目录

本 Buildroot 2017.11 手册是 2017-09-25 从 git revision 3f61400 生成的。Buildroot 手册由 Buildroot 开发人员编写。 它是根据 GNU 通用公共许可证版本 2 授权的。

有关此许可证的全文，请参阅 Buildroot 源中的 COPYING 文件。

第一部分　　入门

第 1 章 关于 Buildroot

Buildroot 是一种工具，可简化并自动化使用交叉编译构建嵌入式系统的完整 Linux 系统的过程。

为了实现这一点，Buildroot 能够为您的目标生成交叉编译工具链、根文件系统、Linux 内核映像和引导加载程序。 Buildroot 可以独立使用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 构建根文件系统）。

Buildroot 主要用于使用嵌入式系统的人员。嵌入式系统通常使用的处理器不是常规的 x86 处理器，每个人都习惯在他的电脑中使用。它们可以是 PowerPC 处理器、MIPS 处理器、ARM 处理器等。

Buildroot 支持众多处理器及其变体；它还提供了可用于现成的几个板的默认配置。除此之外，还有一些第三方的项目是在 Buildroot 之上建立或开发他们的 BSP[1] 或 SDK[2]。

名词解释：

1、 DSP：Board Support Package，板级支持包，是介于主板硬件和操作系统中驱动层程序之间的一层，一般认为它属于操作系统一部分，主要是实现对操作系统的支持，为上层的驱动程序提供访问硬件设备寄存器的函数包，使之能够更好的运行于硬件主板。

2、 SDK：Software Development Kit，软件开发工具包，一般都是一些软件工程师为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件时的开发工具的集合。

第 2 章　系统要求

Buildroot 设计用于在 Linux 系统上运行。

虽然 Buildroot 本身将构建编译所需的大多数主机包，但某些标准 Linux 实用程序预计已经安装在主机系统上。下面将介绍强制性和可选软件包的概述（请注意，软件包名称可能因发行版本而异）。

2.1 强制包

•构建工具：

- which
- sed
- make（版本 3.81 或更高版本）
- binutils
- build-essential（仅适用于基于 Debian 的系统）
- gcc（版本 2.95 或更高版本）
- g ++（版本 2.95 或更高版本）
- bash
- patch
- gzip
- bzip2
- perl（5.8.7 版或更高版本）
- tar
- cpio
- python（版本 2.6 或以后）
- unzip
- rsync
- file（必须在/usr/bin/file）
- bc

•源码下载工具：

- wget

2.2 可选包

•配置接口依赖关系：

对于这些库，您需要安装运行时和开发数据，这些数据在许多发行版中是单独打包的。开发包通常具有-dev 或-devel 后缀。

- ncurses5 使用 menuconfig 界面
- qt4 使用 xconfig 界面
- glib2，gtk2 和 glade2 使用 gconfig 界面

•源码下载工具：

在官方目录中，大部分软件包来源都是使用 ftp，http 或 https 位置的 wget 来检索的。几个软件包只能通过版本控制系统使用。此外，Buildroot 能够通过其他工具（如 rsync 或 scp）下载源代码（有关详细信息，请参阅第 19 章）。如果使用这些方法启用包，则需要在主机系统上安装相应的工具：

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

•与 Java 相关的软件包，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，8.6.3 或更高版本
- w3m
- python 包含 argparse 模块（自动出现在 2.7+和 3.2+以上）
- dblatex（仅适用于 pdf 手册）

•图形生成工具：

- graphviz 使用 graph-depends 和<pkg>-graph-depends
- python-matplotlib 使用图形构建

第 3 章 获得 Buildroot

Buildroot 版本是每 3 个月发布一次版本，分别在每年的 2 月、5 月、8 月和 11 月发布的。版本号格式为 YYYY.MM，例如 2013.02，2014.08。发布版本可以在 http://buildroot.org/downloads/ 下载。

为了方便起见，Buildroot 源代码目录 support/misc/Vagrantfile 中的一个 Vagrantfile 文件可以快速设置一个具有所需依赖关系的虚拟机来开始使用。

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
```

如果你在 Windows 上，将其粘贴到你的 PowerShell 中：

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

如果要跟踪开发，您可以使用每日快照或克隆 Git 仓库。有关详细信息，请参阅 Buildroot 网站的下载页面。

第 4 章 Buildroot 快速启动

重要提示：您可以并且应该用普通用户来使用 Buildroot，没有必要使用 root 来配置和使用 Buildroot。通过普通用户身份运行所有命令，可以在编译和安装过程中保护系统免受包装的影响。

使用 Buildroot 的第一步是创建一个配置。 Buildroot 有一个很好的配置工具，类似于您可以在 Linux 内核或 BusyBox 中找到的配置工具。

进入 buildroot 目录后，如果需要配置原来基于 curses 的配置，可以运行：

```
$ make menuconfig
```

如果要配置新的基于 curses 的配置，可以运行：

```
$ make nconfig
```

如果要配置基于 Qt 的配置，可以运行：

```
$ make xconfig
```

如果要配置基于 GTK 的配置，可以运行：

```
$ make gconfig
```

所有这些“make”命令都需要构建配置实用程序（包括接口），因此您可能需要为配置实用程序使用的相关库安装“开发”软件包。有关更多详细信息，请参阅第 2 章，特别是可选要求第 2.2 节获取您最喜欢的界面的依赖关系。

对于配置工具中的每个菜单项，您可以找到描述输入目的的关联帮助。有关具体配置方面的详细信息，请参阅第 6 章。

一旦配置完成，配置工具就会生成包含整个配置的.config 文件。该文件将被顶级 Makefile 读取。

要开始构建过程，只需运行：

```
$ make
```

你不应该用 Buildroot 使用 make -jN：目前不支持顶级并行 make。相反，使用 BR2_JLEVEL 选项来告诉 Buildroot 运行具有 make -jN 的每个单独包的编译。

make 命令通常执行以下步骤：

•下载源文件（根据需要）；

•配置，构建和安装交叉编译工具链，或简单导入外部工具链；

•配置，构建和安装所选的目标包；

•构建内核映像（如果选择）；

•构建引导加载程序映像（如果选择）；

•以选定的格式创建根文件系统。

Buildroot 输出存储在目录 output 中，该目录包含几个子目录：

•images/ 全部镜像文件（内核映像，引导加载程序和根文件系统映像）的保存目录。这些是您需要放在目标系统上的文件。

•build/ 全部构建的组件（包括 Buildroot 在主机上所需的工具和为目标编译的软件包）。此目录包含每个这些组件的一个子目录。

•staging/ 其中包含类似于根文件系统层次结构的层次结构。此目录包含交叉编译工具链的标题和库以及为目标选择的所有用户空间包。但是，该目录并不是目标的根文件系统：它包含大量的开发文件，未映射的二进制文件和库，这对于嵌入式系统来说太大了。这些开发文件用于为依赖于其他库的目标编译库和应用程序。

•target/ 几乎包含目标的完整根文件系统：除了/dev/（Buildroot 无法创建它们，因为 Buildroot 不以 root 身份运行，并且不希望以 root 身份运行）中的设备文件。此外，它没有正确的权限（例如，为 busybox 二进制文件设置）。因此，此目录不应用于您的目标。相反，您应该使用 images/ 目录中内置的图像之一。如果需要通过 NFS 引导的根文件系统的提取镜像，则使用在 images/ 中生成的 tarball 镜像，并以 root 身份提取。与 staging / 相比，target/ 仅包含运行所选目标应用程序所需的文件和库：开发文件（标题等）不存在，二进制文件被剥离。

•host/ 包含为主机编译的适用于 Buildroot 所需的工具的安装，包括交叉编译工具链。

这些命令，使 menuconfig | nconfig | gconfig | xconfig 和 make 是基本的命令，可以轻松快速地生成适合您需要的映像，以及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，请参见第 8.1 节。

第 5 章 社区资源

像任何开源项目一样，Buildroot 有不同的方式在社区和外部分享信息。

如果您正在寻找一些帮助，想要了解 Buildroot 或为项目做出贡献，那么这些方法中的每一种可能会使您感兴趣。

邮件列表

Buildroot 有一个邮件列表供讨论和开发。它是 Buildroot 用户和开发人员的主要交互方式。只有 Buildroot 邮件列表的订阅者才能发布到此列表。您可以通过邮件列表信息页面订阅。

发送到邮件列表的邮件也可以通过邮件列表存档，并通过 gmane.comp. lib.uclibc.buildroot 访问 Gmane。在提出问题之前，请先搜索邮件列表档案，因为有一个很好的机会有人问过同样的问题。

IRC

Buildroot IRC 频道#buildroot 托管在 Freenode 上。这是一个有用的地方，提出问题或讨论某些话题。

在 IRC 要求帮助时，使用代码共享网站（如 http://code.bulix.org）共享相关的日志或代码段。

请注意，对于某些问题，发布到邮件列表可能会更好，因为它将覆盖更多的人，包括开发人员和用户。

错误跟踪器

Buildroot 中的 Bug 可以通过邮件列表或通过 Buildroot bugtracker 进行报告。在创建错误报告之前，请参阅第 21.6 节。

WIKI

Buildroot wiki 页面托管在 eLinux wiki 上。它包含一些有用的链接，过去和即将到来的事件的概述，以及一个 TODO 列表。

Patchwork

Patchwork 是一种基于网络的补丁跟踪系统，旨在促进对开源项目的贡献和贡献的管理。已发送到邮件列表的修补程序被系统“捕获”，并显示在网页上。引用补丁的任何注释都附加到补丁页面。有关补丁的更多信息，请参见 http://jk.ozlabs.org/projects/patchwork/。

Buildroot 的 Patchwork 网站主要由 Buildroot 的维护者使用，以确保补丁不会错过。它也由 Buildroot 修补程序审阅者使用（另见第 21.3.1 节）。然而，由于网站在干净简洁的网络界面中公开了补丁及其相应的评论意见，所以对于所有的 Buildroot 开发人员来说都是有用的。

Buildroot 补丁管理界面可从 http://patchwork.buildroot.org 获取。

第二部分 用户指南

第 6 章 Buildroot 配置

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

make *config 命令还提供了一个搜索工具。阅读不同前端菜单中的帮助信息，以了解如何使用它：

•在 menuconfig 中，通过按“/”键可以调用搜索工具；
•在 xconfig 中，按 Ctrl + f 调用搜索工具。

搜索结果显示匹配项目的帮助信息。在 menuconfig 中，左列中的数字为相应条目提供了快捷方式。只要输入此号码即可直接跳转到条目，或输入到包含的菜单，以防由于缺少相关性而无法选择条目。

尽管菜单结构和条目的帮助文本应该足够自明，但是一些主题需要额外的解释，这些说明在帮助文本中不能轻易被覆盖，因此将在以下部分中介绍。

6.1 交叉编译工具链

编译工具链是一组可以为您的系统编译代码的工具。它由一个编译器（在我们的例子中是 gcc），二进制文件如汇编器和链接器（在我们的例子中是 binutils）和一个 C 标准库（例如 GNU Libc，uClibc-ng）组成。

安装在您的开发站上的系统肯定已经有一个编译工具链，可以用来编译在系统上运行的应用程序。如果您使用的是 PC，则您的编译工具链可在 x86 处理器上运行，并为 x86 处理器生成代码。在大多数 Linux 系统下，编译工具链使用 GNU libc（glibc）作为 C 标准库。该编译工具链称为“主机编译工具链”，它运行在您工作的机器（称为“主机系统”）上。

编译工具链由您的发行版提供，Buildroot 与其无关（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，您主机系统上的编译工具链在主机系统中运行并生成处理器的代码。当您的嵌入式系统具有不同的处理器时，您需要一个交叉编译工具链 -- 一种在主机系统上运行的编译工具链，但为目标系统（和目标处理器）生成代码。例如，如果您的主机系统使用 x86 并且目标系统使用 ARM，则主机上的常规编译工具链将在 x86 上运行，并生成 x86 的代码，而交叉编译工具链在 x86 上运行并生成 ARM 代码。

Buildroot 为交叉编译工具提供了两种解决方案：

•内部工具链后端，在配置界面中称为 Buildroot 工具链。
•外部工具链后端，在配置界面中称为外部工具链。

这两个解决方案之间的选择使用工具链菜单中的工具链类型选项进行。 一旦选择了一个解决方案，就会出现一些配置选项，它们将在以下部分中详细介绍。

6.1.1 内部工具链后端

内部工具链后端是 Buildroot 在为目标嵌入式系统构建用户空间应用程序和库之前自己构建交叉编译工具链的后端。这个后端支持几个 C 库：uClibc-ng，glibc 和 musl。

选择此后端后，会显示多个选项。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。这个项目值得一些解释。在构建交叉编译工具链的过程中，正在构建 C 库，该库提供用户空间应用程序和 Linux 内核之间的接口。为了知道如何访问 Linux 内核，C 库需要访问 Linux 内核头文件（即内核的.h 文件），它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，用于构建工具链的 Linux 内核头的版本不需要完全匹配您要在嵌入式系统上运行的 Linux 内核的版本。它们只需要与要运行的 Linux 内核版本相等或更旧。如果使用比您在嵌入式系统上运行的 Linux 内核更新的内核头文件，则 C 库可能正在使用未由 Linux 内核提供的接口。

•更改 GCC 编译器，binutils 和 C 库的版本。

•选择多个工具链选项（仅适用于 uClibc）：工具链是否应支持 RPC（主要用于 NFS），广泛支持，语言环境支持（国际化），C ++支持或线程支持。根据您选择的选项，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启用某些工具链选项。大多数软件包在需要某个工具链选项才能启用这些软件包时会显示注释。如果需要，您可以通过运行 make uclibc-menuconfig 来进一步细化 uClibc 配置。请注意，Buildroot 中的所有软件包都将根据 Buildroot 中捆绑的默认 uClibc 配置进行测试：如果通过从 uClibc 中删除功能而偏离此配置，则某些软件包可能不再生成。

值得注意的是，只要其中一个选项被修改，那么整个工具链和系统就必须被重建。见第 8.2 节。

这个后端的优点：

•与 Buildroot 完美整合
•快速，只建立必要的

这个后端的缺点：

•做干净时需要重建工具链，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编译交叉编译工具链。 Buildrdroot 知道一些众所周知的交叉编译工具链（来自 Linaro for ARM，Sourcery CodeBench for ARM，x86-64，PowerPC 和 MIPS），并且能够自动下载它们，或者可以指向一个自定义工具链，可以下载或在本地安装。然后，您有三个解决方案来使用外部工具链：

•使用预定义的外部工具链配置文件，并让 Buildroot 下载，解压缩并安装工具链。 Buildroot 已经知道几个 CodeSourcery 和 Linaro 工具链。只需从可用工具链中选择工具链中的工具链配置文件。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，而不是让 Buildroot 下载并提取工具链，您可以告诉 Buildroot 您的工具链已安装在系统上。只需通过可用的工具链选择工具链中的工具链配置文件，自动取消选择“下载”工具链，然后使用交叉编译工具链的路径填充工具链路径文本条目。

•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“工具链”列表中选择“自定义”工具链解决方案。您需要填写 Toolch ain 路径，Toolchain 前缀和外部工具链 C 库选项。那么你必须告诉 Buildroot 你的外部工具链支持什么。如果您的外部工具链使用 glibc 库，您只需要知道您的工具链是否支持 C ++，以及是否内置 RPC 支持。如果您的外部工具链使用 uClibc 库，那么必须告诉 Buildroot 是否支持 RPC，wide-char，locale，程序调用，线程和 C ++。在执行开始时，Buildroot 会告诉您所选选项是否与工具链配置不匹配。

我们的外部工具链支持已经通过 CodeSourcery 和 Linaro 的工具链测试，由 crosstool-NG 生成的工具链以及 Buildroot 本身生成的工具链。一般来说，支持 sysroot 功能的所有工具链都可以工作。如果没有，请不要犹豫与开发人员联系。

我们不支持 OpenEmbedded 或 Yocto 生成的工具链或 SDK，因为这些工具链不是纯工具链（即编译器，binutils，C 和 C ++库）。相反，这些工具链带有一大批预编译的库和程序。因此，

Buildroot 无法导入工具链的 sysroot，因为它将包含通常由 Buildroot 构建的数百兆字节的预编译库。

我们也不支持使用分发工具链（即您的发行版安装的 gcc / binutils / C 库）作为目标软件的工具链。这是因为您的分发工具链不是“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，则必须使用 Buildroot 或 crosstool-NG 生成交叉编译工具链。

如果要为您的项目生成自定义工具链，可以在 Buildroot 中用作外部工具链，我们的建议绝对是使用 crosstool-NG 来构建的。我们建议与 Buildroot 分开构建工具链，然后使用外部工具链后端将其导入 Buildroot。

这个后端的优点：

•允许使用熟知和经过良好测试的交叉编译工具链。
•避免交叉编译工具链的构建时间，这在嵌入式 Linux 系统的整体构建时间通常是非常重要的。

这个后端的缺点：

•如果您的预先构建的外部工具链有错误，可能很难从工具链厂商获得修复，除非您自己使用 Crosstool-NG 构建外部工具链。

6.1.2.1 外部工具包

使用外部工具链时，Buildroot 会生成一个包装程序，它将透明地将相应的选项（根据配置）传递给外部工具链程序。如果你需要调试这个包装器来检查什么参数被传递，你可以将环境变量 BR2_DEBUG_WRAPPER 设置为以下任一个：

•0：空或未设置：无调试

•1：跟踪单行上的所有参数

•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，称为设备文件，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，您的用户空间应用程序将无法使用硬件设备，即使它们被 Linux 内核正确识别。

在系统配置下，/dev 管理，Buildroot 提供四种不同的解决方案来处理/dev 目录：

•第一个解决方案是静态使用设备表。这是在 Linux 中处理设备文件的旧古典方式。使用这种方法，设备文件将永久存储在根文件系统中（即它们在重新启动之间持续存在），并且当从系统添加或删除硬件设备时，没有任何东西将自动创建和删除这些设备文件。因此，Buildroot 使用设备表创建一组标准的设备文件，默认设备文件存储在 Buildroot 源代码中的 system/device_table_dev.txt 中。当 Buildroot 生成最终根文件系统映像时，此文件被处理，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 使用的默认设备表，或者添加其他设备表，以便 Buildroot 在构建期间创建其他设备文件。因此，如果您使用此方法，并且系统中缺少设备文件，则可以创建包含其他设备文件说明的板/<yourcompany>/<yourproject>/device_table_dev.txt 文件，然后您可以在 board/<yourcompany>/<yourproject>/device_table_dev.txt 中，将 BR2_ROOTFS_STATIC_DEVICE_TABLE 设置为 system/device_table_dev.txt 。有关设备表文件格式的更多详细信息，请参见第 23 章。

•第二个解决方案是仅使用 devtmpfs 的 Dynamic。 devtmpfs 是 Linux 内核中已经在内核 2.6.32 中引入的虚拟文件系统（如果使用较旧的内核，则无法使用此选项）。当安装在 /dev 中时，这个虚拟文件系统会自动使设备文件出现并消失，因为从系统中添加和删除了硬件设备。这个文件系统在重新启动时不会持久：它由内核动态填充。使用 devtmpfs 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，它确保启用这两个选项。但是，如果您在 Buildroot 之外构建您的 Linux 内核，那么您有责任启用这两个选项（如果您未能这样做，您的 Buildroot 系统将无法启动）。

•第三个解决方案是使用 devtmpfs + mdev 进行动态处理。该方法还依赖于上面详细描述的 devtmpfs 虚拟文件系统（因此，在内核配置中启用 CONFIG_DEVTMPFS 和 CONFIG_DEVTMPFS_MOUNT 的要求仍然适用），但在其上添加了 mdev 用户空间实用程序。mdev是BusyBox的一个程序部分，内核每次添加或删除设备时都会调用它。感谢 /etc/mdev.conf 配置文件，mdev 可以配置为例如设置文件的特定权限或所有权，每当设备出现或消失时调用脚本或应用程序等。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。如果您有需要固件的设备，mdev 也很重要，因为它将负责将固件内容推送到内核。 mdev 是 udev 的轻量级实现（具有较少的功能）。有关 mdev

的详细信息及其配置文件的语法，请参见 http://git.busybox.net/-busybox/tree/docs/mdev.txt。

•第四个解决方案是使用 devtmpfs + eudev 进行动态处理。此方法还依赖于上面详细描述的 devtmpfs 虚拟文件系统，但在其上添加了 eudev 用户空间守护程序。 eudev 是一个在后台运行的守护进程，当从系统添加或删除设备时，它将被内核调用。它比 mdev 更重量级的解决方案，但提供更高的灵活性。 eudev 是 udev 的独立版本，它是大多数桌面 Linux 发行版中使用的原始用户空间守护程序，现在它是 Systemd 的一部分。有关详细信息，请访问 http://en.wikipedia.org/wiki/Udev。

Buildroot 开发人员的建议是从动态使用 devtmpfs 的解决方案开始，直到您需要在添加/删除设备或需要固件时通知用户空间，在这种情况下使用 devtmpfs + mdev 通常是一个很好的解决方案。

请注意，如果系统选择为 init 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（它携带 PID 编号 1），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 init 系统，可以从系统配置 Init 系统中选择：

•第一个解决方案是 BusyBox。在许多程序中，BusyBox 具有一个基本的 init 程序的实现，这对大多数嵌入式系统是足够的。启用 BR2_INIT_BUSYBOX 将确保 BusyBox 将构建和安装其 init 程序。这是 Buildroot 中的默认解决方案。 BusyBox init 程序将在启动时读取/etc/inittab 文件，以了解该做什么。该文件的语法可以在 http://git.busybox.net/busybox/tree/examples/initta 中找到（请注意，BusyBox inittab 语法是特别的：不要使用 Internet 上的随机 inittab 文档来了解 BusyBox inittab ）。 Buildroot 中的默认 inittab 存储在 system/skeleton/etc/inittab 中。除了安装几个重要的文件系统之外，默认的 inittab 主要工作是启动 /etc/init.d/rcS 中的 shell 脚本，并启动一个 getty 程序（提供一个登录提示）。

•第二个解决方案是 systemV。该解决方案使用旧的传统 sysvinit 程序，以 package/sysvinit 包装在 Buildroot 中。这是大多数台式机 Linux 发行版中使用的解决方案，直到它们切换到最新的替代方案，如 Upstart 或 Systemd。 sysvinit 还可以使用一个 inittab 文件（与 BusyBox 的语法略有不同）。使用此 init 解决方案安装的默认 inittab 位于 package/sysvinit/inittab 中。

•第三个解决方案是 systemd。 systemd 是 Linux 的新一代 init 系统。它远远超过传统的 init 程序：积极的并行功能，使用套接字和 D-Bus 激活启动服务，提供守护进程的按需启动，跟踪使用 Linux 控制组的进程，支持系统状态的快照和还原，系统将在相对复杂的嵌入式系统中使用，例如需要 D-Bus 和服务在彼此之间进行通信的系统。值得注意的是，systemd 带来了相当大数量的大型依赖： 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 init，因为它对大多数嵌入式系统是足够的。systemd 可以用于更复杂的情况。

第 7 章 配置其他组件

在尝试修改以下任何组件之前，请确认已经配置了 Buildroot 本身，并启用了相应的软件包。

BusyBox：

如果您已经有一个 BusyBox 配置文件，可以使用 BR2_PACKAGE_BUSYBOX_CONFIG 在 Buildroot 配置中直接指定此文件。否则，Buildroot 将从默认的 BusyBox 配置文件开始。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，尽管这不是建议的。有关详细信息，请参见第 8.6 节。

uClibc：

uClibc 的配置与 BusyBox 的相同。用于指定现有配置文件的配置变量是 BR2_UCLIBC_CONFIG。随后更改的命令是 uclibc-menuconfig。

Linux 内核：

如果您已经有一个内核配置文件，可以使用 BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG 在 Buildroot 配置中直接指定此文件。如果还没有内核配置文件，可以使用 BR2_LINUX_KERNEL_USE_DEFCONFIG 在 Buildroot 配置中指定一个 defconfig，或者首先创建一个空文件，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 的配置以与 Linux 内核相同的方式完成。相应的配置变量为 BR2_TARGET_BAREBOX_USE_CUSTOM_CONFIG 和 BR2_TARGET_BAREBOX_USE_DEFCONFIG。要打开配置编辑器，请使用 make barebox-menuconfig。

U-Boot：

U-Boot（版本 2015.04 或更高版本）的配置与 Linux 内核的方式相同。相应的配置变量是 BR2_TARGET_UBOOT_USE_CUSTOM_CONFIG 和 BR2_TARGET_UBOOT_USE_DEFCONFIG。要打开配置编辑器，请使用 make uboot-menuconfig。

第 8 章 一般 Buildroot 用法

8.1 编译提示

这是一系列提示，可帮助您充分利用 Buildroot。

显示由 make 执行的所有命令：

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

使用 defconfig 显示支持的主板列表：

