNAL=/path/to/foo menuconfig
```

从现在开始，将使用/path/to/foo br2-external 目录树中的定义：

```
buildroot/ $ make
buildroot/ $ make legal-info
```

我们可以随时切换到另一个 br2-external 目录树：

```
buildroot/ $ make BR2_EXTERNAL=/where/we/have/bar xconfig
```

我们还可以使用多个 br2-external 目录树：

```
buildroot/ $ make BR2_EXTERNAL=/path/to/foo:/where/we/have/bar menuconfig
```

或者禁用任何 br2-external 目录树：*（译者注：此处将 BR2_EXTERNAL 设置为空）*

```
buildroot/ $ make BR2_EXTERNAL= xconfig
```

9.2.1 br2-external 目录树布局

单个 br2-extarnal 目录树必须至少包含以下三个文件，下面各小节将对此进行介绍：

- external.desc
- external.mk
- Config.in

除了这些强制性文件之外，br2-external 目录树中可能还存在其他可选内容，例如 configs/ 或 providers/目录。下面各小节也对它们进行介绍。

稍后还将介绍完整的 br2-external 目录树布局示例。

9.2.1.1 external.desc 文件

该文件描述了 br2-external 目录树：该 br2-external 目录树的名称和描述。

该文件的格式基于行，每行以一个关键字开头，后面跟一个冒号和一个或多个空格，然后是分配给该关键字的值。目前能识别出两个关键字：

- name，必填项，定义该 br2-external 目录树的名称。该名称只能使用[A-Za-z0-9_]集合中的 ASCII 字符，禁止使用任何其他字符。Buildroot 将 BR2_EXTERNAL_$(NAME)_PATH 变量设置为 br2-external 目录树的绝对路径，以便您可以用来引用 br2-external 目录树。该变量可用于 Kconfig 文件和 Makefile 文件，可用它来为 Kconfig 文件（请参阅下文）提供源，也可用它来包含其他 Makefile 文件（请参阅下文）或者引用 br2-external 目录树中的其他文件（如数据文件）。

请注意：由于可能会同时使用多个 br2-external 目录树，因此 Buildroot 使用该 name 为每个目录树生成变量。该 name 用于标识您的 br2-external 目录树，请尽量提供一个真正描述您的 br2-external 目录树的名称，以使其相对唯一，避免与其他 br2-external 目录树名称冲突，尤其是在您打算以某种方式与第三方共享 br2-external 目录树或使用第三方 br2-external 目录树时。

- desc，可选项，提供该 br2-external 目录树的简短描述。它应该放在一行上，主要是自由格式的（见下文），用于显示 br2-external 目录树的有关信息（例如，在 defconfig 文件列表上方，或者作为 menuconfig 中的提示）。因此，它应该相对简短（40 个字符可能是一个很好的上限）。该描述可在 BR2_EXTERNAL_$(NAME)_DESC 变量中获取。

name 和对应的 BR2_EXTERNAL_$(NAME)_PATH 变量的示例：

- FOO -> BR2_EXTERNAL_FOO_PATH
- BAR_42 -> BR2_EXTERNAL_BAR_42_PATH

在后面示例中，假定将 name 设置为 BAR_42。

请注意：BR2_EXTERNAL_$(NAME)_PATH 和 BR2_EXTERNAL_$(NAME)_DESC 变量都可用于 Kconfig 文件和 Makefile 文件中。它们也会被导出到当前环境中，因此可以在 post-build、post-image 和 in-fakeroot 脚本中使用。

9.2.1.2 Config.in 和 external.mk 文件

这些文件（可能每个都为空）可用来定义软件包配方（例如 foo/Config.in 和 foo/foo.mk，如

Buildroot 本身附带的软件包）或其他自定义配置选项或 make 逻辑应用。

Buildroot 会自动包含每个 br2-external 目录树中的 Config.in，以使其出现在 Buildroot 顶级配置菜单中，并包含每个 br2-external 目录树中的 external.mk 以及其余的 makefile 逻辑。

它的主要用途是保存软件包配方。推荐的实现方式是编写一个 Config.in 文件，如下所示：

```
source "$BR2_EXTERNAL_BAR_42_PATH/package/package1/Config.in"
source "$BR2_EXTERNAL_BAR_42_PATH/package/package2/Config.in"
```

然后，创建一个 external.mk 文件，如下所示：

```
include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/package/*/*.mk))
```

然后在$(BR2_EXTERNAL_BAR_42_PATH)/package/package1 和$(BR2_EXTERNAL_BAR_42_PATH)/package/package2 中创建常规的 Buildroot 软件包配方，如第 17 章中所述。如果您愿意，还可以将软件包分组在名为<boardname>的子目录中，并相应地调整上述路径。

您还可以在 Config.in 中自定义配置选项，并在 external.mk 中自定义 make 逻辑。

9.2.1.3 configs/目录

可以将 Buildroot 的 defconfigs 文件保存在 br2-external 目录树的 configs 子目录中。Buildroot 将在“make list-defconfigs”输出中自动显示它们，并允许使用常规的 make <name>_defconfig 命令加载它们。它们将显示在“make list-defconfigs”输出中的“External configs”标签下，该标签包含了 br2-external 目录树名称。*（译者注：实际内容为 external.desc 文件中“desc”关键字描述）*

请注意：如果一个 defconfig 文件存在于多个 br2-external 目录树中，则使用最后一个 br2-external 目录树中的 defconfig 文件。因此，可能会覆盖 Buildroot 附带的或者另一个 br2-external 目录树中的 defconfig 文件。

9.2.1.4 provides/目录

对于某些软件包，Buildroot 提供了两个（或更多）API 兼容实现的软件包选择。例如，可以选择 libjpeg 或 jpeg-turbo，也可以选择 openssl 或 libressl。最后，选择一个已知的、预配置的工具链。

br2-external 可以通过提供定义替代项的一组文件来扩展这些选择：

- provides/toolchains.in 定义预配置的工具链，它将被列入选择工具链的菜单中；
- provides/jpeg.in 定义替代的 libjpeg 实现；
- provides/openssl.in 定义替代的 openssl 实现。

9.2.1.5 自由格式内容

可以在其中保存特定主板的所有配置文件，例如内核配置、根文件系统 overlay 或者任何其他位置的配置文件（通过 BR2_EXTERNAL_$(NAME)_PATH 设置位置）。例如，您可以按以下方式将路径设置为全局补丁目录、文件系统 overlay 和内核配置文件（通过运行“make menuconfig”并填写以下选项）：

```
BR2_GLOBAL_PATCH_DIR=$(BR2_EXTERNAL_BAR_42_PATH)/patches/
BR2_ROOTFS_OVERLAY=$(BR2_EXTERNAL_BAR_42_PATH)/board/<boardname>/overlay/
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE=$(BR2_EXTERNAL_BAR_42_PATH)/board/
<boardname>/kernel.config
```

9.2.1.6 添加 Linux 内核扩展

可以将其他 Linux 内核扩展保存到 br2-external 目录树根目录下的 linux/目录中，以此添加 Linux 内核扩展（请参阅第 17.20.2 节）。

9.2.1.7 示例布局

这是一个使用了 br2-external 所有功能的示例布局（针对上面所述文件的示例，仅用于解释 br2-external 目录树；下面所有这些都是出于说明目的而组成）：

```
/path/to/br2-ext-tree/
  |- external.desc
  |       |name: BAR_42
  |       |desc: Example br2-external tree
  |       `----
  |
  |- Config.in
  |       |source "$BR2_EXTERNAL_BAR_42_PATH/toolchain/toolchain-external-mine/Config.in.
options"
  |       |source "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-1/Config.in"
  |       |source "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-2/Config.in"
  |       |source "$BR2_EXTERNAL_BAR_42_PATH/package/my-jpeg/Config.in"
  |       |
  |       |config BAR_42_FLASH_ADDR
  |       |     hex "my-board flash address"
  |       |     default 0x10AD
  |       `----
  |
  |- external.mk
  |       |include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/package/*/*.mk))
  |       |include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/toolchain/*/*.mk))
  |       |
  |       |flash-my-board:
  |       |     $(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/flash-image \
  |       |           --image $(BINARIES_DIR)/image.bin \
  |       |           --address $(BAR_42_FLASH_ADDR)
  |       `----
  |
  |- package/pkg-1/Config.in
  |       |config BR2_PACKAGE_PKG_1
  |       |     bool "pkg-1"
  |       |     help
  |       |       Some help about pkg-1
  |       `----
  |- package/pkg-1/pkg-1.hash
  |- package/pkg-1/pkg-1.mk
  |       |PKG_1_VERSION = 1.2.3
```

```
|       |PKG_1_SITE = /some/where/to/get/pkg-1
|       |PKG_1_LICENSE = blabla
|       |
|       |define PKG_1_INSTALL_INIT_SYSV
|       |     $(INSTALL) -D -m 0755 $(PKG_1_PKGDIR)/S99my-daemon \
|       |                                   $(TARGET_DIR)/etc/init.d/S99my-daemon
|       |endef
|       |
|       |$(eval $(autotools-package))
|       `----
|- package/pkg-1/S99my-daemon
|
|- package/pkg-2/Config.in
|- package/pkg-2/pkg-2.hash
|- package/pkg-2/pkg-2.mk
|
|- provides/jpeg.in
|       |config BR2_PACKAGE_MY_JPEG
|       |     bool "my-jpeg"
|       `----
|- package/my-jpeg/Config.in
|       |config BR2_PACKAGE_PROVIDES_JPEG
|       |     default "my-jpeg" if BR2_PACKAGE_MY_JPEG
|       `----
|- package/my-jpeg/my-jpeg.mk
|       |# This is a normal package .mk file
|       |MY_JPEG_VERSION = 1.2.3
|       |MY_JPEG_SITE = https://example.net/some/place
|       |MY_JPEG_PROVIDES = jpeg
|       |$(eval $(autotools-package))
|       `----
|
|- provides/toolchains.in
|       |config BR2_TOOLCHAIN_EXTERNAL_MINE
|       |     bool "my custom toolchain"
|       |     depends on BR2_some_arch
|       |     select BR2_INSTALL_LIBSTDCPP
|       `----
|- toolchain/toolchain-external-mine/Config.in.options
|       |if BR2_TOOLCHAIN_EXTERNAL_MINE
|       |config BR2_TOOLCHAIN_EXTERNAL_PREFIX
|       |     default "arch-mine-linux-gnu"
|       |config BR2_PACKAGE_PROVIDES_TOOLCHAIN_EXTERNAL
```

```
|       |     default "toolchain-external-mine"
|       |endif
|       `----
|- toolchain/toolchain-external-mine/toolchain-external-mine.mk
|       |TOOLCHAIN_EXTERNAL_MINE_SITE = https://example.net/some/place
|       |TOOLCHAIN_EXTERNAL_MINE_SOURCE = my-toolchain.tar.gz
|       |$(eval $(toolchain-external-package))
|       `----
|
|- linux/Config.ext.in
|       |config BR2_LINUX_KERNEL_EXT_EXAMPLE_DRIVER
|       |     bool "example-external-driver"
|       |     help
|       |         Example external driver
|       |---
|- linux/linux-ext-example-driver.mk
|
|- configs/my-board_defconfig
|       |BR2_GLOBAL_PATCH_DIR="$(BR2_EXTERNAL_BAR_42_PATH)/patches/"
|       |BR2_ROOTFS_OVERLAY="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/overlay/"
|       |BR2_ROOTFS_POST_IMAGE_SCRIPT="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/post-image.sh"
|       |BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/kernel.config"
|       `----
|
|- patches/linux/0001-some-change.patch
|- patches/linux/0002-some-other-change.patch
|- patches/busybox/0001-fix-something.patch
|
|- board/my-board/kernel.config
|- board/my-board/overlay/var/www/index.html
|- board/my-board/overlay/var/www/my.css
|- board/my-board/flash-image
`- board/my-board/post-image.sh
        |#!/bin/sh
        |generate-my-binary-image \
        | --root ${BINARIES_DIR}/rootfs.tar \
        | --kernel ${BINARIES_DIR}/zImage \
        | --dtb ${BINARIES_DIR}/my-board.dtb \
        | --output ${BINARIES_DIR}/image.bin
        `----
```

br2-external 目录树将在 menuconfig 中可见（布局已展开）：

```
External options --->
    *** Example br2-external tree (in /path/to/br2-ext-tree/)
    [ ] pkg-1
    [ ] pkg-2
    (0x10AD) my-board flash address
```

如果使用多个 br2-external 目录树，如下所示（布局已展开，且第二个目录树 name 为 FOO_27，在 external.desc 中没有“desc”关键字）

```
External options ---->
    Example br2-external tree --->
        *** Example br2-external tree (in /path/to/br2-ext-tree)
        [ ] pkg-1
        [ ] pkg-2
        (0x10AD) my-board flash address
    FOO_27 --->
        *** FOO_27 (in /path/to/another-br2-ext)
        [ ] foo
        [ ] bar
```

此外，jpeg provider 将在 jpeg 选项中可见：

```
Target packages --->
    Libraries --->
        Graphics --->
            [*] jpeg support
                jpeg variant () --->
                    ( ) jpeg
                    ( ) jpeg-turbo
                        *** jpeg from: Example br2-external tree ***
                    (X) my-jpeg
                        *** jpeg from: FOO_27 ***
                    ( ) another-jpeg
```

类似地，工具链：

```
Toolchain --->
    Toolchain () --->
        ( ) Custom toolchain
            *** Toolchains from: Example br2-external tree ***
        (X) my custom toolchain
```

（译者注：工具链选项展开效果需修改上面布局中 provides/toolchains.in 文件第 3 行“BR2_some_arch”为“BR2_arm”，或者 BR2_其他芯片架构，这与用户所选芯片架构配置有关。）

请注意，toolchain/toolchain-external-mine/Config.in.options 中的 toolchain 选项不会出现在“Toolchain”菜单中。必须在 br2-external 目录树的顶层 Config.in 中明确包含这些内容，才能在“External options”菜单中显示它们。

9.3 存储 Buildroot 配置

可以使用命令“make savedefconfig”来存储 Buildroot 配置。

通过删除默认配置选项，可以简化 Buildroot 配置，将结果存储在名为 defconfig 的文件中。如果要将它保存在其他位置，则需更改 Buildroot 本身配置中的 BR2_DEFCONFIG 选项，或者调用“make savedefconfig BR2_DEFCONFIG=<path-to-defconfig>”。

建议该 defconfig 文件的存储位置是 configs/<boardname>_defconfig。如果遵循此建议，则该配置将列在“make help”中，并且可以通过运行“make <boardname>_defconfig”进行重新设置。

或者，您可以将文件复制到任何其他位置，并使用“make defconfig BR2_DEFCONFIG=<path-to-defconfig-file>”进行重新构建。

9.4 存储其他组件配置

如果更改了 BusyBox、Linux 内核、Barebox、U-Boot 和 uClibc 的配置文件，则需将它们存储起来。对于这些组件中的每一个，都存在一个 Buildroot 配置选项来指向一个配置输入文件，例如 BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE。要存储它们的配置，请将这些配置选项设置为配置文件的存储路径，然后使用下面描述的帮助程序来完成实际存储。

如 9.1 节“推荐的目录结构”所述，建议存储这些配置文件的路径是 board/<company>/<boardname>/foo.config。

在更改 BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE 等选项之前，请确保已创建对应的配置文件。否则，Buildroot 将尝试访问该配置文件，若配置文件尚未存在，则会访问失败。您可以通过运行“make linux-menuconfig”等来创建配置文件。

Buildroot 提供了一些帮助程序，以简化配置文件的保存。

- make linux-update-defconfig 将 linux 配置保存到 BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE 指定的路径。它通过删除默认值来简化配置文件。但是，这仅适用于从 2.6.33 开始的内核。对于较早的内核，请使用“make linux-update-config”。
- make busybox-update-config 将 busybox 配置保存到 BR2_PACKAGE_BUSYBOX_CONFIG 指定的路径。
- make uclibc-update-config 将 uClibc 配置保存到 BR2_UCLIBC_CONFIG 指定的路径。
- make rawbox-update-defconfig 将 barebox 配置保存到 BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE 指定的路径。
- make uboot-update-defconfig 将 U-Boot 配置保存到 BR2_TARGET_UBOOT_CUSTOM_CONFIG_FILE 指定的路径。
- 对于 at91bootstrap3，不存在任何帮助程序，因此您需要手动将配置文件复制到 BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE 指定的路径。

9.5 自定义生成的目标文件系统

除了通过 make *config 更改配置，还有其他几种方法可以自定义生成的目标文件系统。

下面推荐两种方法，它们可以共存，就是根文件系统 overlay(s)和 post-build 脚本。

- **根文件系统 overlay（BR2_ROOTFS_OVERLAY）**

文件系统 overlay 是一个文件目录树，在构建目标文件系统后，可以直接将它复制到目标文件系统上。要启用此功能，请将配置选项 BR2_ROOTFS_OVERLAY（在“System configuration”菜单中）设置为 overlay 的根目录。您甚至可以指定多个以空格分隔的 overlay。如果指定相对路径，则它是相对于 Buildroot 的根目录。版本控制系统的隐藏目录（如.git，.svn，.hg 等），名为.empty 的文件和以“~”结尾的文件都将被排除。

当启用 BR2_ROOTFS_MERGED_USR 时，overlay 不得包含/bin、/lib 或/sbin 目录，因为 Buildroot 会将它们创建为指向/usr 中相关文件夹的符号链接。在这种情况下，如果 overlay 具有任何程序或库，则应将它们放在/usr/bin，/usr/sbin 和/usr/lib 中。

如 9.1 节所示，推荐 overlay 的存储路径是 board/<company>/<boardname>/rootfs-overlay。

- **post-build 脚本（BR2_ROOTFS_POST_BUILD_SCRIPT）**

post-build 脚本是在 Buildroot 构建所有选定软件之后但在生成 rootfs 镜像之前调用的 shell 脚本。要启用此功能，请在配置选项 BR2_ROOTFS_POST_BUILD_SCRIPT（在“System configure”菜单中）中指定以空格分隔的 post-build 脚本列表。如果指定相对路径，则它是相对于 Buildroot 的根目录。

通过使用 post-build 脚本，您可以删除或修改目标文件系统中的任何文件。但是，您应谨慎使用此功能。每当您发现某个软件包生成了错误的或者不需要的文件时，您都应该修复该软件包，而不是使用一些 post-build 的清理脚本来解决它。

如 9.1 节所示，推荐该脚本的存储路径是 board/<company>/<boardname>/post_build.sh。

post-build 脚本以 Buildroot 主目录作为当前工作目录运行。目标文件系统的路径作为第一个参数被传递给每个脚本。如果配置选项 BR2_ROOTFS_POST_SCRIPT_ARGS 不为空，则这些参数也将被传递给脚本。所有脚本都将被传递完全相同的参数集，而不可能将不同的参数集传递给每个脚本。

此外，还可以使用以下环境变量：

- BR2_CONFIG：Buildroot .config 文件的路径；
- HOST_DIR、STAGING_DIR、TARGET_DIR：请参阅第 17.5.2 节；
- BUILD_DIR：提取和构建软件包的目录；
- BINARIES_DIR：所有二进制文件（也称为镜像）的存储位置；
- BASE_DIR：基本输出目录。

下面还介绍了三种自定义目标文件系统的方法，但不建议使用。

- **直接修改目标文件系统**

对于临时修改，您可以直接修改目标文件系统并重建镜像。目标文件系统可在 output/target/目录中获取。进行更改后，通过运行“make”来重建目标文件系统镜像。

此方法允许您对目标文件系统执行任何操作，但是如果您需要使用“make clean”清理 Buildroot，则这些更改将丢失。在某些情况下这种清理是必要的，有关详细信息请参阅第 8.2 节。因此，该方法仅对快速测试有用：此类更改无法在“make clean”命令中得以保留。验证更改后，您应该通过使用根文件系统 overlay 或 post-build 脚本来确保在运行“make clean”后能得以保留。

- **自定义目标框架（BR2_ROOTFS_SKELETON_CUSTOM）**

根文件系统镜像是基于目标框架（target skeleton）创建的，所有软件包都在它上面安装各自的文件。在构建和安装任何软件包之前，会将框架（skeleton）复制到目标目录 output/target 中。默认的目标框架提供了标准的 Unix 文件系统布局以及一些基本的初始化脚本和配置文件。

如果默认框架（在 system/skeleton 下获取）不符合您的需求，则通常使用根文件系统 overlay 或 post-build 脚本来适配它。但是，如果默认框架与您所需的框架完全不同，则使用自定义的框架可能会更合适。

要启用此功能，请启用配置选项 BR2_ROOTFS_SKELETON_CUSTOM，并将 BR2_ROOTFS_SKELETON_CUSTOM_PATH 设置为自定义框架的路径。这两个选项均可在“System configuration”菜单中获取。如果指定相对路径，则它是相对于 Buildroot 的根目录。

自定义框架不需要包含/bin、/lib 或/sbin 目录，因为它们是在构建过程中自动创建的。当启

用 BR2_ROOTFS_MERGED_USR 后，自定义框架不得包含/bin、/lib 或/sbin 目录，因为 Buildroot 会将它们创建为指向/usr 中相关文件夹的符号链接。在这种情况下，如果框架具有任何程序或库，则应将它们放置在/usr/bin、/usr/sbin 和/usr/lib 中。

不建议使用此方法，因为它会复制整个框架，从而阻止 Buildroot 更高版本对默认框架的修复或改进。

- **post-fakeroot 脚本（BR2_ROOTFS_POST_FAKEROOT_SCRIPT）**

生成最终镜像时，该过程的某些部分会需要 root 权限：在/dev 中创建设备节点，设置文件和目录的权限或所有权等等。为了避免需要实际的 root 权限，Buildroot 使用 fakeroot 模拟 root 权限。这不能完全代替实际的 root 用户，但足以满足 Buildroot 的需求。

post-fakeroot 脚本是在 fakeroot 阶段的最后但在调用文件系统镜像生成器之前调用的 shell 脚本。因此，它们是在 fakeroot 上下文中被调用。

如果您需要调整文件系统以执行通常仅由 root 用户才能执行的修改，则 post-fakeroot 脚本会很有用。

注意：建议使用现有机制来设置文件权限或者在/dev 中创建节点（请参阅第 9.5.1 节）或者创建用户（请参阅第 9.6 节）。

注意：post-build 脚本（上述）和 fakeroot 脚本的区别在于，在 fakeroot 上下文中不调用 post-build 脚本。

注意：使用 fakeroot 并不能完全代替实际的 root 用户。fakeroot 只伪造文件访问权限和类型（常规、块或字符设备...）和 uid/gid；这些是在内存中模拟的。

9.5.1 设置文件权限和所有权并添加自定义设备节点

有时需要在文件或者设备节点上设置特定的权限或所有权。例如，某些文件可能需要由 root 拥有。由于 post-build 脚本不是作为 root 用户运行的，因此无法从那里进行此类更改，除非您在 post-build 脚本中使用显式 fakeroot。

相反，Buildroot 通过权限表（permission tables）提供支持。要使用此功能，请将配置选项 BR2_ROOTFS_DEVICE_TABLE 设置为以空格分隔的权限表列表，它是一种遵循 makedev 语法的常规文本文件。

如果您使用的是静态设备表（即未使用 devtmpfs、mdev 或(e)udev），则可以使用相同的语法在设备表（device tables）中添加设备节点。要使用此功能，请将配置选项 BR2_ROOTFS_STATIC_DEVICE_TABLE 设置为以空格分隔的设备表列表。

如 9.1 节所示，推荐此类文件的存储位置是 board/<company>/<boardname>/。

请注意，如果特定权限或设备节点与特定应用程序相关，则应在软件包的.mk 文件中设置变量 FOO_PERMISSIONS 和 FOO_DEVICES（请参阅第 17.5.2 节）。

9.6 添加自定义用户帐户

有时需要在目标系统中添加特定用户。为了满足此要求，Buildroot 通过用户表（users tables）提供支持。要使用此功能，请将配置选项 BR2_ROOTFS_USERS_TABLES 设置为以空格分隔的用户表列表，它是一种遵循 makeusers 语法的常规文本文件。

如 9.1 节所示，推荐此类文件的存储位置是 board/<company>/<boardname>/。

请注意，如果自定义用户与特定应用程序相关，则应在软件包的.mk 文件中设置变量 FOO_USERS（请参阅第 17.5.2 节）。

9.7 创建镜像后进行自定义

post-build 脚本（第 9.5 节）会在构建文件系统镜像、内核和引导加载程序之前运行，而 post-image 脚本可在创建所有镜像之后用于执行某些特定操作。

例如，post-image 脚本可用于自动提取根文件系统压缩包到 NFS 服务器导出的位置，或创建一个绑定于根文件系统和内核镜像的特定固件镜像，或项目所需的任何其他自定义操作。

要启用此功能，请将配置选项 BR2_ROOTFS_POST_IMAGE_SCRIPT（在“System configuration”菜单中）设置为以空格分隔的 post-image 脚本列表。如果指定相对路径，则它是相对于 Buildroot 的根目录。

像 post-build 脚本一样，post-image 脚本会以 Buildroot 主目录作为当前工作目录运行。镜像输出目录的路径作为第一个参数被传递给每个脚本。如果配置选项 BR2_ROOTFS_POST_SCRIPT_ARGS 不为空，则这些参数也将被传递给脚本。所有脚本都将被传递完全相同的参数集，而不可能将不同的参数集传递给每个脚本。

同样，像 post-build 脚本一样，post-image 脚本可以访问环境变量 BR2_CONFIG、HOST_DIR、STAGING_DIR、TARGET_DIR、BUILD_DIR、BINARIES_DIR 和 BASE_DIR。

post-image 脚本将以执行 Buildroot 的用户身份来执行，该用户通常不应该是 root 用户。因此，在这些脚本中需要 root 权限的任何操作都需要作特殊处理（使用 fakeroot 或 sudo），该处理留给脚本开发人员实现。

9.8 添加特定项目的补丁

有时在 Buildroot 提供的补丁基础上，将额外的补丁程序应用于软件包也会很有用。例如，这可能用于支持项目中的自定义功能，或者使用新体系架构。

BR2_GLOBAL_PATCH_DIR 配置选项可用于指定以空格分隔的一个或多个包含软件包补丁程序的目录列表。

对于特定软件包<packagename>的特定版本<packageversion>，将从 BR2_GLOBAL_PATCH_DIR 中应用补丁，如下所示：

1. 对于存在于 BR2_GLOBAL_PATCH_DIR 中的每个<global-patch-dir>目录，<package-patch-dir>取决于：
   - <global-patch-dir>/<packagename>/<packageversion>/（如果目录存在）；
   - 否则，使用<global-patch-dir>/<packagename>（如果目录存在）。
2. 然后将从<package-patch-dir>中应用补丁，如下所示：
   - 如果软件包目录中存在 series 文件，则根据 series 文件应用补丁；
   - 否则，将按照字母顺序应用匹配*.patch 的补丁文件。因此，为确保按正确的顺序应用它们，强烈建议将补丁文件命名为：<number>-<description>.patch，其中<number>是指应用顺序。

有关如何将补丁程序应用于软件包的信息，请参阅第 18.2 节。

BR2_GLOBAL_PATCH_DIR 选项是指定自定义补丁目录的首选方法。它可用于指定 buildroot 中任意软件包的补丁程序目录，也可用于代替 U-Boot 和 Barebox 等软件包的自定义补丁目录选项。这样，它将允许用户从一个顶层目录来管理补丁。

相对于 BR2_GLOBAL_PATCH_DIR 作为指定自定义补丁目录的首选方法，它的例外是 BR2_LINUX_KERNEL_PATH。选项 BR2_LINUX_KERNEL_PATCH 用于指定 URL 上可获取的内核补丁。请注意：BR2_LINUX_KERNEL_PATH 所指定的内核补丁是在 BR2_GLOBAL_PATCH_DIR 中的补丁应用之后才进行应用，因为它是在 Linux 软件包的 post-patch 脚本钩子函数中完成的。

9.9 添加特定项目的软件包

通常，任何新的软件包都应该直接添加到 package 目录中，然后提交给 Buildroot 上游项目。关于如何在 Buildroot 中添加软件包将在第 17 章中详细介绍，在此不再赘述。但是，您的项目可能会需要一些无法上游的专有软件包。本节将说明如何将特定项目的软件包保存在特定项目的目录中。

如 9.1 节所示，特定项目软件包的推荐存储位置是 package/<company>/。如果您正在使用 br2-external 目录树功能（请参阅第 9.2 节），则推荐的位置是将它放在 br2-external 目录树中一个名为 package/的子目录下。

但是，Buildroot 将无法识别此位置的软件包，除非我们执行一些其他步骤。如第 17 章所述，Buildroot 中的软件包基本上由两个文件组成：.mk 文件（描述如何构建软件包）和 Config.in 文件（描述该软件包的配置选项）。

Buildroot 会自动在 package 目录的第一级子目录中包含.mk 文件（使用模式 package/*/*.mk）。如果我们需要 Buildroot 包含更深层次的子目录.mk 文件（如 package/<company>/package1），则只需在第一级子目录中添加一个.mk 文件以包含这些.mk 文件即可。因此，创建一个 package/<company>/<company>.mk 文件，包含以下内容（假设 package/<company>/下仅有一个额外的目录级别）：