```
$ make list-defconfigs
```

显示所有可用的目标：

```
$make help
```

并非所有目标都可用，但.config 文件中的某些设置可能会隐藏一些目标：

•busybox-menuconfig 仅在 busybox 启用时有效;
•linux-menuconfig 和 linux-savedefconfig 仅在启用了 linux 时有效;
•uclibc-menuconfig 仅在内部工具链后端选择 uClibc C 库时可用;
•barebox-menuconfig 和 barebox-savedefconfig 仅在启用了裸机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：当更改任何体系结构或工具链配置选项时，需要进行显式清理。

删除所有构建产品（包括构建目录，主机，分段和目标树，图像和工具链）：

```
$ make clean
```

生成手册：本手册源位于 docs/manual 目录。生成手册：

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

手册输出将在 output/docs /manual 目录中生成。

请注意：

•需要几个工具来构建文档（参见第 2.2 节）。

为新目标重置 Buildroot：删除所有构建产品以及配置：

```
$ make distclean
```

注意如果启用 ccache，则运行 make clean 或 distclean 不会清空 Buildroot 使用的编译器缓存。

要删除它，请参见第 8.12.3 节。

转储内部 make 变量：可以转储已知的所有变量及其值：

```
$ make -s printvars
VARIABLE = value_of_variable
...
```

可以使用一些变量来调整输出：

•VARS 将列表限制为与指定的 make-pattern 匹配的变量
•QUOTED_VARS，如果设置为 YES，将单引号
•RAW_VARS（如果设置为 YES）将打印未展开的值

例如：

```
$make -s printvars
VARS=BUSYBOX_%DEPENDENCIES
BUSYBOX_DEPENDENCIES=skeleton
toolchain
BUSYBOX_FINAL_ALL_DEPENDENCIES=sk
eleton toolchain
BUSYBOX_FINAL_DEPENDENCIES=skeleto
n toolchain

BUSYBOX_FINAL_PATCH_DEPE
NDENCIES=
BUSYBOX_RDEPENDENCIES=n
curses 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_DEPEN
DENCIES=''
BUSYBOX_RDEPENDENCIES='ncu
rses 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_DEPENDENCIES))
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

引用变量的输出可以在 shell 脚本中重用，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建

当通过 make menuconfig、make xconfig 或其他配置工具之一更改系统配置时，Buildroot 不会尝试检测系统的哪些部分、在哪些情况下，应该重建整个系统，或者只要重新构建一个特定的程序包子集。 要想完全可靠地检测这一点是非常困难的，因此 Buildroot 开发人员决定不尝试这样做。

相反，用户有责任知道何时需要完全重建。 作为一个提示，这里有一些经验法则可以帮助您了解如何使用 Buildroot：

•当目标体系结构配置发生更改时，需要进行完整的重建。改变架构变体，例如二进制格式或浮点策略对整个系统有影响。

•当工具链配置更改时，通常需要完全重建。更改工具链配置通常包括更改编译器版本，C 库的类型或其配置或其他一些基本配置项，这些更改对整个系统有影响。

•当向配置添加附加包时，不一定需要完全重建。 Buildroot 将会检测到这个包从未被构建，并且将构建它。但是，如果此软件包是可以由已经构建的软件包可以选择使用的库，则 Buildroot 将不会自动重新构建。要知道哪些软件包应该被重建，你可以手动重建它们，或者你应该做一个完整的重建。例如，假设您已经构建了一个包含 ctorrent 包的系统，但没有 openssl。您的系统可以正常工作，但您会意识到您希望在 ctorrent 中支持 SSL，因此您可以在 Buildroot 配置中启用 openssl 软件包并重新启动该版本。 Buildroot 将检测到 openssl 应该被构建并且将被构建，但是它不会检测到应该重建 ctorrent 以受益于 openssl 来添加 OpenSSL 支持。您将要完成重建，或重建 ctorrent 本身。

•从配置中删除包时，Buildroot 不会做任何特殊的操作。它不会将此程序包安装的文件从目标根文件系统或工具链 sysroot 中删除，需要完全重新编译才能摆脱这个包。但是，一般来说，您可

以等到下一个中午休息时间从头重新编译。

•当软件包的子选项更改时，软件包不会自动重新编译。进行此类更改后，仅重新构建此程序包通常就足够了，除非启用程序包子选项为程序包添加了一些对已经构建的另一个程序包有用的功能。同样，Buildroot 不会跟踪一个包应该重建的时间：一旦建立了一个包，除非明确告诉你这样做，否则它永远不会重新编译。

•当对根文件系统骨架进行更改时，需要进行完全重新编译。但是，当对根文件系统覆盖，更改后脚本或后映像脚本进行更改时，无需进行完全重建：简单的 make 调用将考虑更改。

一般来说，当您遇到构建错误，并且不确定所做的配置更改的潜在后果时，请进行全面重建。 如果您获得相同的构建错误，那么您确定错误与程序包的部分重建无关，并且如果来自官方 Buildroot 的程序包发生此错误，请不要犹豫，以报告问题！ 随着 Buildroot 的进步，您将逐渐了解完全重建是否真正有必要，您将节省越来越多的时间。

作为参考，通过运行完成重建：

```
$ make clean all
```

8.3 了解如何重建软件包

Buildroot 用户询问的最常见的问题之一是如何重建给定的包或如何从头开始重新构建所有包。

Buildroot 不支持删除软件包，无需从头重建。这是因为 Buildroot 不会跟踪哪个软件包在 output/staging 和 output/target 目录中安装什么文件，或者哪个软件包将根据另一个软件包的可用性进行不同的编译。

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 将从头重新提取，重新配置，重新编译和重新安装此软件包。您可以使用 make <package> -dirclean 命令来询问 buildroot。

另一方面，如果您只想从其编译步骤重新启动程序包的构建过程，则可以运行 make <package> -rebuild，然后使用 make 或 make <package>。它将重新启动程序包的编译和安装，但不是从头开始：它基本上重新执行 make 并将其安装在程序包中，因此它只会重建已更改的文

件。

如果要从其配置步骤重新启动程序包的构建过程，可以运行 make <package> -reconfig ure，然后执行 make 或 make <package>。它将重新启动包的配置，编译和安装。

在内部，Buildroot 创建所谓的邮票文件，以跟踪每个包装已经完成了哪些构建步骤。它们存储在包构建目录，output/build/<package>-<version>/ 中，并命名为.stamp_ <step-name>。上面详细描述的命令简单地操纵这些戳记文件，以强制 Buildroot 重新启动包构建过程的特定步骤。

有关包装特殊目标的进一步细节，请参见 8.12.5 节。

8.4 离线版本

如果你打算做一个离线版本，只想下载你以前在配置器（menuconfig，nconfig，xconfig 或 gconfig）中选择的所有源，然后发出：

```
$ make source
```

您现在可以断开或将 dl 目录的内容复制到构建主机。

8.5 构建 out-of-tree

默认情况下，Buildroot 构建的所有内容都存储在 Buildroot 树中的目录输出中。

Buildroot 还支持使用类似于 Linux 内核的语法构建树。要使用它，将 o = <directory>添加到 make 命令行中：

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

或者：

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

所有输出文件将位于/ tmp / build 下。如果 O 路径不存在，Buildroot 将创建它。

注意： O 路径可以是绝对路径或相对路径，但是如果它作为相对路径传递，则重要的是要注意它相对于主 Buildroot 源目录而不是当前工作目录进行解释。

当使用 out-of-tree 构建时，Buildroot .config 和临时文件也存储在输出目录中。这意味着只要使用唯一的输出目录，您可以使用相同的源代码树并行运行多个构建。

为了方便使用，Buildroot 在输出目录中生成一个 Makefile 包装器，所以在第一次运行之后，你不再需要通过

O = <...>和-C <...>，只需运行（在输出目录中）：

```
$ make <target>
```

8.6 环境变量

Buildroot 还会尊重一些环境变量，当它们被传递到环境中进行设置或设置时：

•HOSTCXX，主机使用 C ++编译器

•HOSTCC，主机 C 编译器使用

•UCLIBC_CONFIG_FILE = <path/to/.config>，uClibc 配置文件的路径，用于编译 uClibc，如果正在构建内部工具链。

注意，uClibc 配置文件也可以从配置界面进行设置，所以通过 Buildroot .config 文件；这是推荐的设置方法。

•BUSYBOX_CONFIG_FILE = <path/to/.config>，BusyBox 配置文件的路径。

请注意，BusyBox 配置文件也可以从配置界面设置，所以通过 Buildroot 的 .config 文件；这是推荐的设置方法。

•BR2_CCACHE_DIR，用于覆盖使用 ccache 时 Buildroot 存储缓存文件的目录。

•BR2_DL_DI，覆盖 Buildroot 下载文件的目录 stores/retrieves。

请注意，Buildroot 下载目录也可以从配置界面进行设置，因此可以通过 Buildroot .config 文件进行设置。有关如何设置下载目录的详细信息，请参见第 8.12.4 节。

•BR2_GRAPH_ALT，如果设置为非空，则在构建时图中使用备用颜色方案。

•BR2_GRAPH_OUT，设置生成图形的文件类型，不管是 pdf（默认）还是 png。

•BR2_GRAPH_DEPS_OPTS，将附加选项传递给依赖图;请参阅[simpara]接受的选项。

•将 BR2_GRAPH_DOT_OPTS 作为选项逐字传递给点实用程序绘制依赖图。

使用位于 toplevel 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
```

如果要使用除默认 gcc 或 g ++之外的编译器来构建主机上的辅助二进制文件，那么请执行下列操作

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

8.7 使用文件系统映像进行有效的处理

文件系统映像可以变得相当大，具体取决于您选择的文件系统，软件包数量，是否提供了可用空间。 。 。然而，文件系统图像中的某些位置可能是空的（例如长时间的零）；这样的文件称为稀疏文件。

大多数工具可以有效地处理稀疏文件，并且只能存储或写入不为空的稀疏文件的这些部分。例如：

•tar 接受-S 选项来指示它只存储非零块稀疏文件：

- tar cf archive.tar -S [files ...]将有效地将稀疏文件存储在 tarball 中 - tar xf archive.tar -S 将有效地存储从 tarball 中提取的稀疏文件

•cp 接受--sparse = WHEN 选项（WHEN 是 auto，never 或 always 之一）：

- cp --sparse = always source.file dest.file 将使 dest.file 成为一个稀疏文件，如果 source.file 有很长的零运行

其他工具可能有类似的选择。请咨询他们各自的手册页。

如果您需要存储文件系统映像（例如从一台机器传输到另一台机器），或者您需要将其发送给 Q＆A 团队，则可以使用稀疏文件。

请注意，在使用 dd 的稀疏模式时将文件系统映像闪烁到设备可能会导致文件系统损坏（例如，ext2 文件系统的块位图可能已损坏;或者，如果您的文件系统中有稀疏文件，那些部分可能会读回时不是全零）。处理构建机器上的文件时，应该仅使用稀疏文件，而不是将其传输到将在目标上使用的实际设备。

8.8 绘制包之间的依赖关系

Buildroot 的工作之一是知道软件包之间的依赖关系，并确保它们以正确的顺序构建。这些依赖有时可能是相当复杂的，对于给定的系统，通常不容易理解为什么这样或这样的包被 Buildroot 构建。

为了帮助理解依赖关系，因此更好地了解嵌入式 Linux 系统中不同组件的作用是什么，Buildroot 能够生成依赖关系图。

要生成您编译的完整系统的依赖图，只需运行:

```
make graph-depends
```

您将在 output / graphs / graph-depends.pdf 中找到生成的图。

如果您的系统相当大，依赖图可能太复杂，难以阅读。因此，可以为给定的包生成依赖图:

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

您将在 output/graph/<pkg> -graph-depends.pdf 中找到生成的图形。

请注意，依赖关系图是使用 Graphviz 项目中的点工具生成的，您必须在系统上安装它才能使用此功能。在大多数发行版中，它可以作为 graphviz 包使用。

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，您可以切换到其他输出格式，如 PNG，PostScript 或 SVG。支持点工具的-T 选项支持的所有格式。

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

可以通过在 BR2_GRAPH_DEPS_OPTS 环境变量中设置选项来控制图形依赖行为。

接受的选择是:

- -- depth N，-d N，以将依赖深度限制为 N 级。默认值 0 表示无限制。
- -- stop-onPKG，-s PKG，以停止软件包 PKG 上的图形。 PKG 可以是一个实际的包名，一个 glob，关键字 virtual（在虚拟包上停止）或关键字主机（在主机包上停止）。虽然它不是依赖关系，该包仍然存在于图表上。
- -- exclude PKG，-x PKG，如--stop-on，但也从图中省略 PKG。
- -- transitive， --no-transitive，绘制（或不）传递依赖关系。默认是不绘制传递依赖关系。
- -- colours R，T，H，用于绘制根包（R），目标包（T）和主包（H）的逗号分隔的颜色列表。

默认为：lightblue，gray，gainsboro

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

8.9 绘制构建持续时间

当系统构建需要很长时间时，有助于了解哪些程序包最长才能构建，以查看是否可以完成任何操作以加快构建速度。为了帮助这样的构建时间分析，Buildroot 收集每个包的每个步骤的构建时间，并允许从这些数据生成图形。

要在构建后生成构建时间图，请运行：

```
make graph-build
```

这将在 output/graphs 中生成一组文件：

•build.hist-build.pdf，按照构建顺序排序的每个包的构建时间的直方图。

•build.hist-duration.pdf，每个包的构建时间的直方图，按持续时间排序（最早的）

•build.hist-name.pdf，每个包的构建时间的直方图，按包名称排序。

•build.pie-packages.pdf，每个包的构建时间的饼图

•build.pie-steps.pdf，在包构建过程的每个步骤中花费的全局时间的饼图。

这个图形构建目标需要安装 Python Matplotlib 和 Numpy 库（大多数发行版都有 python-matplotlib 和 python-numpy），如果您使用的是旧版本的 Python 版本（pyt hon-argparse on 大多数发行版）。

默认情况下，图形的输出格式为 PDF，但可以使用 BR2_GRAPH_OUT 环境变量选择不同的格式。唯一支持的其他格式是 PNG：

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

8.10 绘制软件包的文件系统大小贡献

当您的目标系统增长时，有助于了解每个 Buildroot 软件包对整个根文件系统大小的贡献。

为了帮助进行这样的分析，Buildroot 收集有关每个包安装的文件的数据并使用这些数据，生成一个图形和 CSV 文件，详细说明不同包的大小贡献。
要在构建后生成这些数据，请运行：

```
make graph-size
```

这将产生：

•output/graphs/graph-size.pdf，每个包对整个根文件系统大小贡献的饼图

•output/graphs/package-size-stats.csv，一个 CSV 文件，为每个包的大小贡献提供整个根文件系统大小

•output / graphs / file-size-stats.csv，一个 CSV 文件，它将每个安装的文件的大小贡献给它所属的包，以及整个文件系统大小。

这个图形大小的目标需要安装 Python Matplotlib 库（大多数发行版上都有 python-matplotlib），如果您使用的是旧版本的 Python 版本（大多数发行版本都是 python-argparse），则需要使用 argparse 模块。

就像持续时间图一样，支持 BR2_GRAPH_OUT 环境来调整输出文件格式。有关此环境变量的详细信息，请参阅[simpara]。
注意收集的文件系统大小数据仅在完成清洁重建后才有意义。在使用图形大小之前，请务必运行 make clean。

要比较两个不同 Buildroot 编译的根文件系统大小，例如调整配置或切换到另一个 Buildroot 版本后，请使用 size-stats-compare 脚本。它需要两个 file-size-stats.csv 文件（由图形大小生成）作为输入。有关详细信息，请参阅此脚本的帮助文本：

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

8.11 与 Eclipse 集成

虽然嵌入式 Linux 开发人员的一部分喜欢像 Vim 或 Emacs 这样的古典文本编辑器以及基于命令行界面的界面，但是许多其他嵌入式 Linux 开发人员都喜欢更丰富的图形界面来进行开发工作。 Eclipse 是最受欢迎的集成开发环境之一，Buildroot 与 Eclipse 集成，以便于 Eclipse 用户的开发工作。

我们与 Eclipse 的集成简化了构建在 Buildroot 系统之上的应用程序和库的编译，远程执行和远程调试。它不会将 Buildroot 配置和 Eclipse 自身的构建过程整合在一起。因此，Eclipse 集成的典型使用模式是:

•使用 make menuconfig，make xconfig 或随 Buildroot 提供的任何其他配置界面配置 Buildroot 系统。

•通过运行 make 构建您的 Buildroot 系统。

•启动 Eclipse 开发，执行和调试自己的自定义应用程序和库，这些应用程序和库将依赖于 Buildroot 构建和安装的库。

Buildroot Eclipse 集成安装过程和用法在 https://github.com/mbats/eclipse-buildroot-bundle/wiki 中有详细描述。

8.12 高级用法

8.12.1 在 Buildroot 之外使用生成的工具链

您可能希望为您的目标编译您自己的程序或其他未包装在 Buildroot 中的软件。为了做到这一点，你可以使用 Buildroot 生成的工具链。

Buildroot 生成的工具链默认位于 output/host/ 中。使用它的最简单的方法是将 output/host/bin/ 添加到 PATH 环境变量中，然后使用 ARCH-linux-gcc，ARCH-linux-objdump，ARCH-linux-ld 等。

可以重新定位工具链 - 但是，每次调用编译器时，必须传递--sysroot 来告知库和头文件的

位置。

也可以通过使用 Build 选项在除 output/host 之外的目录中生成 Buildroot 工具链! 主机目录选项，如果工具链必须与其他用户共享，这可能很有用。

8.12.2 在 Buildroot 中使用 gdb

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

要达到这个目的:

• 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。这样可以确保十进制 gdb 和 gdbserver 都可以构建，并将 gdbserver 安装到目标上。

•如果使用外部工具链，则应启用 BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY，这将将外部工具链附带的 gdbserver 复制到目标。如果您的外部工具链没有交叉 gdb 或 gdbserver，也可以通过启用与内部工具链后端相同的选项来让 Buildroot 构建它们。

现在，要开始调试一个名为 foo 的程序，你应该运行在目标上:

```
gdbserver：2345 foo
```

这样可以使 gdbserver 在 TCP 端口 2345 上侦听来自 cross gdb 的连接。

然后，在主机上，您应该使用以下命令行启动 cross gdb:

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

当然，foo 必须在当前目录下可用，使用调试符号构建。通常，您从构建 foo 的目录（而不是从 output/target/）启动此命令，因为该目录中的二进制文件被剥离。

<buildroot>/output/staging/usr/share/buildroot/gdbinit 文件将告诉 cross gdb 在哪里找到目标库。

最后，要从 cross gdb 连接到目标:

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

8.12.3 在 Buildroot 中使用 ccache

ccache 是一个编译器缓存。它存储每个编译过程产生的对象文件，并且可以通过使用预先存在的对象文件来跳过同一源文件（具有相同编译器和相同参数）的未来编译。几次从头开始几乎完全相同的构建，它可以很好地加快构建过程。

在 Buildroot 中集成了 ccache 支持。您只需要在 Build 选项中启用编译器缓存。这将自动构建 ccache 并将其用于每个主机和目标编译。

缓存位于 $ HOME/.buildroot-ccache 中。它存储在 Buildroot 输出目录之外，以便它可以由单独的 Buildroot 构建共享。如果要摆脱缓存，只需删除此目录即可。

您可以通过运行 make ccache-stats 来获取缓存的统计信息(其大小, 命中次数, 未命中等)。make 目标 ccache 选项和 CCACHE_OPTIONS 变量提供对 ccache 的更通用访问。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 会产生源文件和编译器选项的哈希值。如果编译器选项不同，缓存对象文件将不会被使用。但是，许多编译器选项包含临时目录的绝对路径。因此，建立在不同的输出目录将导致许多缓存未命中。

为避免此问题，buildroot 具有使用相对路径选项（BR2_CCACHE_USE_BASEDIR）。这将重写指向输出目录内的所有绝对路径到相对路径。因此，更改输出目录不再导致高速缓存未命中。相对路径的缺点是它们也最终成为对象文件中的相对路径。因此，例如，调试器将不再找到该文件，除非您首先 cd 到输出目录。

有关重写绝对路径的更多详细信息，请参阅 ccache 手册“不同目录中的编译”部分。

8.12.4 下载包的位置

由 Buildroot 下载的各种 tarball 都存储在 BR2_DL_DIR 中，默认情况下是 dl 目录。 如果您想保留已知使用关联的压缩包的完整版本的 Buildroot，则可以创建此目录的副本。 这将允许您使用完全相同的版本来重新生成工具链和目标文件系统。

如果您维护了几个 Buildroot 树，那么最好拥有一个共享的下载位置。 这可以通过将 BR2_DL_DIR 环境变量指向一个目录来实现。 如果这样设置，则 Buildroot 配置中的 BR2_DL_DIR 的值将被覆盖。 应该将以下行添加到<~/ .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 文件中的大多数选项不同，该值被 BR2_DL_DIR 环境变量覆盖。

8.12.5 包特定的目标

运行 make <package>构建并安装该特定包及其依赖项。

对于依赖 Buildroot 基础架构的软件包，可以独立调用许多特殊的 make 目标，如下所示：

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

包的构建目标是（按照它们执行的顺序）：

命令/目标	描述
source	提取源码（下载 tarball，克隆源代码库等）
depends	构建并安装构建软件包所需的所有依赖项
extract	提取将源放入包构建目录（提取 tarball，复制源等）
patch	应用补丁（如果有）
configure	执行 configure 命令（如果有）
build	运行编译命令
install-staging	软件包：在分段目录中运行软件包的安装，如果分期必要
install-target	目标包：在目标目录中运行包的安装，如果必要
install	安装目标包：运行 2 个以前的安装命令

	主机包：在主机目录中运行包的安装

此外，还有一些其他有用的 make 目标：

命令/目标	目标描述
show-depends	显示构建包所需的依赖关系
graph-depends	在当前 Buildroot 的上下文中生成包的依赖图组态。有关依赖关系图的更多详细信息，请参阅本节[simpara]。
dirclean	删除整个包构建目录
reinstall	重新运行安装命令
rebuild	重新运行编译命令 - 这仅在使用时才有意义 OVERRIDE_SRCDIR 功能或直接在构建中修改文件时目录
reconfigure	重新运行 configure 命令，然后重建 - 这在使用时才有意义 OVERRIDE_SRCDIR 功能或直接在构建中修改文件时目录

8.12.6 在开发过程中使用 Buildroot

Buildroot 的正常运行是下载 tarball，解压缩，配置，编译和安装在此 tarball 中找到的软件组件。源代码是在 output/build/<package>-<version> 中提取的，它是一个临时目录：每当使用 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。即使使用 Git 或 Subversion 存储库作为软件包源代码的输入，Buildroot 也会创建一个 tarball，然后与 tarball 一样。

当 Buildroot 主要用作集成工具，构建和集成嵌入式 Linux 系统的所有组件时，此行为非常适用。但是，如果在系统某些组件的开发过程中使用 Buildroot，这个行为不是很方便：人们会改变一个包的源代码，而且可以使用 Buildroot 快速重建系统。

直接在 output/build/<package>-<version>中进行更改不是一个合适的解决方案，因为这个目录在 make clean 上被删除。

因此，Buildroot 为此用例提供了一个特定的机制：<pkg>_OVERRIDE_SRCDIR 机制。Buildroot 读取一个覆盖文件，允许用户告诉 Buildroot 某些包的源的位置。默认情况下，此覆盖

文件名为 local.mk，位于 Buildroot 源代码树的顶部目录中，但可以通过 BR2_PACKAGE_OVERRIDE_FILE 配置选项指定其他位置。

在这个覆盖文件中，Buildroot 希望找到以下格式的行：

```
<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 发现给定的包时，已经定义了一个<pkg> _OVERRIDE_SRCDIR，它将不再尝试下载，提取和修补包。相反，它将直接使用指定目录中可用的源代码，并使 clean 不会触及此目录。这允许将 Buildroot 指向您自己的目录，可以由 Git，Subversion 或任何其他版本控制系统进行管理。为了实现这一点，Buildroot 将使用 rsync 将组件的源代码从指定的<pkg> _OVERRIDE_SRCDIR 复制到 output/build/<package>-custom/。

这个机制最好与 make <pkg> -rebuild 一起使用，并使<pkg>-reconfigure 目标。一个 make <pkg> -rebuild 所有的序列将 rsync 的源代码从 <pkg>_OVERRIDE_SRCDIR 输出到 output/build/<package>-custom（感谢 rsync，只有修改的文件被复制），然后重新启动刚才的构建过程包。

在上面的 linux 包的例子中，开发人员可以在 /home/bob/linux 中修改源代码，然后运行：

```
make linux-rebuild all
```

在几秒钟内，您可以在输出/图像中获取更新的 Linux 内核映像。同样，可以改变 BusyBox 源代码在 /home /bob /busybox 中，之后：

```
make busybox-rebuild all
```

output/images 中的根文件系统映像包含更新的 BusyBox。

第 9 章 项目特定的定制

给定项目可能需要执行的典型操作有：

•配置 Buildroot（包括构建选项和工具链，引导加载程序，内核，程序包和文件系统映像类型选择）

•配置其他组件，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）
- 修改或删除目标文件系统上的文件（使用 BR2_ROOTFS_POST_BUILD_SCRIPT）
- 在生成文件系统映像之前运行任意命令（使用 BR2_ROOTFS_POST_BUILD_SCRIPT）
- 设置文件权限和所有权（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）

•生成文件系统映像后运行任意命令（使用 BR2_ROOTFS_POST_IMAGE_SCRIPT）

•将特定于项目的修补程序添加到某些包（使用 BR2_GLOBAL_PATCH_DIR）

•添加专案包

关于这样的项目特定定制的一个重要注意事项：请仔细考虑哪些更改确实是项目特定的，哪些更改对于项目之外的开发人员也是有用的。 Buildroot 社区强烈建议并鼓励对 Buildroot 官方项目的改进，软件包和板级支持进行上游。当然，上游有时是不可能或不可取的，因为这些变化是高度具体的或专有的。

本章介绍如何在 Buildroot 中进行这样的项目特定定制，以及如何以可重复的方式构建相同映像的方式来存储它们，即使在运行 make clean 之后。通过遵循推荐的策略，您甚至可以使用相同的 Buildroot 树来构建多个不同的项目！

9.1 推荐的目录结构

为您的项目定制 Buildroot 时，您将创建一个或多个需要存储在某个地方的专案文件。 虽然大多数这些文件可以放置在任何位置，因为它们的路径要在 Buildroot 配置中指定，但 Buildroot 开发人员建议使用本节中描述的特定目录结构。

与此目录结构正交，您可以选择将此结构本身放置在哪里：在 Buildroot 树内部，或者使用外部树形结构树外部。 这两个选项都是有效的，选择取决于你。

```
+-- 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)
```

有关上述文件的详细信息，请参见本章。

注意：如果您选择将此结构放置在 Buildroot 树之外，但是在一个 br2 外部树中，则<company>和可能的<boardname>组件可能是多余的，可以省略。

9.1.1 实施分层定制

用户有几个相关项目，部分需要相同的自定义项是很常见的。 而不是为每个项目复制这些定制，建议使用分层定制方法，如本节所述。

Buildroot 中几乎可以使用的所有定制方法，如后构建脚本和根文件系统重叠，都可以接受空格分隔的项目列表。 指定的项目始终按照从左到右的顺序进行处理。 通过创建多个这样的项目，一个用于常规定制，另一个用于真正的项目特定的自定义项，您可以避免不必要的重复。每个层通常由 board/<company>/ 中的单独的目录体现。 根据您的项目，您甚至可以引入两层以上。

用户具有两个定制图层的示例目录结构和 fooboard 是：

```
+-- 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，如果用户的 BR2_GLOBAL_PATCH_DIR 配置选项设置为：

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

那么首先应用来自公共层的补丁，然后应用来自 fooboard 层的补丁。

9.2 在 Buildroot 之外保留自定义

如第 9.1 节所述，您可以将项目特定的自定义设置放在两个位置：

•直接在 Buildroot 树中，通常使用版本控制系统中的分支来维护它们，从而轻松升级到较新的 Buildroot 版本。

•在 Buildroot 树之外，使用 br2-外部机制。这种机制允许在 Buildroot 树之外保留包装配方，板支持和配置文件，同时仍然将它们很好地集成到构建逻辑中。我们称这个位置是一个 br2 外部的树。本节将介绍如何使用 br2 外部机制以及在外部树中提供的内容。

可以通过将 BR2_EXTERNAL make 变量设置为要使用的 br2 外部树的路径来告诉 Buildroot 使用一个或多个 br2 外部树。它可以传递给任何 Buildroot 进行调用。它将自动保存在输出目录中隐藏的.br-external.mk 文件中。感谢这样，在每次 make 调用时都不需要传递 BR2_EXTERNAL。然而，它可以随时通过传递一个新值来更改，并且可以通过传递一个空值来删除。

注意 br2 外部树的路径可以是绝对的或相对的。如果它作为相对路径传递，重要的是要注意它相对于主 Buildroot 源目录而不是 Buildroot 输出目录进行解释。

注意：如果在 Buildroot 2016.11 之前使用 br2 外部树，则需要将其转换，然后才能使用 Buildroot 2016.11。有关这方面的帮助，请参阅第 25.1 节。

一些例子：

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

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

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

随时可以切换到另一个 br2 外部树：

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

我们也可以使用多个 br2 外部树：

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

或禁用任何 br2 外部树的使用：

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

9.2.1 br2 外部树的布局

一个 br2-外部的树必须至少包含这三个文件，在以下章节中描述：

•external.desc
•external.mk
•Config.in

除了这些强制性文件外，可能还有一些额外的和可选的内容可能存在于外部的一个树中，如 configs/目录。它们也在以下章节中描述。

稍后将描述完整的示例 br2-外部树形布局。

9.2.1.1 external.desc 文件

该文件描述了 br2 外部树：该 br2 外部树的名称和描述。

此文件的格式为基于行，每行以关键字开头，后跟冒号和一个或多个空格，后跟赋予该关键字的值。目前有两个关键字：

•name，必要的，定义该 br2 外部树的名称。该名称只能在集合[A-Za-z0-9_]中使用 ASCII 字符；禁止任何其他角色。 Buildroot 将变量 BR2_EXTERNAL _ $（NAME）_PATH 设置为 br2 外部树的绝对路径，以便您可以使用它来引用您的 br2 外部树。此变量在 Kconfig 中都可用，因此您可以使用它来为 Kconfig 文件（见下文）和 Makefile 提供源代码，以便您可以使用它来包含其他 Makefile（参见下文）或参考其他文件（如数据文件）从你的 br2 外部树。

注意：由于可以一次使用多个 br2 外部树，Buildroot 将使用该名称为每个树生成变量。该名称用于识别您的 br2 外部树，因此请尝试提供一个真正描述您的 br2 外部树的名称，以使其相对独特，以便它不会与另一个 br2 的另一个名称冲突 - 外部树，特别是如果你打算以某种方式与第三方共享你的 br2 外部树，或者使用来自第三方的 br2 外部树。

•desc，可选，为该 br2 外部树提供简短描述。它应该适合单行，大多是自由形式（见下文），并在显示有关 br2 外部树的信息时使用（例如，在 defconfig 文件列表上方，或作为 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）。描述在 BR2_EXTERNAL _ $（NAME）_DESC 变量中可用。

名称示例和相应的 BR2_EXTERNAL _ $（NAME）_PATH 变量：

•FOO！ BR2_EXTERNAL_FOO_PATH

•BAR_42！ BR2_EXTERNAL_BAR_42_PATH

在下面的例子中，假设名称设置为 BAR_42。

注意：在 Kconfig 文件和 Makefile 中都可以使用 BR2_EXTERNAL _ $（NAME）_PATH 和 BR2_EXTERNAL _ $（NAME）_DESC。它们也在环境中导出，因此可在后构建，后映像和脚本中使用。

9.2.1.2 Config.in 和 external.mk 文件

这些文件（可能都是空的）可以用于定义包配方（即 foo / Config.in 和 foo / foo.mk，像 Buildroot 本身捆绑的包）或其他自定义配置选项或使逻辑。

Buildroot 自动从每个 br2 外部树中包含 Config.in，使其出现在顶级配置菜单中，并将每个 br2 外部树中的 external.mk 与其余的 makefile 逻辑包含在一起。

这主要用于存储包装配方。建议的方法是编写一个如下所示的 Config.in 文件：

```
source “$ BR2_EXTERNAL_BAR_42_PATH / package / package1 / Config.in”
source “$ BR2_EXTERNAL_BAR_42_PATH / package / package2 / Config.in”
```

然后，有一个 external.mk 文件看起来像：

```
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 中创建正常的 Buildroot 包配方，如第 17 章所述。如果您愿意，还可以将包分组到名为<boardname>的子目录中相应地适应上述路径。您还可以在 Config.in 中定义自定义配置选项，并在 external.mk 中定义自定义生成逻辑。

9.2.1.3 configs/ 目录

可以将 Buildroot defconfig 存储在 br2 外部树的 configs 子目录中。 Buildroot 将自动将它们显示在 make list-defconfig 的输出中，并允许它们加载正常的 make <name> _defconfig 命令。它们将在 make list-defconfigs 输出中显示，在外部配置标签下面，该标签包含它们定义的 br2 外部树的名称。

注意：如果一个 defconfig 文件存在于多个 br2 外部树中，则使用最后一个 br2 外部树中的一个。因此可以覆盖在 Buildroot 或另一个 br2 外部树中捆绑的 defconfig。

9.2.1.4 自由格式的内容

可以存储所有特定于板的配置文件，例如内核配置，根文件系统覆盖或 Buildroot 允许其设置位置的任何其他配置文件（通过使用 BR2_EXTERNAL _ $（NAME）_PATH 变量）。例如，您可以将路径设置为全局修补程序目录，rootfs 覆盖和内核配置文件，如下所示（例如，运行 make menuconfig 并填写这些选项）：

```
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_FOO)/board/<board
    name>/kernel. - config
```

9.2.1.5 布局示例

这是一个使用 br2-external 的所有功能的示例布局（示例内容显示在上面的文件中，当它与解释 br2 外部树相关时；当然这完全是为了说明的原因）：

```
/path/to/br2-ext-tree/
  |- external.desc
  |        |name: BAR_42
  |        |desc: Example br2-external tree
  |        '----
```

```
|
|- Config.in
|       |source "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-1/Config.in"
|       |source "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-2/Config.in"
|       |
|       |config BAR_42_FLASH_ADDR
|       |hex "my-board flash address"
|       |default 0x10AD
|       `----
|
|- external.mk
|       |include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/package/*/*.mk))
|       |flash-my-board:
|       |$(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
|
    |- 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_FOO)/boar
          d/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 '---
              -
```

然后，br2 外部树将在 menuconfig 中显示（布局扩展）：

```
External options          --->

    *** Example br2-external tree (in /path/to/br2-ext-tree/)
```

```
[   ] pkg-1
[   ] pkg-2
(0x10AD) my-board flash address
```

如果您使用多个外部的外部树，它将看起来像（布局扩展，第二个名称是 FOO_27 但没有 desc：field 在 external.desc 中）：

```
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 (in /path/to/another-
        br2-ext) [ ] foo
        [ ] bar
```

9.3 存储 Buildroot 配置

可以使用命令 make savedefconfig 存储 Buildroot 配置。

这将通过删除默认值的配置选项来剥离 Buildroot 配置。 结果存储在一个名为 defconfig 的文件中。 如果要将其另存为另一个地方，请更改 Buildroot 配置本身中的 BR2_DEFCONFIG 选项，或使用 make savedefconfig BR2_DEFCONFIG = <path-to-defconfig>调用 make。

存储此 defconfig 的建议位置是 configs / <boardname> _defconfig。 如果按照此建议，配置将在帮助中列出，可以通过运行 make <boardname> _defconfig 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>进行重建。

9.4 存储其他组件的配置

如果更改 BusyBox，Linux 内核，Barebox，U-Boot 和 uClibc 的配置文件也应该存储。对于这些组件中的每一个，存在一个 Buildroot 配置选项来指向输入配置文件，例如。BR2_LINUX_KER

NEL_CUSTOM_CONFIG_FILE。要存储其配置，请将这些配置选项设置为要保存配置文件的路径，然后使用下面描述的辅助对象实际存储配置。

如第 9.1 节所述，存储这些配置文件的建议路径是 board / <company> / <boardname> / foo.config。

在更改 BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE 等选项之前，请确保创建配置文件。否则，Buildroot 将尝试访问这个尚不存在的配置文件，并将失败。您可以通过运行 make linux-menuconfig 等创建配置文件。

Buildroot 提供了一些帮助目标，使配置文件的保存更容易。

•使 linux-update-defconfig 将 linux 配置保存到由 BR2_LINUX_KERNEL_CUST OM_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。

•使 busybox-update-config 将 busybox 配置保存到 BR2_PACKAGE_BUSYBOX_CONFIG 指定的路径。

•使 uclibc-update-config 将 uClibc 配置保存到由 BR2_UCLIBC_CONFIG 指定的路径。

• 让 barebox-update-defconfig 将裸机配置保存到由 BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE 指定的路径。

• 使 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，不存在帮助器，因此您必须手动将配置文件复制到 BR2_TARGET_AT91BOOTSTRAP3_CU STOM_CONFIG_FILE。

9.5 自定义生成的目标文件系统

除了通过 make * config 更改配置外，还有一些其他方法可以自定义生成的目标文件系统。两种可以共存的推荐方法是根文件系统覆盖和后期构建脚本。

根文件系统覆盖（BR2_ROOTFS_OVERLAY）

文件系统覆盖是在构建目标文件系统之后直接复制的文件树。要启用此功能，请将配置选项 BR2_ROOTFS_OVERLAY（在系统配置菜单中）设置为覆盖的根。您甚至可以指定多个叠加层，

空格分隔。如果指定相对路径，它将相对于 Buildroot 树的根。版本控制系统的隐藏目录，例如.git，.svn，.hg 等，名为.empty 的文件和以~结尾的文件被从副本中排除。

如第 9.1 节所示，此覆盖的推荐路径为 board/<company>/<boardname>/rootfs-overlay。

后期制作脚本（BR2_ROOTFS_POST_BUILD_SCRIPT）

后构建脚本是在 Buildroot 构建所有选定的软件之后，但在组合 rootfs 映像之前调用的 shell 脚本。要启用此功能，请在配置选项 BR2_ROOTFS_PO ST_BUILD_SCRIPT（在系统配置菜单中）指定空格分隔的后构建脚本列表。如果指定相对路径，它将相对于 Buildroot 树的根。

使用后构建脚本，可以删除或修改目标文件系统中的任何文件。但是，您应该谨慎使用此功能。每当您发现某个包产生错误或不必要的文件时，您应该修复该包，而不是使用一些后期清理脚本来解决。

如第 9.1 节所示，此脚本的推荐路径是 board/<company>/<boardname>/ post_build.sh.

后构建脚本以主 Buildroot 树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传递给每个脚本。如果配置选项 BR2_ROOTFS_POST_SCRIPT_ARGS 不为空，这些参数也将传递给脚本。所有的脚本将被传递完全相同的参数集合，不可能将不同的参数集传递给每个脚本。

此外，您还可以使用这些环境变量：

•BR2_CONFIG：Buildroot .config 文件的路径

•HOST_DIR，STAGING_DIR，TARGET_DIR：请参见第 17.5.2 节

•BUILD_DIR：提取和构建软件包的目录

•BINARIES_DIR：存储所有二进制文件（也称为图像）的位置

•BASE_DIR：基本输出目录

描述了另外三种定制目标文件系统的方法，但不推荐使用它们。

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映像。 目标文件系统在 output / target /下可用。 进行更改后，运行 make 重建目标文件系统映像。

此方法允许您对目标文件系统执行任何操作，但如果需要使用 make clean 清理 Buildroot 树，则这些更改将丢失。 这种清洁在几种情况下是必要的，详见第 8.2 节。 因此，此解决方案仅适用

于快速测试：更改无法在 make clean 命令中生存。 验证您的更改后，您应确保在使用根文件系统覆盖或后构建脚本进行清理之后它们将保留。

自定义目标骨架（BR2_ROOTFS_SKELETON_CUSTOM）

根文件系统映像是从目标骨架创建的，所有软件包都安装其文件。在构建和安装任何程序包之前，将骨架复制到目标目录 output/target。默认目标骨架提供标准的 Unix 文件系统布局和一些基本的 init 脚本和配置文件。

如果默认框架（在 system/skeleton 下可用）与您的需求不匹配，则通常会使用根文件系统覆盖或后构建脚本进行调整。但是，如果默认框架完全不同于使用自定义框架，则可能更适合。

要启用此功能，请启用配置选项 BR2_ROOTFS_SKELETON_CUSTOM 并将 BR2_ROOTFS_SKELETON_C USTOM_PATH 设置为自定义骨架的路径。系统配置菜单中有两个选项。如果指定相对路径，它将相对于 Buildroot 树的根。

不建议使用此方法，因为它会复制整个 skeleton，从而无法利用修补程序或后续 Buildroot 版本中带来的缺省框架的改进。

Post-fakeroot 脚本（BR2_ROOTFS_POST_FAKEROOT_SCRIPT）

聚合最终图像时，流程的某些部分需要 root 权限：在/ dev 中创建设备节点，为文件和目录设置权限或所有权等。为了避免需要实际的根权限，Buildroot 使用 fakeroot 来模拟根权限。这不是一个完全替代实际上是 root，但足以满足 Buildroot 的需求。

Post-fakeroot 脚本是在 fakeroot 阶段结束时调用的 shell 脚本，就在文件系统映像生成器被调用之前。因此，它们被称为 fakeroot 上下文。

如果您需要调整文件系统以进行通常仅对 root 用户可用的修改，则 post-fakeroot 脚本可能非常有用。

注意：建议使用现有机制设置文件权限或在 /dev 中创建条目（请参见第 9.5.1 节）或创建用户（见第 9.6 节）

注意：后构建脚本（上图）和 fakeroot 脚本之间的区别在于，在 fakeroot 上下文中不调用后构建脚本。

注意：使用 fakeroot 不是实际上是 root 的绝对替代品。 fakeroot 只会伪造文件访问权限和类型（常规，块或字符设备...）和 uid/gid;这些在内存中被仿真。

9.5.1 设置文件权限和所有权并添加自定义设备节点

有时需要在文件或设备节点上设置特定权限或所有权。例如，某些文件可能需要由 root 拥有。由于后构建脚本不以 root 身份运行，因此，除非您使用后构建脚本中的明确的 fakeroot，否则不能执行此类更改。

相反，Buildroot 提供对所谓的权限表的支持。要使用此功能，请将配置选项 BR2_ROOTFS_DEVI CE_TABLE 设置为空格分隔的权限表列表，遵循 makedev 语法的常规文本文件第 23 章。

如果您正在使用静态设备表（即不使用 devtmpfs，mdev 或（e）udev），则可以使用相同的语法在所谓的设备表中添加设备节点。要使用此功能，请将配置选项 BR2_ROOTFS_STATIC_DEVICE_TABLE 设置为空格分隔的设备表列表。

如第 9.1 节所示，这些文件的推荐位置是 board/<company>/<boardname>/。

应该注意的是，如果特定权限或设备节点与特定应用程序相关，则应该在包的.mk 文件中设置变量 FOO_PERMISSIONS 和 FOO_DEVICES（参见第 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的用户。为了满足这一要求，Buildroot 支持所谓的用户表。要使用此功能，请将配置选项 BR2_ROOTFS_USERS_TABLES 设置为空格分隔的用户列表，遵循 makeusers 语法的常规文本文件第 24 章。

如第 9.1 节所示，这些文件的推荐位置是 board/<company>/<boardname>/。

应该注意的是，如果自定义用户与特定应用程序相关，则应该在包的.mk 文件中设置变量 FOO_USERS（参见第 17.5.2 节）。

9.7 创建图像后进行自定义

在构建文件系统映像，内核和引导加载程序之前运行后构建脚本（第 9.5 节）后，可以使用

后映像脚本在创建所有映像后执行一些特定操作。

例如，Post-image 脚本可用于在 NFS 服务器导出的位置中自动提取根文件系统 tarball，或创建捆绑根文件系统和内核映像的特殊固件映像，或项目所需的任何其他自定义操作。

要启用此功能，请在 config 选项 BR2_ROOTFS_POST_IMAGE_SC RIPT（在系统配置菜单中）指定一个空格分隔的后映像脚本列表。如果指定相对路径，它将相对于 Buildroot 树的根。

就像后期构建脚本一样，后台脚本将以主 Buildroot 树作为当前工作目录运行。图像输出目录的路径作为第一个参数传递给每个脚本。如果配置选项 BR2_ROOTFS_POST_SCRIPT_A RGS 不为空，这些参数也将传递给脚本。所有的脚本将被传递完全相同的参数集合，不可能将不同的参数集传递给每个脚本。

再次就像后期制作脚本一样，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 脚本将作为执行 Buildroot 的用户执行，通常不应该是 root 用户。因此，在这些脚本之一中需要 root 权限的任何操作都需要特殊的处理（fakeroot 或 sudo 的使用），这将留给脚本开发人员。

9.8 添加专案补丁

在包装中应用额外的补丁有时是有用的 - 除了在 Buildroot 中提供的补丁之外。这可能用于支持项目中的自定义功能，例如，或在新架构上工作时。

BR2_GLOBAL_PATCH_DIR 配置选项可用于指定包含包修补程序的一个或多个目录的空格分隔列表。

对于特定软件包<packagename>的特定版本<packageversion>，修补程序应用于 BR2_GLOBA L_PATCH_DIR，如下所示：

1.对于存在于 BR2_GLOBAL_PATCH_DIR 中的每个目录 - <global-patch-dir>，将确定<package-

patch-dir>如下：

•如果目录存在，则<global-patch-dir>/<packagename>/<packageversion>/

•否则，如果目录存在，则为<global-patch-dir>/<packagename>。

2.然后从<package-patch-dir>应用补丁，如下所示：

•如果包目录中存在一系列文件，则根据系列文件应用修补程序；

•否则，匹配* .patch 的修补程序文件按字母顺序应用。因此，为了确保按照正确的顺序进行应用，强烈建议将补丁文件命名为：<number> - <description> .patch，其中<number>是指应用顺序。

有关如何应用程序包的修补程序的信息，请参见第 18.2 节

BR2_GLOBAL_PATCH_DIR 选项是为软件包指定自定义修补程序目录的首选方法。它可以用于为 buildroot 中的任何包指定补丁目录。它也应该用于代替可用于诸如 U-Boot 和 Barebox 等软件包的自定义修补程序目录选项。通过这样做，它将允许用户从一个顶级目录管理他们的修补程序。

作为指定自定义修补程序的首选方法 BR2_GLOBAL_PATCH_DIR 的例外是 BR2_LINUX_KERNEL_PATCH。 BR2_LINUX_KERNEL_PATCH 应用于指定 URL 可用的内核修补程序。注意：BR2_LINUX_KERNEL_PATCH 指定在 BR2_GLOBAL_PATCH_DIR 中可用的修补程序之后应用的内核修补程序，因为它是从 Linux 程序包的后修补程序钩子完成的。

9.9 添加项目特定的包

一般来说，任何新软件包都应直接添加到软件包目录中，并提交给 Buildroot 上游项目。一般来说，如何向 Buildroot 添加软件包将在第 17 章中详细介绍，这里不再赘述。但是，您的项目可能需要一些不能上游的专有软件包。本节将介绍如何将特定项目的程序包保存在特定于项目的目录中。

如第 9.1 节所示，项目特定软件包的推荐位置是 package / <company> /。如果使用 br2 外部树功能（参见第 9.2 节），推荐的位置是将它们放在 br2 外部树中的名为 package /的子目录中。

但是，除非我们执行一些额外的步骤，否则 Buildroot 将不会意识到此位置中的包。如第 17 章所述，Buildroot 中的一个包基本上由两个文件组成：一个.mk 文件（描述如何构建包）和一个 Config.in 文件（描述此包的配置选项）。

Buildroot 将自动将.mk 文件包含在包目录的一级子目录中（使用模式包/ * / *。mk）。如果我们希望 Buildroot 从更深入的子目录（如 package / <company> / package1 /）中包含.mk 文件，那么我们只需要在包含这些额外的.mk 文件的一级子目录中添加一个.mk 文件。因此，创建一个包含以下内容的 package / <company> / <company> .mk（假设您在 package / <company> /下面只有一个额外的目录级别）：

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

对于 Config.in 文件，请创建一个包含所有软件包的 Config.in 文件的文件包/ <company> /Config.in。必须提供详尽的列表，因为 kconfig 的 source 命令不支持通配符。例如：

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

从 package / Config.in 中包含这个新的文件包/ <company> /Config.in，最好在一个公司特定的菜单中，使得与未来的 Buildroot 版本的合并更容易。

如果使用 br2 外部树，请参阅第 9.2 节，了解如何填写这些文件。

9.10 存储项目特定自定义的快速指南

在本章的前面部分，描述了制定项目特定定制的不同方法。 现在，这一节将总结所有这一切，提供分步说明来存储项目特定的自定义。 显然，与您的项目无关的步骤可以跳过。

1.使 menuconfig 配置工具链，包和内核。

2.使 linux-menuconfig 更新内核配置，类似于其他配置，如 busybox，uclibc，。。。

3. mkdir -p board / <manufacturer> / <boardname>

4.将以下选项设置为 board / <manufacturer> / <boardname> / <package> .config（只要它们相关）：

•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 barebox-update-defconfig

•make uboot-update-defconfig

6.创建 board/<manufacturer>/<boardname>/rootfs-overlay/，并填写您需要在 rootfs 上的其他文件，例如 board/<manufacturer>/<boardname>/rootfs-overlay/etc/inittab 中。将 BR2_ROOTFS_OVERLAY 设置为 board/<manufacturer>/<boardname>/rootfs-overlay。

7.创建 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. 如果必须设置额外的设置权限或必须创建设备节点，请创建 board/<manufacturer>/<boardname>/ device_table.txt 并将该路径添加到 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 并将该路径添加到 BR2_ROOTFS_USERS_TABLES。

10. 要为某些包添加自定义修补程序，请将 BR2_GLOBAL_PATCH_DIR 设置为 board/<manufacturer>/<board name>/patches 将每个包的修补程序添加到该包命名的子目录中。每个补丁应该被称为<packagename> - <num> - <description> .patch。

11.对于 Linux 内核，还存在选项 BR2_LINUX_KERNEL_PATCH，主要优点是它也可以从 URL 下载

修补程序。如果不需要这个，首选 BR2_GLOBAL_PATCH_DIR。 U-Boot，Barebox，at91bootstrap 和 at91bootstrap3 也有单独的选项，但是这些选项并不比 BR2_GLOBAL_PATCH_DIR 提供任何优势，将来可能会被删除。

12.如果需要添加特定于项目的程序包，请创建程序 package/<manufacturer>/ 并将程序包放在该目录中。创建一个整体的<manufacturer> .mk 文件，其中包含所有包的.mk 文件。创建一个整体 Config.in 文件，该文件来源于所有软件包的 Config.in 文件。从 Buildroot 的 package/ Config.in 文件中包含此 Config.in 文件。

13.使 save savedconfig 保存 buildroot 配置。

14.cp defconfig configs/<boardname>_defconfig

第 10 章 常见问题和疑难解答

10.1 启动网络后启动挂起

如果启动过程似乎在以下消息之后挂起（消息不一定完全相似，具体取决于所选包的列表）：

```
Freeing init memory: 3972K
...Initializingrandomnumbergenerator                    done.
Starting network...
Starting dropbear sshd: generating rsa key...                    generating dsa key...          OK
```

那么这意味着你的系统正在运行，但没有在串行控制台上启动一个 shell。 为了使系统在串行控制台上启动 shell，您必须进入 Buildroot 配置，在系统配置中，修改启动后运行 getty（登录提示），并在 getty 选项子菜单中设置相应的端口和波特率。 这将自动调整生成的系统的/ etc / inittab 文件，以便 shell 在正确的串行端口上启动。

10.2 为什么目标上没有编译器

已经决定，将从 Buildroot-2012.11 版本停止对目标的本机编译器的支持，因为：

•这个特征既没有保持也没有被测试，并且常常被破坏；

•此功能仅适用于 Buildroot 工具链；

•Buildroot 主要针对小型或非常小的目标硬件，其中有限的资源（CPU，RAM，大容量存储），对目标进行编译并不太有意义；

•Buildroot 旨在缓解交叉编译，使目标的本地编译不必要。

如果您需要在目标上编译器，那么 Buildroot 不适合您的目的。 在这种情况下，您需要真正的分发，您应该选择以下内容：

•OpenEmbedded

•yocto

•emdebian

•Fedora

•openSUSE ARM

•Arch Linux ARM

•。。。

10.3 为什么目标没有开发文件

由于目标上没有可用的编译器（请参见第 10.2 节），因此使用标题或静态库来浪费空间是没有意义的。

因此，Buildroot-2012.11 发行版始终会从目标中删除这些文件。

10.4 为什么目标没有文件

因为 Buildroot 主要针对小型或非常小的目标硬件（CPU，RAM，大容量存储）资源有限，所以使用文档数据浪费空间是没有意义的。

如果您仍然需要目标文档数据，那么 Buildroot 不适合您的目的，您应该寻找一个实际的分发（参见第 10.2 节）。

10.5 为什么在 Buildroot 配置菜单中有些软件包不可见

如果一个包存在于 Buildroot 树中，并且没有出现在配置菜单中，这很可能意味着一些包的依赖关系不能满足。

要了解更多关于包的依赖关系，请在配置菜单中搜索包符号（请参见第 8.1 节）。

然后，您可能必须递归启用几个选项（对应于未满足的依赖项）才能最终选择该包。

如果包由于某些未满足的工具链选项而不可见，那么您应该必须运行完整的重建（有关更多说明，请参见第 8.1 节）。

10.6 为什么不将目标目录用作 chroot 目录

有很多理由不能使用目标目录一个 chroot，其中包括：

•文件所有权，模式和权限在目标目录中未正确设置；

•设备节点不会在目标目录中创建。

由于这些原因，使用目标目录作为新根的 chroot 运行的命令很可能会失败。

如果要在 chroot 或 NFS 根目录中运行目标文件系统，请使用图像中生成的 tarball 图像，并以 root 身份提取。

10.7 为什么 Buildroot 不生成二进制包（.deb，.ipkg ...）

Buildroot 列表中经常讨论的一个功能是“软件包管理”的一般主题。总而言之，这个想法是添加一些跟踪哪个 Buildroot 软件包安装什么文件，目标是：

•当从 menuconfig 中取消选择该包时，可以删除包中安装的文件；

•能够生成可以安装在目标上的二进制包（ipk 或其他格式），而无需重新生成新的根文件系统映像。

一般来说，大多数人认为很容易做到：只要跟踪哪个包安装了什么，并在包被取消选择时将其删除。但是，它比这更复杂：

•不仅是 target/目录，还包括 host/ <tuple> / sysroot 中的 sysroot 和 host/ 目录本身。必须跟踪各种软件包安装在这些目录中的所有文件。

•从配置中取消选择包时，仅删除其安装的文件是不够的。还必须删除所有相反的依赖关系（即依赖于它的包）并重建所有这些包。例如，软件包 A 可选地依赖于 OpenSSL 库。两者都被选中，Buildroot 被构建。软件包 A 使用 OpenSSL 构建加密支持。后来，OpenSSL 从配置中被取消选择，但是 A 包仍然存在（因为 OpenSSL 是一个可选的依赖关系，这是可能的）。如果仅删除 OpenSSL 文件，那么包 A 安装的文件就会被破坏：它们使用一个库不再出现在目标上。虽然这在技术上是可行的，但它对 Buildroot 增加了很多复杂性，这违背了我们试图坚持的简单性。

•除了上一个问题，还有一些 Buildroot 甚至不知道可选依赖关系的情况。例如，版本 1.0 中的软件包 A 从未使用 OpenSSL，但是在版本 2.0 中，它会自动使用 OpenSSL（如果可用）。如果

Buildroot .mk 文件尚未更新以考虑到这一点，则软件包 A 将不会成为 OpenSSL 的相反依赖关系的一部分，并且在将 OpenSSL 删除时不会被删除和重建。当然，包 A 的.mk 文件应该被修改为提到这个可选的依赖关系，但同时也可以有不可重现的行为。

•请求还允许将 menuconfig 中的更改应用于输出目录，而无需从头开始重新构建所有内容。然而，这是很难以可靠的方式实现的：当一个包的子选项被改变时会发生什么（我们必须检测这个，并从头开始重新构建包，并且潜在的所有反向依赖），如果工具链选项改变等等。此刻，Buildroot 做的是清晰简单，其行为非常可靠，并且易于支持用户。如果在 menuconfig 中进行的配置更改在下一个 make 之后被应用，那么在所有情况下都必须正确正确地运行，并且没有一些奇怪的角落。风险在于获取错误报告，如“我启用了包 A，B 和 C，然后运行 make，然后禁用包 C 并启用包 D 并运行 make，然后重新启用包 C 并启用包 E，然后有一个建立失败“。或者更糟糕的是，我做了一些配置，然后修改，然后做了一些修改，修改，更多的修改，修改，更多的修改，建立，现在它失败了，但我不记得我所做的所有更改，以及哪个顺序”。这是不可能的支持。

由于所有这些原因，结论是，当未选择包时添加安装文件的跟踪以及生成二进制包的存储库，这是非常难以可靠地实现的，并且会增加很多复杂性。

在这个问题上，Buildroot 开发人员发表了这个立场声明：

•Buildroot 致力于生成根文件系统（因此名称，顺便说一句）。这就是我们想让 Buildroot 擅长：构建根文件系统。

•Buildroot 并不是一个分发（或者说是一个分发生成器）。大多数 Buildroot 开发者都认为这不是我们应该追求的目标。我们相信还有其他工具比 Buildroot 更适合生成发行版。例如，Open Embedded 或 openWRT 是这样的工具。

•我们更倾向于将 Buildroot 推向方便（或更简单）地生成完整的根文件系统。这就是 Buildroot 在人群中脱颖而出（当然有）

•我们认为，对于大多数嵌入式 Linux 系统，二进制包不是必需的，并且可能是有害的。当使用二进制包时，这意味着系统可以部分升级，这样可以在嵌入式设备上进行升级之前创建大量可能的软件包版本组合。另一方面，通过一次升级整个根文件系统映像进行完整的系统升级，部署到嵌入式系统的映像确实是经过测试和验证的映像。

10.8 如何加快构建过程

由于 Buildroot 经常涉及对整个系统进行全面重建，这可能相当长时间，因此我们提供以下一些提示来帮助缩短构建时间：

•使用预先构建的外部工具链，而不是默认的 Buildroot 内部工具链。 通过使用预制的 Linaro 工具链（在 ARM 上）或 Sourcery CodeBench 工具链（对于 ARM，x86，x86-64，MIPS 等），您将在每次完整重建时保存工具链的构建时间，大约 15 到 20 分钟。 请注意，一旦系统的其余部分工作，临时使用外部工具链就不会阻止您切换回内部工具链（可能会提供更高级别的定制）

•使用 ccache 编译器缓存（参见第 8.12.3 节）；

•了解如何重建仅实际关心的几个软件包（请参见第 8.3 节），但请注意，有时需要完全重建（参见第 8.2 节）；

•确保您没有为用于运行 Buildroot 的 Linux 系统使用虚拟机。 大多数虚拟机技术已知会对 I / O 造成重大的性能影响，这对于构建源代码非常重要。

•确保只使用本地文件：不要尝试通过 NFS 进行构建，这会大大减慢构建速度。 让 Buildroot 下载文件夹在本地可用也有所帮助。

•购买新硬件。 SSD 和大量 RAM 是加快构建速度的关键。

第 11 章 已知的问题

•如果这样的选项包含一个$符号，则无法通过 BR2_TARGET_LDFLAGS 传递额外的链接器选项。

例如，已知以下内容中断：BR2_TARGET_LDFLAGS =“ - Wl， -rpath ='$ ORIGIN /../ lib'”

•SuperH 2 和 ARC 架构不支持 libffi 包。

•prboom 包从 Sourcery CodeBench 版本 2012.09 中的 SuperH 4 编译器触发编译器故障。

第 12 章 法律通知和许可

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）都包含在各种许可证下发布的开源软件。

使用开源软件可以让您自由构建丰富的嵌入式系统，从广泛的软件包中进行选择，还可以强制您必须了解和遵守的一些义务。一些许可证要求您在产品文档中发布许可证文本。其他需要您将软件的源代码重新分发给接收产品的源代码。

每个许可证的具体要求在每个包装中都有记录，您的责任（或您的合法办公室）的责任符合这些要求。为了使您更容易，Buildroot 可以为您收集您可能需要的一些材料。要生成此材料后，使用 make menuconfig 配置 Buildroot 后，请执行 xconfig 或 make gconfig，运行：

```
make legal-info
```

Buildroot 将在 legal-info/ 子目录下的输出目录中收集与法律相关的材料。你会发现：

•README 文件，总结了生成的材料，并包含有关 Buildroot 无法生成的材料的警告。

•buildroot.config：这是通常使用 make menuconfig 生成的 Buildroot 配置文件，它是重现构建所必需的。

•所有包的源代码;这分别保存在目标和主机包的 sources/ 和 host-sources/ 子目录中。设置<PKG>_REDISTRIBUTE = NO 的软件包的源代码将不会被保存。已应用的修补程序也保存，以及名为系列的文件，按照其应用顺序列出修补程序。补丁与他们修改的文件的许可证相同。注意：Buildroot 对基于自动工具的程序包的 Libtool 脚本应用其他修补程序。这些补丁可以在 Buildroot 源的 support/libtool 下找到，并且由于技术限制，不能与程序包源一起保存。您可能需要手动收集。

•清单文件（一个用于主机，一个用于目标软件包），列出配置的软件包，其版本，许可证和相关信息。 Buildroot 中可能未定义其中的一些信息;这些物品被标记为“未知”。

•所有软件包的许可证文本，分别在目标和主机软件包的 licenses/ 和 host-licenses/ 子目录中。如果许可证文件未在 Buildroot 中定义，则不会生成该文件，并且 README 中的警告指示此。