```
include $(sort $(wildcard package/<company>/*/*.mk))
```

对于 Config.in 文件，创建一个 package/<company>/Config.in 文件，在其中包含所有软件包的 Config.in 文件。由于 kconfig 的 source 命令不支持通配符，因此必须提供详尽的列表。例如：

```
source "package/<company>/package1/Config.in"
source "package/<company>/package2/Config.in"
```

在 package/Config.in 中包含该新文件 package/<company>/Config.in，且最好包含在特定 company 的菜单中，以更易于合并到以后的 Buildroot 版本中。

如果使用 br2-external 目录树，请参阅第 9.2 节以了解如何填写这些文件。

9.10 特定项目自定义配置快速指南

在本章前面，已经描述了特定项目自定义配置的不同方法。现在，本节将通过一步步说明来总结特定项目的所有自定义配置。当然，与您的项目无关的步骤可以跳过。

1. “make menuconfig”，配置工具链、软件包和内核；
2. “make linux-menuconfig”，更新内核配置，类似于其他配置，例如 busybox、uclibc 等；
3. “mkdir -p board/<manufacturer>/<boardname>”；
4. 将以下选项设置为 board/<manufacturer>/<boardname>/<package>.config（只要相关）：

- BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE
- BR2_PACKAGE_BUSYBOX_CONFIG
- BR2_UCLIBC_CONFIG
- BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE
- BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE
- BR2_TARGET_UBOOT_CUSTOM_CONFIG_FILE

5. 编写配置文件：

- “make linux-update-defconfig”
- “make busybox-update-config”
- “make uclibc-update-config”
- “cp <output>/build/at91bootstrap3-*/.config board/<manufacturer>/<boardname>/at91bootstrap3.config”

- “make badbox-update-defconfig”
- “make uboot-update-defconfig”

6. 创建 board/<manufacturer>/<boardname>/rootfs-overlay/，并在其中添加您 rootfs 所需的文件，例如 board/<manufacturer>/<boardname>/rootfs-overlay/etc/inittab。将 BR2_ROOTFS_OVERLAY 设置为 board/<manufacturer>/<boardname>/rootfs-overlay；

7. 创建一个 post-build 脚本 board/<manufacturer>/<boardname>/post_build.sh。将 BR2_ROOTFS_POST_BUILD_SCRIPT 设置为 board/<manufacturer>/<boardname>/post_build.sh；

8. 如果需要设置其他 setuid 权限或者创建设备节点，请创建 board/<manufacturer>/<boardname>/device_table.txt，并将其路径添加到 BR2_ROOTFS_DEVICE_TABLE；

9. 如果需要创建其他用户帐户，请创建 board/<manufacturer>/<boardname>/users_table.txt，并将其路径添加到 BR2_ROOTFS_USERS_TABLES；

10. 如果需要添加某些软件包的自定义补丁，请将 BR2_GLOBAL_PATCH_DIR 设置为 board/<manufacturer>/<boardname>/patches/，并将每个软件包的补丁添加到以该软件包命名的子目录中。每个补丁应命名为 <packagename>-<num>-<description>.patch。

11. 针对 Linux 内核，还有一个选项 BR2_LINUX_KERNEL_PATCH，它主要优点是可以从 URL 中下载补丁。如果您不需要它，则首选 BR2_GLOAL_PATCH_DIR。U-Boot、Barebox、at91bootstrap 和 at91bootstrap3 也有单独的选项，但与 BR2_GLOBAL_PATCH_DIR 相比，它们没有提供任何优势，并且将来可能会被删除。

12. 如果需要添加特定项目的软件包，请创建 package/<manufacturer>/，并将软件包放至该目录中。创建一个整体的<manufacturer>.mk 文件，在其中包含所有软件包的.mk 文件。创建一个整体的 Config.in 文件，来“source”所有软件包的 Config.in 文件。然后在 Buildroot 的 package/Config.in 文件中包含该 Config.in 文件。

13. “make savedefconfig”，保存 buildroot 配置；

14. “cp defconfig configs/<boardname>_defconfig”。

10 常见问题和故障排除

10.1 启动网络后，引导程序挂起

如果在输出以下信息后，引导程序似乎挂起（以下信息不一定完全相似，具体取决于所选软件包的列表）：

```
Freeing init memory: 3972K
Initializing random number generator... done.
Starting network...
Starting dropbear sshd: generating rsa key... generating dsa key... OK
```

这表示您的系统正在运行，但没有在串行控制台上启动 shell 程序。为了使系统在串行控制台上启动 shell，您必须进入 Buildroot 配置菜单，在“System configuration”中修改“Run a getty(login prompt) after boot”，并在“getty options”子菜单中设置适当的端口和波特率。这将自动调整目标系统的/etc/inittab 文件，以便能在正确的串行端口上启动 shell 程序。

10.2 为什么目标上没有编译器

我们已经决定从 Buildroot-2012.11 发行版本开始，停止对目标上本地编译器（“native compiler on the target”）的支持，因为：

- 此功能既未维护也未测试，并且经常损坏；
- 此功能仅适用于 Buildroot 工具链；
- Buildroot 主要是针对板载资源有限（CPU、内存、大容量存储）的小型或者超小型目标硬件，在此类目标上进行编译没有太大意义；
- Buildroot 旨在简化交叉编译，没有必要在目标上进行本地编译。

如果仍然需要在目标上使用编译器，则 Buildroot 不适合您的用途。在这种情况下，您需要一个真正的发行版，并且应该选择以下内容：

- openembedded
- yocto
- emdebian
- Fedora
- openSUSE ARM
- Arch Linux ARM
- …

10.3 为什么目标上没有开发文件

由于目标上没有可用的编译器（请参阅第 10.2 节），因此耗费空间去存储头文件或静态库是没有意义的。

因此，自 Buildroot-2012.11 发行版本以来，始终会从目标中移除这些文件。

10.4 为什么目标上没有文档

由于 Buildroot 主要是针对板载资源有限（CPU、内存、大容量存储）的小型或超小型目标硬件，因此耗费空间去存储文档数据（documentation data）是没有意义的。

如果仍然需要目标上的文档数据，则 Buildroot 不适合您的用途，您应该寻找一个真正的发行版（请参阅第 10.2 节）。

10.5 为什么在 Buildroot 配置菜单中有些软件包不可见

如果某个软件包存在于 Buildroot 目录树中，却没有出现在配置菜单上，则最有可能意味着某些软件包的依赖关系未得到满足。

要了解有关软件包依赖关系的更多信息，请在配置菜单中搜索该软件包符号（请参阅第 8.1 节）。

然后，您可能需要递归地启用几个选项（与未满足的依赖项相对应），才能最终可选择该软件包。

如果是由于一些未满足依赖关系的工具链选项导致该软件包不可见，那您应当进行完全重新构建（有关更多说明，请参阅第 8.1 节）。

10.6 为什么不将 target 目录用作 chroot 目录

有很多原因不将 target 目录用作 chroot 目录，其中包括：

- 在 target 目录中未正确设置文件所有权、模式和权限；
- 在 target 目录中未创建设备节点。

由于这些原因，若将 target 目录用作新的 root 目录，则通过 chroot 运行的命令很可能会失败。*（译者注：此处的 target 目录是指 output/target/）*

如果您想在 chroot 中运行目标文件系统，或者作为 NFS 根目录，请使用 images/中生成的文件系统镜像压缩包并将其提取为 root。*（译者注：此处的 images/是指 output/images/）*

10.7 为什么 Buildroot 不生成二进制软件包（.deb，.ipkg...）

在 Buildroot 列表上经常讨论的一项功能是“程序包管理”。可概括为，添加对 Buildroot 中哪个软件包安装了哪些文件的跟踪，目标是：

- 当从 menuconfig 菜单中取消选择该软件包时，能够删除该软件包所安装的文件；
- 能够生成可以安装在目标上的二进制软件包（ipk 或其他格式），而无需重新生成新的根文件系统镜像。

通常，大多数人认为这很容易做到：只需跟踪哪个软件包安装了哪些文件，然后在取消选择该软件包时将其删除即可。但是，它比这复杂得多：

- 它不仅与 target/目录有关，还与 host/<tuple>/sysroot 中的 sysroot 和 host/目录本身有关。必须跟踪各种软件包在这些目录中安装的所有文件。
- 从配置中取消选择软件包时，仅删除其安装的文件是不够的。还必须删除它的所有反向依赖关系（即依赖它的软件包）并重新构建所有这些软件包。例如，软件包 A 可选地依赖于 OpenSSL 库。两者都被选中，并且构建了 Buildroot。软件包 A 使用 OpenSSL 进行加密支持。稍后，从配置中取消选择 OpenSSL，但是保留了软件包 A（由于 OpenSSL 是可选的依赖项，所以这是可能的）。如果仅删除 OpenSSL 文件，则软件包 A 所安装的文件将受到破坏：它们使用的库将不再出现在目标上。尽管从技术上来讲这是可行的，但是它给 Buildroot 增加了很多复杂性，这与我们一直坚持的简易性背道而驰。
- 除了前面的问题，还有一种情况是 Buildroot 甚至不知道这个可选依赖项。例如，软件包 A 在版本 1.0 中从未使用过 OpenSSL，但在版本 2.0 中，它会自动使用 OpenSSL（如果有）。如果未更新 Buildroot 的.mk 文件以将其考虑在内，则软件包 A 将不属于 OpenSSL 的反向依赖项，并且在删除 OpenSSL 时也不会删除和重建软件包 A。可以肯定的是，应该修复软件包 A 的.mk 文件，以提及该可选依赖项，但与此同时，您可能具有不可复制的行为。
- 该请求还允许将 menuconfig 菜单中的更改应用于 output 目录而不必从头开始重新构建所有内容。但是，这很难用可靠的方式来实现：软件包子选项在更改时会发生什么（我们必须检测到这一点，并从头开始重新构建该软件包，以及可能会重新构建所有的反向依赖关系），例如更改工具链选项会发生什么。目前，Buildroot 所做的工作既清晰又简单，因此它的行为是非常可靠的，并且易于支持用户。如果在下一次“make”之后应用 menuconfig 菜单中所做的配置更改，则它必须在所有情况下都能正确地工作，并且不会出现一些异常情况。它的风险是将会获取到一些错误报告，例如“我启用了软件包 A、B 和 C，然后运行了 make，然后禁用了软件包 C，然后启用了软件包 D 并运行了 make，然后重新启用了软件包 C 和启用了软件包 E，然后有一个构建失败”。或更糟糕的是，“我先进行了一些配置，然后构建，然后进行了一些更改，再构建，更多的更改，构建，更多的更改，构建，但现在失败了，但是我不记得我所做的所有更改以及更改顺序”。这将不可能得到支持。

由于所有这些原因，得出的结论是，添加对已安装文件的跟踪以在取消选择该软件包时将它们删除，或者生成二进制软件包的存储仓库，是很难可靠实现的，并且这会增加很多复杂性。

在此问题上，Buildroot 开发人员作以下立场声明：

- Buildroot 致力于使生成根文件系统变得容易（顺便说一下，因此得名）。我们想让 Buildroot 更擅长：构建根文件系统。
- Buildroot 并不是要成为发行版（或发行版生成器）。大多数 Buildroot 开发人员认为这不是我们应该追求的目标。我们相信，会有比 Buildroot 更适合生成发行版的其他工具。例如 Ope

n Embedded 或者 openWRT，就是这样的工具。

• 我们倾向于将 Buildroot 推向易于（甚至更易于）生成完整根文件系统的方向。这就是 Buildroot 在人群中脱颖而出的原因（当然还有其他原因！）。

• 我们认为，对于大多数嵌入式 Linux 系统，二进制软件包不是必需的，并且可能是有害的。使用二进制软件包时，意味着可以对系统进行部分升级，这会产生大量可能的软件包版本组合，在嵌入式设备升级之前应先对其进行测试。另一方面，通过一次性升级整个根文件系统镜像来进行完整的系统升级，可以确保部署到嵌入式系统的镜像确实是经过测试和验证的。

10.8 如何加快构建过程

由于 Buildroot 经常涉及对整个系统进行完全重新构建，这可能会耗时很长，因此我们在下面提供一些技巧来帮助减少构建时间：

• 使用预构建的外部工具链，而不是默认的 Buildroot 内部工具链。通过使用预构建的 Linaro 工具链（在 ARM 上）或 Sourcery CodeBench 工具链（用于 ARM、x86、x86-64、MIPS 等），您将在每次完全重建时节省该工具链的构建时间，大约为 15 至 20 分钟。请注意，在系统其余部分正常工作后，临时使用外部工具链并不会阻止您切换回内部工具链（这可能会提供更高级别的自定义功能）；

• 使用 ccache 编译器缓存（请参阅第 8.13.3 节）；

• 了解仅重建您真正关心的少数软件包的有关信息（请参阅第 8.3 节），但请注意，有时仍然需要进行完全重建（请参阅第 8.2 节）；

• 确保您没有让运行 Buildroot 的 Linux 系统使用虚拟机。众所周知，大多数虚拟机技术都会对 I/O 产生显著的性能影响，这对于构建源代码而言确实很重要。

• 确保您仅使用本地文件：请勿尝试通过 NFS 进行构建，这会大大降低构建速度。在本地拥有 Buildroot 的下载文件夹也会有一些帮助。

• 购买新硬件。SSD 和大量 RAM 是加快构建速度的关键。

• 尝试使用顶级并行构建，请参阅第 8.11 节。

11 已知问题

• 如果选项包含$符号，则无法通过 BR2_TARGET_LDFLAGS 传递额外的链接器选项。例如，以下内容将被打断：BR2_TARGET_LDFLAGS="-Wl,-rpath='$ORIGIN/../lib'"。

• SuperH 2 和 ARC 体系架构不支持 libffi 软件包。

• prboom 软件包使用 Sourcery CodeBench 2012.09 版本的 SuperH 4 编译器会触发编译器故障。

12 法律声明和许可

12.1 遵守开源许可证

Buildroot 的所有最终产品（工具链、根文件系统、内核、引导加载程序）都包含以各种许可证发布的开源软件。

使用开源软件，您可以自由地构建丰富的嵌入式系统，可以选择各种软件包，但是您也需要承担一些必须了解和兑现的义务。一些许可证要求您在产品文档中发布许可文本。其他许可证要求您将软件的源代码重新分发给接收您产品的人。

在每个软件包中都记录了每个许可证的确切要求，并且您（或您的法律办公室）有责任遵守这些要求。为了使您更轻松，Buildroot 可以为您收集一些您可能需要的材料。要生成此材料，请在使用“make menuconfig”、“make xconfig”或“make gconfig”配置 Buildroot 后，运行：

```
make legal-info
```

Buildroot 将收集与法律相关的材料至您的 output 目录中 legal-info/子目录下。在那里您会看到：

- README 文件，该文件汇总了生成的材料说明并包含 Buildroot 无法生成的材料的有关警告。
- buildroot.config，这是 Buildroot 的配置文件，通常使用“make menuconfig”生成，并且是复制构建过程所必需的。
- 所有软件包的源代码；目标软件包和宿主机软件包分别保存在 source/和 host-sources/子目录中。设置“<PKG>_REDISTRIBUTE = NO”的软件包将不会保存源代码。该目录也会保存已应用的补丁程序，以及一个名为 series 的文件，它会按照应用顺序来列出补丁程序。补丁程序与它们所修改的文件具有相同的许可证。请注意：Buildroot 会将其他补丁程序应用于基于 autotools 软件包的 Libtool 脚本，这些补丁程序可以在 Buildroot 中 support/libtool 下找到，由于技术限制，它们不会与软件包源文件一起保存。您可能需要手动收集它们。
- mainfest 清单文件（一个用于宿主机，一个用于目标软件包），该文件列出了已配置的软件包、它们的版本、许可证和相关信息。其中一些信息可能在 Buildroot 中未定义；这些条目会被标记为“unknown”。
- 所有软件包的许可文本，目标软件包和宿主机软件包的许可文本分别在 licenses/和 host-licenses/子目录中。如果在 Buildroot 中未定义许可文件，则不会生成该文件，并且在 README 文件中的警告（warning）会指出这一点。

请注意，Buildroot 的 legal-info 功能的目的是生成与软件包许可证法律合规性有关的所有材料。Buildroot 不会尝试生成您必须以某种方式公开的准确材料。当然，在严格遵守法律的前提下，所生成的材料要比需要的多。例如，为根据 BSD-like 许可证发布的软件包生成源代码，您无需以源码形式重新分发。

此外，由于技术限制，Buildroot 不会生成您即将需要或者可能需要的某些材料，例如一些外部工具链的源代码以及 Buildroot 源代码本身。当运行“make legal-info”时，Buildroot 会在 README 文件中生成一些警告（warning），以通知您无法保存的相关材料。

最后，请记住“make legal-info”的输出是基于每个软件包配方中的声明性语句。Buildroot 开发人员会尽其所能，尽最大努力让这些声明性语句尽可能准确。但是，这些声明性语句很可能是不完全准确的，也不是详尽的。您（或您的法律部门）必须先检查“make legal-info”的输出，然后才能将其用作自己的合规性交付。请参阅 Buildroot 发行版根目录下 COPYING 文件中的“NO WARRANTY”条款（第 11 和 12 条）。

12.2 遵守 Buildroot 许可证

Buildroot 本身是一个开源软件，在 GNU 通用公共许可证，版本 2 或（根据您的选择）任何更高版本下发布，下面描述的软件包补丁程序除外。然而，作为一个构建系统，它通常不是最终产品的一部分：如果您为设备开发根文件系统、内核、引导加载程序或工具链，则 Buildroot 的代码仅存在于开发计算机上，而不是在设备中。

尽管如此，Buildroot 开发人员的普遍看法是，在发布包含 GPL 许可软件的产品时，您应该发布 Buildroot 源代码以及其他软件包的源代码。这是因为 GNU GPL 将可执行文件的“完整源代码”定义为“它包含所有模块的所有源代码，以及任何相关的接口定义文件，以及用于控制

可执行文件编译和安装的脚本”。Buildroot 是属于“用于控制可执行文件编译和安装的脚本”中的部分，因此，它被视为必须重新分发的材料的一部分。

请记住，这只是 Buildroot 开发人员的意见，如有任何疑问，请咨询法律部门或者律师。

12.2.1 软件包补丁

Buildroot 还捆绑了应用于各种软件包源码的补丁文件。这些补丁不属于 Buildroot 的许可证范围。但是，它们受到应用补丁的软件的许可保护。当所述软件有多个许可证时，Buildroot 补丁程序仅在可公开访问的许可证下提供。

有关技术上的细节，请参阅第 18 章。

13 Buildroot 之外

13.1 引导生成的镜像

13.1.1 NFS 引导

要实现 NFS 引导，请在“Filesystem images”菜单中启用“tar the root filesystem”。

完成构建后，只需运行以下命令来设置 NFS 根目录：

```
sudo tar -xavf /path/to/output_dir/rootfs.tar -C /path/to/nfs_root_dir
```

请记住将此路径添加到/etc/exports（即“/path/to/nfs_root_dir”）。然后，您可以从目标执行 NFS 引导。

13.1.2 Live CD

要生成 live CD 镜像，请在“Filesystem images”菜单中启用“iso image”选项。注意，此选项仅在 x86 和 x86-64 体系架构上可用，并且需要使用 Buildroot 来构建内核。

您可以使用 IsoLinux、Grub 或 Grub 2 作为 bootloader 来构建 live CD 镜像，但只有 IsoLinux 支持将该镜像同时用作 live CD 和 live USB（通过“Build hybrid image”选项）。

您可以使用 QEMU 来测试 live CD 镜像：

```
qemu-system-i386 -cdrom output/images/rootfs.iso9660
```

如果是 hybrid ISO，也可以将其用作硬盘镜像：

```
qemu-system-i386 -hda output/images/rootfs.iso9660
```

可以使用 dd 命令轻松将其刷新到 USB 驱动器：

```
dd if=output/images/rootfs.iso9660 of=/dev/sdb
```

13.2 Chroot

如果要在生成的镜像中使用 chroot，则应注意以下内容：

- 您应该从 tar 格式的根文件系统镜像中设置新的 root；
- 所选目标体系架构需要与您的主机兼容，或者您应该使用一些 qemu-*二进制文件并在 binfmt 属性中正确设置它，以便能在主机上运行为目标构建的二进制文件；
- Buildroot 目前不提供正确构建和设置的 host-qemu 和 binfmt 来用于这种用途。

第三部分 开发人员指南

14 Buildroot 如何工作

如前面所述，Buildroot 基本上是一组 Makefile 文件，可以使用正确的选项来对所需软件进行下载、配置和编译。它还包含各种软件包的补丁-主要是那些涉及交叉编译工具链（gcc、binutils 和 uClibc）的软件包。

每个软件包基本上只有一个 Makefile 文件，它们以.mk 扩展名命名。Makefile 被分为许多不同的部分。

- toolchain/目录包含与交叉编译工具链相关的所有软件的 Makefile 和相关文件：binutils、gcc、gdb、kernel-header 和 uClibc。
- arch/目录包含 Buildroot 支持的所有处理器体系架构的定义。
- package/目录包含所有用户空间工具和库（Buildroot 可以将它们编译并添加到目标根文件系统）的 Makefile 和相关文件。每个软件包都有一个子目录。
- linux/目录包含 Linux 内核的 Makefile 和相关文件。
- boot/目录包含 Buildroot 支持的 Bootloader 的 Makefile 和相关文件。
- system/目录包含对系统集成的支持，例如目标文件系统框架 skeleton 和 init 系统的选择。
- fs/目录包含与生成目标根文件系统镜像有关的软件的 Makefile 和相关文件。

每个目录至少包含 2 个文件：

- something.mk 是用于下载、配置、编译和安装软件包 something 的 Makefile。
- Config.in 是配置工具描述文件的一部分。它描述与软件包有关的选项。

主 Makefile 执行以下步骤（一旦配置完成）：

- 创建所有输出目录：staging、target、build 等（默认在 output/目录中，可以使用“O=”来指定另一个路径）。
- 生成工具链目标。当使用内部工具链时，这意味着将生成交叉编译工具链。当使用外部工具链时，这意味着将检查外部工具链的功能并将其导入到 Buildroot 环境。
- 生成 TARGETS 变量中列出的所有目标。该变量由所有单个组件的 Makefile 填充。生成这些目标将触发用户空间软件包（库、程序集）、内核、引导加载程序的编译以及根文件系统镜像的生成，具体取决于配置。

15 编码风格

总体而言，这些编码风格规则可以帮助您在 Buildroot 中添加新文件或者重构现有文件。

如果您需要稍微修改现有的一些文件，那么保持整个文件的一致性是很重要的，因此您可以：

- 要么遵循此文件中使用的可能被弃用的编码风格；
- 要么完全重做，以使其符合这些规则。

15.1 Config.in 文件

Config.in 文件包含 Buildroot 中几乎所有可配置的条目。

条目具有以下样式：

```
config BR2_PACKAGE_LIBFOO
```

```
        bool "libfoo"
        depends on BR2_PACKAGE_LIBBAZ
        select BR2_PACKAGE_LIBBAR
        help
          This is a comment that explains what libfoo is. The help text
          should be wrapped.

          http://foosoftware.org/libfoo/
```

- bool、depends on、select 和 help 均以一个制表符 tab 缩进（8 字符）。
- help 文本本身应缩进一个制表符 tab 和两个空格。
- help 文本应换行以适配 72 列，其中制表符为 8 个字符，因此文字本身为 62 个字符。

Config.in 文件是 Buildroot 所使用的配置工具（即常规 Kconfig）的输入。有关 Kconfig 语言的更多详细信息，请参阅 http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt。

15.2 .mk 文件

- 标头：该文件以标头起始。它包含模块名称，最好用小写字母，并包含在由 80 个“#”字符组成的分隔符之间。标头后必须有一行空白行：

```
################################################################################
#
# libfoo
#
################################################################################
```

- 赋值：使用“=”，前后加一空格：

```
LIBFOO_VERSION = 1.0
LIBFOO_CONF_OPTS += --without-python-support
```

不必将“=”对齐。

- 缩进：仅使用制表符 tab：

```
define LIBFOO_REMOVE_DOC
        $(RM) -fr $(TARGET_DIR)/usr/share/libfoo/doc \
                $(TARGET_DIR)/usr/share/man/man3/libfoo*
endef
```

注意，define 块内的命令应始终以制表符 tab 开头，make 才会识别它们为命令。

- 可选依赖项：

- 首选多行语法。

Yes：

```
ifeq ($(BR2_PACKAGE_PYTHON),y)
LIBFOO_CONF_OPTS += --with-python-support
LIBFOO_DEPENDENCIES += python
else
LIBFOO_CONF_OPTS += --without-python-support
endif
```

NO：

```
LIBFOO_CONF_OPTS += --with$(if $(BR2_PACKAGE_PYTHON),,out)-python-support
```

```
LIBFOO_DEPENDENCIES += $(if $(BR2_PACKAGE_PYTHON),python,)
```

- 保持配置选项和依赖项紧密。
- 可选钩子 hooks：将钩子变量定义和赋值放在一个 if 块中。

  YES：

```
ifneq ($(BR2_LIBFOO_INSTALL_DATA),y)
define LIBFOO_REMOVE_DATA
	$(RM) -fr $(TARGET_DIR)/usr/share/libfoo/data
endef
LIBFOO_POST_INSTALL_TARGET_HOOKS += LIBFOO_REMOVE_DATA
endif
```

  NO：

```
define LIBFOO_REMOVE_DATA
	$(RM) -fr $(TARGET_DIR)/usr/share/libfoo/data
endef

ifneq ($(BR2_LIBFOO_INSTALL_DATA),y)
LIBFOO_POST_INSTALL_TARGET_HOOKS += LIBFOO_REMOVE_DATA
endif
```

15.3 documentation 文档

documentation 文档使用 asciidoc 格式。

有关 asciidoc 语法的更多信息，请参阅 http://www.methods.co.nz/asciidoc/userguide.html。

15.4 支持脚本

support/和 utils/目录中的一些脚本是用 Python 编写的，它们应遵循“PEP8 Style Guide for Python Code”。

16 添加对特定硬件板的支持

Buildroot 包含了一些公开可用的硬件板的基本配置，因此该硬件板的用户可以很容易地构建一个可正常工作的系统。欢迎您为 Buildroot 添加对其他硬件板的支持。

为了做到这一点，您需要创建一个常规的 Buildroot 配置来为硬件建立一个基本系统：（内部）工具链、内核、引导加载程序、文件系统和一个简单的只有 BusyBox 的用户空间。不应选择特定的软件包：配置应尽可能少，并且应仅为目标平台构建可正常工作的 BusyBox 基本系统。当然，您可以为您的内部项目使用更复杂的配置，但 Buildroot 项目将仅集成基本的硬件板配置。这是因为软件包的选择是高度特定于应用程序的。

一旦您有了一个已知的工作配置后，可运行“make savedefconfig”。这将在 Buildroot 根目录下生成一个最小的 defconfig 文件。可将此文件移动到 configs/目录中，并将其重命名为<boardname>_defconfig。如果配置有些复杂，则最好手动将它重新格式化并分成几个部分，且在每部分前面添加对应的注释。典型部分是 Architecture、Toolchain options（通常只是 linux 标头版本）、Firmware、Bootloader、Kernel 和 Filesystem。*（译者注：可参考现有 defconfig 文件的做法）*

对于不同的组件，请始终使用固定的版本或 commit 哈希值，而不是“latest”版本。例如，设置“BR2_LINUX_KERNEL_CUSTOM_VERSION=y”和设置 BR2_LINUX_KERNEL_CUSTO

M_VERSION_VALUE 为您所测试的内核版本。

建议尽量使用 Linux 内核和 bootloader 的上游版本，并尽量使用内核和 bootloader 的默认配置。如果它们不适用于您的硬件板或不存在默认值，我们支持您将修复程序发送到对应的上游项目。

与此同时，您可能需要存储特定于目标平台的内核或引导加载程序的配置或补丁。为此，请创建目录 board/<manufacturer>和子目录 board/<manufacturer>/<boardname>。然后，您可以将补丁程序和配置存储在这些目录中，并从 Buildroot 主配置中引用它们。

有关更多详细信息，请参阅第 9 章。

17 向 Buildroot 添加新软件包

本章介绍如何集成新软件包（用户空间库或应用程序）到 Buildroot 中。本章还展示现有软件包是如何集成的，这是修复问题或调整它们配置所必需的。

添加新软件包时，需确保在各种条件下进行测试（请参阅第 17.23.3 节），并检查其编码风格（请参阅第 17.23.2 节）。

17.1 软件包目录

首先，为您的软件在 package 目录下创建一个目录，例如 libfoo。

某些软件包已在子目录中按主题分组：x11r7、qt5 和 gstreamer。如果您的软件包适合其中的类别，请在这些类别中创建软件包目录。但是，不建议使用新的子目录。

17.2 配置文件

为了让软件包显示在配置工具中，您需要在软件包目录中创建一个 Config 文件。该文件有两种类型：Config.in 和 Config.in.host。

17.2.1 Config.in 文件

对于在目标上使用的软件包，创建一个 Config.in 文件。该文件包含与我们 libfoo 软件相关的选项说明，将在配置工具中使用和显示它。它应基本包含：