请注意，Buildroot 的法律信息功能的目的是生产与某种方式相关的所有材料，以符合包装许可证的合法性。 Buildroot 不会尝试制作必须以某种方式公开的确切材料。当然，生产的材料比严格遵守法规所需要的更多。例如，它生成了根据 BSD 类许可证发布的软件包的源代码，您不需

要以源代码形式重新分发。

此外，由于技术上的限制，Buildroot 不会生成一些您将需要的材料，例如工具链源代码和 Buildroot 源代码本身（包括需要源分发的软件包的补丁）。运行法律信息时，Buildroot 会在 README 文件中生成警告，以通知您无法保存的相关资料。

最后，请记住，make legal-info 的输出是基于每个包配方中的声明性语句。 Buildroot 开发人员竭尽全力尽可能保持声明性声明尽可能准确。然而，很可能这些声明性陈述并不完全准确，也不完整。您（或您的法律部门）必须在使用它作为您的合规交付之前检查法律信息的输出。请参阅 Buildroot 分发根目录下的 COPYING 文件中的 NO WARRANTY 子句（第 11 和 12 条）。

12.2 遵守 Buildroot 许可证

Buildroot 本身是一个开放源代码软件，根据 GNU 通用公共许可证，版本 2 或（在您的选择）任何更新版本发布，除了以下详细的软件补丁。然而，作为构建系统，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机器上，而不存在于设备存储。

然而，Buildroot 开发人员的一般观点是，在发布包含 GPL 授权软件的产品时，您应该释放 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“它包含的所有模块的所有源代码，加上任何关联的接口定义文件，以及用于控制可执行文件的编译和安装的脚本”。 Buildroot 是用于控制可执行程序的编译和安装的脚本的一部分，因此它被认为是必须重新分发的材料的一部分。

请记住，这只是 Buildroot 开发者的意见，如果有任何疑问，您应该咨询您的法律部门或律师。

12.2.1 修补包装

Buildroot 还捆绑补丁文件，这些文件应用于各种软件包的源代码。这些补丁不包括 Buildroot 的许可证。相反，它们被应用补丁的软件的许可证覆盖。当所述软件在多个许可证下可用时，Buildroot 修补程序仅在可公开访问的许可证下提供。

有关技术细节，请参见第 18 章。

第 13 章 超越 Buildroot

13.1 引导生成的图像

13.1.1 NFS 引导

要实现 NFS 引导，请在文件系统映像菜单中启用 tar 根文件系统。

完成构建后，只需运行以下命令即可设置 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 中。

然后，您可以从目标执行 NFS 引导。

13.1.2 Live CD

要构建实时 CD 映像，请在文件系统映像菜单中启用 iso 映像选项。请注意，此选项仅适用于 x86 和 x86-64 架构，如果您正在使用 Buildroot 构建内核。

您可以使用 IsoLinux，Grub 或 Grub 2 作为引导程序构建实时 CD 映像，但只有 Isolinux 支持将此映像用作实时 CD 和实时 USB（通过 Build hybrid image 选项）。

您可以使用 QEMU 测试您的实时 CD 映像：

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

或者如果它是混合 ISO，则将其用作硬盘驱动器映像：

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

它可以很容易地使用 dd 命令写入到一个 USB 驱动器：

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

13.2 chroot

如果要在生成的图像中进行 chroot，那么您应该注意的事情很少：

•您应该从 tar 根文件系统映像中设置新根;

•选定的目标架构与主机兼容，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，以便能够运行为主机上的目标构建的二进制代码;

•Buildroot 当前不提供正确构建的 host-qemu 和 binfmt，并可用于此类用途。

第三部分 开发者指南

第 14 章 Buildroot 如何工作

如上所述，Buildroot 基本上是一组使用正确选项下载，配置和编译软件的 Makefile。它还包括各种软件包的补丁 - 主要是交叉编译工具链（gcc，binutils 和 uClibc）中的补丁。

每个软件包基本上有一个 Makefile，它们以.mk 扩展名命名。 Makefile 分成许多不同的部分。

•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-headers 和 uClibc。

•arch/ 目录包含 Buildroot 支持的所有处理器体系结构的定义。

•package/ 目录包含所有用户空间工具和库的 Makefile 和相关文件，Buildroot 可以编译并添加到目标根文件系统。每个包都有一个子目录。

•linux/ 目录包含 Linux 内核的 Makefile 和相关文件。

•boot/ 目录包含 Buildroot 支持的引导程序的 Makefile 和关联文件。

•system/ 目录包含对系统集成的支持，例如目标文件系统框架和 init 系统的选择。

•fs/ 目录包含与生成目标根文件系统映像相关的软件的 Makefile 和相关文件。

每个目录至少包含 2 个文件：

•something.mk 是下载，配置，编译和安装软件包的 Makefile。

•Config.in 是配置工具描述文件的一部分。它描述与包相关的选项。

主 Makefile 执行以下步骤（配置完成后）：

•在输出目录中创建所有输出目录：分段，目标，构建等（输出/默认情况下，可以使用 O =指定另一个值）

•生成工具链目标。当使用内部工具链时，这意味着生成交叉编译工具链。当使用外部工具链时，这意味着检查外部工具链的功能并将其导入到 Buildroot 环境中。

•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。根据配置，生成这些目标将触发用户空间包（库，程序），内核，引导加载程序和生成根文件系统映像的编译。

第 15 章 编码风格

总的来说，这些编码风格规则在这里可以帮助您在 Buildroot 中添加新文件或重构现有文件。如果稍微修改一些现有的文件，重要的是保持整个文件的一致性，所以你可以：

•或者遵循此文件中使用的潜在弃用的编码风格，

•或完全重做，以使其符合这些规则。

15.1 Config.in 文件

Config.in 文件包含几乎任何可在 Buildroot 中配置的条目。

条目具有以下模式：

```
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，取决于，选择和帮助行缩进一个选项卡。

•帮助文本本身应缩进一个选项卡和两个空格。

•帮助文本应该被包装以适合 72 列，其中选项卡计数为 8，因此文本本身为 62 个字符。

Config.in 文件是 Buildroot 中使用的配置工具的输入，这是常规的 Kconfig。 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

•标题：文件以标题开始。它包含由 80 个哈希值组成的分隔符之间的模块名称，最好是小写的。标题后面必须有空行：

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

```
################################################################################
```

•分配：use =的后面跟着一个空格：

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

不要对齐=符号。

•缩进：仅使用选项卡：

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

请注意，定义块中的命令应始终以一个选项卡开头，因此请将其识别为命令。

•可选依赖关系：

- 偏好多行

语法：

是

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

语法：否

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

- 将配置选项和依赖关系保持在一起。

•可选钩子：将钩子定义和分配一起保存在一个块中。

是：

```
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
```

否:

```
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 文件

文档使用 asciidoc 格式。

有关 asciidoc 语法的更多详细信息，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，应该遵循 PEP8“Python 代码样式指南”。

第 16 章 添加对特定板的支持

Buildroot 包含几个公开可用的硬件板的基本配置，这样一个板卡的用户可以轻松构建一个已知可以工作的系统。欢迎您加入对 Buildroot 的其他板卡的支持。

为此，您需要创建一个正常的 Buildroot 配置，以构建硬件的基本系统：工具链，内核，引导加载程序，文件系统和简单的仅限 BusyBox 的用户空间。不应该选择特定的包：配置应尽可能的小，只能为目标平台构建一个基本的 BusyBox 系统。您当然可以为内部项目使用更复杂的配置，但是 Buildroot 项目只会集成基本的电路板配置。这是因为程序包选择是高度特定于应用程序的。

一旦你有一个已知的工作配置，运行 make savedefconfig。这将在 Buildroot 源代码树的根目录下生成一个最小的 defconfig 文件。将此文件移动到 configs/ 目录中，并将其重命名为 <boardname>_defconfig。

建议尽可能多地使用 Linux 内核和引导加载程序的上游版本，并尽可能使用默认的内核和引导加载程序配置。如果您的电路板不正确，或者没有默认值，我们建议您将修复发送到相应的上游项目。

但是，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，您可以将补丁和配置存储在这些目录中，并从主 Buildroot 配置中引用它们。
有关详细信息，请参阅第 9 章。

第 17 章 向 Buildroot 添加新包

本节介绍如何将新包（用户空间库或应用程序）集成到 Buildroot 中。它还显示了如何集成现有的软件包，这是修复问题或调整其配置所需的。

当您添加新包装时，请务必在各种条件下进行测试;见第 17.20.2 节

17.1 包目录

首先，在软件包的目录下创建一个目录，例如 libfoo。

某些包已按主题分为子目录：x11r7，qt5 和 gstreamer。如果您的包适合这些类别之一，则在这些类别中创建您的包目录。不过，新的子目录是不鼓励的。

17.2 配置文件

要在配置工具中显示包，需要在包目录中创建一个 Config 文件。有两种类型：Config.in 和 Config.in.host。

17.2.1 Config.in 文件

对于目标上使用的包，请创建一个名为 Config.in 的文件。该文件将包含与我们的 libfoo 软件相关的选项描述，将在配置工具中使用和显示。它应该基本上包含：

```
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/
```

有关配置选项的布尔线，帮助行和其他元数据信息必须缩进一个选项卡。帮助文本本身应该缩进一个选项卡和两个空格，线条应该包装以适应 72 列，其中选项卡计数为 8，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 Buildroot 特有的约定，属性的顺序如下：

1.选项的类型：bool，string。 。 。与提示

2.如果需要，默认值

3.任何依赖于形式

4.选择表单的任何依赖关系

5.帮助关键字和帮助文本。

您可以将其他子选项添加到如果 BR2_PACKAGE_LIBFOO ... endif 语句中，以配置软件中的特定内容。您可以在其他包中查看示例。Config.in 文件的语法与内核 Kconfig 文件的语法相同。此语法的文档可从 http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt 下载。

最后，您必须将新的 libfoo / Config.in 添加到 package/Config.in（或在类别子目录中，如果您决定将包放在现有类别之一）。包括在内的文件按类别按字母顺序排序，除了包的裸名外，不应该包含任何东西。

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

17.2.2 Config.in.host 文件

还需要为主机系统构建一些软件包。 这里有两个选择：

•主机软件包只需要满足一个或多个目标软件包的建立时间依赖关系。 在这种情况下，将 host-foo 添加到目标包的 BAR_DEPENDENCIES 变量中。 不应该创建 Config.in.host 文件。

•主机包应由用户从配置菜单中明确选择。 在这种情况下，创建一个 Config.in。 该主机包的主机文件：

```
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/
```

与 Config.in 文件相同的编码风格和选项是有效的。

最后，您必须将新的 libfoo/Config.in.host 添加到 package/Config.in.host 中。 包含在其中的文件按字母顺序排列，除了包的裸名外，不应该包含任何东西。

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

然后主机包将从 Host Utility 菜单中可用。

17.2.3 选择取决于或选择

您的包的 Config.in 文件还必须确保启用相关性。 通常，Buildroot 使用以下规则：

•使用选择类型的依赖关系对库的依赖。 这些依赖关系一般不明显，因此，使 kconfig 系统确保选择依赖关系是有意义的。 例如，libgtk2 包使用 sele ct BR2_PACKAGE_LIBGLIB2 来确保此库也被启用。 select 关键字表示具有反向语义的依赖关系。

•当用户真正需要了解依赖关系时，使用依赖关系的类型。 通常，Buildroot 使用这种依赖关系来依赖于目标架构，MMU 支持和工具链选项（见第 17.2.4 节），或者依赖于“大”事物，如 X.org 系统。 依赖关键词表达了与前向语义的依赖关系。

注意 kconfig 语言的当前问题是这两个依赖关系语义不是内部链接的。 因此，可能会选择一个不符合其依赖/要求的包。

一个例子说明了 select 的使用和依赖。

```
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，这两个依赖关系类型只能与相同类型的依赖关系传递。

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
```

•如果选择了软件包 B，则选择软件包 C 将会可见，只有当软件包 A 被选中时，才会显示该软件包 C。

•选择软件包 E 将选择软件包 D，它将选择软件包 B，它不会检查软件包 B 的依赖关系，因此不会选择软件包 A.

•由于程序包 B 已被选择但程序包 A 不存在，这违反程序包 B 对程序包 A 的依赖性。因此，在这种情况下，必须显式添加传递依赖关系：

```
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
```

总的来说，对于软件包库依赖性，选择应该是首选的。

请注意，这些依赖关系将确保依赖关系选项也启用，但不一定构建在您的包之前。

为此，还需要在包的.mk 文件中表示依赖关系。

进一步格式化细节：参见编码风格第 15.1 节。

17.2.4 对目标和工具链选项的依赖

许多软件包依赖于工具链的某些选项：选择 C 库，C ++支持，线程支持，RPC 支持，wchar 支持或动态库支持。某些软件包只能在某些目标架构上构建，或者 MMU 在处理器中可用。必须使用适当的依赖关系来表达这些依赖关系，这取决于 Config.in 文件中的语句。此外，对于工具链选项的依赖，当该选项未启用时，应显示注释，以便用户知道软件包为何不可用。对目标架构或 MMU 支持的依赖不应该在注释中显示：由于用户不太可能自由选择另一个目标，所以显式显示这些依赖关系是没有意义的。

如果配置选项本身在满足工具链选项依赖性时可见，则该注释应该是可见的。这意味着必须在注释定义上重复该包的所有其他依赖关系（包括对目标体系结构和 MMU 支持的依赖）。为了保持清楚，对于这些非工具链选项的依赖声明应与工具链选项的依赖语句保持分开。如果对同一文件（通常是主包）中的配置选项有依赖关系，则最好具有全局 if ... endif 构造，而不是重复对注释和其他配置选项的依赖语句。

软件包 foo 的依赖注释的一般格式是：

```
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，此文本的目的是简短的，以便它适合于 80 个字符的终端。

本节的其余部分列举了不同的目标和工具链选项，要依赖的相应配置符号以及注释中要使用的文本。

•目标架构

- 依赖符号：BR2_powerpc，BR2_mips，。。。 （参见 arch / Config.in）
- 评论字符串：没有评论被添加

•MMU 支持

- 依赖符号：BR2_USE_MMU
- 评论字符串：没有评论被添加

•用于原子操作的 Gcc _sync *内置函数。 它们可用于 1 字节，2 字节，4 字节和 8 字节的变体。

由于不同的架构支持不同大小的原子操作，因此每个大小都有一个依赖符号:

- 相关性符号：1 个字节的 BR2_TOOLCHAIN_HAS_SYNC_1，2 个字节的 BR2_TOOLCHAIN_HAS_SYNC_2，4 个字节的 BR2 _TOOLCHAIN_HAS_SYNC_4，8 个字节的 BR2_TOOLCHAIN_HAS_SYNC_8。
- 评论字符串：没有评论被添加

•用于原子操作的 Gcc _atomic *内置函数。

- 依赖符号：BR2_TOOLCHAIN_HAS_ATOMIC。
- 评论字符串：没有评论被添加

•内核头

- 依赖符号: BR2_TOOLCHAIN_HEADERS_AT_LEAST_X_Y,（用适当的版本替换 X_Y，请参阅 toolchain / toolchain-common.in）
- 注释字符串：headers> = X.Y 和/或标题<= X.Y（用适当的版本替换 X.Y）

•GCC 版本

- 依赖符号：BR2_TOOLCHAIN_GCC_AT_LEAST_X_Y,（用正确的版本替换 X_Y，请参阅工具 ain / toolchain-common.in）
- 注释字符串：gcc> = X.Y 和/或 gcc <= X.Y（用适当的版本替换 X.Y）

•主机 GCC 版本

- 依赖符号：BR2_HOST_GCC_AT_LEAST_X_Y,（用正确的版本替换 X_Y，请参阅 Config.in）
- 评论字符串：没有评论被添加
- 请注意，封装本身通常不具有最小的主机 GCC 版本，而是一个取决于其的主机包。

•C 库

- 依赖符号：BR2_TOOLCHAIN_USES_GLIBC，BR2_TOOLCHAIN_USES_MUSL，BR2_TOOLCHAIN_USES_uClibc 的
- 注释字符串：对于 C 库，使用稍微不同的注释文本：foo 需要一个 glibc 工具链，或者 foo 需要一个 glibc 工具链 w / C ++

•C ++支持

- 依赖符号：BR2_INSTALL_LIBSTDCPP
- 注释字符串：C ++

•Fortran 支持

- 依赖符号：BR2_TOOLCHAIN_HAS_FORTRAN
- 评论字符串：fortran

•线程支持

- 依赖符号：BR2_TOOLCHAIN_HAS_THREADS
- 注释字符串：threads（除非还需要 BR2_TOOLCHAIN_HAS_THREADS_NPTL，在这种情况下，仅指定 NPTL 就足够了）

•NPTL 线程支持

- 依赖符号：BR2_TOOLCHAIN_HAS_THREADS_NPTL
- 评论字符串：NPTL

•RPC 支持

- 依赖符号：BR2_TOOLCHAIN_HAS_NATIVE_RPC
- 注释字符串：RPC

•wchar 支持

- 依赖符号：BR2_USE_WCHAR
- 评论字符串：wchar

•动态库

- 依赖符号：!BR2_STATIC_LIBS
- 注释字符串：动态库

foo needs a Linux kernel to be built

如果对工具链选项和 Linux 内核有依赖关系，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

如果一个包需要 udev / dev 管理，它应该取决于符号 BR2_PACKAGE_HAS_UDEV，并且应该添加以下内容：

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

如果对工具链选项和 udev / dev 管理有依赖关系，请使用以下格式：

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

17.2.7 虚拟包提供的功能的依赖

一些功能可以由多个包提供，例如 openGL 库。

有关虚拟软件包的更多信息，请参阅[simpara]。

17.3 .mk 文件

最后，这是最难的部分。创建一个名为 libfoo.mk 的文件。它描述了如何下载，配置，构建，安装等包。

根据软件包类型，.mk 文件必须以不同的方式编写，使用不同的基础架构：

•通用软件包的 Makefile（不使用自动工具或 CMake）：这些基于类似于用于基于自动工具的软件包的基础架构，但需要开发人员更多的工作。它们规定了包的配置，编译和安装应该做什么。这种基础设施必须用于不使用自动工具作为其构建系统的所有软件包。将来可能会为其他构建系统编写其他专门的基础设施。我们在教程第 17.5.1 节和参考文献 17.5.2 中介绍了它们。

•用于基于自动工具的软件（autoconf，automake 等）的 Makefile：我们为这些软件包提供专用的基础架构，因为自动工具是非常常见的构建系统。这种基础设施必须用于依赖自动工具作为构建系统的新软件包。我们通过教程 17.6.1 节和参考 17.6.2 节来介绍它们。

•用于基于 cmake 的软件的 Makefile：我们为这样的软件包提供专用的基础架构，因为 CMake 是一个越来越常用的构建系统，并具有标准化的行为。这种基础设施必须用于依赖于 CMake 的新软件包。我们通过教程 17.7.1 节和参考 17.7.2 节来介绍它们。

•Python 模块的 Makefile：我们为 Python 模块提供专用的基础架构，可以使用 distutils 或 setuptools 机制。我们通过教程 17.8.1 和参考部分 17.8.2 来介绍它们。

•Lua 模块的 Makefile：我们通过 LuaRocks 网站提供 Lua 模块专用基础架构。我们通过教程 17.9.1 节和参考部分 17.9.2 来介绍它们。

进一步的格式细节：见第 15.2 节的写作规则。

17.4 .hash 文件

如果可能，您必须添加名为 libfoo.hash 的第三个文件，该文件包含 libfoo 软件包下载的文件的哈希值。没有添加.hash 文件的唯一原因是，由于如何下载包，哈希检查是不可能的。
存储在该文件中的散列用于验证下载的文件和许可证文件的完整性。

该文件的格式为每个文件的一行，用于检查散列，每一行都是空格分隔的，具有以下三个字段：

•哈希类型，其中之一：

- md5，sha1，sha224，sha256，sha384，sha512，none

•文件的散列：

- 没有，一个或多个非空格字符，通常只是字符串 xxx
- 对于 md5，32 个十六进制字符
- 用于 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
- 用于 sha384，96 个十六进制字符

- 用于 sha512，128 个十六进制字符

•文件的名称:

- 对于源存档：文件的基本名称，没有任何目录组件，
- 许可证文件：出现在 FOO_LICENSE_FILES 中的路径。

以 # 号开始的行被视为注释，并被忽略。空行被忽略。

单个文件可以有多个哈希，每个文件都在其自己的行上。在这种情况下，所有散列必须匹配。

注意理想情况下，存储在此文件中的哈希应与上游发布的哈希匹配。在他们的网站上，在电子邮件公告。 。 。如果上游提供多种类型的散列（例如 sha1 和 sha512），那么最好在.hash 文件中添加所有这些散列。如果上游没有提供任何哈希，或者只提供 md5 哈希，那么自己计算至少一个强哈希（最好是 sha256，而不是 md5），并且在哈希上方的注释行中提到这一点。

注意许可证文件的哈希值用于在程序包版本被碰撞时检测许可证更改。 对于具有多个版本（如 Qt5）的包，请在该包的子目录<packageversion>中创建哈希文件（另见第 18.2 节）。

注意空格的数量并不重要，因此可以使用空格（或制表符）来正确对齐不同的字段。

无哈希类型保留给从存储库下载的存档，如 git clone，subversion checkout。。。

下面的示例定义了主 libLoo-1.2.3.tar.bz2 tarball，上游的 md5 和二进制 blob 的本地计算的 sha256 哈希，用于下载的补丁的 sha256 的上游的 sha1 和 sha256，以及 没有哈希的档案:

```
# 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.hash 文件存在，并且它包含一个或多个下载文件的哈希值，则 Buildroot 计算的哈希（下载后）必须与存储在.hash 文件中的哈希匹配。如果一个或多个哈希值不匹配，则 Buildroot 会将其视为错误，删除已下载的文件并中止。