```
config BR2_PACKAGE_LIBFOO
        bool "libfoo"
        help
          This is a comment that explains what libfoo is. The help text
          should be wrapped.

          http://foosoftware.org/libfoo/
```

关于配置选项中的 bool 行、help 行和其他元数据信息，都必须以一个制表符 tab 缩进。help 文本本身应缩进一个制表符 tab 和两个空格，每一行应换行以适配 72 列，其中制表符为 8 个字符，因此文本本身为 62 个字符。帮助文本必须在空行后提及项目的上游 URL。

作为针对于 Buildroot 的约定，属性的顺序如下：

1. 选项类型：bool、string…以及相关提示；
2. default 默认值，如果需要；
3. 目标的任何依赖，以“depends on”形式；
4. 工具链的任何依赖，以“depends on”形式；

5. 其他软件包的任何依赖，以“depends on”形式；
6. 以“select”形式的任何依赖；
7. help 关键字和 help 文本。

您可以将其他子选项添加到“if BR2_PACKAGE_LIBFOO…endif”语句中，以配置特定内容。您可以查看其他软件包的示例。Config.in 文件的语法与内核 Kconfig 文件的语法相同。有关此语法的文档，请参阅：http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt。

最后，您必须将新的 libfoo/Config.in 添加到 package/Config.in 中（或者如果您决定将软件包放入现有的类别中，则应添加到该类别的子目录下）。每个类别包含的文件将按字母顺序排序，除了软件包本身的名字外，不应包含任何其他内容。

```
source "package/libfoo/Config.in"
```

17.2.2 Config.in.host 文件

还需要为宿主机系统构建一些软件包。这里有两种情况：

• 宿主机软件包仅需满足一个或多个目标软件包在构建时的依赖关系。在这种情况下，请将 host-foo 添加到目标软件包的 BAR_DEPENDENCIES 变量，而不需创建 Config.in.host 文件。

• 用户能在配置菜单中明确选择宿主机软件包。在这种情况下，请为该宿主机软件包创建一个 Config.in.host 文件：

```
config BR2_PACKAGE_HOST_FOO
        bool "host foo"
        help
          This is a comment that explains what foo for the host is.

          http://foosoftware.org/foo/
```

它的编码样式和选项与 Config.in 文件相同。

最后，您必须将新的 libfoo/Config.in.host 添加到 package/Config.in.host 中。其中包含的文件将按字母顺序排序，除软件包本身的名字外，不应包含任何其他内容。

```
source "package/foo/Config.in.host"
```

然后可以从“Host utilities”菜单中获得宿主机软件包。

17.2.3 选择 depends on 或 select

软件包的 Config.in 文件还必须确保启用了依赖关系。通常，Buildroot 使用以下规则：

• 对库的依赖关系使用“select”的依赖关系类型。通常这些依赖关系并不明显，因此让具有 kconfig 的系统来确保已选择依赖关系是很有意义的。例如，libgtk2 软件包使用“select BR2_PACKAGE_LIBGLIB2”来确保启用该库。“select”关键字使用向后语义来表示依赖关系。

• 当用户确实需要在乎依赖关系时，请使用“depends on”的依赖关系类型。通常，Buildroot 使用这种依赖关系类型来处理对目标体系架构、MMU 支持和工具链选项的依赖（请参阅第 17.2.4 节），或者对“大”的事物的依赖，例如 X.org 系统。“depends on”关键字使用向前语义来表示依赖关系。

请注意，kconfig 语言目前存在的问题是这两个依赖关系的语义未在内部关联。因此，有可能选择了一个软件包，而它的依赖关系或要求却不能满足。

以下示例将说明“select”和“depends on”的用法。

```
config BR2_PACKAGE_RRDTOOL
        bool "rrdtool"
```

```
        depends on BR2_USE_WCHAR
        select BR2_PACKAGE_FREETYPE
        select BR2_PACKAGE_LIBART
        select BR2_PACKAGE_LIBPNG
        select BR2_PACKAGE_ZLIB
        help
          RRDtool is the OpenSource industry standard, high performance
          data logging and graphing system for time series data.

          http://oss.oetiker.ch/rrdtool/

comment "rrdtool needs a toolchain w/ wchar"
        depends on !BR2_USE_WCHAR
```

请注意，这两种依赖关系类型仅在具有相同类型的依赖关系中传递。这意味着，在以下示例中：

```
config BR2_PACKAGE_A
        bool "Package A"

config BR2_PACKAGE_B
        bool "Package B"
        depends on BR2_PACKAGE_A

config BR2_PACKAGE_C
        bool "Package C"
        depends on BR2_PACKAGE_B

config BR2_PACKAGE_D
        bool "Package D"
        select BR2_PACKAGE_B

config BR2_PACKAGE_E
        bool "Package E"
        select BR2_PACKAGE_D
```

- 如果 Package B 已被选中，则 Package C 将是可见的；如果 Package A 已被选中，则 Package B 是可见的。
- 选择 Package E 时，将选择 Package D，而 Package D 将选择 Package B，可它不检查 Package B 的依赖关系，因此不会选择 Package A。
- 由于选择了 Package B，但没有选择 Package A，这违反了 Package B 对 Package A 的依赖性。因此，在这种情况下，必须显式添加可传递的依赖关系：

```
config BR2_PACKAGE_D
        bool "Package D"
        select BR2_PACKAGE_B
        depends on BR2_PACKAGE_A
```

```
config BR2_PACKAGE_E
        bool "Package E"
        select BR2_PACKAGE_D
        depends on BR2_PACKAGE_A
```

总体而言，对于软件包库的依赖关系，应该首选“select”。

请注意，此类依赖关系规则将确保启用了依赖关系选项，但是它们没必要在您的软件包之前进行构建。因此，还需要在软件包的.mk 文件中表达依赖关系。

进一步的格式细节，请参阅“编码风格”章节。

17.2.4 对目标和工具链选项的依赖

有许多软件包依赖于工具链的某些选项：C 库选择、C++支持、线程支持、RPC 支持、wchar 支持或动态库支持。有些软件包只能在某些目标体系架构上进行构建，或者在处理器带有 MMU 的情况下构建。

在 Config.in 文件中必须使用适当的“depends on”语句来表达这些依赖关系。另外，对于工具链选项的依赖，当未启用某个选项时，则应显示注释内容*（译者注：即 comment 选项描述的内容）*，以便让用户知道为什么该软件包不可用。对于目标体系架构或 MMU 支持的依赖，则不应在注释（comment）中显示：由于用户不太可能自由地选择另一个目标，因此明确显示这些依赖关系几乎没有意义。

当满足工具链选项的依赖关系时，如果 config 选项本身可见，则 comment 选项才应该可见。这意味着软件包的所有其他依赖关系（包括对目标体系架构和 MMU 支持的依赖关系）必须在 comment 选项上进行重复定义。为了保持清晰，应将这些非工具链选项的“depends on”语句与工具链选项的“depends on”语句分开。如果在同一文件中（通常是主软件包）对某个 config 选项有依赖性，则最好有一个全局的 if ... endif 结构，而不要在 comment 选项和其他 config 选项上重复定义“depends on”语句。

软件包 foo 依赖项注释的一般格式为：

```
foo needs a toolchain w/ featA, featB, featC
```

例如：

```
mpd needs a toolchain w/ C++, threads, wchar
```

或者

```
crda needs a toolchain w/ threads
```

请注意，此文本需保持简短，以使其适合 80 个字符的终端。

本节其余部分将列举不同的目标和工具链选项、要依赖的相应配置符号以及在 comment 选项中使用的文本。

- Target architecture
  - 依赖项符号：BR2_powerpc、BR2_mips、...（请参阅 arch/Config.in）
  - comment 字符串：没有要添加的注释
- MMU support
  - 依赖项符号：BR2_USE_MMU
  - comment 字符串：没有要添加的注释
- gcc _sync*内置函数，用于原子操作。可对 1 字节、2 字节、4 字节和 8 字节变量进行原子操作。由于不同的体系架构支持不同大小的原子操作，因此每种大小都可以使用一个依赖项符号：

- 依赖项符号：BR2_TOOLCHAIN_HAS_SYNC_1 表示 1 字节，BR2_TOOLCHAIN_HAS_SYNC_2 表示 2 字节，BR2_TOOLCHAIN_HAS_SYNC_4 表示 4 字节，BR2_TOOLCHAIN_HAS_SYNC_8 表示 8 字节。
- comment 字符串：没有要添加的注释

- gcc _atomic*内置函数，用于原子操作。
  - 依赖项符号：BR2_TOOLCHAIN_HAS_ATOMIC
  - comment 字符串：没有要添加的注释
- Kernel headers
  - 依赖项符号：BR2_TOOLCHAIN_HEADERS_AT_LEAST_X_Y，（用适当的版本替换 X_Y，请参阅 toolchain/Config.in）
  - comment 字符串：headers >= X.Y and/or headers <= X.Y（用适当的版本替换 X.Y）
- GCC version
  - 依赖项符号：BR2_TOOLCHAIN_GCC_AT_LEAST_X_Y，（用适当的版本替换 X_Y，请参阅 toolchain/Config.in）
  - comment 字符串：gcc >= X.Y and/or gcc <= X.Y（用适当的版本替换 X.Y）
- Host GCC version
  - 依赖项符号：BR2_HOST_GCC_AT_LEAST_X_Y，（用适当的版本替换 X_Y，请参阅 Config.in）
  - comment 字符串：没有要添加的注释
  - 请注意，通常不是软件包本身有一个最小的主机 GCC 版本，而是其所依赖的宿主机软件包。
- C library
  - 依赖项符号：BR2_TOOLCHAIN_USES_GLIBC，BR2_TOOLCHAIN_USES_MUSL，BR2_TOOLCHAIN_USES_UCLIBC
  - comment 字符串：对于 C 库，使用的注释文本略有不同：foo needs a glibc toolchain，或者 foo needs a glibc toolchain w/ C++
- C++ support
  - 依赖项符号：BR2_INSTALL_LIBSTDCPP
  - comment 字符串：C++
- D support
  - 依赖项符号：BR2_TOOLCHAIN_HAS_DLANG
  - comment 字符串：Dlang
- Fortran support
  - 依赖项符号：BR2_TOOLCHAIN_HAS_FORTRAN
  - comment 字符串：fortran
- thread support
  - 依赖符号：BR2_TOOLCHAIN_HAS_THREADS
  - comment 字符串：threads（除非还需要 BR2_TOOLCHAIN_HAS_THREADS_NPTL，在这种情况下，只指定 NPTL 即可）
- NPTL thread support
  - 依赖项符号：BR2_TOOLCHAIN_HAS_THREADS_NPTL
  - comment 字符串：NPTL
- RPC support

- 依赖项符号：BR2_TOOLCHAIN_HAS_NATIVE_RPC
- comment 字符串： RPC

• wchar support
- 依赖项符号：BR2_USE_WCHAR
- comment 字符串：wchar

• dynamic library
- 依赖项符号：!BR2_STATIC_LIBS
- comment 字符串：dynamic library

17.2.5 对 buildroot 构建的 Linux 内核的依赖

有些软件包需要 buildroot 构建 Linux 内核。这些软件包通常是内核模块或固件。应该在 Config.in 文件中添加注释（comment）以表达这种依赖性，类似于对工具链选项的依赖。通用格式为：

```
foo needs a Linux kernel to be built
```

如果同时依赖于工具链选项和 Linux 内核，请使用以下格式：

```
foo needs a toolchain w/ featA, featB, featC and a Linux kernel to be built
```

17.2.6 对 udev /dev 管理的依赖

如果软件包需要 udev /dev 管理，则它应依赖于符号 BR2_PACKAGE_HAS_UDEV，并应添加以下注释：

```
foo needs udev /dev management
```

如果同时依赖于工具链选项和 udev /dev 管理，请使用以下格式：

```
foo needs udev /dev management and a toolchain w/ featA, featB, featC
```

17.2.7 对 virtual 软件包提供的功能的依赖

某些功能可以由多个软件包提供，例如 openGL 库。
有关 virtual 软件包的更多信息，请参阅第 17.11 节。

17.3 .mk 文件

最后，这是最难的一部分。创建一个名为 libfoo.mk 的文件。它描述软件包该如何下载、配置、构建和安装等等。

根据软件包的类型，必须使用不同的基础结构以不同的方式来编写.mk 文件：

• 通用软件包的 Makefiles 文件（不使用 autotools 或 CMake）：这些文件所使用的基础结构类似于基于 autotools 的软件包的基础结构，但需要开发人员多做一点工作。他们指定应如何配置、编译和安装软件包。此基础结构必须用于所有不使用 autotools 作为其构建系统的软件包。在将来，我们可能会为其他构建系统编写其他的特定基础结构。我们将在它的 tutorial 和 reference 章节中介绍它们。

• 基于 autotools 的软件（autoconf，automake 等）的 Makefiles：autotools 是一种非常常见的构建系统，我们为此类软件包提供了专用的基础结构。此基础结构必须用于依赖 autotools 作为其构建系统的新软件包。我们将在它的 tutorial 和 reference 章节中介绍它们。

• 基于 cmake 的软件的 Makefiles：由于 CMake 是越来越常用的构建系统，并且具有标准化的行为，因此我们为此类软件包提供了专用的基础结构。此基础结构必须用于依赖 CMake 的新软件包。我们将在它的 tutorial 和 reference 章节中介绍它们。

• Python 模块的 Makefiles：我们为使用 distutils 或 setuptools 机制的 Python 模块提供专用的基础结构。我们将在它的 tutorial 和 reference 章节中介绍它们。

• Lua 模块的 Makefiles：我们为通过 LuaRocks 网站获取的 Lua 模块提供专用的基础结构。我们将在它的 tutorial 和 reference 章节中介绍它们。

更多的格式细节，请参阅第 15.2 节。

17.4 .hash 文件

如果可能，您必须添加第三个文件，文件名为 libfoo.hash，其中包含为 libfoo 软件包下载的文件的哈希值。不添加.hash 文件的唯一原因是，由于软件包的下载方式，导致不可能进行 hash 检查。

当软件包具有版本选择选项时，则 hash 文件可以存储在以版本命名的子目录中，例如 package/libfoo/1.2.3/libfoo.hash。如果不同版本有不同的许可条款，但它们存储在同一文件中，则这一点特别重要。否则，hash 文件应保留在软件包的主目录中。

存储在此文件中的哈希值用于校验下载文件和许可文件的完整性。

此.hash 文件的格式是一行检查一个文件的哈希值，每一行包含以下三个字段，并以两个空格隔开：

- 哈希类型，可选其中之一：
  - md5，sha1，sha224，sha256，sha384，sha512，none
- 文件哈希：
  - 对于 none，一个或多个非空格字符，通常是字符串 xxx
  - 对于 md5，32 个十六进制字符
  - 对于 sha1，40 个十六进制字符
  - 对于 sha224，56 个十六进制字符
  - 用于 sha256，64 个十六进制字符
  - 对于 sha384，96 个十六进制字符
  - 对于 sha512，128 个十六进制字符
- 文件名：
  - 对于源码包：文件的基本名称，不包含任何目录组件；
  - 对于许可证文件：FOO_LICENSE_FILES 中显示的路径。

以“#”符号开头的行被视为注释，并被忽略。空行也将被忽略。

单个文件可以有多个哈希值，每个哈希值都在自己的行上。在这种情况下，所有哈希值必须都匹配。

请注意，在理想情况下，此文件中存储的哈希值应与上游发布的哈希值相匹配，例如在它网站上的电子邮件公告中...如果上游提供了多种类型哈希值（例如 sha1 和 sha512），则最好将所有这些哈希值添加到.hash 文件中。如果上游不提供任何哈希值，或者只提供一个 md5 哈希值，则您至少要自己计算一个强哈希值（最好是 sha256，而不是 md5），并在哈希值上方的注释行中提及这一点。

另外请注意，许可证文件的哈希值是用于检测软件包版本迭代时的许可证更改。在运行“make legal-info”目标期间会检查哈希值。对于具有多个版本的软件包（例如 Qt5），请在该软件包的子目录<packageversion>中创建 hash 文件（请参阅第 18.2 节）。

哈希类型 none 保留给从存储库下载的那些软件包，例如 git clone、subversion checkout...

下面示例中，定义了 libfoo-1.2.3.tar.bz2 源码包的上游 sha1 和 sha256 哈希值，二进制 blob 的上游 md5 和本地计算的 sha256 哈希值，下载补丁的 sha256 哈希值和没有哈希值的源码包：

```
# Hashes from: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.{sha1,sha256}:
sha1 486fb55c3efa71148fe07895fd713ea3a5ae343a libfoo-1.2.3.tar.bz2
sha256 efc8103cc3bcb06bda6a781532d12701eb081ad83e8f90004b39ab81b65d4369 libfoo-1.2.3.tar.bz2

# md5 from: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.md5, sha256 locally computed:
md5 2d608f3c318c6b7557d551a5a09314f03452f1a1 libfoo-data.bin
sha256 01ba4719c80b6fe911b091a7c05124b64eeece964e09c058ef8f9805daca546b libfoo-data.bin

# Locally computed:
sha256 ff52101fb90bbfc3fe9475e425688c660f46216d7e751c4bbdb1dc85cdccacb9 libfoo-fix-blabla.patch

# No hash for 1234:
none xxx libfoo-1234.tar.gz

# Hash for license files:
sha256 a45a845012742796534f7e91fe623262ccfb99460a2bd04015bd28d66fba95b8 COPYING
sha256 01b1f9f2c8ee648a7a596a1abe8aa4ed7899b1c9e5551bda06da6e422b04aa55 doc/COPYING.LGPL
```

如果存在.hash 文件，并且该文件中包含下载文件的一个或多个哈希值，则 Buildroot 计算出的哈希值（在下载后）必须与.hash 文件中存储的哈希值相匹配。如果一个或多个哈希值不匹配，则 Buildroot 会认为这是一个错误，将删除已下载的文件并中止继续下载。

如果存在.hash 文件，但该文件中不包含下载文件的哈希值，则 Buildroot 会认为这是一个错误并中止下载。但是，下载的文件会保留在下载目录中，因为这通常表明.hash 文件本身存在错误，而下载的文件可能没有问题。

目前 Buildroot 会检查从 http/ftp 服务器或 Git 存储库获取的、使用 scp 复制的以及本地的文件的哈希值。不检查其他版本控制系统（例如 Subversion，CVS 等）的哈希值，因为当从此类版本控制系统中获取源代码时，Buildroot 目前并不会生成可重用的源码压缩包。

只有确保文件是稳定的，才应在.hash 文件中添加哈希值。例如，不能保证由 Github 自动生成的补丁是稳定的，因此其哈希值会随着时间推移而变化。不应下载此类补丁，而应将补丁添加到本地的软件包文件夹中。

如果.hash 文件丢失，则完全不进行检查。

17.5 具有特定构建系统的软件包的基础结构

对于具有特定构建系统的软件包，我们是指那些不具有标准构建系统（例如 autotools 或 CMake）的所有软件包。这通常包括基于手写的 Makefile 或 Shell 脚本作为构建系统的软件包。

17.5.1 generic-package tutorial

```
01: ################################################################################
02: #
```

```
03: # libfoo
04: #
05: ################################################################################
##
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_LICENSE = GPL-3.0+
11: LIBFOO_LICENSE_FILES = COPYING
12: LIBFOO_INSTALL_STAGING = YES
13: LIBFOO_CONFIG_SCRIPTS = libfoo-config
14: LIBFOO_DEPENDENCIES = host-libaaa libbbb
15:
16: define LIBFOO_BUILD_CMDS
17:       $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D) all
18: endef
19:
20: define LIBFOO_INSTALL_STAGING_CMDS
21:       $(INSTALL) -D -m 0755 $(@D)/libfoo.a $(STAGING_DIR)/usr/lib/libfoo.a
22:       $(INSTALL) -D -m 0644 $(@D)/foo.h $(STAGING_DIR)/usr/include/foo.h
23:       $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(STAGING_DIR)/usr/lib
24: endef
25:
26: define LIBFOO_INSTALL_TARGET_CMDS
27:       $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(TARGET_DIR)/usr/lib
28:       $(INSTALL) -d -m 0755 $(TARGET_DIR)/etc/foo.d
29: endef
30:
31: define LIBFOO_USERS
32:       foo -1 libfoo -1 * - - - LibFoo daemon
33: endef
34:
35: define LIBFOO_DEVICES
36:       /dev/foo c 666 0 0 42 0 - - -
37: endef
38:
39: define LIBFOO_PERMISSIONS
40:       /bin/foo f 4755 foo libfoo - - - - -
41: endef
42:
43: $(eval $(generic-package))
```

该 Makefile 从第 7 行开始至第 11 行包含元数据信息：软件包的版本（LIBFOO_VERSIO

N），软件包的压缩包名称（LIBFOO_SOURCE）（建议使用 xz 格式的压缩包），软件包的 Internet 下载位置（LIBFOO_SITE），许可证（LIBFOO_LICENSE）和许可证文本文件（LIBFOO_LICENSE_FILES）。所有变量必须以相同的前缀 LIBFOO_开头。此前缀始终是软件包名称的大写版本（请参阅下文了解在何处定义软件包名称）。

第 12 行，我们指定该软件包要在 staging 空间中安装一些东西。通常库文件需要这么做，因为它们必须在 staging 空间中安装头文件和其他开发文件。这将确保执行由 LIBFOO_INSTALL_STAGING_CMDS 变量列出的命令。

第 13 行，我们指定要对在 LIBFOO_INSTALL_STAGING_CMDS 阶段安装的 libfoo-config 文件进行一些修复。这些*-config 文件是位于$(STAGING_DIR)/usr/bin 目录中的可执行 Shell 脚本文件，由第三方软件包执行，以找到该特定软件包的位置和链接标志。

问题是，所有这些给出的*-config 文件在默认情况下都有一些错误，宿主机系统链接标志不适合进行交叉编译。例如：使用-I/usr/include 而不是-I$(STAGING_DIR)/usr/include，或者：使用-L/usr/lib 而不是-L$(STAGING_DIR)/usr/lib。

因此，需要对这些脚本进行一些 sed 修改，以使它们给出正确的链接标志。提供给 LIBFOO_CONFIG_SCRIPTS 变量的参数是需要修复的 Shell 脚本的文件名称。所有这些名称都是相对于$(STAGING_DIR)/usr/bin 路径的，如果需要，可以指定多个名称。

此外，已从$(TARGET_DIR)/usr/bin 中删除 LIBFOO_CONFIG_SCRIPTS 列出的脚本，因为目标系统不需要它们。

示例 17.1 配置脚本：divine 软件包

软件包 divine 安装 shell 脚本：$(STAGING_DIR)/usr/bin/divine-config

因此它的修复脚本将是：

```
DIVINE_CONFIG_SCRIPTS = divine-config
```

示例 17.2 配置脚本：imagemagick 软件包

软件包 imagemagick 安装以下脚本：$(STAGING_DIR)/usr/bin/{Magick,Magick++,MagickCore,MagickWand,Wand}-config

因此它的修复脚本将是：

```
IMAGEMAGICK_CONFIG_SCRIPTS = \
    Magick-config Magick++-config \
    MagickCore-config MagickWand-config Wand-config
```

第 14 行，我们指定该软件包的依赖项列表。这些依赖项以小写的软件包名称列出，可以是目标软件包（不带 host-前缀），也可以是宿主机软件包（带 host-前缀）。Buildroot 将确保在当前软件包开始配置之前，已构建并安装好所有这些依赖项软件包。

Makefile 的其余部分（第 16 至 29 行）定义在软件包配置、编译和安装过程的不同步骤中应执行哪些操作。LIBFOO_BUILD_CMDS 指示应执行哪些步骤来构建该软件包。LIBFOO_INSTALL_STAGING_CMDS 指示应执行哪些步骤来将软件包安装到 staging 空间。LIBFOO_INSTALL_TARGET_CMDS 指示应执行哪些步骤来将软件包安装到 target 空间。

所有这些步骤都依赖于$(@D)变量，该变量包含软件包源代码的提取目录。

第 31 至 33 行，我们定义该软件包的使用用户（例如，以非 root 用户身份运行守护程序）（LIBFOO_USERS）。

第 35 至 37 行，我们定义该软件包使用的设备节点文件（LIBFOO_DEVICES）。

第 39 至 41 行，我们定义该软件包安装的特定文件的权限（LIBFOO_PERMISSIONS）。

最后，第 43 行，我们调用 generic-package 函数，该函数将根据前面定义的变量生成所有必需的 Makefile 代码，来让您的软件包正常工作。

17.5.2 generic-package reference

通用目标有两种宏形式。generic-package 宏用于需要交叉编译的目标软件包。host-generic-package 宏用于宿主机软件包，是针对宿主机进行编译的。可以在单个.mk 文件中同时调用它们：一次是创建规则以生成目标软件包，一次是创建规则以生成宿主机软件包：

```
$(eval $(generic-package))
$(eval $(host-generic-package)
```

如果目标软件包的编译需要在宿主机上安装一些工具，这可能会很有用。如果软件包名称为 libfoo，则目标软件包的名称也为 libfoo，而宿主机软件包的名称为 host-libfoo。如果其他软件包依赖于 libfoo 或 host-libfoo，则应在其他软件包的 DEPENDENCIES 变量中使用这些名称。

在所有变量定义之后，对 generic-package 宏和 host-generic-package 宏的调用必须位于.mk 文件末尾。对 host-generic-package 宏的调用必须在 generic-package 宏调用之后（如果有）。

对于目标软件包，generic-package 宏使用由.mk 文件定义的、以大写软件包名称 LIBFOO_*为前缀的变量（*译者：此处是以 libfoo 软件包为例*）。host-generic-package 宏使用 HOST_LIBFOO_*变量。对于某些变量，如果 HOST_LIBFOO_前缀变量不存在，则软件包基础结构将使用以 LIBFOO_为前缀的相应变量。对于目标软件包和宿主机软件包，它们可能具有相同值的变量。有关更多信息，请参阅下文。

以下是变量列表，可在.mk 文件中设置，以提供元数据信息（假设软件包名称为 libfoo）：

• LIBFOO_VERSION，必填项，必须包含软件包的版本。请注意，如果 HOST_LIBFOO_VERSION 不存在，则假定它与 LIBFOO_VERSION 相同。它可以是修订版本号，也可以是从软件包版本控制系统中获取的 tag 标签。示例：

- 发布版本：LIBFOO_VERSION = 0.1.2
- git tree 的 sha1：LIBFOO_VERSION = cb9d6aa9429e838f0e54faa3d455bcbab5eef057
- git tree 的标签：LIBFOO_VERSION = v0.1.2

注意：不支持将 git 分支名称用作 FOO_VERSION，因为它不能像人们所期望的那样起作用：

1. 由于是本地缓存，Buildroot 不会重新获取 repository 存储库，因此希望能够访问远程存储库的人会感到非常惊讶和失望。

2. 因为两个构建永远不可能完美地同时进行，并且由于远程存储库随时可能在分支上获得新的提交，所以两个用户使用相同的 Buildroot 源码树并以相同的配置进行构建可能会获得不同的源，从而导致该构建不可重现，人们会因此感到非常惊讶和失望。

• LIBFOO_SOURCE 包含软件包的压缩包名称，Buildroot 将使用该名称从 LIBFOO_SITE 下载压缩包。如果未指定 HOST_LIBFOO_SOURCE，则它默认为 LIBFOO_SOURCE。如果未指定 LIBFOO_SOURCE，则假定该值为 libfoo-$(LIBFOO_VERSION).tar.gz。示例：

```
LIBFOO_SOURCE = foobar-$(LIBFOO_VERSION).tar.bz2
```

• LIBFOO_PATCH 包含以空格分隔的补丁程序名称列表，Buildroot 将下载该列表并将它应用到软件包源代码中。如果该条目包含“://”，则 Buildroot 将假定它为完整的 URL，然后从该位置下载补丁程序。否则，Buildroot 将假定应从 LIBFOO_SITE 下载该补丁程序。如果未指定 HOST_LIBFOO_PATCH，则它默认为 LIBFOO_PATCH。请注意，Buildroot 本身包含的补丁程序使用不同的工作机制：Buildroot 的 package 目录中所有*.patch 格式的文件都将在软件包解压后应用于该软件包（请参阅第 18 章）。最后，LIBFOO_PATCH 变量中列出的补丁将在 Buildroot 的 package 目录中的补丁应用之前进行应用。

• LIBFOO_SITE 提供软件包的位置，可以是 URL 或本地文件系统的路径。URL 支持的类型有 HTTP、FTP 和 SCP，用于检索软件压缩包。在这种情况下，请不要在结尾加上斜杠：Bui

ldroot 会在目录和文件名之间适当地添加斜杠。支持 Git、Subversion、Mercurial 和 Bazaar 的 URL 类型，用于直接从源码管理系统中检索软件包。有一个 helper 帮助程序，可以让从 GitHub 下载源码包变得更加容易（有关详细信息，请参阅第 17.23.4 节）。文件系统路径可用于指定源码包或者包含软件源码的目录。有关如何检索的更多详细信息，请参阅下面的 LIBFOO_SITE_METHOD。请注意，SCP 的 URL 格式应为 scp://[user@]host:filepath，并且此 filepath 是相对于用户 home 目录的，因此您可能需要在路径前面添加一个斜杠以用作绝对路径：scp://[user@]host:/absolutepath。如果未指定 HOST_LIBFOO_SITE，则它默认为 LIBFOO_SITE。示例：