如果.hash 文件存在，但它不包含用于下载文件的哈希，则 Buildroot 会将其视为错误并中止。但是，下载的文件保留在下载目录中，因为这通常表示.hash 文件错误，但是下载的文件可能正常。

目前，检查从 http / ftp 服务器，Git 存储库，使用 scp 和本地文件复制的文件的哈希。因为当从这样的版本控制系统中获取源代码时，Buildroot 目前不会生成可重复的压缩包，所以不会检查其他版本控制系统（如 Subversion，CVS 等）的哈希。

只能在.hash 文件中添加哈希值，以确保稳定的文件。例如，Github 自动生成的补丁不能保证稳定，因此它们的哈希可以随着时间而改变。这样的补丁不应该被下载，而是在本地添加到包文件夹。

如果缺少.hash 文件，那么根本不进行任何检查。

17.5 具有特定构建系统的软件包的基础架构

通过具有特定构建系统的软件包，我们意味着其构建系统不是标准系统之一的所有软件包，例如 autotools 或 CMake。这通常包括其构建系统基于手写的 Makefile 或 shell 脚本的包。

17.5.1 通用包教程

```
1:  ################################################################################
2:  #
3:  # libfoo
4:  #
5:  ################################################################################
6:
7:  LIBFOO_VERSION = 1.0
8:  LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
9:  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))
```

Makefile 从第 7 行到第 11 行开始，包含元数据信息：软件包版本（LIBFOO_VERSION），包含该软件包（LIBFOO_SOURCE）的 tarball 的名称（推荐 xz-ed tarball）可以从其下载 tarball 的 Internet 位置（LIBFOO_SITE），许可证（LIBFOO_LICENSE）和具有许可证文本（LIBFOO_LICENSE_FILES）的文件。在这种情况下，所有变量必须以相同的前缀 LIBFOO_开头。该前缀始终是软件包名称的高级版本（请参阅下文以了解软件包名称的定义位置）。

在第 12 行，我们指定这个包想要安装某些东西到分段空间。库通常需要这样做，因为它们必须在分段空间中安装头文件和其他开发文件。这将确保 LIBFOO_INSTALL_STAGING_CMDS 变量中列出的命令将被执行。

在第 13 行，我们指定对 LIB FOO_INSTALL_STAGING_CMDS 阶段期间安装的某些 libfoo-config 文件进行了一些修复。这些* -config 文件是位于$（STAG-ING_DIR）/ usr / bin 目录中的可执行 shell 脚本文件，由其他第三方软件包执行，以查找该特定软件包的位置和链接标志。
问题是，默认情况下，所有这些* -config 文件不正确，主机系统链接标志不适合交叉编译。
例如：-I / usr / include 而不是-I $（STAGING_DIR）/ usr / include 或：-L / usr / lib 而不是-L $（STAGING_DIR）/usr/lib

所以对这些脚本做了一些 sed 魔术，使它们给出正确的标志。给予 LIBFOO_CONFIG_S CRIPTS 的参数是需要修复的 shell 脚本的文件名。所有这些名称都与$（STAGING_DIR）/usr/bin 相关，如果需要，可以给出多个名称。

此外，LIBFOO_CONFIG_SCRIPTS 中列出的脚本从$（TARGET_DIR）/usr/bin 中删除，因为它们不需要在目标上。

例 17.1 配置脚本： divine package

```
Package divine installs shell script $(STAGING_DIR)/usr/bin/divine-config.
```

可以修改为：

```
DIVINE_CONFIG_SCRIPTS = divine-config
```

例 17.2 配置脚本：imagemagick package

```
Package imagemagick installs the following scripts:
$(STAGING_DIR)/usr/bin/{Magick,Magick++,MagickCore,MagickWand,Wand}-config
```

可以修改为：

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

在第 14 行，我们指定此包依赖的依赖关系列表。这些依赖关系以小写包名称列出，它们可以是目标（不含主机前缀）或主机（具有 host-）前缀的包）的包。 Buildroot 将确保在当前程序包启动其配置之前构建和安装所有这些程序包。

Makefile 的其余部分，第 16..29 行定义了在程序包配置，编译和安装的不同步骤中应该做什么。LIBFOO_BUILD_CMDS 告诉您应该执行哪些步骤来构建程序包。 LIBFOO_INSTALL_STAGING_CMDS 告诉您在分段空间中安装软件包时应执行哪些步骤。 LIBFOO_INSTALL_TARGET_CMDS 告诉您在目标空间中安装软件包时应执行哪些步骤。

所有这些步骤依赖于$（@ D）变量，其中包含已经提取了包的源代码的目录。

在第 31..43 行，我们定义了该包使用的用户（例如，以非 root 方式运行守护程序）（LIBFOO_USERS）。

在第 35..37 行，我们定义了一个设备节点文件（LIBFOO_DEVICES）。

在第 39..41 行，我们定义了由此包安装的特定文件（LIBFOO_PERMISSIONS）设置的权限。

最后，在第 43 行，我们调用 generic-package 函数，它根据前面定义的变量生成所有 Makefile 代码，使你的包工作。

17.5.2 通用包参考

通用目标有两种变体。通用包宏用于为目标交叉编译的包。 host-generic-package 宏用于主机包，为主机本机编译。可以在单个.mk 文件中调用它们：一次创建规则以生成目标包，一次创建生成主机包的规则：

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

如果目标包的编译需要在主机上安装一些工具，这可能很有用。如果包名是 libfoo，那么目标的包的名称也是 libfoo，而主机的包的名称是 host-libfoo。这些名称应该在其他软件包的 DEPENDENCIES 变量中使用，如果它们依赖于 libfoo 或 host-libfoo。

在所有变量定义之后，对 generic-package 和/或 host-generic-package 宏的调用必须位于.mk 文件的末尾。

对于目标包，generic-package 使用由.mk 文件定义的变量，前缀为大写的包名：LIBFOO_ *。 host-generic-package 使用 HOST_LIBFOO_ *变量。对于某些变量，如果 HOST_LIBFOO_前缀变量不存在，则包基础架构使用以 LIBF OO_为前缀的相应变量。对于目标和主机包可能具有相同值的变量，这是完成的。详见下文。

可以在.mk 文件中设置以提供元数据信息的变量列表（假定包名为 libfoo）：

•LIBFOO_VERSION 必须包含该包的版本。请注意，如果 HOST_LIBFOO_VERSION 不存在，则假定它与 LIBFOO_VERSION 相同。它也可以是直接从其版本控制系统中获取的软件包的修订版本，分支或标签。

例子：

```
LIBFOO_VERSION =0.1.2
LIBFOO_VERSION =cb9d6aa9429e838f0e54faa3d455bcbab5eef057
LIBFOO_VERSION =stable
```

•LIBFOO_SOURCE 可能包含该包的 tarball 的名称，Buildroot 将使用该名称从 LIBFOO_SITE 下载 tarball。如果未指定 HOST_LIBFOO_SOURCE，则默认为 LIBFOO_SOURCE。如果没有指定，则该值假定为 libfoo - $（LIBFOO_VERSION）.tar.gz。

示例：LIBFOO_SOURCE = foobar - $（LIBFOO_VERSION）.tar.bz2

•LIBFOO_PATCH 可能包含一个空格分隔的补丁文件名列表，Buildroot 将下载并应用到软件包源代码。如果一个条目包含：//，那么 Buildroot 将假定它是一个完整的 URL，并从该位置下载修补程序。否则，Buildroot 会假设补丁应该从 LIBFOO_SITE 下载。如果未指定 HOST_LIBFOO_PATCH，则默认为 LIBFOO_PATCH。请注意，Buildroot 中包含的修补程序本身使用不同的机制：Buildroot 中的包目录中存在的所有形式的* .patch 文件将在提取后应用于该包（请参阅第 18 章修补包）。最后，LIBFOO_PATCH 变量中列出的修补程序将应用于存储在 Buildroot 软件包目录中的补丁。

•LIBFOO_SITE 提供程序包的位置，可以是 URL 或本地文件系统路径。 HTTP，FTP 和 SCP 是用于检索包 tarball 的支持的 URL 类型。在这些情况下，不包括尾部斜杠：它将由 Buildroot 在目录和文件名之间根据需要添加。 Git，Subversion，Mercurial 和 Bazaar 是用于直接从源代码管理系统检索包的 URL 类型。有一个帮助功能，可以更容易地从 GitHub 下载源 tarball（详见 17.20.3 节）。文件系统路径可用于指定 tarball 或包含包源代码的目录。有关检索的工作原理的详细信息，请参阅下面的 LIBFOO_SITE_METHOD。请注意，SCP URL 的格式应为 scp：// [user @] host：filepath，该文件路径是相对于用户的主目录，因此您可能希望使用斜杠为绝对路径添加路径：scp：/ / [user @] host：/absolute path。

如果未指定 HOST_LIBFOO_SITE，则默认为 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 是一个空格分隔的附加选项列表，用于传递给下载器。对于使用服务器端检查用户登录和密码或使用代理检索文档非常有用。支持 LIBFOO_SITE _METHOD 有效的所有下载方法;有效选项取决于下载方法（请参阅相关下载实用程序的手册页）。

•LIBFOO_EXTRA_DOWNLOADS 是 Buildroot 应下载的附加文件的空格分隔列表。如果一个条目包含：//然后 Buildroot 会认为它是一个完整的 URL，并使用此 URL 下载该文件。否则，Buildroot 将假定要下载的文件位于 LIBFOO_SITE。 Buildroot 不会对这些附加文件执行任何操作，除了下载它们：它将由$（DL_DIR）使用它们的包配方来完成。

•LIBFOO_SITE_METHOD 确定用于获取或复制包源代码的方法。在很多情况下，Buildroot 从 LIBFOO_SITE 的内容中猜出该方法，并且不需要设置 LIBFOO_SITE_METHOD。当 HOST _LIBFOO_SITE_METHOD 未指定时，默认值为 LIBFOO_SITE_METHOD。

LIBFOO_SITE_METHOD 的可能值为：

- 用于正常 FTP / HTTP 下载 tarball 的 wget。 当 LIBFOO_SITE 以 http：//，https：//或 ftp：//

开头时，默认使用。

- scp 通过 SSH 下载 tarballs。 当 LIBFOO_SITE 以 scp：//开始时，默认使用。

- svn 用于从 Subversion 存储库检索源代码。 当 LIBFOO_SITE 以 svn 开始时，默认使用：/ /。当在 LIBFOO_SITE 中指定 http：// Subversion 存储库 URL 时，必须指定 LIBFOO_SITE_MET HOD = svn。 Buildroot 执行一个检出，保存为下载缓存中的 tarball；随后的构建使用 tarball 而不是执行另一个检出。

- cvs 用于从 CVS 存储库检索源代码。 当 LIBFOO_SITE 以 cvs：//开头时，默认使用。 下载的源代码与 svn 方法一样被缓存。假设在 LIBFOO_SITE 上明确地定义匿名 pserver 模式。LIBFOO_SITE = cvs://libfoo.net:/cvsroot/ libfoo 和 LIBFOO_SITE =cvs://:ext:libfoo.net:/cvsroot/libfoo 都是可以接受的，在前一个匿名 pserver 访问模式下是等效的。 LIBFOO_SITE 必须包含源 URL 以及远程存储库目录。该模块是包名称 LIBFOO_VERSION 是强制性的，必须是标签，分支或日期（例如“2014-10-20”，“2014-10-20 13:45”，“2014-10-20 13：45 + 01”，请参见“man cvs“更多细节）。

- git 用于从 Git 存储库检索源代码。当 LIBFOO_SITE 以 git：//开头时，默认使用。下载的源代码与 svn 方法一样被缓存。

- 用于从 Mercurial 存储库检索源代码。当 LIBFO O_SITE 包含 Mercurial 存储库 URL 时，必须指定 LIBFOO_SITE_METHOD = hg。下载的源代码与 svn 方法一样被缓存。

- bzr 用于从 Bazaar 存储库检索源代码。当 LIBFOO_SITE 以 bzr：//开头时，默认使用。下载的源代码与 svn 方法一样被缓存。

- 本地压缩文件。当 LIBFOO_SITE 将包 tarball 指定为本地文件名时，应使用此选项。对于不公开或版本控制的软件非常有用。

- 本地的本地源代码目录。当 LIBFOO_SITE 指定包含包源代码的本地目录路径时，应该使用它。Buildroot 将源目录的内容复制到程序包的构建目录中。请注意，对于本地软件包，不应用任何修补程序。如果您还需要修补源代码，请使用 LIBFOO_POST_RSYNC_ HOOKS，请参见第 17.18.1 节。

•可以将 LIBFOO_GIT_SUBMODULES 设置为 YES，以使用存储库中的 git 子模块创建归档。这仅适用于使用 git 下载的软件包（即当 LIBFOO_SITE_METHOD = git 时）。请注意，当它们包含捆绑

库时，我们尽量不要使用这样的 git 子模块，在这种情况下，我们更喜欢从自己的包中使用这些库。

•LIBFOO_STRIP_COMPONENTS 是提取时 tar 必须从文件名中剥离的主要组件（目录）的数量。大多数软件包的 tarball 有一个名为“<pkg-name> - <pkg-version>”的主要组件，因此 Buildroot 将--strip-components = 1 传递给 tar 以将其删除。对于不具有此组件或具有多个前导组件来剥离的非标准软件包，请将此变量设置为要传递给 tar 的值。默认值：1。

•LIBFOO_EXCLUDES 是提取存档时要排除的模式的空格分隔列表。该列表中的每个项都作为 tar 的--exclude 选项传递。默认为空。

•LIBFOO_DEPENDENCIES 列出了当前目标包编译所需的依赖关系（以包名称表示）。这些依赖项保证在当前包的配置启动之前进行编译和安装。以类似的方式，HOST_LIBFOO_DEPENDENCIES 列出了当前主机包的依赖关系。

•LIBFOO_PATCH_DEPENDENCIES 列出了要修补的当前打包时间所需的依赖关系（以包名称表示）。这些依赖关系在当前包被修补之前被保证被提取和修补。以类似的方式，HOST_LIBFOO_PATCH_DEPENDENCIES 列出了当前主机包的依赖关系。这很少使用;通常，LIBFOO_DEPENDENCIES 是您真正想要使用的。

•LIBFOO_PROVIDES 列出了 libfoo 实现的所有虚拟包。见[simpara]。

•LIBFOO_INSTALL_STAGING 可以设置为 YES 或 NO（默认）。如果设置为 YES，则执行 LIBFOO_INSTALL_STAGING_CMDS 变量中的命令以将软件包安装到暂存目录中。

•LIBFOO_INSTALL_TARGET 可以设置为 YES（默认）或 NO。如果设置为 YES，则执行 LIBFOO_INSTAL L_TARGET_CMDS 变量中的命令将程序包安装到目标目录中。

•LIBFOO_INSTALL_IMAGES 可以设置为 YES 或 NO（默认）。如果设置为 YES，则执行 LIBFOO_INSTALL_IMAGES_CMDS 变量中的命令将程序包安装到 images 目录中。

•LIBFOO_CONFIG_SCRIPTS 列出$（STAGING_DIR）/ usr / bin 中需要进行特殊修复的文件的名称，以使它们交叉编译成为友好。可以给出以空格分隔的多个文件名，并且都与$（STAG-ING_DIR）/ usr / bin 相关。 LIBFOO_CONFIG_SCRIPTS 中列出的文件也从$（TARGET_DIR）/ usr / bin 中删除，因为它们不需要在目标上。

•LIBFOO_DEVICES 列出使用静态设备表时由 Buildroot 创建的设备文件。 使用的语法是 makedevs。 您可以在第 23 章中找到此语法的一些文档。此变量是可选的。

•LIBFOO_PERMISSIONS 列出了在构建过程结束时要完成的权限更改。 语法再次是 makedevs 的语法。 您可以在第 23 章中找到此语法的一些文档。此变量是可选的。

•如果要安装要作为特定用户运行的程序（例如作为守护程序或作为 cron-job），LIBFOO_USERS 将列出为此程序包创建的用户。语法在精神上与 makedevs 相似，并在第 24 章中描述。此变量是可选的。

•LIBFOO_LICENSE 定义软件包被释放的许可证（或许可证）。此名称将显示在由法律信息生成的清单文件中。如果许可证出现在 SPDX 许可证列表中，请使用 SPDX 短标识符使清单文件统一。否则，以精确和简明的方式描述许可证，避免实际上命名一个许可证系列的不明确的名称，如 BSD。此变量是可选的。如果未定义，未知将显示在该包的清单文件的许可证字段中。

此变量的预期格式必须符合以下规则:

- 如果包的不同部分在不同的许可证下被释放，那么逗号分开许可证（例如 LIBFOO_LICE NSE = GPL-2.0 +，LGPL-2.1 +）。如果明确区分哪个组件在许可证下获得许可，则在该括号之间（例如 LIBFOO_LICENSE = GPL-2.0 +（程序 s），LGPL-2.1 +（库））中注明该组件的许可证。

- 如果包是双重许可的，则使用或关键字（例如 LIBFOO_LICENSE = AFL-2.1 或 GPL-2.0 +）分开许可证。

•LIBFOO_LICENSE_FILES 是包 tarball 中包含释放包的许可证的空格分隔的文件列表。使法律信息在 legal-info 目录中复制所有这些文件。有关更多信息，请参阅第 12 章。此变量是可选的。如果没有定义，将会产生警告，让您知道，并且不保存将显示在此软件包的清单文件的许可证文件字段中。

•LIBFOO_ACTUAL_SOURCE_TARBALL 仅适用于其 LIBFOO_SITE / LIBTOO_SOURCE 对指向不包含源代码但不包含二进制代码的归档的软件包。这是一个非常罕见的情况，只知道应用于已经编译的外部工具链，但理论上它可能适用于其他软件包。在这种情况下，实际的源代码通常可以提供单独的 tarball。将 LIBFOO_ACTUAL_SOURCE_TARBALL 设置为实际源代码归档的名称，Buildroot 将下载该文件，并在运行 make legal-info 收集与法律相关的资料时使用它。注意，这个文件将不会在常规构建和源代码下载。

•LIBFOO_ACTUAL_SOURCE_SITE 提供实际源压缩包的位置。默认值为 LIBFOO_SITE，因此如果二进制和源存档托管在同一目录下，则不需要设置此变量。如果未设置 LIBFOO_ACTUAL_SOURCE_TARBALL，则定义 LIBFOO_ACTUAL_SOURCE_SITE 是没有意义的。

•LIBFOO_REDISTRIBUTE 可以设置为 YES（默认值）或 NO，以指示是否允许重新分配软件包源代码。 将其设置为非开源软件包：Buildroot 收集法律信息时不会保存此软件包的源代码。

•LIBFOO_FLAT_STACKSIZE 定义了 FLAT 二进制格式内置的应用程序的堆栈大小。NOMMU 架构处理器上的应用程序堆栈大小无法在运行时放大。 FLAT 二进制格式的默认堆栈大小只有 4k 字

节。 如果应用程序消耗更多的堆栈，请在此处附加所需的数字。

建议的定义这些变量的方法是使用以下语法：

```
LIBFOO_VERSION = 2.32
```

现在，定义在构建过程的不同步骤中应该执行什么的变量。

•LIBFOO_EXTRACT_CMDS 列出要执行的解压缩操作。这通常不需要，因为压缩包由 Buildroot 自动处理。但是，如果程序包使用非标准归档格式（例如 ZIP 或 RAR 文件）或具有非标准组织的 tarball，则此变量允许覆盖程序包基础结构的默认行为。

•LIBFOO_CONFIGURE_CMDS 列出了在编译之前配置包的操作。

•LIBFOO_BUILD_CMDS 列出了要编译包的操作。

•HOST_LIBFOO_INSTALL_CMDS 列出了当程序包是主机程序包时要执行的安装程序包的操作。软件包必须将其文件安装到由$（HOST_DIR）指定的目录中。应该安装所有文件，包括头文件等开发文件，因为其他软件包可能会在此软件包的顶部进行编译。

•LIBFOO_INSTALL_TARGET_CMDS 列出了当程序包是目标程序包时要将程序包安装到目标目录时要执行的操作。软件包必须将其文件安装到由$（TARGET_DIR）给出的目录中。必须安装执行程序包所需的文件。当目标文件系统完成时，头文件，静态库和文档将被再次删除。

•LIBFOO_INSTALL_STAGING_CMDS 列出了当程序包是目标程序包时要将程序包安装到临时目录时要执行的操作。软件包必须将其文件安装到由$（STAGING_DIR）给出的目录中。应该安装所有开发文件，因为它们可能需要编译其他软件包。

•LIBFOO_INSTALL_IMAGES_CMDS 列出了当程序包是目标程序包时，要将程序包安装到 images 目录时要执行的操作。软件包必须将其文件安装到由$（BINARIES_DIR）给出的目录中。只有不属于 TARGET_DIR，但是引导电路板所必需的二进制映像（也称为映像）的文件应放在此处。例如，如果一个包具有类似于内核映像，引导加载程序或根文件系统映像的二进制文件，则应该使用此步骤。

•LIBFOO_INSTALL_INIT_SYSV 和 LIBFOO_INSTALL_INIT_SYSTEMD 列出了为系统 V 类 init 系统（busybox，sysvinit 等）或 systemd 设备安装 init 脚本的操作。只有在安装相关的 init 系统时才会运行这些命令（即如果在配置中选择 systemd 作为 init 系统，则只运行 LIBFOO_INSTALL_INIT_SYSTEMD）。

•LIBFOO_HELP_CMDS 列出了打印包帮助的操作，其中包含在主要帮助输出中。这些命令可以打印任何格式的任何东西。这很少使用，因为包很少有自定义规则。不要使用这个变量，除非你真

的知道你需要打印帮助。

定义这些变量的首选方法是：

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

在动作定义中，可以使用以下变量：

•$（LIBFOO_PKGDIR）包含包含 libfoo.mk 和 Config.in 文件的目录的路径。当需要安装 Buildroot 中捆绑的文件时，该变量很有用，如运行时配置文件，闪屏图像。 。 。