```
LIBFOO_SITE=http://www.libfoosoftware.org/libfoo
LIBFOO_SITE=http://svn.xiph.org/trunk/Tremor
LIBFOO_SITE=/opt/software/libfoo.tar.gz
LIBFOO_SITE=$(TOPDIR)/.../src/libfoo
```

• LIBFOO_DL_OPTS 是以空格分隔的、传递给下载器的其他选项列表。支持所有对 LIBFOO_SITE_METHOD 有效的下载方法；具体有效选项取决于对应下载方法（请参阅相应下载工具的手册）。

• LIBFOO_EXTRA_DOWNLOADS 是以空格分隔的、Buildroot 应下载的其他文件列表。如果条目包含“://”，则 Buildroot 将假定它是完整的 URL，并从该 URL 下载文件。否则，Buildroot 将假定要下载的文件位于 LIBFOO_SITE。Buildroot 不会对这些文件执行任何操作，除了下载它们：将由该软件包配方决定如何从$(LIBFOO_DL_DIR)中使用它们。

• LIBFOO_SITE_METHOD 指定获取或复制软件包源代码的方法。在许多情况下，Buildroot 会通过 LIBFOO_SITE 的内容来猜测该方法，而无需设置 LIBFOO_SITE_METHOD。如果未指定 HOST_LIBFOO_SITE_METHOD，则它默认为 LIBFOO_SITE_METHOD 的值。LIBFOO_SITE_METHOD 的值可能为：

- wget，用于压缩包的常规 FTP/HTTP 下载。当 LIBFOO_SITE 以 http://、https://或 ftp://开头时，默认使用。
- scp，使用 scp 通过 SSH 方式下载压缩包。当 LIBFOO_SITE 以 scp://开头时，默认使用。
- svn，用于从 Subversion 存储库中检索源代码。当 LIBFOO_SITE 以 svn://开头时，默认使用。当 LIBFOO_SITE 指定为 http://Subversion 存储库 URL 时，必须指定“LIBFOO_SITE_METHOD=svn”。Buildroot 执行一次 checkout，并将源代码以压缩包的形式保存在下载缓存中；随后使用该压缩包执行构建而不是执行另一次 checkout。
- cvs，用于从 CVS 存储库中检索源代码。当 LIBFOO_SITE 以 cvs://开头时，默认使用。通过 cvs 下载的源代码与 svn 方法一样会被缓存。默认情况下会假设使用匿名 pserver 模式，否则需要在 LIBFOO_SITE 中显式定义。在假设使用匿名 pserver 访问模式前提下，可接受定义 LIBFOO_SITE=cvs://libfoo.net:/cvsroot/libfoo 和 LIBFOO_SITE=cvs://:ext:libfoo.net:/cvsroot/libfoo。LIBFOO_SITE 必须包含源 URL 以及远程存储库目录。LIBFOO_VERSION 是必填项，并且必须是 tag 标签、分支或日期（例如，“2014-10-20”，“2014-10-20 13:45”，“2014-10-20 13:45+01”，更多详细信息，请参阅“man cvs”）。
- git，用于从 Git 存储库中检索源代码。当 LIBFOO_SITE 以 git://开头时，默认使用。通过 git 下载的源代码与 svn 方法一样会被缓存。
- hg，用于从 Mercurial 存储库中检索源代码。当 LIBFOO_SITE 包含 Mercurial 存储库 URL 时，必须指定“LIBFOO_SITE_METHOD=hg”。通过 hg 下载的源代码与 svn 方法一样会被缓存。
- bzr，用于从 Bazaar 存储库中检索源代码。当 LIBFOO_SITE 以 bzr://开头时，默认使用。通过 bzr 下载的源代码与 svn 方法一样会被缓存。

- file，用于本地压缩包。当 LIBFOO_SITE 指定软件压缩包为本地文件名时，应使用此方法。这对非公开的或未进行版本控制的软件很有用。

- local，用于本地源代码目录。当 LIBFOO_SITE 指定包含软件包源代码的本地目录路径时，应使用此方法。Buildroot 会将源目录的内容复制到该软件包的 build 目录中。请注意，对于本地软件包，不会应用补丁程序。如果仍需要给源码打补丁，请使用 LIBFOO_POST_RSYNC_HOOKS，请参阅第 17.21.1 节。

• LIBFOO_GIT_SUBMODULES 可设置为 YES，以使用存储库中的 git 子模块创建软件压缩包。该项仅适用于 git 下载的软件包（即当 LIBFOO_SITE_METHOD=git 时）。请注意，当它们包含绑定的库时，我们尽量不要使用此类 git 子模块，在这种情况下，我们更喜欢使用软件包自身的那些库。

• LIBFOO_STRIP_COMPONENTS 指定在 tar 命令解压源码包时必须剥离的文件名前导组件（目录）的数量。大多数软件压缩包有一个名为<pkg-name>-<pkg-version>的前导组件，因此 Buildroot 将参数“--strip-components=1”传递给 tar 命令来将其删除。对于不具有前导组件的非标准软件包或者要剥离的前导组件不止一个的软件包，请将此变量设置为传递给 tar 命令的值。默认值：1。

• LIBFOO_EXCLUDES 是以空格分隔的 pattern 列表，在 tar 命令解压源码包时要将它们排除在外。该列表中的每一项都作为 tar 的--exclude 选项传递。默认情况下为空。

（译者注：LIBFOO_STRIP_COMPONENTS 和 LIBFOO_EXCLUDES 两个选项为 tar 命令参数）

• LIBFOO_DEPENDENCIES 列出了当前软件包需要编译的依赖项（以软件包名称的形式）。我们确保在配置当前软件包之前编译并安装这些依赖项。但是，对这些依赖项配置进行修改将不会强制重新构建当前软件包。类似地，HOST_LIBFOO_DEPENDENCIES 列出了当前宿主机软件包的依赖项。

• LIBFOO_EXTRACT_DEPENDENCIES 列出当前软件包需要提取（解压）的依赖项（以软件包名称的形式）。我们确保在当前软件包进行提取之前编译并安装这些依赖项。它仅由软件包基础结构在内部使用，通常不应该由软件包直接使用。

• LIBFOO_PATCH_DEPENDENCIES 列出当前软件包需要修补（即 pacth，打补丁）的依赖项（以软件包名称的形式）。我们确保在修补当前软件包之前提取并修补（不必要构建）这些依赖项。类似地，HOST_LIBFOO_PATCH_DEPENDENCIES 列出当前宿主机软件包的依赖项。这很少使用；通常，LIBFOO_DEPENDENCIES 才是您真正想要使用的。

• LIBFOO_PROVIDES 列出所有由 libfoo 实现的 virtual 软件包。参阅第 17.11 节。

• LIBFOO_INSTALL_STAGING 可设置为 YES 或 NO（默认）。如果设置为 YES，则将执行 LIBFOO_INSTALL_STAGING_CMDS 变量中的命令，以将软件包安装到 staging 目录中。

• LIBFOO_INSTALL_TARGET 可设置为 YES（默认）或 NO。如果设置为 YES，则将执行 LIBFOO_INSTALL_TARGET_CMDS 变量中的命令，以将软件包安装到 target 目录中。

• LIBFOO_INSTALL_IMAGES 可设置为 YES 或 NO（默认）。如果设置为 YES，则将执行 LIBFOO_INSTALL_IMAGES_CMDS 中的命令，以将软件包安装到 images 目录中。

• LIBFOO_CONFIG_SCRIPTS 列出$(STAGING_DIR)/usr/bin 中的文件名，这些文件需进行一些特殊的修复才能使其易于交叉编译。可以列出多个以空格分隔的文件名，它们位置都是相对于$(STAGING_DIR)/usr/bin 的。会从$(TARGET_DIR)/usr/bin 中删除 LIBFOO_CONFIG_SCRIPTS 列出的文件，因为在目标上不需要它们。

• LIBFOO_DEVICES 列出当使用静态设备表时由 Buildroot 创建的设备文件。使用 makedevs 语法。您可以在第 24 章中找到有关此语法的一些说明。此变量是可选的。

• LIBFOO_PERMISSIONS 列出在构建过程结束时要进行的权限更改。使用 makedevs 语

法。您可以在第 24 章中找到有关此语法的一些说明。此变量是可选的。

• LIBFOO_USERS 列出要为此软件包创建的用户，如果它安装了一个你想以特定用户身份运行的程序（例如，作为守护程序或 cron-job）。该语法在本质上与 makedevs 相似，在第 25 章中进行了描述。此变量是可选的。

• LIBFOO_LICENSE 定义软件包发布时所依据的一个或多个许可证。该许可证名称将出现在执行“make legal-info”后生成的 manifest 文件中。如果许可证出现在 SPDX 许可证列表中，请使用 SPDX 标识符的简称来让 manifest 文件统一。否则，请以准确简明的方式描述许可证，避免使用不明确的名称，例如“BSD”实际上会命名一个许可证系列。此变量是可选的。如果未定义，则 manifest 文件中该软件包的“license”字段将显示“unknown”。

此变量格式需要符合以下规则：

- 如果软件包的不同部分以不同的许可证发布，则需要以逗号分隔不同的许可证（例如 LIBFOO_LICENSE = GPL-2.0+, LGPL-2.1+）。如果哪个组件在哪个许可证下有明显的区别，则在括号之间注释该许可证组件（例如 LIBFOO_LICENSE = GPL-2.0+ (programs), LGPL-2.1+ (libraries)）。

- 如果某些许可证需要以启用子选项为条件，则可追加以逗号分隔的有条件的许可证（例如 FOO_LICENSE += , GPL-2.0+ (programs)）；基础结构将在内部删除逗号前的空格。

- 如果该软件包是双重许可的，则以 or 关键字隔开（例如 LIBFOO_LICENSE = AFL-2.1 or GPL-2.0+）。

• LIBFOO_LICENSE_FILES 是以空格分割的、用于发布软件包的许可证文件列表，这些文件位于软件压缩包中。运行“make legal-info”将复制所有这些文件到 legal-info 目录中。有关更多信息，请参阅第 12 章。此变量是可选的。如果未定义，则会生成一条 warning 通知您，并且 manifest 文件中该软件包的“license files”字段将显示“not saved”。

• LIBFOO_ACTUAL_SOURCE_TARBALL 仅适用于那些在 LIBFOO_SITE/LIBFOO_SOURCE 中指向的压缩包实际上并不包含源代码而包含二进制码（binary code）的软件包。这是一种非常罕见的情况，仅已知适用于已编译的外部工具链，尽管从理论上来讲它可能适用于其他软件包。在这种情况下，通常会有一个单独的压缩包来提供实际的源代码。将 LIBFOO_ACTUAL_SOURCE_TARBALL 设置为实际源代码压缩包的名称，Buildroot 将下载它，并在运行“make legal-info”收集与法律相关的材料时使用它。请注意，在常规构建期间或者运行“make source”时都不会下载此文件。

• LIBFOO_ACTUAL_SOURCE_SITE 提供实际源代码压缩包的位置。默认值为 LIBFOO_SITE，因此如果二进制压缩包和源码压缩包托管在同一目录中，则无需设置此变量。如果未设置 LIBFOO_ACTUAL_SOURCE_TARBALL，则定义 LIBFOO_ACTUAL_SOURCE_SITE 毫无意义。

• LIBFOO_REDISTRIBUTE 可设置为 YES（默认）或 NO，以指示是否允许重新分发软件包源代码。对于非开源软件包，将其设置为 NO：在收集 legal-info 时，Buildroot 将不会保存该软件包的源代码。

• LIBFOO_FLAT_STACKSIZE 定义 FLAT 二进制格式应用程序的堆栈大小。在 NOMMU 体系架构处理器上的应用程序其堆栈大小无法在运行时扩大。FLAT 二进制格式应用程序的默认堆栈大小仅为 4k 字节。如果应用程序需要占用更多堆栈，请在此处附加所需的数字。

• LIBFOO_BIN_ARCH_EXCLUDE 是以空格分隔的路径列表（相对于 target 目录），在检查软件包是否正确安装交叉编译的二进制文件时可忽略这些路径。您几乎不需设置此变量，除非软件包将其二进制 blobs 安装在默认位置之外。这些位置/lib/firmware、/usr/lib/firmware、/lib/modules、/usr/lib/modules 和/usr/share 会被自动排除。

• LIBFOO_IGNORE_CVES 是以空格分隔的 CVE 编号列表，它告诉 Buildroot 的 CVE 跟踪工具，该软件包应忽略哪些 CVE 编号。当 CVE 漏洞由软件包中的补丁程序修复时，或者由于某些原因 CVE 漏洞不影响 Buildroot 软件包时，通常使用此方法。在添加 CVE 编号之前，必须先在 Makefile 中添加注释。示例：

```
# 0001-fix-cve-2020-12345.patch
LIBFOO_IGNORE_CVES += CVE-2020-12345
# only when built with libbaz, which Buildroot doesn't support
LIBFOO_IGNORE_CVES += CVE-2020-54321
```

定义这些变量的推荐方法是使用以下语法：

```
LIBFOO_VERSION = 2.32
```

现在，下面一些变量，定义在构建过程的不同步骤中应执行哪些操作。

• LIBFOO_EXTRACT_CMDS 列出提取（解压）软件包时需执行的操作。由于压缩包是由 Buildroot 自动处理的，因此通常不需要设置该变量。但是，如果软件包使用非标准的压缩格式（例如 ZIP 或 RAR 文件），或者使用非标准组织结构的压缩包，则此变量允许用户覆盖软件包基础结构的默认行为。

• LIBFOO_CONFIGURE_CMDS 列出在编译软件包之前需执行的配置操作。

• LIBFOO_BUILD_CMDS 列出编译软件包时需执行的操作。

• HOST_LIBFOO_INSTALL_CMDS 列出安装宿主机软件包时需执行的操作。该软件包必须将其文件安装到$(HOST_DIR)给定的目录中。应该安装所有文件，包括开发文件（如 headers），因为其他软件包可能会在此软件包上进行编译。

• LIBFOO_INSTALL_TARGET_CMDS 列出将目标软件包安装到 target 目录时需执行的操作。该软件包必须将其文件安装到$(TARGET_DIR)给定的目录中。只需安装软件包在执行时所需的文件。当目标文件系统完成后，头文件、静态库和文档（documentation）将被再次移除。

• LIBFOO_INSTALL_STAGING_CMDS 列出将目标软件包安装到 staging 目录时需执行的操作。该软件包必须将其文件安装到$(STAGING_DIR)指定的目录中。应安装所有开发文件，因为编译其他软件包时可能会需要这些文件。

• LIBFOO_INSTALL_IMAGES_CMDS 列出将目标软件包安装到 images 目录时需执行的操作。该软件包必须将其文件安装到$(BINARIES_DIR)给定的目录中。只有那些不属于 TARGET_DIR 的但对启动开发板有必要的二进制镜像文件（也称为镜像）才应该放置在这里。例如，如果一个软件包有类似于内核镜像、引导加载程序或根文件系统镜像的二进制文件，则应使用该步骤。

• LIBFOO_INSTALL_INIT_SYSV、LIBFOO_INSTALL_INIT_OPENRC 和 LIBFOO_INSTALL_INIT_SYSTEMD 列出为 SystemV-like init 系统（busybox、sysvinit 等）、openrc 或 systemd units 安装 init 脚本时需执行的操作。这些命令仅在安装了相关的初始化系统后才运行（例如，如果在配置中选择 systemd 作为初始化系统，则仅运行 LIBFOO_INSTALL_INIT_SYSTEMD）。唯一的例外是，当选择 OpenRC 作为初始系统却没有设置 LIBFOO_INSTALL_INIT_OPENRC 时，在这样情况下，LIBFOO_INSTALL_INIT_SYSV 将被调用，因为 openrc 支持 sysv init 脚本。当将 systemd 用作初始化系统时，buildroot 将在 image 镜像构建的最后阶段使用 systemctl preset-all 命令自动启用所有服务。您可以添加预置文件，以防止 buildroot 自动启用特定服务。

• LIBFOO_HELP_CMDS 列出打印软件包帮助信息需执行的操作，该操作被包含在“make help”的输出中。这些命令能以任何格式打印任何内容。由于软件包很少有自定义的规则，因此很少使用。除非您真的需要打印帮助信息，否则不要使用此变量。

定义这些变量的首选方式是：

```
define LIBFOO_CONFIGURE_CMDS
        action 1
        action 2
        action 3
endef
```

在定义上面 action 时，可以使用以下变量：

- $(LIBFOO_PKGDIR)指向该软件包的目录路径，该目录包含 libfoo.mk 和 Config.in 文件。当需要安装 Buildroot 附带的文件时（例如运行时配置文件，启动画面镜像等等），此变量会很有用。
- $(@D)，指向已解压的源码目录路径。*（译者注：即 output/build/下对应软件包目录）*
- $(LIBFOO_DL_DIR)，Buildroot 为 libfoo 进行的所有下载都存储在该目录中。
- $(TARGET_CC)、$(TARGET_LD)等获取目标交叉编译实用工具
- $(TARGET_CROSS)获取交叉编译工具链前缀
- 当然，可以使用$(HOST_DIR)、$(STAGING_DIR)和$(TARGET_DIR)变量来安装软件包。

这些变量默认指向全局的 host 目录、staging 目录和 target 目录，除非使用“per-package directories”支持，这种情况下它们将指向当前软件包的 host 目录、staging 目录和 target 目录。对于这两种情况，从软件包的角度看，它们没有任何区别：它们应该仅使用 HOST_DIR、STAGING_DIR 和 TARGET_DIR。有关“per-package directories”的更多详细信息，请参阅第 8.11 节。

最后，您也可以使用 hook 钩子变量。有关更多信息，请参阅第 17.21 节。

17.6 基于 autotools 的软件包的基础结构

17.6.1 autotools-package tutorial

首先，让我们看一下如何为基于 autotools 的软件包编写一个.mk 文件，示例如下：

```
01: ################################################################################
##
02: #
03: # libfoo
04: #
05: ################################################################################
##
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = LIBFOO-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_INSTALL_STAGING = YES
11: LIBFOO_INSTALL_TARGET = NO
12: LIBFOO_CONF_OPTS = --disable-shared
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf
14：
15: $(eval $(autotools-package))
```

第 7 行，我们声明该软件包的版本。

第 8 行和第 9 行，我们声明该软件包的压缩包名称（建议使用 xz 格式的压缩包）以及该压

缩包在 Web 上的位置。Buildroot 将自动从该位置下载压缩包。

第 10 行，我们告诉 Buildroot 将该软件包安装到 staging 目录。staging 目录位于 output/staging/中，所有软件包（包括其开发文件）将安装到该目录。默认情况下，不安装软件包至 staging 目录，通常只需将库文件安装在 staging 目录：Buildroot 需要这些库的开发文件来编译依赖于它们的其他库或应用程序。同样，默认情况下，启用 staging 安装后，将使用“make install”命令将该软件包安装至此位置。

第 11 行，我们告诉 Buildroot 不要将该软件包安装到 target 目录。该目录包含了在目标机上运行的根文件系统的目录。对于纯静态库，没必要将它们安装在 target 目录中，因为它们不会在运行时使用。默认情况下，target 安装处于启用状态，几乎不需将此变量设置为 NO。同样，默认情况下，启用 target 安装后，将使用“make install”命令将该软件包安装至此位置。

第 12 行，我们告诉 Buildroot 传递一个自定义配置选项，该选项将在软件包配置和构建之前传递给./configure 脚本。

第 13 行，我们声明依赖项，以便在软件包构建开始之前构建依赖项。

最后，第 15 行，我们调用 autotools-package 宏，该宏生成构建软件包的所有 Makefile 规则。

17.6.2 autotools-package reference

autotools 软件包基础结构的主要宏是 autotools-package。它类似于 generic-package 宏。通过 host-autotools-package 宏，还可以获得目标软件包和宿主机软件包。

像 generic 基础结构一样，autotools 基础结构工作时需要在调用 autotools-package 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 autotools 基础结构中：LIBFOO_VERSION，LIBFOO_SOURCE，LIBFOO_PATCH，LIBFOO_SITE，LIBFOO_SUBDIR，LIBFOO_DEPENDENCIES，LIBFOO_INSTALL_STAGING，LIBFOO_INSTALL_TARGET。

还可以定义一些 autotools 基础结构特定的附加变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中几个。

- LIBFOO_SUBDIR，指定包含软件包配置脚本的子目录名称。例如，如果软件包主配置脚本不在解压后的目录树的根目录中，则此变量会很有用。如果 HOST_LIBFOO_SUBDIR 未指定，则默认为 LIBFOO_SUBDIR。
- LIBFOO_CONF_ENV，指定传递给配置脚本的其他环境变量。默认情况下为空。
- LIBFOO_CONF_OPTS，指定传递给配置脚本的其他配置选项。默认情况下为空。
- LIBFOO_MAKE，指定备用 make 命令。当在配置中（使用 BR2_JLEVEL）启用并行 make 时，这通常很有用，但出于某种原因，应对给定的软件包禁用此功能。默认情况下，将其设置为$(MAKE)。如果软件包不支持并行构建，则应将其设置为“LIBFOO_MAKE=$(MAKE1)”。
- LIBFOO_MAKE_ENV，指定在构建步骤中传递给 make 命令的其他环境变量。在 make 命令执行之前传递。默认情况下为空。
- LIBFOO_MAKE_OPTS，指定在构建步骤中传递给 make 命令的其他变量。在 make 命令执行之后传递。默认情况下为空。
- LIBFOO_AUTORECONF，说明是否自动重新配置软件包（即当重新运行 autoconf、automake、libtool 时，是否重新生成配置脚本和 Makefile.in 文件）。有效值为 YES 和 NO。默认情况下，该值为 NO。
- LIBFOO_AUTORECONF_ENV，如果 LIBFOO_AUTORECONF=YES，则该变量指定传

递给 autoreconf 程序的其他环境变量。在 autoreconf 命令的执行环境中传递。默认情况下为空。

• LIBFOO_AUTORECONF_OPTS，如果 LIBFOO_AUTORECONF=YES，则该变量指定传递给 autoreconf 程序的其他配置选项。默认情况下为空。

• LIBFOO_GETTEXTIZE，说明是否对该软件包进行 gettext 化（即如果该软件包使用的 gettext 版本与 Buildroot 提供的版本不同，是否需要运行 gettextize 命令）。仅在 LIBFOO_AUTORECONF=YES 时有效。有效值为 YES 和 NO。默认为 NO。*（译者注：gettext 是一个多语种国际化服务）*

• LIBFOO_GETTEXTIZE_OPTS，如果 LIBFOO_GETTEXTIZE=YES，则该变量指定传递给 gettextize 程序的其他选项。例如，如果.po 文件不在标准位置（即软件包根目录下的 po/目录），则可以使用该变量。默认情况下，该值为-f。

• LIBFOO_LIBTOOL_PATCH，说明是否应用可修复 libtool 交叉编译问题的 Buildroot 补丁程序。有效值为 YES 和 NO。默认情况下，该值为 YES。

• LIBFOO_INSTALL_STAGING_OPTS，包含用于将软件包安装到 staging 目录的 make 选项。默认情况下，该值为“DESTDIR=$(STAGING_DIR) install”，这适用于大多数 autotools 软件包。有需要的话，也可以覆盖它。

• LIBFOO_INSTALL_TARGET_OPTS，包含用于将软件包安装到 target 目录的 make 选项。默认情况下，该值为“DESTDIR=$(TARGET_DIR) install”。这适用于大多数 autotools 软件包。有需要的话，也可以覆盖它。

通过 autotools 基础结构，已经定义了构建和安装软件包所需的所有步骤，并且它们通常适用于大多数基于 autotools 的软件包。但在需要时，仍然可以自定义在任何特定步骤中需完成的操作：

• 通过添加 post-operation hook 钩子变量（在 extract，patch，configure，build 或 install 之后）。有关详细信息，请参阅第 17.21 节。

• 通过覆盖其中一个步骤。例如，即使使用了 autotools 基础结构，如果软件包.mk 文件定义了自己的 LIBFOO_CONFIGURE_CMDS 变量，那么将使用它来代替 autotools 中的默认值。但是，使用此方法应仅限于非常特殊的情况。一般情况下，请勿使用它。

17.7 基于 CMake 的软件包的基础结构

17.7.1 cmake-package tutorial

首先，让我们看一下如何为基于 CMake 的软件包编写.mk 文件，示例如下：

```
01: ################################################################################
##
02: #
03: # libfoo
04: #
05: ################################################################################
##
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = LIBFOO-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_INSTALL_STAGING = YES
```

```
11: LIBFOO_INSTALL_TARGET = NO
12: LIBFOO_CONF_OPTS = -DBUILD_DEMOS=ON
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf
14：
15: $(eval $(cmake-package)
```

第 7 行，我们声明该软件包的版本。

第 8 和 9 行，我们声明该软件包的压缩包名称（建议使用 xz 格式的压缩包）以及该压缩包在 Web 上的位置。Buildroot 将自动从该位置下载压缩包。

第 10 行，我们告诉 Buildroot 将该软件包安装到 staging 目录。staging 目录位于 output/staging/中，所有软件包（包括其开发文件）将安装到该目录。默认情况下，不安装软件包至 staging 目录，通常只需要将库文件安装在 staging 目录：Buildroot 需要这些库的开发文件来编译依赖于它们的其他库或应用程序。同样，默认情况下，启用 staging 安装后，将使用“make install”命令将该软件包安装至此位置。

第 11 行，我们告诉 Buildroot 不要将该软件包安装到 target 目录。该目录包含了在目标机上运行的根文件系统的目录。对于纯静态库，没必要将它们安装在 target 目录中，因为它们不会在运行时使用。默认情况下，target 安装处于启用状态，几乎不需将此变量设置为 NO。同样，默认情况下，启用 target 安装后，将使用“make install”命令将该软件包安装至此位置。

第 12 行，我们告诉 Buildroot 在配置该软件包时将传递自定义选项给 CMake。

第 13 行，我们声明依赖项，以便在我们软件包构建开始之前构建依赖项。

最后，第 15 行，我们调用 cmake-package 宏，该宏生成构建软件包的所有 Makefile 规则。

17.7.2 cmake-package reference

CMake 软件包基础结构的主要宏是 cmake-package。它类似于 generic-package 宏。通过 host-cmake-package 宏，还可以获得目标软件包和宿主机软件包。

像 generic 基础结构一样，CMake 基础结构工作时需要在调用 cmake-packag 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 CMake 基础结构中：LIBFOO_VERSION，LIBFOO_SOURCE，LIBFOO_PATCH，LIBFOO_SITE，LIBFOO_SUBDIR，LIBFOO_DEPENDENCIES，LIBFOO_INSTALL_STAGING，LIBFOO_INSTALL_TARGET。

还可以定义一些 CMake 基础结构特定的附加变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中几个。

- LIBFOO_SUBDIR，指定包含软件包主 CMakeLists.txt 文件的子目录名称。例如，如果软件包主 CMakeLists.txt 文件不在解压后的目录树的根目录中，则此变量会很有用。如果 HOST_LIBFOO_SUBDIR 未指定，则默认为 LIBFOO_SUBDIR。
- LIBFOO_CONF_ENV，指定传递给 CMake 的其他环境变量。默认情况下为空。
- LIBFOO_CONF_OPTS，指定传递给 CMake 的其他配置选项。默认情况下为空。一些常见的 CMake 选项是由 cmake-package 基础结构设置的*（译者注：可参阅 package/pkg-cmake.mk 文件）*。因此通常无需在软件包的*.mk 文件中设置它们，除非您想覆盖它们：
  - CMAKE_BUILD_TYPE 由 BR2_ENABLE_DEBUG 驱动；
  - CMAKE_INSTALL_PREFIX；
  - BUILD_SHARED_LIBS 由 BR2_STATIC_LIBS 驱动；
  - BUILD_DOC，BUILD_DOCS 已禁用；

- BUILD_EXAMPLE，BUILD_EXAMPLES 已禁用；
- BUILD_TEST，BUILD_TESTS，BUILD_TESTING 已禁用。

• 当软件包无法在其源码目录树中构建，但需要单独的构建目录时，则应设置“LIBFOO_SUPPORTS_IN_SOURCE_BUILD = NO”。

• LIBFOO_MAKE，指定备用 make 命令。当在配置中（使用 BR2_JLEVEL）启用并行 make 时，这通常很有用，但出于某种原因，应对给定的软件包禁用此功能。默认情况下，将其设置为$(MAKE)。如果软件包不支持并行构建，则应将其设置为“LIBFOO_MAKE=$(MAKE1)”。

• LIBFOO_MAKE_ENV，指定在构建步骤中传递给 make 命令的其他环境变量。在 make 命令执行之前传递。默认情况下为空。

• LIBFOO_MAKE_OPTS，指定在构建步骤中传递给 make 命令的其他变量。在 make 命令执行之后命令。默认情况下为空。

• LIBFOO_INSTALL_STAGING_OPTS，包含用于将软件包安装到 staging 目录的 make 选项。默认情况下，该值为“DESTDIR=$(STAGING_DIR) install”，这适用于大多数 CMake 软件包。有需要的话，也可以覆盖它。

• LIBFOO_INSTALL_TARGET_OPTS,包含用于将软件包安装到 target 目录的 make 选项。默认情况下，该值为“DESTDIR=$(TARGET_DIR) install”。这适用于大多数 CMake 软件包。有需要的话，也可以覆盖它。

通过 CMake 基础结构，已经定义了构建和安装软件包所需的所有步骤，并且它们通常适用于大多数基于 CMake 的软件包。但在需要时，仍然可以自定义在任何特定步骤中需完成的操作：

• 通过添加 post-operation hook 钩子变量（在 extract，patch，configure，build 或 install 之后）。有关详细信息，请参阅第 17.21 节。

• 通过覆盖其中一个步骤。例如，即使使用了 CMake 基础结构，如果软件包.mk 文件定义了自己的 LIBFOO_CONFIGURE_CMDS 变量，那么将使用它代替默认的 CMake 变量。但是，使用此方法应仅限于非常特殊的情况。一般情况下，请勿使用它。

17.8 Python 软件包的基础结构

此基础结构适用于使用标准 Python setuptools 机制作为构建系统的 Python 软件包，通常可通过 setup.py 脚本来识别。

17.8.1 python-package tutorial

首先，让我们看一下如何为 Python 软件包编写.mk 文件，示例如下：

```
01: ################################################################################
##
02: #
03: # python-foo
04: #
05: ################################################################################
##
06:
07: PYTHON_FOO_VERSION = 1.0
08: PYTHON_FOO_SOURCE = python-foo-$(PYTHON_FOO_VERSION).tar.xz
09: PYTHON_FOO_SITE = http://www.foosoftware.org/download
```

```
10: PYTHON_FOO_LICENSE = BSD-3-Clause
11: PYTHON_FOO_LICENSE_FILES = LICENSE
12: PYTHON_FOO_ENV = SOME_VAR=1
13: PYTHON_FOO_DEPENDENCIES = libmad
14: PYTHON_FOO_SETUP_TYPE = distutils
15:
16: $(eval $(python-package))
```

第 7 行，我们声明该软件包的版本。

第 8 和 9 行，我们声明该软件包的压缩包名称（建议使用 xz 格式的压缩包）以及该压缩包在 Web 上的位置。Buildroot 将自动从该位置下载压缩包。

第 10 行和第 11 行，我们提供该软件包的许可证相关信息（第 10 行是许可证，第 11 行是包含该许可证的文本文件）。

第 12 行，我们告诉 Buildroot 在配置该软件包时将传递自定义选项给 Python 的 setup.py 脚本。

第 13 行，我们声明依赖项，以便在我们软件包构建开始之前构建依赖项。

第 14 行，我们声明将要使用的 Python 特定构建系统。在这种情况下，将使用 distutils 作为 Python 构建系统。支持两种选择：distutils 和 setuptools。

最后，第 16 行，我们调用 python-package 宏，该宏生成构建软件包的所有 Makefile 规则。

17.8.2 python-package reference

作为一项策略，仅提供 Python 模块的软件包在 Buildroot 中应命名为 python-<something>。而其他使用了 Python 构建系统但不是 Python 模块的软件包则可以自由命名其名称（在 Buildroot 中现有示例为 scons 和 supervisor）。

只能兼容一个 Python 版本（例如：Python 2 或 Python 3）的软件包，应在其 Config.in 文件中明确指明该软件包依赖于哪个版本（对于 Python 2 为 BR2_PACKAGE_PYTHON，对于 Python 3 为 BR2_PACKAGE_PYTHON3）。与两个 Pyhon 版本都兼容的软件包则不应在其 Config.in 文件中指明依赖版本，因为 Buildroot 已经为整个“External python modules”菜单表达了这种条件。*（译者注：该菜单位于 package/Config.in 中，可通过 menuconfig 菜单中“Target packages”-> “Interpreter languages and scripting”-> “python”或者“python3”子选项“External python modules”打开）*

Python 软件包基础结构的主要宏是 python-package。它类似于 generic-package 宏。也可以使用 host-python-package 宏来创建 Python 宿主机软件包。

像 generic 基础结构一样，Python 基础结构工作时需要在调用 python-package 宏或者 host-python-package 宏之前定义多项变量。

在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 Python 基础结构中：PYTHON_FOO_VERSION，PYTHON_FOO_SOURCE，PYTHON_FOO_PATCH，PYTHON_FOO_SITE，PYTHON_FOO_SUBDIR，PYTHON_FOO_DEPENDENCIES，PYTHON_FOO_LICENSE，PYTHON_FOO_LICENSE_FILES，PYTHON_FOO_INSTALL_STAGING 等等。

请注意：

- 无需在软件包的 PYTHON_FOO_DEPENDENCIES 变量中添加 python 或 host-python，因为 Python 软件包基础结构会根据需要自动添加这些基本依赖项。
- 同样，无需为基于 setuptools 的软件包将 host-setuptools 添加到 PYTHON_FOO_DEPENDENCIES 中，因为 Python 基础结构会根据需要自动添加它。

Python 基础结构中有一个变量是必需的：

• PYTHON_FOO_SETUP_TYPE，定义该软件包使用哪个 Python 构建系统。支持的两个值是 distutils 和 setuptools。如果您不知道该软件包使用哪个构建系统，则请查看软件包源代码中的 setup.py 文件，查看它是从 distutils 模块还是 setuptools 模块导入东西。

根据软件包的需要，可以选择性地定义一些 Python 基础结构特定的附加变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中几个，或者不使用。

• PYTHON_FOO_SUBDIR，指定包含软件包主 setup.py 文件的子目录名称。例如，如果软件包主 setup.py 文件不在解压后的目录树的根目录中，则此变量会很有用。如果 HOST_PYTHON_FOO_SUBDIR 未指定，则默认为 PYTHON_FOO_SUBDIR。

• PYTHON_FOO_ENV，指定传递给 Python setup.py 脚本（用于 build 和 install 两步骤）的其他环境变量。注意，基础结构会自动传递一些标准变量，这些变量定义在 PKG_PYTHON_DISTUTILS_ENV（用于 distutils 目标软件包），HOST_PKG_PYTHON_DISTUTILS_ENV（用于 distutils 宿主机软件包），PKG_PYTHON_SETUPTOOLS_ENV（用于 setuptools 目标软件包）和 HOST_PKG_PYTHON_SETUPTOOLS_ENV （用于 setuptools 宿主机软件包）。

• PYTHON_FOO_BUILD_OPTS，指定在构建步骤中传递给 Python setup.py 脚本的其他选项。对于 distutils 目标软件包，基础结构会自动传递 PKG_PYTHON_DISTUTILS_BUILD_OPTS 选项。

• PYTHON_FOO_INSTALL_TARGET_OPTS，PYTHON_FOO_INSTALL_STAGING_OPTS，HOST_PYTHON_FOO_INSTALL_OPTS，分别指定在 target 安装步骤、staging 安装步骤或 host 安装期间传递给 Python setup.py 脚本的其他选项。注意，基础结构会自动传递一些选项，这些选项定义在 PKG_PYTHON_DISTUTILS_INSTALL_TARGET_OPTS 或 PKG_PYTHON_DISTUTILS_INSTALL_STAGING_OPTS（用于 distutils 目标软件包），HOST_PKG_PYTHON_DISTUTILS_INSTALL_OPTS（用于 distutils 宿主机软件包），PKG_PYTHON_SETUPTOOLS_INSTALL_TARGET_OPTS 或 PKG_PYTHON_SETUPTOOLS_INSTALL_STAGING_OPTS （用于 setuptools 目标软件包）以及 HOST_PKG_PYTHON_SETUPTOOLS_INSTALL_OPTS（用于 setuptools 宿主机软件包）。

• HOST_PYTHON_FOO_NEEDS_HOST_PYTHON，定义宿主机 python 解释器。此变量的用法仅限于宿主机软件包。支持的两个值是 python2 和 python3。这将确保使用正确的宿主机 python 软件包，并在构建时调用该解析器。如果某些构建步骤是重载的，则必须在命令中显式调用正确的 python 解释器。

通过 Python 基础结构，已经定义了构建和安装软件包所需的所有步骤，并且它们通常适用于大多数基于 Python 的软件包。但在需要时，仍然可以自定义在任何特定步骤中需完成的操作：

• 通过添加 post-operation hook 钩子变量（在 extract，patch，configure，build 或 install 之后）。有关详细信息，请参阅第 17.21 节。

• 通过覆盖其中一个步骤。例如，即使使用了 Python 基础结构，如果软件包.mk 文件定义了自己的 PYTHON_FOO_BUILD_CMDS 变量，那么将使用它代替默认的 Python 变量。但是，使用此方法应仅限于非常特殊的情况。一般情况下，请勿使用它。

17.8.3 从 PyPI 存储库生成 python 软件包

如果您想为从 PyPI 存储库获取的 Python 软件包创建 Buildroot 软件包，则您可能需要使用位于 utils/中的 scanpypi 工具来自动执行该过程。

您可以在这里找到现有 PyPI 软件包的列表：https://pypi.org/

scanpypi 需要您在宿主机上安装 Python 的 setuptools 软件包。

当您在 buildroot 目录的根目录下时，可执行以下操作：

```
utils/scanpypi foo bar -o package
```

如果软件包 foo 和 bar 存在于 https://pypi.org/中*（译者注：此处是使用 foo 和 bar 举例）*，则将在 package 文件夹中生成软件包 python-foo 和 python-bar。然后找到“External python modules”菜单，将您的软件包插入其中。请记住，该菜单中的条目应按字母顺序排列。*（译者注：该菜单位于 package/Config.in 中）*

请记住，由于生成器无法猜测某些事情（比如，对任何 python 核心模块的依赖，如 BR2_PACKAGE_PYTHON_ZLIB），您很可能需要手动检查软件包中是否存在任何错误。请注意，许可证和许可证文件是猜测的，必须进行检查。您还需要手动将软件包添加到 package/Config.in 文件中。

如果您的 Buildroot 软件包不在官方的 Buildroot 目录树中而是在 br2-external 目录树中，请使用-o 标志，如下所示：

```
utils/scanpypi foo bar -o other_package_dir
```

这将在 other_package_dir 目录中生成软件包 python-foo 和 python-bar，而不是在 package 目录。

选项-h 将列出可用的选项：

```
utils/scanpypi -h
```

17.8.4 python-package CFFI 后端

Python C 外部函数接口（CFFI）提供了一种方便可靠的方法，可以通过由 C 语言编写的接口声明从 Python 中调用已编译的 C 代码。依赖于此后端的 Python 软件包可以通过它们 setup.py 文件“install_requires”字段中的 cffi 依赖项来标识。

这样的软件包应该：

- 添加 python-cffi 作为运行时依赖项，以便在目标上安装已编译的 C 库封装程序。这是通过添加“select BR2_PACKAGE_PYTHON_CFFI”到软件包 Config.in 文件中来实现。

```
config BR2_PACKAGE_PYTHON_FOO
        bool "python-foo"
        select BR2_PACKAGE_PYTHON_CFFI # runtime
```

- 添加 host-python-cffi 作为构建时依赖项，以便可以交叉编译 C 封装程序。这是通过添加 host-python-cffi 到 PYTHON_FOO_DEPENDENCIES 变量中来实现。

```
################################################################################
# python-foo
#
################################################################################
…
PYTHON_FOO_DEPENDENCIES = host-python-cffi
$(eval $(python-package))
```

17.9 基于 LuaRocks 的软件包的基础结构

17.9.1 luarocks-package tutorial

首先，让我们看一下如何为基于 LuaRocks 的软件包编写.mk 文件，示例如下：

```
01: ################################################################################
```

```
##
02: #
03: # lua-foo
04: #
05: ################################################################################
##
06:
07: LUA_FOO_VERSION = 1.0.2-1
08: LUA_FOO_NAME_UPSTREAM = foo
09: LUA_FOO_DEPENDENCIES = bar
10:
11: LUA_FOO_BUILD_OPTS += BAR_INCDIR=$(STAGING_DIR)/usr/include
12: LUA_FOO_BUILD_OPTS += BAR_LIBDIR=$(STAGING_DIR)/usr/lib
13: LUA_FOO_LICENSE = luaFoo license
14: LUA_FOO_LICENSE_FILES = $(LUA_FOO_SUBDIR)/COPYING
15:
16: $(eval $(luarocks-package))
```

第 7 行，我们声明该软件包的版本（与 rockspec 中的版本相同，它是上游版本和 rockspec 修订版本的串联，并用连字符“-”分隔）。

第 8 行，我们声明该软件包在 LuaRocks 上称为“foo”。在 Buildroot 中，我们为 Lua 相关的软件包指定一个以“lua”开头的名称，因此 Buildroot 中的名称与上游名称不同。而 LUA_FOO_NAME_UPSTREAM 变量可在两个名称之间建立连接。

第 9 行，我们声明对本地库的依赖关系，以便在我们软件包构建开始之前构建依赖项。

第 11-12 行，我们告诉 Buildroot 在构建该软件包时将传递自定义选项给 LuaRocks。

第 13-14 行中，我们指定该软件包的许可条款。

最后，第 16 行，我们调用 luarocks-package 宏，该宏生成构建软件包的所有 Makefile 规则。

这些细节大部分可以从 rock 和 rockspec 中获取。因此，可以通过在 Buildroot 目录中运行命令“luarocks buildroot foo lua-foo”来生成此文件和 Config.in 文件。此命令会运行 Buildroot 特定的插件 luarocks（*译者注：位于 package/luarocks*），将自动生成 Buildroot 软件包。执行结果仍然需要手动检查并可能修改。

- package/Config.in 文件必须手动更新，以包含新生成的 Config.in 文件。

17.9.2 luarocks-package reference

LuaRocks 是 Lua 模块的部署和管理系统，支持各种 build.type：builtin，make 和 cmake。在 Buildroot 上下文中，luarocks 软件包基础结构仅支持 builtin 模式。使用 make 或 cmake 构建机制的 LuaRocks 软件包应分别使用 Buildroot 中的 generic-package 和 cmake-package 基础结构来进行打包。

LuaRocks 软件包基础结构的主要宏是 luarocks-package。像 generic-package 一样，它通过定义一些变量来提供有关该软件包的元数据信息，然后调用 luarocks-package。值得一提的是，Buildroot 不支持为宿主机构建 LuaRocks 软件包，因此未实现 host-luarocks-package 宏。（*译者注：此处为当前译本的 Buildroot 版本不支持* host-luarocks-package *宏*）

像 generic 基础结构一样，LuaRocks 基础结构工作时需要在调用 luarocks-package 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 LuaRocks 基础结构中：LUA_FOO_VERSION、LUA_FOO_SOURCE、LUA_FOO_SITE、LUA_FOO_DEPENDENCIES、LUA_FOO_LICENSE、LUA_FOO_FILES。

上述变量的其中两个由 LuaRocks 基础结构填充（用于 download 步骤）。如果您的软件包没有托管在 LuaRocks 镜像站点$(BR2_LUAROCKS_MIRROR)上，则可以覆盖它们：

- LUA_FOO_SITE，默认为$(BR2_LUAROCKS_MIRROR)；
- LUA_FOO_SOURCE，默认为$(lowercase LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION).src.rock。

还定义了一些 LuaRocks 基础结构特定的其他变量。可以在特定情况下覆盖它们。

- LUA_FOO_NAME_UPSTREAM，默认为 lua-foo，即 Buildroot 软件包名称；
- LUA_FOO_ROCKSPEC，默认为$(lowercase LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION).rockspec；
- LUA_FOO_SUBDIR，默认为$(LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION_WITHOUT_ROCKSPEC_REVISION；
- LUA_FOO_BUILD_OPTS 包含 luarocks build 调用的其他构建选项。

17.10 Perl/CPAN 软件包的基础结构

17.10.1 perl-package tutorial

首先，让我们看一下如何为 Perl/CPAN 软件包编写.mk 文件，示例如下：

```
01: ################################################################################
##
02: #
03: # perl-foo-bar
04: #
05: ################################################################################
##
06:
07: PERL_FOO__BAR_VERSION = 0.02
08: PERL_FOO_BAR_SOURCE = Foo-Bar-$(PERL_FOO_BAR_VERSION).tar.gz
09: PERL_FOO_BAR_SITE = $(BR2_CPAN_MIRROR)/authors/id/M/MO/MONGER
10： PERL_FOO_BAR_DEPENDENCIES = perl-strictures
11: PERL_FOO_BAR_LICENSE = Artistic or GPL-1.0+
12: PERL_FOO_BAR_LICENSE_FILES = LICENSE
13: PERL_FOO_BAR_DISTNAME = Foo-Bar
14:
15: $(eval $(perl-package))
```

第 7 行，我们声明该软件包的版本。

第 8 行和第 9 行，我们声明该软件包的压缩包名称以及在 CPAN 服务器上的位置。Buildroot 将自动从该位置下载压缩包。

第 10 行，我们声明依赖项，以便在我们软件包构建开始之前构建依赖项。

第 11 和 12 行，我们提供该软件包的许可证相关信息（第 10 行是许可证，第 11 行是包含该许可证的文本文件）。

第 13 行，脚本 utils/scancpan 所需的该软件包发行版名称（为了重新生成或者升级软件包文件）。

最后，第 15 行，我们调用 perl-package 宏，该宏生成构建软件包的所有 Makefile 规则。

这些数据大多数可以从 https://metacpan.org/中获取。因此，可以通过在 Buildroot 目录（或 br2-external 目录树）中运行脚本“utils/scancpan Foo-Bar”来生成此文件和 Config.in 文件。该脚本会为请求的软件包创建 Config.in 文件和 foo-bar.mk 文件，并递归创建 CPAN 指定的所有依赖项。不过您仍然需要手动编辑执行结果。特别是，应检查以下事项。

- 如果 perl 模块链接到另一个软件包（非 perl）提供的共享库，则不会自动添加此依赖项。必须手动将其添加到 PERL_FOO_BAR_DEPENDENCIES。
- package/Config.in 文件必须手动更新，以包含新生成的 Config.in 文件。作为提示，scancpan 脚本会打印出所需的 source "…"语句，并按字母顺序排序。

17.10.2 perl-package reference

作为一项策略，提供 Perl/CPAN 模块的软件包在 Buildroot 中应全部命名为 perl-<something>。

该基础结构可处理不同的 Perl 构建系统：ExtUtils-MakeMaker（EUMM），Module-Build（MB）和 Module-Build-Tiny。当软件包同时提供 Makefile.PL 和 Build.PL 两种构建系统时，默认情况下首选 Build.PL。

Perl/CPAN 软件包基础结构的主要宏是 perl-package。它类似于 generic-package 宏。通过 host-perl-package 宏，还可以获得目标软件包和宿主机软件包。

像 generic 基础结构一样，Perl/CPAN 基础结构工作时需要在调用 perl-package 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也存在于 Perl/CPAN 基础结构中：PERL_FOO_VERSION，PERL_FOO_SOURCE，PERL_FOO_PATCH，PERL_FOO_SITE，PERL_FOO_SUBDIR，PERL_FOO_DEPENDENCIES，PERL_FOO_INSTALL_TARGET。

注意，将 PERL_FOO_INSTALL_STAGING 设置为 YES 是无效的，除非定义了 PERL_FOO_INSTALL_STAGING_CMDS 变量。而 Perl 基础结构没有定义这些命令（*译者注：这个变量会定义一些操作命令*），因为 Perl 模块通常不需要安装到 staging 目录中。

还可以定义一些 Perl/CPAN 基础结构特定的附加变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中几个。

- PERL_FOO_PREFER_INSTALLER / HOST_PERL_FOO_PREFER_INSTALLER，指定首选的安装方法。可能的值为 EUMM（使用 ExtUtils-MakeMaker 的软件包是基于 Makefile.PL 的安装方法）和 MB（使用 Module-Build 的软件包是基于 Build.PL 的安装方法）。仅当该软件包提供两种安装方法时才使用此变量。
- PERL_FOO_CONF_ENV / HOST_PERL_FOO_CONF_ENV，指定传递给“perl Makefile.PL”或“perl Build.PL”的其他环境变量。默认情况下为空。
- PERL_FOO_CONF_OPTS / HOST_PERL_FOO_CONF_OPTS，指定传递给“perl Makefile.PL”或“perl Build.PL”的其他配置选项。默认情况下为空。
- PERL_FOO_BUILD_OPTS / HOST_PERL_FOO_BUILD_OPTS，指定在构建步骤中传递给“make pure_all”或“perl Build build”的其他选项。默认情况下为空。
- PERL_FOO_INSTALL_TARGET_OPTS，指定在安装步骤中传递给“make pure_install”或“perl Build install”的其他选项。默认情况下为空。
- HOST_PERL_FOO_INSTALL_OPTS，指定在安装步骤中传递给“make pure_install”或

“perl Build install”的其他选项。默认情况下为空。

17.11 virtual 软件包的基础结构

在 Buildroot 中，virtual 软件包是一种由一个或多个被称为供应者（provider）的其他程序提供其功能的软件包。virtual 软件包管理采用一种可扩展的机制，它允许用户选择已在 rootfs 中使用的供应者程序。*（译者注：此处将 provider 译为供应者，下文同）*

例如，OpenGL ES 是用于嵌入式系统上 2D 和 3D 图形的 API。对于全志科技 Sunxi 系列平台和德州仪器（TI）OMAP35xx 平台，此 API 的实现有所不同。因此，libgles 将是一个 virtual 软件包，而 sunxi-mali 和 ti-gfx 将成为供应者。

17.11.1 virtual-package tutorial

在下面示例中，我们将说明如何添加新的 virtual 软件包（something-virtual）和供应者程序（some-provider）。

首先，让我们来创建 virtual 软件包。

17.11.2 Virtual 软件包 Config.in 文件

virtual 软件包“something-virtual”的 Config.in 文件应包含：

```
01: config BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
02:         bool
03:
04: config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
05:         depends on BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
06:         string
```

在此文件中，我们声明两个选项，BR2_PACKAGE_HAS_SOMETHING_VIRTUAL 和 BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL，供应者程序将使用它们的值。

17.11.3 Virtual 软件包.mk 文件

Virtual 软件包的.mk 文件应该只包含 virtual-package 宏：

```
01: ################################################################################
##
02: #
03: # something-virtual
04: #
05: ################################################################################
##
06:
07: $(eval $(virtual-package))
```

通过 host-virtual-package 宏，还可以获得目标软件包和宿主机软件包。

17.11.4 Provider Config.in 文件

当添加某个软件包作为供应者程序时，仅需对 Config.in 文件进行一些修改。

给 something-virtual 软件包提供功能的 some-provider 供应者程序，它的 Config.in 文件应包含：

```
01: config BR2_PACKAGE_SOME_PROVIDER
02:     bool "some-provider"
03:     select BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
04:     help
05:       This is a comment that explains what some-provider is.
06:
07:       http://foosoftware.org/some-provider/
08:
09: if BR2_PACKAGE_SOME_PROVIDER
10: config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
11:     default "some-provider"
12: endif
```

第 3 行，我们选择 BR2_PACKAGE_HAS_SOMETHING_VIRTUAL，和第 11 行，我们设置 BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL 的值为供应者程序的名称，但仅限于已“select”的情况。

17.11.5 Provider .mk 文件

.mk 文件还应该声明一个附加变量 SOME_PROVIDER_PROVIDES，以包含所有 virtual 软件包的名称：

```
01: SOME_PROVIDER_PROVIDES = something-virtual
```

当然，不要忘记为供应者程序添加适当的构建和运行时依赖项！

17.11.6 关于依赖于 virtual 软件包的注意事项

当添加一个需要由 virtual 软件包提供一定功能的软件包时，必须使用“depends on BR2_PACKAGE_HAS_FEATURE”，如下所示：

```
config BR2_PACKAGE_HAS_FEATURE
    bool

config BR2_PACKAGE_FOO
    bool "foo"
    depends on BR2_PACKAGE_HAS_FEATURE
```

17.11.7 关于依赖于特定 provider 的注意事项

如果您的软件包确实需要特定的 provider 程序，则您的软件包必须“depends on”该 provider，而不能“select”某个 provider。

下面示例中，有两个 provider 提供一个“FEATURE”：

```
config BR2_PACKAGE_HAS_FEATURE
    bool

config BR2_PACKAGE_FOO
    bool "foo"
    select BR2_PACKAGE_HAS_FEATURE
```

```
config BR2_PACKAGE_BAR
    bool "bar"
    select BR2_PACKAGE_HAS_FEATURE
```

而您添加的软件包仅需 foo 提供功能，不需要 bar 提供。

如果您要使用“select BR2_PACKAGE_FOO”，则用户仍然可以在 menuconfig 菜单中选择 BR2_PACKAGE_BAR。这就会造成配置上的不一致，即同一 FEATURE 的两个供应者 provider 会被同时启用，一个是由用户明确设置的，另一个是由您“select”隐含设置的。

相反，您必须使用“depends on BR2_PACKAGE_FOO”，这样才可以避免任何隐含的配置不一致。

17.12 使用 kconfig 作为配置文件的软件包的基础结构

软件包处理用户指定配置的一种流行方式是使用 kconfig。其中，Linux 内核、Busybox 和 Buildroot 本身都使用它。使用 kconfig 的两个众所周知的体现是存在一个.config 文件和一个 menuconfig 目标。

Buildroot 为使用 kconfig 进行配置的软件包提供了一个基础结构。此基础结构提供了必要的处理逻辑，以将软件包的 menuconfig 目标展现为 Buildroot 中的 foo-menuconfig，并以正确的方式处理该软件包的配置文件。*（译者注：此处是以 foo 软件包为例）*

kconfig-package 基础结构是基于 generic-package 基础结构的。generic-package 基础结构中支持的所有变量在 kconfig-package 基础结构中也同样支持。有关详细信息，请参阅第 17.5.2 节。

为了让某个 Buildroot 软件包使用 kconfig-package 基础结构，除了 generic-package 基础结构所需的变量之外，该软件包.mk 文件中最低要求需包含以下行：

```
FOO_KCONFIG_FILE = reference-to-source-configuration-file
$(eval $(kconfig-package))
```

此代码片段将创建以下 make 目标：

- foo-menuconfig，调用软件包的 menuconfig 目标 *（译者注：即该软件包的配置菜单）*
- foo-update-config，将软件包当前配置复制回原本的配置文件，即更新源配置文件。设置片段文件 fragment file 后，将无法使用该目标。
- foo-update-defconfig，将软件包当前配置复制回原本的配置文件。此配置文件仅列出与默认值不同的选项。设置片段文件 fragment file 后，将无法使用该目标。
- foo-diff-config，它输出当前配置与此软件包在 Buildroot 中定义的 kconfig 配置之间的差异。例如，该输出可用于识别必须传播到配置片段的配置更改。

并且该代码片段会确保在适当的时候将源配置文件复制到构建目录。

有两个选项可指定要使用的配置文件，即 FOO_KCONFIG_FILE（如上述所示）或 FOO_KCONFIG_DEFCONFIG。必须提供其中一个，但不能同时提供：

- FOO_KCONFIG_FILE，指定一个 defconfig 文件或完整配置文件的路径来配置软件包。
- FOO_KCONFIG_DEFCONFIG，指定要调用的 defconfig 文件 make 规则来配置软件包。

除了这些最低要求的行之外，还可以设置几个可选变量来满足软件包的需求：

- FOO_KCONFIG_EDITORS：以空格分隔的所支持的 kconfig 编辑器列表，例如“menuconfig xconfig”。默认情况下为 menuconfig。
- FOO_KCONFIG_FRAGMENT_FILES：以空格分隔的配置片段文件列表，这些配置片段将合并到主配置文件中。配置片段文件通常在需要与上游(def)config 文件保持同步并进行较小范围修改时使用。
- FOO_KCONFIG_OPTS：指定调用 kconfig 编辑器时需要传递的其他选项。例如，它可能

需要包含$(FOO_MAKE_OPTS)。默认情况下为空。

• FOO_KCONFIG_FIXUP_CMDS：指定在复制配置文件或运行 kconfig 编辑器后需要修复的 Shell 命令列表。例如，可能需要这些命令来确保软件包的配置与 Buildroot 的其他配置一致。默认情况下为空。*（译者注：可参考 package/busybox/busybox.mk 中的用法）*

• FOO_KCONFIG_DOTCONFIG：指定.config 文件的路径（带有文件名），该路径是相对于软件包的源目录树。默认值为.config，这非常适合所有使用从 Linux 内核继承来的标准 kconfig 基础结构的软件包；有些软件包会使用派生 kconfig，它们位于不同的位置。

• FOO_KCONFIG_DEPENDENCIES：指定在解析此软件包 kconfig 之前需要构建的软件包列表（很可能大多数是宿主机软件包）。很少使用。默认情况下为空。

17.13 基于 rebar 的软件包的基础结构

17.13.1 rebar-package tutorial

首先，让我们看一下如何为基于 rebar 的软件包编写.mk 文件，示例如下：