•$（@ D），其中包含程序包源代码未解压缩的目录。

•$（DL_DIR）包含存储 Buildroot 所有下载的目录的路径。

•$（TARGET_CC），$（TARGET_LD）等，以获取目标交叉编译实用程序

•$（TARGET_CROSS）获取交叉编译工具链前缀

•当然，$（HOST_DIR），$（STAGING_DIR）和$（TARGET_DIR）变量正确安装包。

最后，你也可以使用钩子。有关详细信息，请参见第 17.18 节。

17.6 基于自动工具的软件包的基础设施

17.6.1 autotools-package 教程

首先，我们来看一下如何为一个基于自动工具的软件包编写一个.mk 文件，例如：

```
1:  ################################################################################
2:  #
3:  # libfoo
4:  #
5:  ################################################################################
6:
7:  LIBFOO_VERSION = 1.0
8:  LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
```

```
9:  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))
```

在第 7 行，我们声明包的版本。

在第 8 行和第 9 行，我们声明 tarball 的名称（xz-ed tarball 推荐）和 tarball 在 Web 上的位置。

Buildroot 会自动从这个位置下载 tarball。

在第 10 行，我们告诉 Buildroot 将软件包安装到登台目录。位于 output/staging/ 中的登台目录是安装所有软件包的目录，包括其开发文件等。默认情况下，软件包不会安装到登台目录，因为通常只需要安装库分期目录：他们的开发文件需要根据它们编译其他库或应用程序。默认情况下，当启用分段安装时，使用 make install 命令将软件包安装在此位置。

在第 11 行，我们告诉 Buildroot 不要将包安装到目标目录。此目录包含将成为目标上运行的根文件系统的内容。对于纯静态库，不需要将它们安装在目标目录中，因为它们不会在运行时使用。默认情况下，目标安装已启用;将此变量设置为 NO 几乎不需要。默认情况下，使用 make install 命令将软件包安装在此位置。

在第 12 行，我们告诉 Buildroot 传递一个自定义配置选项，这将在配置和构建包之前传递给./configure 脚本。

在第 13 行，我们声明我们的依赖关系，以便它们在我们的包的构建过程开始之前构建。

最后，在第 15 行，我们调用 autotools-package 宏，该宏生成实际允许构建程序包的所有 Makefile 规则。

17.6.2 autotools-package 引用

autotools 包基础设施的主要宏是 autotools-package。它类似于通用包宏。使用 host-autotools-package 宏，也可以使用目标和主机包的功能。

就像通用基础架构一样，autotools 基础设施的工作原理是在调用 auto tools-package 宏之前定义一些变量。

首先，通用基础设施中存在的所有包元数据信息变量也存在于自动工具基础设施中：LIBFOO_VERSION， LIBFOO_SOURCE， LIBFOO_PATCH， LIBFOO_SITE， LIBFOO_SUBDIR，LIBFOO_DEPENDE NCIES，LIBFOO_INSTALL_STAGING，LIBFOO_INSTALL_TARGET。

还可以定义一些特定于自动工具基础设施的附加变量。其中许多仅在非常具体的情况下有用，因此典型的包将仅使用其中的几个。

•LIBFOO_SUBDIR 可能包含包含 configure 脚本的包中的子目录的名称。这是有用的，例如，主配置脚本不在 tarball 提取的树的根目录下。如果未指定 HOST_LIBFOO_SUBDIR，则默认为 LIBFOO_SUBDIR。

•LIBFOO_CONF_ENV，用于指定要传递给 configure 脚本的其他环境变量。默认为空。

•LIBFOO_CONF_OPTS，以指定其他配置选项传递给配置脚本。默认为空。

•LIBFOO_MAKE，指定备用的 make 命令。在配置（使用 BR2_JLEVEL）中启用并行使能时，通常使用此功能，但是由于某种原因，应该禁用此功能对于给定的包。默认设置为$（MAKE）。如果程序包不支持并行构建，则应将其设置为 LIBFOO_MAKE = $（MAKE1）。

•LIBFOO_MAKE_ENV，用于指定在构建步骤中要传递的附加环境变量。这些在 make 命令之前传递。默认为空。

•LIBFOO_MAKE_OPTS，用于指定在构建步骤中传递给其他变量。这些在 make 命令之后传递。默认为空。

•LIBFOO_AUTORECONF，告诉软件包是否应该自动配置（即如果配置脚本和 Makefile 文件应该通过重新运行 autoconf，automake，libtool 等重新生成）。有效值为 YES 和 NO。默认值为 NO

•LIBFOO_AUTORECONF_ENV, 用于指定额外的环境变量传递给 autoreconf 程序, 如果 LIBFOO_AUTORECONF = YES。这些在 autoreconf 命令的环境中传递。默认为空。

• LIBFOO_AUTORECONF_OPTS 指定传递给 autoreconf 程序的其他选项，如果 LIBFOO_AUTORECONF = YES。默认为空。

•LIBFOO_GETTEXTIZE，告诉软件包是否应该被取消文字化（即如果软件包使用与 Buildroot 不同的 gettext 版本，并且需要运行 gettextize）。仅当 LIBFOO_AUTORECONF = YES 时才有效。有效值为 YES 和 NO。默认值为 NO。

• LIBFOO_GETTEXTIZE_OPTS，用于指定传递给 gettextize 程序的其他选项，如果

LIBFOO_GETTEXTIZE = YES。您可以使用，例如，.po 文件不在标准位置（即 po /在包的根目录下）。默认情况下，-f。

•LIBFOO_LIBTOOL_PATCH 会告知是否应用 Buildroot 修补程序修复 libtool 交叉编译问题。有效值为 YES 和 NO。默认值为 YES

•LIBFOO_INSTALL_STAGING_OPTS 包含用于将包安装到临时目录的 make 选项。默认情况下，DESTDIR = $（STAGING_DIR）安装，这对于大多数 autotools 包是正确的。它仍然可以覆盖它。

•LIBFOO_INSTALL_TARGET_OPTS 包含用于将包安装到目标目录的 make 选项。默认情况下，值为 DESTDIR = $（TARGET_DIR）安装。大多数自动工具包的默认值是正确的，但如果需要，仍然可以覆盖它。

使用自动工具基础架构，构建和安装软件包所需的所有步骤已经定义，它们通常适用于大多数基于自动工具的软件包。 但是，如果需要，还可以自定义在任何特定步骤中所做的工作:

•通过添加一个后操作钩子（提取，修补，配置，构建或安装之后）。 详见第 17.18 节。

•覆盖其中一个步骤。 例如，即使使用自动工具基础设施，如果包.mk 文件定义了自己的 LIBFOO_CONFIGURE_CMDS 变量，那么将使用它来代替默认的自动工具。 然而，使用这种方法应该限于非常具体的情况。 在一般情况下不要使用它。

17.7 基于 CMake 的软件包的基础设施

17.7.1 cmake-package 教程

首先，我们来看一下如何为一个基于 CMake 的软件包编写一个.mk 文件，例如:

```
1:  ################################################################################
2:  #
3:  # libfoo
4:  #
5:  ################################################################################
6:
7:  LIBFOO_VERSION = 1.0
8:  LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
9:  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))
```

在第 7 行，我们声明包的版本。

在第 8 行和第 9 行，我们声明 tarball 的名称（xz-ed tarball 推荐）和 tarball 在 Web 上的位置。

Buildroot 会自动从这个位置下载 tarball。

在第 10 行，我们告诉 Buildroot 将软件包安装到登台目录。位于 output/staging/ 中的登台目录是安装所有软件包的目录，包括其开发文件等。默认情况下，软件包不会安装到登台目录，因为通常只需要安装库分期目录：他们的开发文件需要根据它们编译其他库或应用程序。默认情况下，当启用分段安装时，使用 make install 命令将软件包安装在此位置。

在第 11 行，我们告诉 Buildroot 不要将包安装到目标目录。此目录包含将成为目标上运行的根文件系统的内容。对于纯静态库，不需要将它们安装在目标目录中，因为它们不会在运行时使用。默认情况下，目标安装已启用;将此变量设置为 NO 几乎不需要。默认情况下，使用 make install 命令将软件包安装在此位置。

在第 12 行，我们告诉 Buildroot 在配置包时将自定义选项传递给 CMake。

在第 13 行，我们声明我们的依赖关系，以便它们在我们的包的构建过程开始之前构建。最后，在第 15 行，我们调用了 cmake-package 宏，该宏生成实际允许构建程序包的所有 Makefile 规则。

17.7.2 cmake-package 参考

CMake 软件包基础架构的主要宏是 cmake-package。它类似于通用包宏。

具有目标和主机软件包的功能也可以使用主机 cmake-package 宏。

就像通用基础设施一样，CMake 基础设施通过在调用 cmake-package 宏之前定义一些变量来工作。

首先，通用基础架构中存在的所有包元数据信息变量也存在于 CMake 基础设施中：LIBFOO_VERSION， LIBFOO_SOURCE， LIBFOO_PATCH， LIBFOO_SITE， LIBFOO_SUBDIR，LIBFOO_DEPENDE NCIES，LIBFOO_INSTALL_STAGING，LIBFOO_INSTALL_TARGET。

还可以定义特定于 CMake 基础架构的一些附加变量。其中许多仅在非常具体的情况下有用，因此典型的包将仅使用其中的几个。

•LIBFOO_SUBDIR 可能包含包含主 CMakeLists.txt 文件的包中的子目录的名称。这是有用的，如果例如，主要的 CMakeLists.txt 文件不在树的根目录提取的 tarball。如果未指定 HOST_LIBFOO_SUBDIR，则默认为 LIBFOO_SUBDIR。

•LIBFOO_CONF_ENV，用于指定要传递给 CMake 的其他环境变量。默认为空。

•LIBFOO_CONF_OPTS, 用于指定要传递给 CMake 的其他配置选项。默认为空。一些常见的 CMake 选项由 cmake-package 基础设施设置;因此通常不需要将它们设置在包的* .mk 文件中，除非您要覆盖它们:

– CMAKE_BUILD_TYPE is driven by BR2_ENABLE_DEBUG;
– CMAKE_INSTALL_PREFIX;
– BUILD_SHARED_LIBS is driven by BR2_STATIC_LIBS;
– BUILD_DOC, BUILD_DOCS are disabled;
– BUILD_EXAMPLE, BUILD_EXAMPLES are disabled;
– BUILD_TEST, BUILD_TESTS, BUILD_TESTING are disabled.

• 当程序包无法在源代码树中构建但需要单独的构建目录时，应设置 LIBFOO_SUPPORTS_IN_SOURCE_BUILD = NO。

•LIBFOO_MAKE，指定备用的 make 命令。在配置（使用 BR2_JLEVEL）中启用并行使能时，通常使用此功能，但是由于某种原因，应该禁用此功能对于给定的包。默认设置为$（MAKE）。如果程序包不支持并行构建，则应将其设置为 LIBFOO_MAKE = $（MAKE1）。

•LIBFOO_MAKE_ENV，用于指定在构建步骤中要传递的附加环境变量。这些在 make 命令之前传递。默认为空。

•LIBFOO_MAKE_OPTS，用于指定在构建步骤中传递给其他变量。这些在 make 命令之后传递。默认为空。

•LIBFOO_INSTALL_STAGING_OPTS 包含用于将包安装到临时目录的 make 选项。默认情况下，值为 DESTDIR = $（STAGING_DIR）install，这对于大多数 CMake 软件包是正确的。它仍然可以覆盖它。

•LIBFOO_INSTALL_TARGET_OPTS 包含用于将包安装到目标目录的 make 选项。默认情况下，值为 DESTDIR = $（TARGET_DIR）安装。大多数 CMake 软件包的默认值是正确的，但如果需要，它仍然可以覆盖它。

使用 CMake 基础架构，构建和安装软件包所需的所有步骤已经定义，并且它们通常适用于大多数基于 CMake 的软件包。但是，如果需要，还可以自定义在任何特定步骤中所做的工作:

•通过添加一个后操作钩子（提取，修补，配置，构建或安装之后）。详见第 17.18 节。

•覆盖其中一个步骤。例如，即使使用 CMake 基础架构，如果包.mk 文件定义了自己的 LIBFOO_CONFIGURE_CMDS 变量，则将使用它来代替默认的 CMake。然而，使用这种方法应该限于非常具体的情况。在一般情况下不要使用它。

17.8 Python 包的基础设施

该基础架构适用于使用标准 Python setuptools 机制作为其构建系统的 Python 包，通常可以通过使用 setup.py 脚本来识别。

17.8.1 python-package 教程

首先，我们来看一下如何为 Python 包编写一个.mk 文件，并附上一个例子：

```
1:  ################################################################################
2:  #
3:  # python-foo
4:  #
5:  ################################################################################
6:
7:  PYTHON_FOO_VERSION = 1.0
8:  LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
9:  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))
```

在第 7 行，我们声明包的版本。

在第 8 行和第 9 行，我们声明 tarball 的名称（xz-ed tarball 推荐）和 tarball 在 Web 上的位置。

Buildroot 会自动从这个位置下载 tarball。

在第 10 行和第 11 行，我们提供有关该软件包的许可详细信息（其第 10 行的许可证以及包含第 11 行许可证文本的文件）。

在第 12 行，我们告诉 Buildroot 在配置包时将自定义选项传递给 Python setup.py 脚本。

在第 13 行，我们声明我们的依赖关系，以便它们在我们的包的构建过程开始之前构建。

在第 14 行，我们声明正在使用的特定 Python 构建系统。 在这种情况下，使用 distutils Python 构建系统。

两个支持的是 distutils 和 setuptools。

最后，在第 16 行，我们调用了生成实际允许构建包的所有 Makefile 规则的 python-package 宏。

17.8.2 python-package 参考

作为一种策略，只需提供 Python 模块的软件包都应该在 Buildroot 中被命名为 python-<something>。使用 Python 构建系统但不是 Python 模块的其他软件包可以自由选择其名称（Buildroot 中的现有示例是 scons 和 supervisor）。

在它们的 Config.in 文件中，它们应该依赖于 BR2_PACKAGE_PYTHON，所以当 Buildroot 将为模块启用 Python 3 使用时，我们将能够在 Python 3 上逐步启用 Python 模块。

Python 包基础架构的主要宏是 python-package。它类似于通用包宏。

也可以使用 host-python-package 宏来创建 Python 主机包。

就像通用基础架构一样，Python 基础设施通过在调用 python-package 或 host-python-package 宏之前定义一些变量来实现。

通用包基础架构中存在的所有包元数据信息变量第 17.5.2 节也存在于 Python 基础架构中：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

注意：

•没有必要在包的 PYTHON_FOO_DEPENDENCIES 变量中添加 python 或 host-python，因为

Python 包基础架构需要自动添加这些基本依赖关系。

•同样，由于根据需要可以自动添加 Python 基础架构，因此无需为基于 setuptools 的软件包将主机设置工具和/或主机 - distutilscross 依赖项添加到 PYTHON_FOO_ DEPENDENCIES。

一个特定于 Python 基础架构的变量是必需的：

•PYTHON_FOO_SETUP_TYPE，用于定义软件包使用的 Python 构建系统。两个支持的值是 distutils 和 setuptools。如果您不知道在程序包中使用哪个程序包，请查看程序包源代码中的 setup.py 文件，并查看是否从 distutils 模块或 setuptools 模块导入内容。

根据软件包的需要，可以根据需要定义一些特定于 Python 基础架构的附加变量。

其中许多仅在非常具体的情况下才有用，因此典型的包将仅使用其中的一些，或者不使用。

•PYTHON_FOO_ENV，用于指定要传递给 Python setup.py 脚本的其他环境变量（对于构建和安装步骤）。请注意，基础架构自动传递几个标准变量，这些变量定义在 PKG_PYTHON_DISTUTILS_ENV（对于 distutils 目标包），HOST_PKG_PYTHON_DISTUTILS_ENV（用于 distutils 主机包），PKG_PYTHON_SETUPTOOLS_ENV（用于 setuptools 目标包）和 HOST_PKG_PYTHON_SETUPTOOLS_ENV（用于 setuptools 主机包）中。

•PYTHON_FOO_BUILD_OPTS，用于指定在构建步骤期间传递给 Python setup.py 脚本的其他选项。对于目标 distutils 软件包，PKG_PYTHON_DISTUTILS_BUILD_OPTS 选项已由基础架构自动传递。

• PYTHON_FOO_INSTALL_TARGET_OPTS，PYTHON_FOO_INSTALL_STAGING_OPTS，HOST_PYTHON_FOO_INSTA LL_OPTS，分别指定在目标安装步骤，安装步骤或主机安装期间传递给 Python setup.py 脚本的其他选项。需要注意的是基础设施自动传递一些选项，在 PKG_PYTHON_DISTUTILS_INSTALL_TARGET_OPTS 或 PKG_PYTHON_DISTUTILS_INSTALL_STAGING_OPTS 定义（为目标的 distutils 包），HOST_PKG_PYTHON_DISTUTILS_INSTALL_OPTS（主机的 distutils 包），PKG_PYTHON_SETUPTOOLS_INSTALL_TARGET_OPTS 或 PKG_PYTHON_SETUPTOOLS_INSTALL_STAGIN G_OPTS（针对目标 setuptools 的包）和 HOST_PKG_PYTHON_SETUPTOOLS_INSTALL_OPTS（主机 setuptools 的包）。

•HOST_PYTHON_FOO_NEEDS_HOST_PYTHON，定义主机 python 解释器。此变量的用法仅限于主机包。两个支持的值是 python2 和 python3。它将确保正确的主机 python 包可用，并将调用它进行构建。如果一些构建步骤重载，则必须在命令中明确调用正确的 python 解释器。

使用 Python 基础架构，构建和安装软件包所需的所有步骤已经定义，它们通常适用于大多数基

于 Python 的软件包。但是，如果需要，还可以自定义在任何特定步骤中所做的工作:

•通过添加一个后操作钩子（提取，修补，配置，构建或安装之后）。详见第 17.18 节。

•覆盖其中一个步骤。例如，即使使用了 Python 基础设施，如果包.mk 文件定义了自己的 PYTHON_FOO_BUILD_CMDS 变量，它将被使用，而不是默认的 Python。然而，使用这种方法应该限于非常具体的情况。在一般情况下不要使用它。

17.8.3 从 PyPI 存储库生成 python-package

如果您希望创建一个 Buildroot 软件包的 Python 软件包在 PyPI 上可用，您可能需要使用 utils /中的 scanpypi 工具来自动化该进程。

您可以在这里找到现有的 PyPI 软件包列表。

scanpypi 需要在主机上安装 Python 的 setuptools 包。

当你的 buildroot 目录的根目录只是做:

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

这将在包文件夹中生成包 python-foo 和 python-bar，如果它们存在于 https://pypi.python.org 上。

找到外部的 python 模块菜单，并将您的软件包插入。 请记住，菜单中的项目应按字母顺序排列。

请记住，您很可能必须手动检查软件包是否有任何错误，因为有些事情不能被发生器猜测（例如，依赖于任何 python 核心模块，如 BR2_PACKAGE_PYTHON_ZLIB）。 另外，请注意，许可证和许可证文件被猜到并且必须被检查。 您还需要手动将包添加到 package / Config.in 文件中。

如果您的 Buildroot 软件包不在官方 Buildroot 树中，而是在一个 br2 外部树中，请使用-o 标志如下:

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

这将在 other_package_directory 而不是包中生成包 python-foo 和 python-bar。

选项-h 将列出可用选项:

```
utils/scanpypi -h
```

17.8.4 python-package CFFI 后端

C Python 的外部函数接口（CFFI）提供了一种方便可靠的方法，可以使用 C 编写的接口声明从 Python 调用编译的 C 代码。依赖该后端的 Python 包可以通过在 install_requires 字段中出现 cffi 依赖关系来标识 他们的 setup.py 文件。

这样一个包应该:

•将 python-cffi 添加为运行时依赖关系，以便在目标上安装编译的 C 库包装器。 这通过向包 Config.in 中添加选择 BR2_PACKAGE_PYTHON_CFFI 来实现。

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

•添加 host-python-cffi 作为构建时依赖关系，以便交叉编译 C 包装器。 这通过在 PYTHON_FOO_DEPENDENCIES 变量中添加 host-python-cffi 来实现。

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

17.9 基于 LuaRock 的软件包的基础设施

17.9.1 luarocks-package 教程

首先，我们来看一下如何为基于 LuaRocks 的程序包编写一个.mk 文件，并提供一个例子:

```
1:  ################################################################################
2:  #
3:  # lua-foo
4:  #
5:  ################################################################################
```

```
6:
7:    LUA_FOO_VERSION = 1.0.2-1
8:    LUA_FOO_NAME_UPSTREAM = foo
9:    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))
```

在第 7 行，我们声明包的版本（与 rockspec 相同，即上游版本和 rockspec 版本的连接，用连字符分隔）。

在第 8 行，我们声明该包在 LuaRocks 上被称为“foo”。 在 Buildroot 中，我们给 Lua 相关软件包一个以“lua”开头的名称，所以 Buildroot 名称与上游名称不同。LUA_FOO_NAME_UPSTREAM 在两个名称之间建立链接。

在第 9 行，我们声明对本机库的依赖关系，以便它们在我们的包的构建过程开始之前构建。

在第 11-12 行，我们告诉 Buildroot 在构建包时将自定义选项传递给 LuaRocks。

在第 13-14 行，我们指定包的许可条款。

最后，在第 16 行，我们调用生成实际允许构建包的所有 Makefile 规则的 luarocks-package 宏。

17.9.2 luarocks-package 参考

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 的上下文中，luarocks-package 基础架构仅支持内置模式。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-packages 基础架构进行打包。

LuaRocks 包基础设施的主要宏是 luarocks-package：像通用包，它通过定义一些变量来提供有关该包的元数据信息，然后调用 luarocks-package。值得一提的是，不支持为主机构建 LuaRocks 软件包，因此宏主机-luarocks-package 未被实现。