```
01: ################################################################################
##
02: #
03: # erlang-foobar
04: #
05: ################################################################################
##
06:
07: ERLANG_FOOBAR_VERSION = 1.0
08: ERLANG_FOOBAR_SOURCE = erlang-foobar-$(ERLANG_FOOBAR_VERSION).tar.xz
09: ERLANG_FOOBAR_SITE = http://www.foosoftware.org/download
10: ERLANG_FOOBAR_DEPENDENCIES = host-libaaa libbbb
11：
12: $(eval $(rebar-package))
```

第 7 行，我们声明该软件包的版本。

第 8 和 9 行，我们声明该软件包的压缩包名称（建议使用 xz 格式的压缩包）以及该压缩包在 Web 上的位置。Buildroot 将自动从该位置下载压缩包。

第 10 行，我们声明依赖项，以便在我们软件包构建开始之前构建依赖项。

最后，第 12 行，我们调用 rebar-package 宏，该宏生成构建软件包的所有 Makefile 规则。

17.13.2 rebar-package reference

rebar 软件包基础结构的主要宏是 rebar-package。它类似于 generic-package 宏。通过 host-rebar-package 宏，还可以获得宿主机软件包。*（译者注：rebar 是一种 Erlang 构建工具）*

像 generic 基础结构一样，rebar 基础结构工作时需要在调用 rebar-package 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 rebar 基础结构中：ERLANG_FOOBAR_VERSION，ERLANG_FOOBAR_SOURCE，ERLANG_FOOBAR_PATCH，ERLANG_FOOBAR_SITE，ERLANG_FOOBAR_SUBDIR，ERLANG_FOOBAR_DEPENDENCIES，ERLANG_FOOBAR_INSTALL_STAGING，ERLANG_FOOBAR_INSTALL_TARGET，

ERLANG_FOOBAR_LICENSE 和 ERLANG_FOOBAR_LICENSE_FILES。

还可以定义一些 rebar 基础结构特定的附加变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中几个。

• ERLANG_FOOBAR_USE_AUTOCONF，指定软件包在 configuration 步骤中使用 autoconf。当软件包设置此变量为 YES 时，将使用 autotools 基础结构。

注意，您也可以使用 autotools 基础结构中的一些变量：ERLANG_FOOBAR_CONF_ENV，ERLANG_FOOBAR_CONF_OPTS，ERLANG_FOOBAR_AUTORECONF，ERLANG_FOOBAR_AUTORECONF_ENV 和 ERLANG_FOOBAR_AUTORECONF_OPTS。

• ERLANG_FOOBAR_USE_BUNDLED_REBAR，指定该软件包具有绑定版本的 rebar，并且应使用此版本。有效值为 YES 或 NO（默认值）。

注意，如果该软件包捆绑了一个 rebar 实用程序，但它可以使用 Buildroot 提供的通用实用程序，则只需设置为“NO”（即不要指定此变量）。仅在强制使用此软件包中捆绑的 rebar 实用程序时才需要设置。

• ERLANG_FOOBAR_REBAR_ENV，指定传递给 rebar 实用程序的其他环境变量。

• ERLANG_FOOBAR_KEEP_DEPENDENCIES，指定保留 rebar.config 文件中描述的依赖项。有效值为 YES 或 NO（默认值）。除非将此变量设置为 YES，否则 rebar 基础结构会在 post-patch hook 钩子变量中删除此类依赖项，以确保 rebar 不会下载或编译它们。*（译者注：可参考 package/pkg-rebar.mk 中用法）*

通过 rebar 基础结构，已经定义了构建和安装软件包所需的所有步骤，并且它们通常适用于大多数基于 rebar 的软件包。但在需要时，仍然可以自定义在任何特定步骤中需完成的操作：

• 通过添加 post-operation hook 钩子变量（在 extract，patch，configure，build 或 install 之后）。有关详细信息，请参阅第 17.21 节。

• 通过覆盖其中一个步骤。例如，即使使用了 rebar 基础结构，如果软件包.mk 文件定义了自己的 ERLANG_FOOBAR_BUILD_CMDS 变量，那么将使用它代替默认的 rebar 变量。但是，使用此方法应仅限于非常特殊的情况。一般情况下，请勿使用它。

17.14 基于 Waf 的软件包的基础结构

17.14.1 waf-package tutorial

首先，让我们看一下如何为基于 Waf 的软件包编写.mk 文件，示例如下：

```
01: ################################################################################
##
02: #
03: # libfoo
04: #
05: ################################################################################
##
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_CONF_OPTS = --enable-bar --disable-baz
11: LIBFOO_DEPENDENCIES = bar
```

```
12:
13: $(eval $(waf-package))
```

第 7 行，我们声明该软件包的版本。

第 8 和 9 行，我们声明该软件包的压缩包名称（建议使用 xz 格式的压缩包）以及该压缩包在 Web 上的位置。Buildroot 将自动从该位置下载压缩包。

第 10 行，我们告诉 Buildroot 为 libfoo 启用哪些选项。

第 11 行，我们告诉 Buildroot libfoo 的依赖项。

最后，第 13 行，我们调用 waf-package 宏，该宏生成构建软件包的所有 Makefile 规则。

17.14.2 waf-package reference

Waf 软件包基础结构的主要宏是 waf-package。它类似于 generic-package 宏。

像 generic 基础结构一样，Waf 基础结构工作时需要在调用 waf-package 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 Waf 基础结构中：LIBFOO_VERSION，LIBFOO_SOURCE，LIBFOO_PATCH，LIBFOO_SITE，LIBFOO_SUBDIR，LIBFOO_DEPENDENCIES，LIBFOO_INSTALL_STAGING，LIBFOO_INSTALL_TARGET。

还可以定义一些 Waf 基础结构特定的附加变量。

- LIBFOO_SUBDIR，指定包含软件包主 wscript 脚本的子目录名称。例如，如果软件包主 wscript 脚本不在解压后的目录树的根目录中，则此变量会很有用。如果 HOST_LIBFOO_SUBDIR 未指定，则默认为 LIBFOO_SUBDIR。
- LIBFOO_NEEDS_EXTERNAL_WAF，可设置为 YES 或 NO，以告诉 Buildroot 是否使用捆绑的 WAF 可执行文件。如果设置为 NO（默认值），则 Buildroot 将使用软件包源码目录树中提供的 waf 可执行文件；如果设置为 YES，则 Buildroot 将下载并安装 waf 作为宿主机工具，并使用它来构建软件包。
- LIBFOO_WAF_OPTS，指定在软件包 build 过程的每个步骤中（configure，build 和 installation）传递给 waf 脚本的其他选项。默认情况下为空。
- LIBFOO_CONF_OPTS，指定在 configure 步骤期间传递给 waf 脚本的其他选项。默认情况下为空。
- LIBFOO_BUILD_OPTS，指定在 build 步骤期间传递给 waf 脚本的其他选项。默认情况下为空。
- LIBFOO_INSTALL_STAGING_OPTS，指定在 staging 安装步骤期间传递给 waf 脚本的其他选项。默认情况下为空。
- LIBFOO_INSTALL_TARGET_OPTS，指定在 target 安装步骤期间传递给 waf 脚本的其他选项。默认情况下为空。

17.15 基于 Meson 的软件包的基础结构

17.15.1 meson-package tutorial

Meson 是一个开源的构建系统，旨在实现以极快的速度进行构建，并且更重要的是尽可能地与用户友好。它使用 Ninja 作为辅助工具，来执行实际的构建操作。

让我们看一下如何为基于 Meson 的软件包编写.mk 文件，示例如下：

```
01: ################################################################################
```

```
##
02: #
03: # foo
04: #
05: ################################################################################
##
06:
07: FOO_VERSION = 1.0
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-3.0+
11: FOO_LICENSE_FILES = COPYING
12: FOO_INSTALL_STAGING = YES
13:
14: FOO_DEPENDENCIES = host-pkgconf bar
15:
16: ifeq ($(BR2_PACKAGE_BAZ),y)
17: FOO_CONF_OPTS += -Dbaz=true
18: FOO_DEPENDENCIES += baz
19: else
20: FOO_CONF_OPTS += -Dbaz=false
21: endif
22:
23: $(eval $(meson-package))
```

Makefile 从定义软件包声明的标准变量开始（第 7 至 11 行）。

第 23 行，我们调用 meson-package 宏，该宏生成构建软件包的所有 Makefile 规则。

在该示例中，第 14 行 FOO_DEPENDENCIES 变量将 host-pkgconf 和 bar 声明为依赖项，因为 foo 的 Meson 构建文件使用 pkg-config 来确定软件包 bar 的编译标志和库。

注意，不必在软件包的 FOO_DEPENDENCIES 变量中添加 host-meson，因为 Meson 软件包基础结构会根据需要自动添加此基本依赖项。

如果选择了“baz”软件包，则通过第 17 行向 FOO_CONF_OPTS 变量中添加-Dbaz=true 选项来激活“foo”中对“baz”功能的支持，如同在“foo”源码目录树中的 meson_options.txt 文件中所指定的那样。“baz”软件包也添加到 FOO_DEPENDENCIE 变量中。请注意，如果未选择“baz”软件包，则在第 20 行显式禁用“baz”功能。

总结来说，要添加一个新的基于 meson 的软件包，可以逐个复制 Makefile 中的示例，然后对其进行编辑，将所有出现的“FOO”替换为新软件包的大写名称，并更新其标准变量的值。

17.15.2 meson-package reference

Meson 软件包基础结构的主要宏是 meson-package。它类似于 generic-package 宏。通过 host-meson-package 宏，还可以获得目标软件包和宿主机软件包。

像 generic 基础结构一样，Meson 基础结构工作时需要在调用 meson-packag 宏之前定义多项变量。

首先，在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 Meson 基础结

构中：FOO_VERSION，FOO_SOURCE，FOO_PATCH，FOO_SITE，FOO_SUBDIR，FOO_DEPENDENCIES，FOO_INSTALL_STAGING，FOO_INSTALL_TARGET。

还可以定义一些 Meson 基础结构特定的附加变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中几个。

- FOO_SUBDIR，指定包含软件包主 meson.build 文件的子目录名称。例如，如果主 meson.build 文件不在解压后的目录树的根目录中，则此变量会很有用。如果 HOST_FOO_SUBDIR 未指定，则默认为 FOO_SUBDIR。
- FOO_CONF_ENV，指定在 configuration 步骤中传递给 meson 的其他环境变量。默认情况下为空。
- FOO_CONF_OPTS，指定在 configuration 步骤中传递给 meson 的其他选项。默认情况下为空。
- FOO_CFLAGS，指定要添加到软件包特定的 cross-compile.conf 文件中 c_args 属性的编译器参数。默认情况下，值为 TARGET_CFLAGS。
- FOO_CXXFLAGS，指定要添加到软件包特定的 cross-compile.conf 文件中 cpp_args 属性的编译器参数。默认情况下，值为 TARGET_CXXFLAGS。
- FOO_LDFLAGS，指定要添加到软件包特定的 cross-compile.conf 文件中 c_link_args 和 cpp_link_args 属性的编译器参数。默认情况下，值为 TARGET_LDFLAGS。
- FOO_MESON_EXTRA_BINARIES，指定要添加到 meson cross-compilation.conf 配置文件中[binaries]部分的、以空格分隔的程序列表。格式为 program-name='/path/to/program'，其中在“=”号周围没有空格，并且在单引号之间包含目标程序的路径。默认情况下为空。请注意，Buildroot 已经为 c、cpp、ar、strip 和 pkgconfig 设置了正确的值。
- FOO_NINJA_ENV，指定传递给负责执行构建操作的 meson 辅助工具 ninja 的其他环境变量。默认情况下为空。
- FOO_NINJA_OPTS，指定所需构建的、以空格分隔的 target 列表。默认情况下为空，来构建默认 target。

17.16 集成基于 Cargo 的软件包

Cargo 是 Rust 编程语言的软件包管理器。它允许用户构建使用 Rust 编写的程序或库，但是它也下载并管理它们的依赖项，以确保可重复性地构建。Cargo 软件包被称为“crates”。

17.16.1 基于 Cargo 的软件包 Config.in 文件

基于 Cargo 的软件包 foo 的 Config.in 文件应包含：

```
01: config BR2_PACKAGE_FOO
02:     bool "foo"
03:     depends on BR2_PACKAGE_HOST_RUSTC_TARGET_ARCH_SUPPORTS
04:     select BR2_PACKAGE_HOST_CARGO
05:     help
06:       This is a comment that explains what foo is.
07:
08:       http://foosoftware.org/foo/
```

17.16.2 基于 Cargo 的软件包.mk 文件

Buildroot 尚未为基于 Cargo 的软件包提供专用的软件包基础结构。因此，我们将说明如何

为此类软件包编写.mk 文件。让我们从一个示例开始：

```
01: ################################################################################
##
02: #
03: # foo
04: #
05: ################################################################################
##
06:
07: FOO_VERSION = 1.0
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-3.0+
11: FOO_LICENSE_FILES = COPYING
12:
13: FOO_DEPENDENCIES = host-cargo
14:
15: FOO_CARGO_ENV = CARGO_HOME=$(HOST_DIR)/share/cargo
16:
17: FOO_BIN_DIR = target/$(RUSTC_TARGET_NAME)/$(FOO_CARGO_MODE)
18:
19: FOO_CARGO_OPTS = \
20:         $(if $(BR2_ENABLE_DEBUG),,--release) \
21:         --target=$(RUSTC_TARGET_NAME) \
22:         --manifest-path=$(@D)/Cargo.toml
23:
24: define FOO_BUILD_CMDS
25:         $(TARGET_MAKE_ENV) $(FOO_CARGO_ENV) \
26:                 cargo build $(FOO_CARGO_OPTS)
27: endef
28:
29: define FOO_INSTALL_TARGET_CMDS
30:         $(INSTALL) -D -m 0755 $(@D)/$(FOO_BIN_DIR)/foo \
31:                 $(TARGET_DIR)/usr/bin/foo
32: endef
33:
34: $(eval $(generic-package))
```

Makefile 从定义软件包声明的标准变量开始（第 7 至 11 行）。

如第 34 行所示，它是基于 generic-package 基础结构。因此，它定义了此特定基础结构所需的变量，而调用 Cargo 的位置在：

- FOO_BUILD_CMDS：调用 Cargo 来执行构建操作。通过 FOO_CONF_OPTS 变量来传递该软件包交叉编译时所需的配置选项。
- FOO_INSTALL_TARGET_CMDS：将生成的二进制可执行文件安装在目标上。

为了让 Cargo 能用于构建，FOO_DEPENDENCIES 变量需要包含 host-cargo。

总结来说，要添加一个新的基于 Cargo 的软件包，可以逐个复制 Makefile 中的示例，然后对其进行编辑，将所有出现的“FOO”替换为新软件包的大写名称，并更新其标准变量的值。

17.16.3 关于依赖项管理

Crate 可以依赖于其 Cargo.toml 文件列出的 crates.io 或 git 仓库中的其他库。在开始构建之前，Cargo 通常会自动下载它们。此步骤也可以通过“cargo fetch”命令独立执行。

Cargo 维护了一个注册表索引和 crates 的 git checkout 的本地缓存，由$CARGO_HOME 给出位置。如第 15 行的软件包 Makefile 示例所示，此环境变量设置为$(HOST_DIR)/share/cargo。

执行脱机构建时，此依赖项下载机制不方便，因为 Cargo 将无法获取依赖项。在这种情况下，建议使用“cargo vendor”生成依赖项的压缩包，并将其添加到 FOO_EXTRA_DOWNLOADS 变量中。

17.17 Go 软件包的基础结构

此基础结构适用于使用标准构建系统并使用捆绑依赖项的 Go 软件包。

17.17.1 golang-package tutorial

首先，让我们看一下如何为 go 软件包编写.mk 文件，示例如下：

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: FOO_VERSION = 1.0
08: FOO_SITE = $(call github,bar,foo,$(FOO_VERSION))
09: FOO_LICENSE = BSD-3-Clause
10: FOO_LICENSE_FILES = LICENSE
11:
12: $(eval $(golang-package))
```

第 7 行，我们声明该软件包的版本。

第 8 行，我们声明该软件包的上游位置，这里是从 Github 上获取的，因为有大量的 Go 软件包托管在 Github 上。

第 9 行和第 10 行，我们提供有关该软件包的详细许可信息。

最后，第 12 行，我们调用 golang-package 宏，该宏生成构建软件包的所有 Makefile 规则。

17.17.2 golang-package reference

在其 Config.in 文件中，使用 golang-package 基础结构的软件包应添加“depends on BR2_PACKAGE_HOST_GO_TARGET_ARCH_SUPPORTS”，因为 Buildroot 会自动向此类软件包添加 host-go 依赖项。如果您的软件包需要 CGO 支持，则必须添加对 BR2_PACKAGE_HOST_GO_TARGET_CGO_LINKING_SUPPORTS 的依赖。

Go 软件包基础结构的主要宏是 golang-package。它类似于 generic-package 宏。通过 host-golang-package 宏，还可以构建宿主机软件包。由 host-golang-package 宏构建的宿主机软件包应取决于 BR2_PACKAGE_HOST_GO_HOST_ARCH_SUPPORTS。

像 generic 基础结构一样，Go 基础结构工作时需要在调用 go-packag 宏之前定义多项变量。

在 generic 基础结构中存在的所有软件包元数据信息变量也都存在于 Go 基础结构中：FOO_VERSION，FOO_SOURCE，FOO_PATCH，FOO_SITE，FOO_SUBDIR，FOO_DEPENDENCIES，FOO_LICENSE，FOO_LICENSE_FILES，FOO_INSTALL_STAGING 等。

请注意，不必在软件包的 FOO_DEPENDENCIES 变量中添加 host-go，因为 Go 软件包基础结构会根据需要自动添加此基本依赖项。

根据软件包的需要，可以选择性地定义一些 Go 基础结构特定的其他变量。这些变量中的多数仅在非常特定的情况下才有用，因此典型的软件包仅使用其中的少数几个，或者不使用。

- 如果您的软件包需要编译自定义的 GOPATH，则可以使用 FOO_WORKSPACE 变量。GOPATH 将会变为<package-srcdir>/<FOO_WORKSPACE>。如果 FOO_WORKSPACE 未指定，则默认为_gopath。
- FOO_SRC_SUBDIR 是编译源代码的子目录，它相对于 GOPATH。一个示例值是 github.com/bar/foo。如果 FOO_SRC_SUBDIR 未指定，则默认为从 FOO_SITE 变量推断出的值。
- FOO_LDFLAGS 和 FOO_TAGS 用于将 LDFLAGS 或 TAGS 分别传递给“go build”命令。
- FOO_BUILD_TARGETS 用于传递应构建的 target 列表。如果 FOO_BUILD_TARGETS 未指定，则默认为“.”。这里有两种情况：
  - FOO_BUILD_TARGETS 为“.”。在这种情况下，我们假设该软件包只会生成一个二进制文件，并且默认情况下，我们将其命名为软件包的名称。如果不合适，则可以使用 FOO_BIN_NAME 覆盖生成的二进制文件的名称。
  - FOO_BUILD_TARGETS 不是“.”。在这种情况下，我们将遍历它的值以构建每个目标，并为每个目标生成一个二进制文件，此二进制文件是该目标的非目录组件。例如，如果 FOO_BUILD_TARGETS=cmd/docker cmd/dockerd，则生成的二进制文件是 docker 和 dockerd。
- FOO_INSTALL_BINS 用于传递应安装在目标上/usr/bin 中的二进制文件列表。如果 FOO_INSTALL_BINS 未指定，则默认为软件包的小写名称。

通过 Go 基础结构，已经定义了构建和安装软件包所需的所有步骤，并且它们通常适用于大多数基于 Go 的软件包。但在需要时，仍然可以自定义在任何特定步骤中需完成的操作：

- 通过添加 post-operation hook 钩子变量（在 extract，patch，configure，build 或 install 之后）。有关详细信息，请参阅第 17.21 节。
- 通过覆盖其中一个步骤。例如，即使使用了 Go 基础结构，如果软件包.mk 文件定义了自己的 FOO_BUILD_CMDS 变量，那么将使用它来代替 Go 中的默认值。但是，使用此方法应仅限于非常特殊的情况。一般情况下，请勿使用它。

17.18 可构建内核模块的软件包的基础结构

Buildroot 提供了一个辅助基础结构，让编写构建和安装 Linux 内核模块的软件包变得容易。一些软件包仅包含内核模块，另一些软件包除内核模块之外还包含程序集和库。Buildroot 的辅助基础结构都支持这两种情况。

17.18.1 kernel-module tutorial

让我们从一个示例开始，说明如何准备一个仅构建内核模块的简单软件包，不包含其他组

件：

```
01: ################################################################################
##
02: #
03: # foo
04: #
05: ################################################################################
##
06:
07: FOO_VERSION = 1.2.3
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: $(eval $(kernel-module))
14: $(eval $(generic-package))
```

第 7-11 行定义通用的元数据变量，以指定该软件包版本、压缩包名称、能够找到软件包源的远程 URI 和许可信息。

第 13 行，我们调用 kernel-module 辅助基础结构，它将生成所有适当的 Makefile 规则和变量以构建该内核模块。

最后，第 14 行，我们调用了 generic-package 基础结构。

对“linux”的依赖关系会自动添加，因此不需要在 FOO_DEPENDENCIES 中指定它。

您可能已经注意到，与其他软件包基础结构不同，我们显式调用了另一个基础结构。这允许软件包构建内核模块，但如果需要，还可以使用任何其他软件包基础结构来构建普通的用户空间组件（库、可执行文件...）。仅仅使用 kernel-module 基础结构是不够的；必须使用其他软件包基础结构。

让我们看一个更复杂的例子：

```
01: ################################################################################
##
02: #
03: # foo
04: #
05: ################################################################################
##
06:
07: FOO_VERSION = 1.2.3
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: FOO_MODULE_SUBDIRS = driver/base
```

```
14: FOO_MODULE_MAKE_OPTS = KVERSION=$(LINUX_VERSION_PROBED)
15:
16: ifeq ($(BR2_PACKAGE_LIBBAR),y)
17: FOO_DEPENDENCIES = libbar
18: FOO_CONF_OPTS = --enable-bar
19: FOO_MODULE_SUBDIRS += driver/bar
20: else
21: FOO_CONF_OPTS = --disable-bar
22: endif
23:
24: $(eval $(kernel-module))
26: $(eval $(autotools-package))
```

在这里，我们可以看到有一个基于 autotools 的软件包，该软件包会构建位于子目录 driver/base 中的内核模块，如果启用了 libbar，则还构建位于子目录 driver/bar 中的内核模块；并定义了变量 KVERSION，在构建模块时将其传递给 Linux 构建系统。

17.18.2 kernel-module reference

内核模块基础结构的主要宏是 kernel-module。与其他软件包基础结构不同，它不是独立的，需要在它后面调用其他*-package 宏。

kernel-module 宏定义了用于构建内核模块的 post-build 和 post-target-install hook 钩子变量。如果软件包的.mk 文件需要访问已构建的内核模块，则应在 post-build hook 变量中进行访问，该变量需要在调用 kernel-module 后进行注册。同样，如果软件包的.mk 文件需要访问安装后的内核模块，则应在 post-install hook 变量中进行访问，该变量需要在调用 kernel-module 后进行注册。示例如下：

```
$(eval $(kernel-module))

define FOO_DO_STUFF_WITH_KERNEL_MODULE
    # Do something with it...
endef
FOO_POST_BUILD_HOOKS += FOO_DO_STUFF_WITH_KERNEL_MODULE

$(eval $(generic-package))
```

最后，与其他基础结构不同，它没有用于构建宿主机内核模块的 host-kernel-module 宏。

可以选择定义以下附加变量，以进一步配置内核模块的构建：

- FOO_MODULE_SUBDIR，指定内核模块源码所在的一个或多个子目录位置（相对于软件包源码顶层目录）。如果为空或未设置，则视为内核模块源码位于软件包源码的顶层目录。
- FOO_MODULE_MAKE_OPTS，指定传递给 Linux 构建系统的其他变量。

您还可以引用（但不能设置！）这些**内核变量**：

- LINUX_DIR，指定提取和构建 Linux 内核的路径。
- LINUX_VERSION，指定由用户配置的内核版本字符串。
- LINUX_VERSION_PROBED，指定内核的真实版本字符串，可通过运行命令“make -C $(LINUX_DIR) kernelrelease”来获取。
- KERNEL_ARCH，指定当前架构的名称，例如 arm、mip…

17.19 asciidoc 文档的基础结构

您目前正在阅读的 Buildroot 手册是完全使用 AsciiDoc 标记语法编写的。该手册可渲染成多种格式：

- html
- split-html
- pdf
- epub
- txt

尽管 Buildroot 仅包含一个使用 AsciiDoc 语法编写的文档，但对于软件包，存在一种使用 AsciiDoc 语法来渲染文档的基础结构。

同样对于软件包，还可以从 br2-external 目录树中获得 AsciiDoc 基础结构。这将使 br2-external 目录树的文档与 Buildroot 文档相匹配，因为它将被渲染为相同的格式并使用相同的布局和主题。

17.19.1 asciidoc-document tutorial

由于软件包基础结构带有-package 后缀，而文档基础结构带有-document 后缀。因此，AsciiDoc 基础结构被命名为 asciidoc-document。

以下是渲染一个简单的 AsciiDoc 文档的示例：

```
01: ################################################################################
##
02: #
03: # foo-document
04: #
05: ################################################################################
##
06:
07: FOO_SOURCES = $(sort $(wildcard $(pkgdir)/*))
08: $(eval $(call asciidoc-document))
```

第 7 行，Makefile 声明文档的来源。目前 Buildroot 所预期的文档来源仅是本地的；Buildroot 不会尝试下载任何内容以渲染文档。因此，您必须指出来源。通常，上面的字符串对于没有子目录结构的文档就足够了。

第 8 行，我们调用 asciidoc-document 宏，该宏生成渲染文档所需的所有 Makefile 代码。

17.19.2 asciidoc-document reference

以下是变量列表，可在.mk 文件中设置，以提供元数据信息（假设文档名称为 foo）：

- FOO_SOURCES，必填项，定义文档的源文件。
- FOO_RESOURCES，可选项，指定以空格分隔的路径列表，指向包含所谓资源（如 CSS 或 images）的一个或多个目录。默认情况下为空。
- FOO_DEPENDENCIES，可选项，指定在构建此文档之前必须构建的软件包列表（很可能是宿主机软件包）。

还有设置其他 hook 变量（有关 hook 变量的信息，请参阅第 17.21 节），以定义在各个步骤中要执行的附加操作：

• FOO_POST_RSYNC_HOOKS，指定在 Buildroot 复制源后要运行的其他命令。例如，它可用于从目录树中提取信息来生成手册中的一部分。例如，Buildroot 使用此 hook 变量来生成附录中的表格。

• FOO_CHECK_DEPENDENCIES_HOOKS，指定对必需的组件运行附加测试以生成文档。在 AsciiDoc 中，可以调用过滤器，即可以解析单个 AsciiDoc 块并对其进行适当渲染的程序（例如，ditaa 或者 aafigure)。

• FOO_CHECK_DEPENDENCIES_<**FMT**>_HOOKS，指定针对特定的格式<FMT>运行附加测试（见上述提供的格式）。

以下是使用所有变量和所有 hook 的完整示例：

```
01: ################################################################################
##
02: #
03: # foo-document
04: #
05: ################################################################################
##
06:
07: FOO_SOURCES = $(sort $(wildcard $(pkgdir)/*))
08: FOO_RESOURCES = $(sort $(wildcard $(pkgdir)/ressources))
09:
10: define FOO_GEN_EXTRA_DOC
11:         /path/to/generate-script --outdir=$(@D)
12: endef
13: FOO_POST_RSYNC_HOOKS += FOO_GEN_EXTRA_DOC
14:
15: define FOO_CHECK_MY_PROG
16:         if ! which my-prog >/dev/null 2>&1; then \
17:             echo "You need my-prog to generate the foo document"; \
18:             exit 1; \
19:         fi
20: endef
21: FOO_CHECK_DEPENDENCIES_HOOKS += FOO_CHECK_MY_PROG
22:
23: define FOO_CHECK_MY_OTHER_PROG
24:         if ! which my-other-prog >/dev/null 2>&1; then \
25:             echo "You need my-other-prog to generate the foo document as PDF"; \
26:             exit 1; \
27:         fi
28: endef
29: FOO_CHECK_DEPENDENCIES_PDF_HOOKS += FOO_CHECK_MY_OTHER_PROG
30:
31: $(eval $(call asciidoc-document))
```

17.20 Linux 内核软件包特定的基础结构

Linux 内核软件包可以使用基于软件包 hook 变量的某些特定基础结构来构建 Linux 内核工具或者 Linux 内核扩展。

17.20.1 linux 内核工具

Buildroot 提供了一个辅助基础结构，可为 Linux 内核源码中可用的目标构建一些用户空间工具。由于它们的源代码是内核源代码的一部分，因此存在一个特殊的软件包 linux-tools，它重用了在目标上运行的 Linux 内核的源代码。

让我们看一个 Linux 工具的例子。对于一个名为 foo 的全新的 Linux 工具，在现有的 package/linux-tools/Config.in 文件中创建一个新的菜单项。该文件将包含有关每个内核工具的选项描述，在配置工具中会使用和显示它们。它基本上看起来像：

```
config BR2_PACKAGE_LINUX_TOOLS_FOO
        bool "foo"
        select BR2_PACKAGE_LINUX_TOOLS
        help
          This is a comment that explains what foo kernel tool is.

          http://foosoftware.org/foo/
```

选项名称以前缀 BR2_PACKAGE_LINUX_TOOLS_开头，后面跟该工具的大写名称。

请注意，与其他软件包不同，linux-tools 软件包的选项是显示在 linux 内核菜单中的“Linux Kernel Tools”子菜单下，而不是在“Target packages”主菜单下。

然后，为每个 linux 工具添加一个名为 package/linux-tools/linux-tool-foo.mk.in 的.mk.in 文件。它基本上看起来像：

```
################################################################################
#
# foo
#
################################################################################

LINUX_TOOLS += foo

FOO_DEPENDENCIES = libbbb

define FOO_BUILD_CMDS
        $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools foo
endef

define FOO_INSTALL_STAGING_CMDS
        $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
                DESTDIR=$(STAGING_DIR) \
```

```
18:                 foo_install
19: endef
20:
21: define FOO_INSTALL_TARGET_CMDS
22:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
23:                 DESTDIR=$(TARGET_DIR) \
24:                 foo_install
25: endef
```

第 7 行，我们将 Linux 工具 foo 注册到可用的 Linux 工具列表中。

第 9 行，我们指定此工具的依赖项列表。只有当选择了“foo”工具，这些依赖项才会被添加到 Linux 软件包的依赖项列表中。

该 Makefile 的其余部分，第 11-25 行定义了在 Linux 工具构建过程的不同步骤中应该执行的操作（如同 generic package 那样）。只有当选择了“foo”工具，才会执行它们。所支持的命令有_BUILD_CMDS、_INSTALL_STAGING_CMDS 和_INSTALL_TARGET_CMDS。

请注意一点，一定不能调用$(eval $(generic-package))或任何其他软件包基础结构！Linux 工具本身不是软件包，它们是 linux-tools 软件包的一部分。

17.20.2 linux 内核扩展

一些软件包提供了新的功能，它需要对 Linux 内核树进行修改。这可以采用打补丁到内核树的方式，或者采用添加新文件到内核树的方式。Buildroot 的 Linux 内核扩展基础结构提供了一个简单的解决方案，可以在提取内核源码之后和在应用内核补丁之前自动执行此类操作。采用此机制的扩展软件包示例是实时扩展 Xenomai 和 RTAI，以及内核树外 LCD 屏幕驱动程序集 fbtft。

下面让我们看一个示例，如何添加一个新的 Linux 扩展 foo。

首先，创建可提供扩展的软件包 foo：该软件包是一个标准软件包；有关如何创建这样的软件包，请参阅前面章节。该软件包负责下载源码压缩包、检查哈希值、定义许可证信息以及构建用户空间工具（如果有）。

然后创建适当的 Linux 扩展：在现有的 linux/Config.ext.in 文件中创建一个新的菜单项。该文件包含有关每个内核扩展的选项描述，将在配置工具中使用和显示它们。它基本上看起来像：

```
01: config BR2_LINUX_KERNEL_EXT_FOO
02:     bool "foo"
03:     help
04:       This is a comment that explains what foo kernel extension is.
05:
06:       http://foosoftware.org/foo/
```

然后为每个 linux 扩展，添加一个名为 linux/linux-ext-foo.mk 的.mk 文件。它基本上应包含：

```
01: ################################################################################
##
02: #
03: # foo
04: #
05: ################################################################################
##
```

```
06:
07: LINUX_EXTENSIONS += foo
08:
09: define FOO_PREPARE_KERNEL
10:         $(FOO_DIR)/prepare-kernel-tree.sh --linux-dir=$(@D)
11: endef
```

第 7 行，我们将 Linux 扩展 foo 添加到可用的 Linux 扩展列表中。

第 9-11 行，我们定义扩展应执行的操作以修改 Linux 内核树。这是特定于 linux 扩展的，其中可以使用 foo 软件包定义的变量，例如：$(FOO_DIR)或$(FOO_VERSION)…以及所有 Linux 变量，例如：$(LINUX_VERSION)或$(LINUX_VERSION_PROBED)，$(KERNEL_ARCH)…请参阅内核变量的定义。

17.21 各个构建步骤中可用的 Hook 钩子变量

generic 基础结构（以及派生的 autotools 和 cmake 基础结构）允许软件包指定不同的 hook 钩子变量。这些钩子变量定义了在现有步骤之后要执行的其他操作。大多数 hook 钩子变量对于 generic 软件包并不是真正有用，因为.mk 文件可以完全控制在软件包构建过程中每个步骤所需执行的操作。

可用的 hook 钩子变量如下所示：

- LIBFOO_PRE_DOWNLOAD_HOOKS
- LIBFOO_POST_DOWNLOAD_HOOKS
- LIBFOO_PRE_EXTRACT_HOOKS
- LIBFOO_POST_EXTRACT_HOOKS
- LIBFOO_PRE_RSYNC_HOOKS
- LIBFOO_POST_RSYNC_HOOKS
- LIBFOO_PRE_PATCH_HOOKS
- LIBFOO_POST_PATCH_HOOKS
- LIBFOO_PRE_CONFIGURE_HOOKS
- LIBFOO_POST_CONFIGURE_HOOKS
- LIBFOO_PRE_BUILD_HOOKS
- LIBFOO_POST_BUILD_HOOKS
- LIBFOO_PRE_INSTALL_HOOKS (for host packages only)
- LIBFOO_POST_INSTALL_HOOKS (for host packages only)
- LIBFOO_PRE_INSTALL_STAGING_HOOKS (for target packages only)
- LIBFOO_POST_INSTALL_STAGING_HOOKS (for target packages only)
- LIBFOO_PRE_INSTALL_TARGET_HOOKS (for target packages only)
- LIBFOO_POST_INSTALL_TARGET_HOOKS (for target packages only)
- LIBFOO_PRE_INSTALL_IMAGES_HOOKS
- LIBFOO_POST_INSTALL_IMAGES_HOOKS
- LIBFOO_PRE_LEGAL_INFO_HOOKS
- LIBFOO_POST_LEGAL_INFO_HOOKS

这些 hook 钩子变量是变量名列表，它包含要在此 hook 钩子变量下执行的操作。这允许在给定 hook 钩子变量中注册多个钩子操作。示例如下：

```
define LIBFOO_POST_PATCH_FIXUP
```

```
        action1
        action2
endif

LIBFOO_POST_PATCH_HOOKS += LIBFOO_POST_PATCH_FIXUP
```

17.21.1 使用 POST_RSYNC hook 钩子变量

仅针对使用本地源码的软件包，即通过“local”site 站点方式或通过 OVERRIDE_SRCDIR 机制来获取源码的软件包，POST_RSYNC hook 钩子变量才会运行。在这种情况下，将使用 rsync 命令将软件包源码从本地位置复制到 buildroot 构建目录中。但是，rsync 命令不会从源目录中复制所有文件。属于版本控制系统的文件（如目录.git，.hg 等）将不会被复制。对于大多数软件包而言，这已经足够了，但是对于给定的软件包也可以使用 POST_RSYNC hook 钩子变量来执行其他操作。

原则上，hook 钩子变量中可以包含您想要的任何命令。一种特定的用例是使用 rsync 命令故意复制版本控制目录。在 hook 钩子变量中使用 rsync 命令时，除其他变量外，可以使用以下变量：

- $(SRCDIR)：源目录的路径
- $(@D)：构建目录的路径

17.21.2 Target-finalize hook 钩子变量

软件包也可以在 LIBFOO_TARGET_FINALIZE_HOOKS 变量中注册 hook 钩子操作。这些 hook 钩子操作是在构建所有软件包之后但在生成文件系统镜像之前运行。它们很少使用，您的软件包可能不需要它们。

17.22 Gettext 和其他软件包的集成与交互

许多支持国际化的软件包都使用 gettext 库。该库的依赖关系相当复杂，因此应当进行一些解释。

glibc C 库集成了 gettext 的完整实现，支持翻译功能。因此，对本地语言的支持（Native Language Support）已内置在 glibc 中。

另一方面，uClibc 和 musl C 库仅提供 gettext 功能的存根实现（*译者注：即只提供实现接口*），该实现允许使用 gettext 函数来编译库和程序，但不提供 gettext 完整实现中的翻译功能。使用此类 C 库，如果需要真正的本地语言支持，则可以由 gettext 软件包的 libintl 库提供。

因此，为了确保正确处理本地语言支持（Native Language Support），Buildroot 中可以使用 NLS 支持的软件包应该：

1. 当 BR2_SYSTEM_ENABLE_NLS=y 时，确保启用了 NLS 支持。这对于 autotools 软件包是自动完成的，因此这仅应在使用其他软件包基础结构的软件包中进行。

2. 将$(TARGET_NLS_DEPENDENCIES)添加到软件包<pkg>_DEPENDENCIES 变量中。这种添加应该无条件地完成：核心基础结构会自动调整此变量的值，以包含相关的软件包列表。如果禁用了 NLS 支持，则此变量为空。如果启用了 NLS 支持，则此变量会包含 host-gettext，以便在宿主机上编译翻译文件时可获取所需工具。另外，如果使用 uClibc 或 musl，则此变量还包含 gettext，以获取完整的 gettext 实现。

3. 如果需要，将$(TARGET_NLS_LIBS)添加到链接器标志，以便该软件包与 libintl 链接。通常，autotools 软件包不需要它，因为它们通常会自动检测到它们应该与 lbintl 链接。但是，使

用其他构建系统的软件包或基于 autotools 的有问题的软件包可能会需要它。$(TARGET_NLS_LIBS)应该无条件地添加到链接器标志中，因为核心基础结构会自动根据配置将其设置为空或定义为-lintl。

不应对 Config.in 文件进行任何更改以支持 NLS。

最后，某些软件包在目标上可能需要一些 gettext 实用程序，例如 gettext 程序本身，该程序允许从命令行检索翻译的字符串。在这种情况下，软件包应该：

- 请在其 Config.in 文件中使用“select BR2_PACKAGE_GETTEXT”，在它上面的 comment 注释中指出这仅是 rutime 依赖项。
- 不要在其.mk 文件的 DEPENDENCIES 变量中添加任何 gettext 依赖项。

17.23 提示和技巧

17.23.1 软件包名称，配置条目名称和 makefile 变量关系

在 Buildroot 中，它们之间存在一些关系：

- 软件包名称，即软件包目录的名称（和*.mk 文件的名称）；
- 配置条目名称，即在 Config.in 文件中所声明的配置条目；
- makefile 变量前缀。

必须使用以下规则来保持这些元素之间的一致性：

- 软件包目录和*.mk 名称本身就是软件包名称（例如：package/foo-bar_boo/foo-bar_boo.mk）；
- 执行“make”后生成目标的名称是软件包名称本身（例如：foo-bar_boo）；
- config 条目带有大写的软件包名称，“.”和“-”将以“_”字符代替，前缀为 BR2_PACKAGE_（例如：BR2_PACKAGE_FOO_BAR_BOO）；
- *.mk 文件里面的变量前缀是大写的软件包名称，“.”和“-”将以“_”字符替换（例如：FOO_BAR_BOO_VERSION）

17.23.2 如何检查编码风格

Buildroot 在 utils/check-package 中提供了一个脚本，该脚本检查新文件或者更改过的文件的编码风格。它不是一个完整的语言验证程序，但它会捕获许多常见错误。它旨在在创建、提交补丁之前，在您创建或修改的实际文件中运行。

该脚本可用于软件包、文件系统 makefile、Config.in 文件等。它不会检查定义软件包基础结构的文件以及其他包含相似通用代码的文件。

要使用它，请运行 check-package 脚本，并告诉脚本您所创建或更改的文件：

```
$ ./utils/check-package package/new-package/*
```

如果您的路径中有 utils 目录，则还可以运行：

```
$ cd package/new-package/
$ check-package *
```

该工具还可以用于位于 br2-external 中的软件包：

```
$ check-package -b /path/to/br2-ext-tree/package/my-package/*
```

17.23.3 如何测试您的软件包

一旦您已经添加新的软件包，那么重要的一件事是，您必须在各种条件下对其进行测试：它是否适用于所有体系架构？是否能使用不同的 C 库构建？是否需要线程，NPTL？等等...

Buildroot 会运行 autobuilders 持续测试随机配置。但是，这些仅构建 git 树的 master 分支源码，而您的新的软件包尚未存在。

Buildroot 在 utils/test-pkg 中提供了一个脚本，该脚本使用与 autobuilders 相同的基本配置，因此您可以在相同条件下测试软件包。

首先，创建一个配置片段，其中包含启用软件包所需的所有必需选项，但不包含任何体系架构或工具链选项。例如，让我们创建一个仅启用 libcurl 而没有任何 TLS 后端的配置代码段：

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
```

如果您的软件包需要更多的配置选项，则可以将它们添加到 config 片段中。例如，下面是测试将 openssl 作为 TLS 后端的 libcurl 和 curl 程序的配置：

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
BR2_PACKAGE_LIBCURL_CURL=y
BR2_PACKAGE_OPENSSL=y
```

然后运行 test-pkg 脚本，并告诉它要使用的配置片段以及要测试的软件包：

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
```

默认情况下，test-pkg 将针对 autobuilders 所使用的工具链的子集来构建软件包，该子集已被 Buildroot 开发人员选为最有用和最具代表性的子集。如果要测试所有工具链，请传递-a 选项。请注意，在任何情况下，都将排除内部工具链，因为它们耗费的时间太长。

上述指令的执行结果列出了所有经过测试的工具链以及相应的测试结果（下面输出结果非真实的，仅用于说明）：

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
                armv5-ctng-linux-gnueabi [ 1/11]: OK
              armv7-ctng-linux-gnueabihf [ 2/11]: OK
                    br-aarch64-glibc [ 3/11]: SKIPPED
                       br-arcle-hs38 [ 4/11]: SKIPPED
                       br-arm-basic [ 5/11]: FAILED
                br-arm-cortex-a9-glibc [ 6/11]: OK
                br-arm-cortex-a9-musl [ 7/11]: FAILED
                 br-arm-cortex-m4-full [ 8/11]: OK
                           br-arm-full [ 9/11]: OK
                  br-arm-full-nothread [10/11]: FAILED
                      br-arm-full-static [11/11]: OK
11 builds, 2 skipped, 2 build failed, 1 legal-info failed
```

（译者注：经测试，首次执行该命令时需加 sudo，buildroot 将联网下载对应的工具链压缩包）

结果表示：

- OK：构建成功。
- SKIPPED：config 片段中所列出的一个或多个配置选项在软件包的最终配置中不存在。这是由于其中的选项具有工具链无法满足的依赖项，例如软件包配置“depends on BR2_USE_MMU”，却使用 noMMU 工具链。缺少的选项会在输出目录（默认为~/br-test-pkg/TOOLCHAIN_NAME/）中的 missing.config 中报告。
- FAILED：构建失败。检查输出目录中的 logfile 文件，看看出了什么问题：
  - build 失败；

- legal-info 失败；
- 初始步骤之一（下载配置文件，应用配置，为软件包运行 dirclean）失败。

当出现失败时，您可以使用相同的选项重新运行该脚本（在修复软件包后）；该脚本将尝试为所有工具链重新构建用-p 选项指定的软件包，而无需重新构建该软件包的所有依赖项。

test-pkg 脚本可接受一些选项，通过运行以下命令来获得帮助：

```
$ ./utils/test-pkg -h
```

17.23.4 如何添加从 GitHub 上获取的软件包

GitHub 上的软件包通常没有下载发行版压缩包的区域。但是，可以直接从 GitHub 上的存储库下载压缩包。由于 GitHub 过去已经更改了下载机制，因此应该使用 github helper 功能，如下所示。

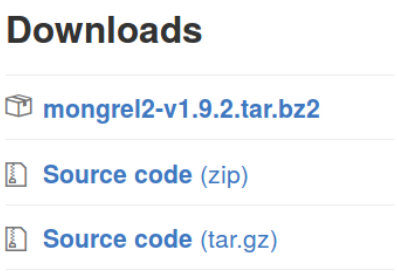
```
# Use a tag or a full commit ID
FOO_VERSION = v1.0
FOO_SITE = $(call github,<user>,<package>,$(FOO_VERSION))
```

注意：

- FOO_VERSION 可以是 tag 或者 commit ID。
- github 生成的压缩包名称与 Buildroot 中默认的名称相匹配（例如：foo-f6fb6654af62045239caed5950bc6c7971965e60.tar.gz），因此没必要在.mk 文件中指定它。
- 当使用 commit ID 作为版本时，应使用完整的 40 个十六进制字符。

如果您要添加的软件包确实在 GitHub 上有 release 部分，则维护者可能已经上传了发行版压缩包，或者该 release 可能只是指向了 git tag 中自动生成的压缩包。如果维护者上传了发行版压缩包，则我们最好使用该版本，因为它可能会略有不同（例如，它已包含配置脚本，因此我们不需要执行 AUTORECONF）。

您可以在 release 页面上看到它是上传的压缩包还是 git tag：



- 如果它看起来像上面的图片，那么它是由维护者上传的，您应该使用该链接（在该示例中：mongrel2-v1.9.2.tar.bz2）来指定 FOO_SITE，而不要使用 github helper。
- 另一方面，如果只有“Source code”链接，则它是一个自动生成的压缩包，您应该使用 github helper 功能。

17.24 结论

如您所见，向 Buildroot 添加一个软件包只是使用现有的例子来编写 Makefile 并根据软件包所需的编译过程对其进行修改。

如果您打包了可能对其他人有用的软件，请不要忘记将补丁发送到 Buildroot 邮件列表（请参阅第 21.5 节）！

18 软件包应用补丁

在集成一个新软件包或更新一个现有软件包时，可能需要对软件的源代码应用补丁，以使其在 Buildroot 中进行交叉编译构建。

Buildroot 提供了一个基础结构，可在构建过程中自动处理该问题。它支持三种应用补丁集的方式：已下载的补丁，buildroot 中提供的补丁以及位于用户自定义的全局补丁目录中的补丁。

18.1 提供补丁

18.1.1 已下载的补丁

如果软件包有必要应用可下载的补丁程序，则将其添加到<packagename>_PATCH 变量中。如果条目中包含“://”，则 Buildroot 将假定它为完整的 URL，然后从该位置下载补丁。否则，Buildroot 将假定应从<packagename>_SITE 下载补丁。它可以是单个补丁，也可以是包含补丁系列的压缩包。

像所有下载的内容一样，补丁文件的哈希值应添加到<packagename>.hash 文件中。

此方法通常用于来自 Debian 的软件包。

18.1.2 在 Buildroot 中的补丁

大多数补丁都在 Buildroot 的 package 目录中提供；这些补丁通常旨在修复交叉编译、libc 支持或其他此类问题。

这些补丁文件应命名为<number>-<description>.patch。

请注意：

- 在 Buildroot 中提供的补丁文件，其文件名不应包含任何软件包版本引用。
- 补丁文件名称中的<number>字段是指应用顺序，应从 1 开始；最好用 0 填充至最大 4 位数字，像 git-format-patch 一样。例如：0001-foobar-the-buz.patch。
- 以前，补丁文件名的前缀必须是软件包名称，例如<package>-<number>-<description>.patch，但现在事实已不再如此。现有的软件包将随着时间推移而修复。不要使用软件包名称作为补丁文件名前缀。
- 以前，series 文件，由 quilt 所管理的，也可以添加到 package 目录中。在这种情况下，series 文件定义了补丁的应用顺序。这是不推荐使用的，Buildroot 将来会移除它。不要使用 series 文件。

18.1.3 全局补丁目录

BR2_GLOBAL_PATCH_DIR 配置文件选项可用于指定一个或多个包含全局软件包补丁的、以空格分隔的目录列表。有关详细信息，请参阅第 9.8 节。

18.2 如何应用补丁

1. 运行<packagename>_PRE_PATCH_HOOKS 命令（如果已定义）；
2. 清理构建目录，删除所有存在的*.rej 文件；
3. 如果定义了<packagename>_PATCH，则将应用来自这些压缩包的补丁；
4. 如果在软件包的 Buildroot 目录或名为<packageversion>的软件包子目录中有一些*.patch 文件，则：

- 如果软件包目录中存在 series 文件，则根据 series 文件应用补丁；
- 否则，匹配*.patch 的补丁文件将按字母顺序进行应用。因此，为确保以正确的顺序应用它们，强烈建议将补丁文件命名为：<number>-<description>.patch，其中<number>是指应用顺

序。

5. 如果定义了 BR2_GLOBAL_PATCH_DIR，则将按照指定的顺序列举目录。按前面所述的步骤应用补丁。

6. 运行<packagename>_POST_PATCH_HOOKS 命令（如果已定义）。

如果在步骤 3 或 4 中出了问题，则构建失败。

18.3 软件包补丁的格式和许可

补丁是在与所应用的软件相同的许可证下发行（请参阅第 12.2 节）。

应当在补丁文件的标题注释中添加一条信息，说明补丁的作用以及为什么需要补丁。

您应在每个补丁文件的标题中添加一个 Signed-off-by 语句，以帮助跟踪更改并确认该补丁是在与所修改的软件相同的许可证下发行。

如果软件包受版本控制，则建议使用上游 SCM 软件生成补丁集。否则，会将标题与“diff -purN package-version.orig/ package-version/”命令的输出连接在一起。

如果您要更新现有的补丁（例如，修改软件包版本），则需确保没有删除已有的 From 标题和 Signed-off-by 标签，根据需要更新其余补丁注释内容。

最后，补丁文件应看起来像：

```
configure.ac: add C++ support test

Signed-off-by: John Doe john.doe@noname.org

--- configure.ac.orig
+++ configure.ac
@@ -40,2 +40,12 @@

AC_PROG_MAKE_SET
+
+AC_CACHE_CHECK([whether the C++ compiler works],
+                [rw_cv_prog_cxx_works],
+                [AC_LANG_PUSH([C++])
+                 AC_LINK_IFELSE([AC_LANG_PROGRAM([], [])],
+                                [rw_cv_prog_cxx_works=yes],
+                                [rw_cv_prog_cxx_works=no])
+                 AC_LANG_POP([C++])])
+
+AM_CONDITIONAL([CXX_WORKS], [test "x$rw_cv_prog_cxx_works" = "xyes"])
```

18.4 集成在 Web 上找到的补丁

当集成一个作者不是您的补丁时，必须在补丁本身的标题中添加一些内容。

这取决于是从项目存储库本身，还是从网络上的某个地方来获取补丁，添加以下标签之一：

```
Backported from: <some commit id>
```

或者

```
Fetch from: <some url>
```

对于可能需要对补丁进行的任何更改，添加一些文字说明也是明智的。

19 下载基础结构

TODO（*译者注：原文未写，仅显示“TODO”*）

20 调试 Buildroot

在构建软件包时，可以检测 Buildroot 所执行的步骤。定义 BR2_INSTRUMENTATION_SCRIPTS 变量，指定您想要在每个步骤之前和之后调用的一个或多个脚本（或其他可执行文件）的路径，以空格分隔。这些脚本按顺序调用，带有三个参数：

- start 或 end，表示步骤的开始或结束；
- 即将开始或刚刚结束的步骤的名称；
- 软件包的名称。

例如：

```
make BR2_INSTRUMENTATION_SCRIPTS="/path/to/my/script1 /path/to/my/script2"
```

该步骤的列表是：

- extract
- patch
- configure
- build
- install-host，当 host-package 安装在$(HOST_DIR)时
- install-target，当 target-package 安装在$(TARGET_DIR)时
- isntall-staging，当 target-package 安装在$(STAGING_DIR)时
- install-image，当 target-package 安装在$(BINARIES_DIR)时

该脚本可以访问以下变量：

- BR2_CONFIG：Buildroot .config 文件的路径
- HOST_DIR，STAGING_DIR，TARGET_DIR：请参阅第 17.5.2 节
- BUILD_DIR：在该目录提取和构建软件包
- BINARIES_DIR：所有二进制文件（也称为镜像）的存储位置
- BASE_DIR：基本输出目录

21 为 Buildroot 做贡献

您可以通过多种方式为 Buildroot 做出贡献：分析和修复错误，分析和修复由 autobuilders 检测到的软件包构建失败，测试和查看其他开发人员发送的补丁，处理我们 TODO 列表中的项目，以及将您自己的改进发送给 Buildroot 或其手册。以下各节提供了有关这些项目的更多详细信息。

如果您有兴趣为 Buildroot 做出贡献，那么您应该做的第一件事就是订阅 Buildroot 邮件列表。此列表是与其他 Buildroot 开发人员进行交互并向其发送贡献的主要方式。如果您尚未订阅，请参阅第 5 章中的订阅链接。

如果您要修改代码，那么强烈建议使用 Buildroot 的 git 存储库，而不是从提取的源码压缩包中开始。Git 是从中进行开发并直接发送补丁到邮件列表的最简单方法。有关获取 Buildroot git 树的更多信息，请参阅第 3 章。

21.1 复现、分析和修复错误

做出贡献的第一种方法是查看 Buildroot bug tracker 中公开的 bug 报告。正如我们努力保持 bug 数量尽可能少，我们非常欢迎在有关复现、分析和修复已报告 bug 方面的所有帮助。即使您还没有看到全貌，也不要犹豫，在错误报告中添加评论，以报告您的发现。

21.2 分析和修复自动构建失败

Buildroot 自动构建器（autobuilders）是一组构建机器，它们基于随机配置来连续运行 Buildroot 构建。这适用于 Buildroot 支持的所有体系架构、各种工具链以及随机选择的软件包。由于 Buildroot 上的大量提交活动，这些自动生成器在提交后尽早发现问题方面提供了很大帮助。

所有构建结果均位于 http://autobuild.buildroot.org，统计资料位于 http://autobuild.buildroot.org/stats.php。每一天，所有构建失败的软件包的概述都会发送到邮件列表。

发现问题是很好的，但是显然也必须解决这些问题。非常欢迎您的贡献！基本上可以完成两件事：

- 分析问题。每日摘要邮件不包含有关实际故障的详细信息：要查看所发生的情况，您必须打开构建日志并检查最后的输出。让其他人对邮件中的所有软件包执行此操作对于其他开发人员来说非常有用，因为他们可以单独基于此输出来进行快速的初始分析。
- 解决问题。当修复自动构建失败问题时，应遵循以下步骤：

1. 检查是否可以通过使用相同的构建配置来复现该问题。您可以手动执行此操作，也可以使用 br-reproduce-build 脚本，该脚本将自动克隆 Buildroot git 存储库，checkout 正确的修订版，下载并设置正确的配置，然后开始构建。
2. 分析问题并创建修复程序。
3. 从干净的 Buildroot 目录树开始并仅应用您的修复程序，以验证问题是否确实得到解决。
4. 将修复程序发送到 Buildroot 邮件列表（请参阅第 21.5 节）。如果针对软件源创建了补丁程序，则您还应该向上游发送补丁程序，以便在以后的版本中解决该问题，并且您可以删除 Buildroot 中的补丁程序。在修复自动构建失败的补丁的提交消息中，添加对构建结果目录的引用，如下所示：

```
Fixes: http://autobuild.buildroot.org/results/51000a9d4656afe9e0ea6f07b9f8ed374c2e4069
```

21.3 审查和测试补丁

每天都会有大量的补丁发送到邮件列表，维护人员很难判断哪些补丁已经准备好应用，哪些补丁尚未准备好。因此贡献者可以通过审查和测试这些补丁，来对此提供极大的帮助。

在审查过程中，请不要犹豫对补丁提交评论、建议或任何有助于所有人理解补丁并使之变得更好的内容。回复补丁提交时，请在纯文本电子邮件中使用 Internet 样式回复。

为了表示补丁已批准，有三个正式标签可用于跟踪此批准行为。将您的标签添加到补丁程序中，请在原始作者的“Signed-off-by”行下方添加批准标签，以进行回复。这些标签将由 patchwork（请参阅第 21.3.1 节）自动提取，并且在补丁程序被接受后将成为提交日志的一部分。

Tested-by

表示补丁已成功测试。我们鼓励您详细说明所执行的测试类型（例如，在体系架构 X 和 Y 上进行编译测试，在目标 A 上进行运行测试等等）。这些附加信息可以帮助其他测试人员和维护人员。

Reviewed-by

表示您已对补丁进行了代码审查，并尽最大努力发现问题，但是您对所接触的区域不够熟

悉，无法提供 Acked-by 标签。这意味着在补丁中可能还存在着其他问题，这些问题将由在该区域有更多经验的人发现。如果检测到此类问题，那么您的 Reviewed-by 标签仍然适用，您不会受到指责。

Acked-by

表示您已对补丁进行了代码审查，并且对所接触的区域非常熟悉，感觉到可以按原样来提交补丁程序（无需进行其他更改）。万一以后发现补丁有问题，您的 Acked-by 标签可能会被认为是不合适的。因此，Acked-by 和 Reviewed-by 之间的区别主要在于，您准备将责任归咎于“Acked”补丁，而不是“Reviewed”补丁。

如果您查看了补丁程序并对其发表了评论，则只需回复说明这些评论，而无需提供 Reviewed-by 或 Acked-by 标签。仅当判断该补丁是正确的时候才应提供这些标签。

请注意，Reviewed-by 或 Acked-by 标签都不意味着已执行测试。为了表明您已经审查并测试了补丁，请提供两个单独的标签（Reviewed/Acked-by 和 Tested-by）。

还要注意，任何开发人员都可以毫无例外地提供 Tested/Reviewed/Acked-by 标签，我们鼓励所有人都这样做。Buildroot 没有定义好的核心开发人员组，恰好有些开发人员比其他开发人员更活跃。维护者将根据其提交者的跟踪记录对标签进行评估。与新手提供的标签相比，常规贡献者提供的标签自然会得到更多的信任。随着您更有规律地提供标签，您的信任度（在维护人员的眼中）将会提高，但是任何所提供的标签都是有价值的。

Buildroot 的 Patchwork 网站可用于拉取补丁以进行测试。有关使用 Buildroot 的 Patchwork 网站来应用补丁的更多信息，请参阅第 21.3.1 节。

21.3.1 应用来自 Patchwork 的补丁

对于开发人员，Buildroot 的 Patchwork 网站的主要用途是将补丁拉取到其本地 git 存储库中来进行测试。

当在 patchwork 管理界面中浏览补丁时，页面顶部会提供一个 mbox 链接。复制此链接地址并运行以下命令：

```
$ git checkout -b <test-branch-name>
$ wget -O - <mbox-url> | git am
```

应用补丁的另一种方法是创建捆绑包（bundles）。捆绑包是一组补丁程序，您可以使用 patchwork 界面将它们分组在一起。一旦捆绑包被创建并将其公开后，您可以复制捆绑包的 mbox 链接并使用上述命令应用该捆绑包。

21.4 处理待办事项清单中的项目

如果您想为 Buildroot 做出贡献，但不知道该从哪里开始，并且不喜欢上述任何主题，则可以始终处理 Buildroot TODO list 中的项目。请先在邮件列表或 IRC 上讨论项目。编辑 Wiki 以表明您何时开始处理项目，从而避免我们重复工作。

21.5 提交补丁

请注意

请不要将补丁附加到 bug 中，而是将其发送到邮件列表中。

如果您对 Buildroot 进行了一些更改，并且希望将其贡献给 Buildroot 项目，请按照以下步骤操作。

21.5.1 补丁格式

我们希望以特定的方式格式化补丁。这对于简化补丁审查、易于应用补丁到 git 存储库、易于在历史记录中发现更改方式和原因以及让使用“git bisect”找到问题根源成为可能，非常必要。

首先，补丁必须具有良好的提交消息（commit message）。提交消息应以单独的一行开头，其中包含更改的简短摘要，并以补丁所涉及的区域作为前缀。一些良好的 commit 标题示例如下：

- package/linuxptp: bump version to 2.0
- configs/imx23evk: bump Linux version to 4.19
- package/pkg-generic: postpone evaluation of dependency conditions
- boot/uboot: needs host-{flex,bison}
- support/testing: add python-ubjson tests

前缀后面的描述应以小写字母开头（即上述示例中的“bump”，“postpone”，“needs”，“add”）。

其次，提交消息的正文应描述为什么需要进行此更改，并在必要时还需提供有关更改方式的详细信息。在编写提交消息时，请考虑审查者将如何阅读它，也要考虑几年后您再次查看此更改时您将如何阅读它。

第三，补丁本身应该只做一个更改，但要做得彻底。通常两个不相关或弱相关的更改应该在两个单独的补丁中完成。这通常意味着补丁仅影响单个软件包。如果涉及多个更改，通常可以将它们分成多个小补丁并按特定的顺序应用它们。小补丁可以让审查变得更容易，并且常常在以后更容易理解为什么进行了此更改。但是，每个补丁都必须完整。不允许在只应用第一个补丁而不应用第二个补丁时，软件包构建受到破坏。为了以后可以使用“git bisect”，这是必需的。

当然，在进行开发时，您可能会在软件包之间来回改动，并且肯定不会立即以一种足够干净的提交方式来提交更改。因此，大多数开发人员会重写提交的历史记录以生成适合提交的干净提交集合。为此，您需要使用交互式变基（interactive rebasing）。您可以了解它 in the Pro Git book。有时，使用“git reset --soft origin/master”放弃历史记录并使用“git add -i”或“git add -p”选择单个更改甚至更容易。

最后，补丁应该被签名。这可以通过在提交消息的末尾添加“Signed-off-by: Your Real Name <your@email.address>”来完成。如果配置正确，“git commit -s”会为您完成添加。Sign-off-by 标签表示您是以 Buildroot 许可证发布补丁（即 GPL-2.0+，但是具有上游许可的软件包补丁除外），并且您可以这样做。有关详细信息，请参阅 the Developer Certificate of Origin 。

当添加新的软件包时，应以单独的补丁提交每个软件包。该补丁应更新 package/Config.in 文件，软件包 Config.in 文件，.mk 文件，.hash 文件，任何初始化脚本以及软件包所有补丁。如果软件包中有许多子选项，则有时最好将它们作为单独的补丁在后续添加。摘要行应类似于“<packagename>: new package”。对于简单的软件包，提交消息的正文可以为空，也可以包含软件包的描述（例如 Config.in 帮助文本）。如果需要进行任何特殊操作来构建该软件包，则还应在提交消息的正文中对此进行明确说明。

当您将软件包升级为新版本时，则还应为每个软件包提交单独的补丁。请不要忘记更新.hash 文件，如果它不存在，则添加它。同样不要忘记检查“_LICENSE”和“_LICENSE_FILES”是否仍然有效。摘要行应类似于“<packagename>: bump to version <new version>”。如果新版本相对于现有版本仅包含安全更新，则摘要行应为“<packagename>: security bump to version <new version>”，并且提交消息的正文应指示所修复的 CVE 编号。如果某些软件包补丁可以在新版本中删除，则应明确说明为什么可以删除它们，并且最好是使用上游提交 ID 来删除。此

外，还应明确说明任何其他所必需的更改，例如一些配置选项不再存在或者不再需要。

如果您希望收到有关您所添加或修改的软件包中构建失败以及需要进一步更改的通知，请将您自己添加到 DEVELOPERS 文件中。这应在创建或修改软件包的同一补丁中完成。有关更多信息，请参阅 DEVELOPERS 文件。

Buildroot 提供了一个方便的工具来检查您所创建或修改的文件中常见的编码格式错误，该工具名称为 check-package。（有关更多信息，请参阅第 17.23.2 节）。

21.5.2 准备补丁系列

从您本地 git 视图中提交的更改开始，在生成补丁集之前，将您的开发分支重置于上游树的顶部。为此，请运行：

```
$ git fetch --all –tags
$ git rebase origin/master
```

现在，您可以生成并提交补丁集了。

要生成它，请运行：

```
$ git format-patch -M -n -s -o outgoing origin/master
```

这将在 outgoing 子目录中生成补丁文件，并自动添加“Signed-off-by”行。

生成补丁文件后，您可以在提交之前使用自己喜欢的文本编辑器来查看或编辑提交消息。

Buildroot 提供了一个方便的工具，可以知道应将补丁发送给谁，该工具名称为 get-developers（有关更多信息，请参阅第 22 章）。该工具会读取您的补丁并输出适当的“git send-email”命令：

```
$ ./utils/get-developers outgoing/*
```

使用 get-developers 的输出来发送您的补丁：

```
$ git send-email --to buildroot@buildroot.org --cc bob --cc alice outgoing/*
```

另外，可以将“get-developers -e”直接与--cc-cmd 参数一起使用，以“git send-email”自动抄送给受影响的开发人员：

```
$ git send-email --to buildroot@buildroot.org --cc-cmd './utils/get-developers -e' origin/master
```

git 可以配置为自动执行此操作：

```
$ git config sendemail.to buildroot@buildroot.org
$ git config sendemail.ccCmd "$(pwd)/utils/get-developers -e"
```

然后只需执行：

```
$ git send-email origin/master
```

请注意，应将 git 配置为使用您的邮件帐户。要配置 git，请参阅“man git-send-email”或用 google 搜索。

如果您不使用“git send-email”，请确保**发布的补丁没有换行**，否则将无法轻松应用。在这种情况下，请修复您的电子邮件客户端，或者更好的是，学习使用“git send-email”。*（译者注：由于 send-email 命令不是 git 必需的组件，所以在首次安装 git 时，需安装“git-email”组件，如用户是使用 ubuntu，则安装命令为“sudo apt install git-email”）*

21.5.3 Cover letter

如果您想要在单独的邮件中显示整个补丁集，则请在“git format-patch”命令中添加--cover-letter 参数（有关更多信息，请参阅“man git-format-patch”）。这将为您补丁系列的介绍性电子邮件生成一个模板。

在以下情况，cover letter 可能会有助于介绍您所提出的更改：

- 补丁系列中的大量提交；
- 深入影响项目其他部分的更改；
- RFC[1]；
- 只要您认为有助于展示您的工作、您的选择、审查过程等。

[1]RFC:（征求意见）更改提议

21.5.4 补丁修订更改日志

当被要求改进补丁时，则每次提交的新修订版应包含每次提交之间的修改的更改日志。请注意，当您的补丁系列由 cover letter 介绍时，除个别提交中的更改日志外，全部更改日志可能会被添加到 cover letter 中。重做补丁系列的最佳方法是通过交互式变基：“git rebase -i origin/master”。有关更多信息，请参阅 git 手册。

当添加更改日志到单个提交时，此更改日志将在编辑提交消息时添加。在“Signed-off-by”部分下方，添加“---”和您的更改日志。

尽管更改日志在邮件主题及 patchwork 中对审查者是可见的，但当补丁被合并时，git 将自动忽略“---”下面的行。这是预期的行为：更改日志并不是永久保留在项目的 git 历史中。

以下是推荐的布局：

```
Patch title: short explanation, max 72 chars

A paragraph that explains the problem, and how it manifests itself. If
the problem is complex, it is OK to add more paragraphs. All paragraphs
should be wrapped at 72 characters.

A paragraph that explains the root cause of the problem. Again, more
than one paragraph is OK.

Finally, one or more paragraphs that explain how the problem is solved.
Don't hesitate to explain complex solutions in detail.

Signed-off-by: John DOE john.doe@example.net

---
Changes v2 -> v3:
  - foo bar (suggested by Jane)
  - bar buz

Changes v1 -> v2:
  - alpha bravo (suggested by John)
  - charly delta
```

任何补丁修订版都应包含版本号。版本号仅由字母 v 后跟一个大于或等于 2 的整数组成（即“PATCH v2”，“PATCH v3”...）。

使用“git format-patch”通过--subject-prefix 选项可以轻松处理：

```
$ git format-patch --subject-prefix "PATCH v4" -M -s -o outgoing origin/master
```

从 git 版本 1.8.1 开始，您还可以使用-v <n>（其中<n>是版本号）：

```
$ git format-patch -v4 -M -s -o outgoing origin/master
```

当您提供补丁的新版本时，请在 pacthwork 中将旧的补丁标记为“Superseded”（作废的）。您需要在 pacthwork 上创建一个帐户以便能够修改补丁的状态。请注意，您只能更改自己提交的补丁的状态，这意味着您在 pacthwork 上注册的电子邮件地址应该与您用于将补丁发送到邮件列表的那个邮件地址相匹配。

当将补丁提交到邮件列表时，您还可以添加--in-reply-to <message-id>选项。可以在 pacthwork 的“Message ID”标签下找到要回复的邮件的 ID。in-reply-to 的优点是，patchwork 将自动标记以前的补丁版本为“Superseded”。

21.6 报告 issues/bugs 或获取帮助

在报告任何 issue 之前，请检查在 mailing list archive 中是否有人已经报告或修复了类似问题。

当您选择报告 bug 或获取帮助时，可以通过在 bug tracker 中打开 bug 或通过向邮件列表发送邮件，那里会提供许多详细信息，以帮助人们复现问题并找到解决问题的方法。

试着想象如果您正在尝试帮助别人，在这种情况下，您需要什么？

以下是需要提供的详细信息的简短列表：

- 主机（操作系统/发行版）
- Buildroot 版本
- 构建失败的目标
- 编译失败的软件包
- 执行失败的命令及其输出
- 任何您认为可能相关的信息

另外，您应该添加.config 文件（或 defconfig 文件；请参阅第 9.3 节）。

如果其中一些细节内容太多，请不要犹豫使用 pastebin 服务。请注意，下载原始粘贴内容时，并非所有可用的 pastebin 服务都会保留 Unix 样式的行终止符。已知以下 pastebin 服务可以正常工作：- https://gist.github.com/ - http://code.bulix.org/。*（译者注：1.最后一个地址已失效，此处提供一个 ubuntu 的 pastebin 服务地址示例：https://paste.ubuntu.com/；2.Pastebin 是一个专供粘贴配置、debug、log 等文本信息的服务，提交问题时先把相关 log、出错信息等粘贴到 pastebin 服务端，再把对应地址链接发送到邮件列表或 IRC 中，就可以让别人协助检查出问题的地方。）*

21.7 使用 run-tests 框架

Buildroot 包含一个运行时测试框架，称为 run-tests，它建立在 Python 脚本和 QEMU 运行时执行的基础上。该框架有两种类型的测试用例，一种用于构建时测试，另一种用于具有 QEMU 依赖性的运行时测试（run-time test）。该框架的目标如下：

- 构建一个定义明确的配置
- 可选项，验证构建输出的某些属性
- 如果是运行时测试：
  - 在 QEMU 下启动
  - 运行一些测试条件以验证给定功能是否正常运行

run-tests 工具-h 帮助参数中记录了一系列选项描述。一些常见的选项包括设置下载文件夹、输出文件夹、保留构建输出以及对于多个测试用例可以为每个测试用例设置 JLEVEL。

以下是一个运行测试用例的示例。

• 首先，让我们看看所有测试用例选项是什么。可以通过执行“support/testing/run-tests -l”来列出测试用例。这些测试可以在控制台测试开发过程中单独运行。一次运行一个测试和一次运行一组测试子集都可以。

```
$ support/testing/run-tests -l
List of tests
test_run (tests.utils.test_check_package.TestCheckPackage)
Test the various ways the script can be called in a simple top to ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainBuildrootMusl) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainBuildrootuClibc) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainCCache) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainCtngMusl) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainLinaroArm) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainSourceryArmv4) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainSourceryArmv5) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainSourceryArmv7) ... ok
[snip]
test_run (tests.init.test_systemd.TestInitSystemSystemdRoFull) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRoIfupdown) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRoNetworkd) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRwFull) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRwIfupdown) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRwNetworkd) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRo) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRoNet) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRw) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRwNet) ... ok
Ran 157 tests in 0.021s
OK
```

这些运行时测试是由 Buildroot Gitlab CI 基础结构定期执行，请参阅.gitlab.yml 和 https://gitlab.com/buildroot.org/buildroot/-/jobs。

21.7.1 创建测试用例

熟悉如何创建一个测试用例的最好方法是查看一些基本文件系统 support/testing/tests/fs/以及初始化 support/testing/tests/init/的测试脚本。这些测试提供了基本构建以及具有运行类型测试构建的良好示例。还有其他更高级的示例，它们使用诸如嵌套的 br2-external 文件夹的东西来提供 skeleton 框架和其他软件包。

默认情况下，测试用例使用 br-arm-full-* uClibc-ng 工具链和 armv5/7 cpu 的预构建内核。建议使用默认的 defconfig 测试配置，除非需要 Glibc/musl 或更新的内核。通过使用默认值，可以节省构建时间，并且当默认值更新时，测试将自动继承内核/std 库升级。

基本的测试用例定义涉及：

• 创建一个新的测试文件

• 定义唯一的测试类别

- 确定是否可以使用默认的 defconfig plus 测试选项
- 实现“def test_run(self):”函数，可选性启动模拟器并提供测试用例条件。

除了创建测试脚本之外，一旦您拥有初始的测试用例脚本，则还应采取一些其他步骤。首先是将您自己添加到 DEVELOPERS 文件中，以维护该测试用例。其次是通过执行“make .gitlab-ci.yml”来更新 Gitlab CI yml。

21.7.2 调试测试用例

在 Buildroot 存储库中，测试框架是由 support/testing/顶层中的 conf、infra 和 tests 文件夹组成的。所有测试用例都位于 tests 文件夹下，并由不同的文件夹进行组织，代表不同的测试类别。

让我们看一个示例。

- 将 Busybox 初始化系统测试用例与可读/写文件系统 tests.init.test_busybox.TestInitSystemBusyboxRw 一起使用。
- 当调试测试用例时，最少的命令行参数应包含-d 指定 dl 文件夹，-o 指定输出文件夹，以及-k 指定在测试通过或者失败时保留任何输出内容。使用这些选项，测试过程将保留日志记录和构建组件，以提供测试用例的构建和执行状态。

（译者注：运行以下命令时需加 sudo，否则提示权限不足）

```
$ support/testing/run-tests -d dl -o output_folder -k tests.init.test_busybox.TestInitSystemBusyboxRw
15:03:26 TestInitSystemBusyboxRw Starting
15:03:28 TestInitSystemBusyboxRw Building
15:08:18 TestInitSystemBusyboxRw Building done
15:08:27 TestInitSystemBusyboxRw Cleaning up
.
Ran 1 test in 301.140s
OK
```

- 对于成功的构建，output_folder 将包含一个<test name>文件夹，其中包含 Buildroot 构建组件、构建日志和运行时日志。如果构建失败，则控制台将显示其失败的阶段（setup/build/run）。根据失败阶段，可以检查和检测 build/run 日志或 Buildroot 构建组件。如果需要启动 QEMU 实例来进行其他测试，则运行时日志的前几行会捕获该实例，这将允许进行一些增量测试，而无需重新运行 support/testing/run-tests。
- 您还可以对 output_folder 中的源码进行修改（出于调试目的），并重新运行标准的 Buildroot make 目标（为了使用新的修改来重新生成完整的镜像），然后重新运行测试。相对于添加补丁文件、删除输出目录以及重新开始测试，直接修改源码可以加快调试速度。

```
$ ls output_folder/
TestInitSystemBusyboxRw/
TestInitSystemBusyboxRw-build.log
TestInitSystemBusyboxRw-run.log
```

- 在 support/testing/tests/init/test_busybox.py 中可以找到用于实施此示例测试的源文件。该文件概述了创建构建所需的最小 defconfig、启动构建镜像的 QEMU 配置和测试用例断言。

要在 Gitlab CI 中测试现有的或新的测试用例，有一种方法：可以通过在您帐户下的 Gitlab 中创建 Buildroot 分支来调用特定的测试。当添加或更改运行时测试（run-time test）或修复由运行时测试用例测试的 bug 时，这将很方便。

下面示例中，分支名称的<name>部分是一个唯一的字符串，用于标识正在创建的特定任务。

- 触发所有 run-test 测试用例任务：

```
$ git push gitlab HEAD:<name>-runtime-tests
```

- 触发一个测试用例任务，使用一个包含完整测试用例名称的特定分支命名字符串。

```
$ git push gitlab HEAD:<name>-<test case name>
```

22 DEVELOPERS 文件和 get-developers

Buildroot 主目录中包含一个名为 DEVELOPERS 的文件，该文件列出了涉及 Buildroot 不同领域的开发人员。由于有了这个文件，get-developers 工具将允许：

- 通过分析补丁并将修改后的文件与相关的开发人员进行匹配，计算出应向其发送补丁的开发人员列表。有关详细信息，请参阅第 21.5 节。
- 查找哪些开发人员正在处理给定的体系架构或软件包，以便在此体系架构或软件包发生构建失败时可以通知他们。这是通过与 Buildroot 的 autobuild 基础结构进行交互来完成的。

我们要求开发人员在向 Buildroot 添加新的软件包、新的板子或者新的功能时，将其自身注册到 DEVELOPERS 文件中。例如，我们希望贡献新软件包的开发人员在其补丁中包含对 DEVELOPERS 文件的适当修改。

DEVELOPERS 文件格式已详细记录在该文件本身内部。

位于 utils/中的 get-developers 工具允许将 DEVELOPERS 文件用于各种任务：

- 当将一个或几个补丁作为命令行参数传递时，get-developers 将返回适当的“git send-email”命令。如果传递了-e 选项，则仅以适合“git send-email –cc-cmd”的格式打印电子邮件地址。
- 当使用-a <arch>命令行选项时，get-developers 将返回负责给定体系架构的开发人员列表。
- 当使用-p <package>命令行选项时，get-developers 将返回负责给定软件包的开发人员列表。
- 当使用-c 命令行选项时，get-developers 将查看 Buildroot 存储库中版本控制下的所有文件，列出未被任何开发人员处理的文件。此选项的目的是帮助完成 DEVELOPERS 文件。
- 当不带任何参数时，它会验证 DEVELOPERS 文件的完整性，并为不匹配的项目提示“WARNINGS”。

23 发布工程

23.1 发布

Buildroot 项目按季度发布，每月发布错误修复版本（bugfix releases）。每年的第一个版本是长期支持版本，LTS。

- 季度发布版本：2020.02，2020.05，2020.08 和 2020.11
- 错误修复版本：2020.02.1，2020.02.2，…
- LTS 版本：2020.02，2021.02，…

发布版本受支持，直到下一个版本的第一个错误修复版本发布。例如，当 2020.08.1 发布时，2020.05.x 即为 EOL。

LTS 版本受支持，直到下一个 LTS 版本的第一个错误修复版本发布。例如，2020.02.x 版本受支持，直到 2021.02.1 版本发布。

23.2 开发

每个发布周期由在 master 主分支上进行两个月的开发以及在发布之前进行一个月的稳定化组成。在此阶段（稳定化阶段），不会将任何新功能添加到 master 分支中，仅会修复错误。

稳定化阶段从标记-rc1 开始，每周都会对另一个候选发布版本进行标记，直到发布为止。

为了在稳定化阶段处理新功能和版本变更，可以为这些功能创建一个 next 分支。在当前版本发布后，next 分支会合并到 master 主分支中，并在那里继续下一个版本的开发周期。

第四部分 附录

24 Makedev 语法文档

在 Buildroot 中 makedev 语法用于定义对权限的更改，或者定义要创建的设备文件以及如何创建它们，以避免调用 mknod。

该语法是从 makedev 实用程序中派生的，可以在 package/makedevs/README 文件中找到更完整的文档。

它采用以空格分隔的字段列表形式，一行定义一个文件；字段如下：

name	type	mode	uid	gid	major	minor	start	Inc	count

有一些重要的部分：

- name 是您要创建或修改的文件的路径。
- type 是文件的类型，为以下之一：
  - f: 常规文件
  - d: 目录
  - r: 递归目录
  - c: 字符设备文件
  - b: 块设备文件
  - p: 命名管道
- mode 是通常的权限设置（仅允许使用数值）。
- uid 和 gid 是要在此文件上设置的 UID 和 GID；可以是数值或者实际名称。
- major 和 minor 这里用于设备文件，其他文件需设置为“-”。
- start，inc 和 count 适用于您要创建一批文件的情况，可以简化为一个循环，从 start 开始，以 inc 为单位递增计数器，直至达到 count。

假设要更改一个给定文件的权限；使用此语法，您将需要编写：

```
/usr/bin/foo f 755 0 0 - - - - -
/usr/bin/bar f 755 root root - - - - -
/data/buz f 644 buz-user buz-group - - - - -
```

或者，如果要递归更改一个目录的所有者或者权限，则您需要编写（针对目录/usr/share/myapp 以及它下面的所有文件和目录，将 UID 设置为 foo，将 GID 设置为 bar，将访问权限设置为 rwxr-x--）：

```
/usr/share/myapp r 750 foo bar - - - - -
```

另一方面，如果您要创建设备文件/dev/hda 和相应的 15 个分区文件，则需要针对/dev/hda：

```
/dev/hda b 640 root root 3 0 0 0 -
```

然后针对设备文件/dev/hda 进行对应的分区/dev/hdaX，其中 X 范围为 1 到 15：

```
/dev/hda b 640 root root 3 1 1 1 15
```

如果启用了 BR2_ROOTFS_DEVICE_TABLE_SUPPORTS_EXTENDED_ATTRIBUTES，则该语法支持扩展属性。这是通过在描述文件的行之后添加以“|xattr”开头的行来完成的。目前，仅支持将功能作为扩展属性。

|xattr	capability

- |xattr 是指示一个扩展属性的“标志”
- capability 是添加到前面文件的功能

如果要将功能 cap_sys_admin 添加到二进制 foo 程序中，则需编写：

```
/usr/bin/foo f 755 root root - - - - -
|xattr cap_sys_admin+eip
```

可以使用多行“|xattr”给文件添加多种功能。如果要将功能 cap_sys_admin 和 cap_net_admin 添加到二进制 foo 程序中，则需编写：

```
/usr/bin/foo f 755 root root - - - - -
|xattr cap_sys_admin+eip
|xattr cap_net_admin+eip
```

25 Makeusers 语法文档

创建用户的语法受上面的 makedev 语法启发，但是它是特定于 Buildroot 的。

添加用户的语法是以空格分隔的字段列表，一行定义一个用户。字段如下：

username	uid	group	gid	password	home	shell	groups	comment

- username 是用户名（也称登录名）。它不能是 root，并且必须是唯一的。如果设置为“-”，则仅创建一个用户群组。
- uid 是用户所需的 UID。它必须是唯一的，并且不是 0。如果设置为-1，则 Buildroot 将在[1000..1999]范围内计算唯一的 UID。
- group 是用户主群组名。它不能是 root。如果该群组不存在，则创建它。
- gid 是用户主群组所需的 GID。它必须是唯一的，并且不是 0。如果设置为-1，并且该群组不存在，则 Buildroot 将在[1000..1999]范围内计算唯一的 GID。
- password 是 crypt(3)编码的密码。如果以“!”为前缀，则禁用登录。如果以“=”为前缀，则将其解释为明文密码，并进行加密编码（使用 MD5）。如果以“!=”为前缀，则明文密码将被加密编码（使用 MD5），并且禁用登录。如果设置为“*”，则不允许登录。如果设置为“-”，则不会设置密码值。
- home 是用户主目录。如果设置为“-”，则不会创建 home 目录，并且用户的 home 目录将是“/”。不允许将 home 明确设置为“/”。
- shell 是用户所需的 shell 程序。如果设置为“-”，则/bin/false 会被设置为用户的 shell 程序。
- groups 是以逗号分隔的、用户应加入的其他组列表。如果设置为“-”，则用户将成为一个不属于任何其他组的成员。缺少的组将使用任意 gid 来创建。
- comment（也称 GECOS 字段）是几乎自由格式的文本。

每个字段的内容都有一些限制：

- 除 comment 外，所有字段均为必填项。
- 除 comment 外，字段不得包含空格。
- 任何字段都不能包含冒号“:”。

如果 home 不是“-”，则 home 目录以及其下的所有文件都将属于该用户及其主群组。

例子：

```
foo -1 bar -1 !=blabla /home/foo /bin/sh alpha,bravo Foo user
```

这将创建此用户：

- username（也称登录名）为：foo
- uid 由 Buildroot 计算
- 主群组 group 是：bar

- 主群组 gid 由 Buildroot 计算
- 明文密码为：blabla，将进行 crypt(3)编码加密，并且禁用登录。
- home 是：/home/foo
- shell 是：/bin/sh
- foo 也是 groups 中 alpha 和 bravo 的成员
- comment 是：Foo user

```
test 8000 wheel -1 = - /bin/sh - Test user
```

这将创建此用户：

- username（也称登录名）为：test
- uid 是：8000
- 主群组 group 是：wheel
- 主群组 gid 由 Buildroot 计算，并将使用 rootfs skeleton 中定义的值
- password 为空（也称无密码）。
- home 是“/”但不属于 test
- shell 是：/bin/sh
- test 不是任何其他 groups 中的成员
- comment 是：Test user

26 从较早的 Buildroot 版本迁移

一些版本引入了向后不兼容的地方。本节将说明这些不兼容之处，并针对每个不兼容之处说明所需执行的操作来完成迁移。

26.1 迁移至 2016.11

在 Buildroot 2016.11 之前，一次只能使用一个 br2-external 树。而 Buildroot 2016.11 则提供了同时使用多个 br2-external 树的可能性（有关详细信息，请参阅第 9.2 节）。

但是，这意味着较早的 br2-external 树无法按原样使用。必须做一个较小的更改：在 br2-external 树中添加一个名称。

仅需几个步骤，即可轻松完成此操作：

- 首先，在 br2-external 树的根目录下创建一个名为 external.desc 的新文件，并单独用一行来定义 br2-external 树的名称：

```
$ echo 'name: NAME_OF_YOUR_TREE' >external.desc
```

请注意，选择名称时要小心：它必须是唯一的，并且只能由[A-Za-z0-9_]中的 ASCII 字符组成。

- 然后，使用新变量来更改 br2-external 树中每次出现的 BR2_EXTERNAL：

```
$ find . -type f | xargs sed -i 's/BR2_EXTERNAL/BR2_EXTERNAL_NAME_OF_YOUR_TREE_PATH/g'
```

现在，您的 br2-external 树可以与 Buildroot 2016.11 及更高版本一起使用。

请注意：此更改将使您的 br2-external 树与 Buildroo 2016.11 之前的版本不兼容。

26.2 迁移至 2017.08

在 Buildroot 2017.08 之前，宿主机软件包安装在$(HOST_DIR)/usr 中（例如，autotools 的--prefix=$(HOST_DIR)/usr）。而 Buildroot 2017.08，它们现在可以直接安装在$(HOST_DIR)中。

每当软件包安装一个与$(HOST_DIR)/lib 中的库相连接的可执行文件时，它必须具有指向该目录的 RPATH。

指向$(HOST_DIR)/usr/lib 的 RPATH 不再被接受。