就像通用基础设施一样，LuaRocks 基础设施的工作原理是在调用 luarocks-package 宏之前

定义一些变量。

首先，通用基础设施中存在的所有包元数据信息变量也存在于 LuaRocks 基础结构中：LUA_FOO_VERSION， LUA_FOO_SOURCE， LUA_FOO_SITE， LUA_FOO_DEPENDENCIES，LUA_FOO_LIC ENSE，LUA_FOO_LICENSE_FILES。

其中两个由 LuaRocks 基础架构（下载步骤）填充。如果你的包不是托管的 LuaRocks 镜像$（BR2_LUAROCKS_MIRROR），您可以覆盖它们：

•LUA_FOO_SITE，默认为$（BR2_LUAROCKS_MIRROR）

• LUA_FOO_SOURCE ， 默 认 为 $ （ 小 写 LUA_FOO_NAME_UPSTREAM ） - $（LUA_FOO_VERSION）.src.rock

还定义了一些特定于 LuaRocks 基础架构的附加变量。在具体情况下可以覆盖它们。

•LUA_FOO_NAME_UPSTREAM，默认为 lua-foo，即 Buildroot 软件包名称

• LUA_FOO_ROCKSPEC ， 默 认 为 $ （ 小 写 LUA_FOO_NAME_UPSTREAM ） - $（LUA_FOO_VERSION）.rockspec

• LUA_FOO_SUBDIR ， 默 认 为 $ （ LUA_FOO_NAME_UPSTREAM ） - $（LUA_FOO_VERSION_WITHOUT_ROCKSP EC_REVISION）

•LUA_FOO_BUILD_OPTS 包含 luarocks 构建调用的其他构建选项。

17.10 Perl / CPAN 软件包的基础设施

17.10.1 perl-package 教程

首先，我们来看一下如何为 Perl / CPAN 包编写一个.mk 文件，例如：

```
1:  ################################################################################
2:  #
3:  # perl-foo-bar
4:  #
5:  ################################################################################
6:
7:  PERL_FOO_BAR_VERSION = 0.02
8:  PERL_FOO_BAR_SOURCE = Foo-Bar-$(PERL_FOO_BAR_VERSION).tar.gz
```

```
9:  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:
14: $(eval $(perl-package))
```

在第 7 行，我们声明包的版本。

在第 8 行和第 9 行，我们在 CPAN 服务器上声明 tarball 的名称和 tarball 的位置。 Buildroot 会自动从这个位置下载 tarball。

在第 10 行，我们声明我们的依赖关系，以便它们在我们的包的构建过程开始之前构建。

在第 11 行和第 12 行，我们提供有关该软件包（其第 11 行的许可证以及包含第 12 行许可证文本的文件）的许可详细信息。

最后，在第 14 行，我们调用生成实际允许构建包的所有 Makefile 规则的 perl-package 宏。

大部分数据可以从 https://metacpan.org/ 检索。因此，该文件和 Config.in 可以通过在 Buildroot 目录（或在外部的外部树中）运行脚本 supports / scripts / scancpan Foo-Bar 来生成。此脚本为请求的包创建一个 Config.in 文件和 foo-bar.mk 文件，并递归地为 CPAN 指定的所有依赖项递归。您仍然应该手动编辑结果。特别要检查以下事项。

•如果 perl 模块与由另一（非 perl）软件包提供的共享库链接，则不会自动添加此依赖关系。必须手动添加到 PERL_FOO_BAR_DEPENDENCIES。

•package/Config.in 文件必须手动更新才能包含生成的 Config.in 文件。作为提示，scanc pan 脚本打印出所需的源“...”语句，按字母顺序排列。

17.10.2 perl-package 参考

作为一项政策，提供 Perl / CPAN 模块的软件包都应该在 Buildroot 中命名为 perl-<something>。

该基础设施处理各种 Perl 构建系统：ExtUtils-MakeMaker（EUMM），模块构建（MB）和 Module-Build-Tiny。默认情况下，当包提供 Makefile.PL 和 Build.PL 时，Build.PL 是首选项。

Perl / CPAN 软件包基础设施的主要宏是 perl-package。它类似于通用包宏。

使用 host-perl-package 宏，也可以使用目标和主机包的功能。

就像通用基础架构一样，Perl/CPAN 基础架构通过在调用 perl-package 宏之前定义一些变量来工作。

首先，通用基础架构中存在的所有包元数据信息变量也存在于 Perl / CPAN 基础架构中：PERL_FOO_VERSION，PERL_FOO_SOURCE，PERL_FOO_PATCH，PERL_FOO_SITE，PERL_FOO_SUBDIR，PERL_FOO_DEPENDENCIES，PERL_FOO_INSTALL_TARGET。

请注意，除非定义了 PERL_FOO_INSTALL_STAGING_CMDS 变量，否则将 PERL_FOO_INSTALL_STAGING 设置为 YES 无效。 perl 基础结构没有定义这些命令，因为 Perl 模块通常不需要安装到登台目录。

还可以定义特定于 Perl / CPAN 基础架构的一些附加变量。其中许多仅在非常具体的情况下有用，因此典型的包将仅使用其中的几个。

•PERL_FOO_PREFER_INSTALLER / HOST_PERL_FOO_PREFER_INSTALLER，指定首选的安装方法。可能的值是 EUMM（对于使用 ExtUtils-MakeMaker 的 Makefile.PL 安装）和 MB（对于基于 Build.PL 的使用 Module-Build 的安装）。此变量仅在程序包提供两种安装方法时使用。

•PERL_FOO_CONF_ENV / HOST_PERL_FOO_CONF_ENV，以指定传递给 perl Makefile.PL 或 perl Build.PL 的其他环境变量。默认为空。

•PERL_FOO_CONF_OPTS / HOST_PERL_FOO_CONF_OPTS，以指定其他配置选项传递给 perl Makefile.PL 或 perl Build.PL。默认为空。

•PERL_FOO_BUILD_OPTS / HOST_PERL_FOO_BUILD_OPTS，以指定要在生成步骤中生成 pure_all 或 perl Build 生成的附加选项。默认为空。

•PERL_FOO_INSTALL_TARGET_OPTS，以指定其他选项，以便在安装步骤中进行 pure_install 或 perl Build 安装。默认为空。

•HOST_PERL_FOO_INSTALL_OPTS，以指定其他选项，以便在安装步骤中进行 pure_install 或 perl Build 安装。默认为空。

17.11 虚拟包的基础设施

在 Buildroot 中，虚拟包是一个包，其功能由一个或多个包（称为提供者）提供。

虚拟包管理是一种可扩展的机制，允许用户选择在 rootfs 中使用的提供者。

例如，OpenGL ES 是嵌入式系统中 2D 和 3D 图形的 API。对于 Allwinner Tech Sunxi 和德州

仪器 OMAP35xx 平台，此 API 的实现是不同的。所以 libgles 将是一个虚拟包，sunxi-mali 和 ti-gfx 将是提供者。

17.11.1 虚拟包教程

在下面的示例中，我们将解释如何添加一个新的虚拟包（something-virtual）和一个提供者（some-provider）。

首先，我们来创建虚拟包。

17.11.2 虚拟包的 Config.in 文件

虚拟包的虚拟包的虚拟文件应包含：

```
1: config BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
2:         bool
3:
4:     config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
5:         depends on BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
6:         string
```

在此文件中，我们声明了两个选项：BR2_PACKAGE_HAS_SOMETHING_VIRTUAL 和 BR2_PACKAGE_PROVIDES_SOME THING_VIRTUAL，其值将由提供者使用。

17.11.3 虚拟包的.mk 文件

虚拟包的.mk 应该只是评估虚拟包宏：

```
1:  ################################################################################
2:  #
3:  # something-virtual
4:  #
5:  ################################################################################
6:
7:  $(eval $(virtual-package))
```

具有目标和主机包的能力也可以使用主机 - 虚拟包宏。

17.11.4 Provider 的 Config.in 文件

将包添加为提供者时，只有 Config.in 文件需要进行一些修改。

package-provider 的 Config.in 文件，它提供了一些虚拟的功能，它应该包含：

```
1: config BR2_PACKAGE_SOME_PROVIDER
2:         bool "some-provider"
3:         select BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
4:         help
5:           This is a comment that explains what some-provider is.
6:
7:           http://foosoftware.org/some-provider/
8:
9:   if BR2_PACKAGE_SOME_PROVIDER
10:  config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
11:        default "some-provider"
12:  endif
```

在第 3 行，我们选择 BR2_PACKAGE_HAS_SOMETHING_VIRTUAL，在第 11 行，我们将 BR2_PACKAGE_PRO VIDES_SOMETHING_VIRTUAL 的值设置为提供者的名称，但只有当它被选中时。

17.11.5 提供者的.mk 文件

.mk 文件还应声明一个附加变量 SOME_PROVIDER_PROVIDES，以包含其实现的所有虚拟包的名称：

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

当然，不要忘了为这个包添加正确的构建和运行时依赖关系！

17.11.6 取决于虚拟包的注意事项

当添加需要虚拟包提供的特定功能的包时，必须使用依赖于 BR2_PACKAGE_HAS_FEATURE，像

这样：

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

17.11.7 取决于具体的提供者的注意事项

如果你的包真的需要一个特定的提供商，那么你必须使你的包取决于这个提供者；您不能选择提供者。

让我们举个例子说明两个提供者的特征：

```
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
```

而您正在添加一个需要由 foo 提供的 FEATURE 的包，但不是由 bar 提供的。

如果要使用选择 BR2_PACKAGE_FOO，那么用户仍然可以在 menuconfig 中选择 BR2_PACKAGE_BAR。这将创建一个配置不一致，由此两个同一 FEATURE 的提供者将一次启用，一个由用户明确设置，另一个由您的选择隐式启用。

相反，您必须使用依赖于 BR2_PACKAGE_FOO，这避免任何隐式配置不一致。

17.12 使用 kconfig 配置文件的软件包的基础架构

一个软件包处理用户指定配置的流行方式是 kconfig。其中，它由 Linux 内核，Busybox 和 Buildroot 本身使用。存在.config 文件和 menuconfig 目标是使用 kconfig 的两个众所周知的症状。

Buildroot 具有使用 kconfig 进行配置的软件包的基础架构。该基础架构提供了必要的逻辑，

以便将 BuildMot 中的软件包 menuconfig 目标公开为 foo-menuconfig，并以正确的方式来处理配置文件的来回复制。

kconfig-package 基础架构基于通用包基础架构。基因 ric-package 支持的所有变量也可以在 kconfig-package 中使用。有关详细信息，请参见第 17.5.2 节。

为了对 Buildroot 软件包使用 kconfig-packages 基础架构，除了通用软件包基础架构所需的变量外，.mk 文件中最低限度的要求是：

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

此代码段创建以下 make 目标：

•foo-menuconfig，它调用该包的 menuconfig 目标

•foo-update-config，将配置文件复制回源配置文件。 当设置片段文件时，不可能使用此目标。

•foo-update-defconfig，将配置文件复制回源配置文件。 配置文件将仅列出与默认值不同的选项。 当设置片段文件时，不可能使用此目标。并确保源配置文件在正确的时刻被复制到构建目录。有两个选项来指定要使用的配置文件：FOO_KCONFIG_FILE（如上例所示）或 FOO_KCONFIG_DEFCONFIG。 必须提供，但不能同时提供：

•FOO_KCONFIG_FILE 指定要用于配置包的 defconfig 或 full-config 文件的路径。

•FOO_KCONFIG_DEFCONFIG 指定要调用的 defconfig make 规则来配置包。

除了这些最低限度的要求线之外，还可以设置几个可选变量，以满足考虑的包装需求：

•FOO_KCONFIG_EDITORS：用于支持的 kconfig 编辑器的空格分隔列表，例如 menuconfig xconfig。默认情况下，menuconfig。

•FOO_KCONFIG_FRAGMENT_FILES：合并到主配置文件的空格分隔的配置片段文件列表。当希望与上游（def）配置文件保持同步时，通常会使用片段文件，稍作修改。

•FOO_KCONFIG_OPTS：调用 kconfig 编辑器时要传递的额外选项。这可能需要包括$（FOO_MAKE_OPTS），例如。默认为空。

•FOO_KCONFIG_FIXUP_CMDS：在复制或运行 kconfig 编辑器后修复配置文件所需的 shell 命令列表。可能需要这样的命令来确保与 Buildroot 的其他配置一致的配置。默认为空。

•FOO_KCONFIG_DOTCONFIG：相对于包源代码树的.config 文件的路径（带文件名）。默认，。config，应该适合于使用从 Linux 内核继承的标准 kconfig 基础架构的所有软件包;一些软件包使

用使用不同位置的 kconfig 派生。

17.13 rebar-based packages 的基础

17.13.1 rebar-package 教程

首先，我们来看一下如何为基于螺纹的软件包编写一个.mk 文件，并附上一个例子：

```
1:  ################################################################################
2:  #
3:  # erlang-foobar
4:  #
5:  ################################################################################
6:
7:  ERLANG_FOOBAR_VERSION = 1.0
8:  ERLANG_FOOBAR_SOURCE = erlang-foobar-$(ERLANG_FOOBAR_VERSION).tar.xz
9:  ERLANG_FOOBAR_SITE = http://www.foosoftware.org/download
10: ERLANG_FOOBAR_DEPENDENCIES = host-libaaa libbbb
11:
12: $(eval $(rebar-package))
```

在第 7 行，我们声明包的版本。

在第 8 行和第 9 行，我们声明 tarball 的名称（xz-ed tarball 推荐）和 tarball 在 Web 上的位置。

Buildroot 会自动从这个位置下载 tarball。

在第 10 行，我们声明我们的依赖关系，以便它们在我们的包的构建过程开始之前构建。

最后，在第 12 行，我们调用生成所有实际允许构建包的 Makefile 规则的 rebar-package 宏。

17.13.2 rebar-package 参考

rebar package 基础设施的主要宏是 rebar-package。它类似于通用包宏。

主机包宏的功能也是可用的。

就像通用基础设施一样，rebar 基础设施的工作原理是在调用 rebar package 宏之前定义一些变量。

首先，通用基础设施中存在的所有包元数据信息变量也存在于 rebar 架构中：

ERLANG_FOOBAR_VERSION ， ERLANG_FOOBAR_SOURCE ， ERLANG_FOOBAR_PATCH ， ERLANG_FOOBAR_SITE ， ERLA NG_FOOBAR_SUBDIR ， ERLANG_FOOBAR_DEPENDENCIES ， ERLANG_FOOBAR_INSTALL_STAGING ， ERLANG_FOOBA R_INSTALL_TARGET ， ERLANG_FOOBAR_LICENSE 和 ERLANG_FOOBAR_LICENSE_FILES。

还可以定义一些特定于钢筋基础设施的附加变量。其中许多仅在非常具体的情况下有用，因此典型的包将仅使用其中的几个。

•ERLANG_FOOBAR_USE_AUTOCONF，用于指定程序包在配置步骤中使用 autoconf。当包将此变量设置为 YES 时，将使用自动工具基础架构。

注意您还可以使用自动工具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_FOO BAR_AUTORECONF_OPTS。

•ERLANG_FOOBAR_USE_BUNDLED_REBAR，用于指定该软件包具有捆绑版本的钢筋，并将使用该版本。有效值为 YES 或 NO（默认值）。

注意如果包装捆绑一个钢筋实用程序，但可以使用 Buildroot 提供的通用工具，只需说出 NO（即不要指定此变量）。只有在强制使用此包中捆绑的钢筋实用工具时才设置。

•ERLANG_FOOBAR_REBAR_ENV，用于指定要传递给 rebar 实用程序的其他环境变量。

使用钢筋基础设施，构建和安装软件包所需的所有步骤已经定义，并且它们通常适用于大多数基于 rebar 的软件包。但是，如果需要，还可以自定义在任何特定步骤中所做的工作：

•通过添加一个后操作钩子（提取，修补，配置，构建或安装之后）。详见第 17.18 节。

•覆盖其中一个步骤。例如，即使使用了 rebar 基础设施，如果包.mk 文件定义了自己的 ERLANG_FOOBAR_BUILD_CMDS 变量，那么将使用它来代替默认的 rebar 架。然而，使用这种方法应该限于非常具体的情况。在一般情况下不要使用它。

17.14 基于 waf-package 的基础设施

17.14.1 waf-package 教程

首先，我们来看一下如何为基于 Waf 的软件包编写一个.mk 文件，例如：

```
1:  ################################################################################
2:  #
3:  # libfoo
4:  #
5:  ################################################################################
6:
7:  LIBFOO_VERSION = 1.0
8:  LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
9:  LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_CONF_OPTS = --enable-bar --disable-baz
11: LIBFOO_DEPENDENCIES = bar
12:
13: $(eval $(waf-package))
```

在第 7 行，我们声明包的版本。

在第 8 行和第 9 行，我们声明 tarball 的名称（xz-ed tarball 推荐）和 tarball 在 Web 上的位置。

Buildroot 会自动从这个位置下载 tarball。

在第 10 行，我们告诉 Buildroot 为 libfoo 启用哪些选项。

在第 11 行，我们告诉 Buildroot libfoo 的不同。

最后，在第 13 行，我们调用生成实际允许构建包的所有 Makefile 规则的 waf-package 宏。

17.14.2 waf-package 参考

Waf 包的基础设施的主要宏是 waf-package。它类似于通用包宏。

就像通用基础设施一样，Waf 基础设施通过在调用 waf-package 宏之前定义一些变量来实现。

首先，通用基础设施中存在的所有包元数据信息变量也存在于 Waf 基础设施中：LIBFOO_VERSION， LIBFOO_SOURCE， LIBFOO_PATCH， LIBFOO_SITE， LIBFOO_SUBDIR，LIBFOO_DEPENDE NCIES，LIBFOO_INSTALL_STAGING，LIBFOO_INSTALL_TARGET。

还可以定义一个特定于 Waf 基础设施的附加变量。

•可以将 LIBFOO_NEEDS_EXTERNAL_WAF 设置为 YES 或 NO，告诉 Buildroot 使用捆绑的 waf 可执行文件。如果设置为 NO，那么默认情况下，Buildroot 将使用包源码树中提供的 waf 可执行文件;如果设置为 YES，则 Buidlroot 将下载，安装 waf 作为主机工具，并使用它来构建软件包。

•LIBFOO_WAF_OPTS，用于指定在程序包构建过程的每个步骤传递给 waf 脚本的其他选项：配置，构建和安装。默认为空。

•LIBFOO_CONF_OPTS，用于指定其他选项传递给配置步骤的 waf 脚本。默认为空。

•LIBFOO_BUILD_OPTS，用于指定在构建步骤期间传递给 waf 脚本的其他选项。默认为空。

•LIBFOO_INSTALL_STAGING_OPTS，用于指定在分段安装步骤期间传递给 waf 脚本的其他选项。默认为空。

•LIBFOO_INSTALL_TARGET_OPTS，用于指定在目标安装步骤期间传递给 waf 脚本的其他选项。默认为空。

17.15 构建内核模块的软件包的基础架构

Buildroot 提供了一个帮助基础设施，可以轻松编写构建和安装 Linux 内核模块的软件包。一些软件包只包含内核模块，除了内核模块之外，其他软件包还包含程序和库。 Buildroot 的帮助基础设施支持两种情况。

17.15.1 内核模块教程

我们从一个例子介绍如何准备一个只构建一个内核模块的简单软件包，没有其他组件：

```
1:  ################################################################################
2:  #
3:  # foo
4:  #
```

```
5:  ################################################################################
6:
7:  FOO_VERSION = 1.2.3
8:  FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
9:  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 行定义了通常的元数据，以指定版本，归档名称，远程 URI 在哪里可以找到包源，许可信息。

在第 13 行，我们调用内核模块助手基础结构，生成所有适当的 Makefile 规则和变量来构建该内核模块。

最后，在第 14 行，我们调用了通用包基础架构第 17.5.1 节。

自动添加对 linux 的依赖，所以不需要在 FOO_DEPENDENCIES 中指定它。

您可能已经注意到，与其他软件包基础架构不同，我们显式地调用了第二个基础架构。 这允许包构建内核模块，但是如果需要，也可以使用任何一个其他软件包基础架构来构建正常的用户空间组件（库，可执行文件...）。 单独使用内核模块基础架构是不够的；必须使用另一个包装基础设施。

我们来看一个更复杂的例子：

```
1:  ################################################################################
2:  #
3:  # foo
4:  #
5:  ################################################################################
6:
7:  FOO_VERSION = 1.2.3
8:  FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
9:  FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: FOO_MODULE_SUBDIRS = driver/base
14: FOO_MODULE_MAKE_OPTS = KVERSION=$(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，我们看到我们有一个基于自动工具的软件包，它还构建了位于子目录驱动程序/base 中的内核模块，如果启用了 libbar，内核模块位于子目录 driver / bar 中，并定义了变量 KVERSION 在构建模块时传递给 Linux 构建系统。

17.15.2 内核模块参考

内核模块基础架构的主要宏是内核模块。 与其他软件包基础架构不同，它不是独立的，并且需要在其后面调用任何其他*包装宏。

内核模块宏定义后构建和后目标安装钩子来构建内核模块。 如果包的.mk 需要访问内置的内核模块，那么应该在调用 kernel-module 之后注册的后构建钩子。 类似地，如果包的.mk 在安装完成后需要访问内核模块，那么应该在调用 kernel-module 后注册的安装后的挂钩中这样做。 这里有一个例子：

```
$(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，与其他软件包基础架构不同，没有主机内核模块变体来构建主机内核模块。

可以选择定义以下附加变量来进一步配置内核模块的构建：

•FOO_MODULE_SUBDIRS 可能设置为内核模块源所在的一个或多个子目录（相对于包源顶级目录）。如果为空或未设置，则内核模块的源将被视为位于包源树的顶部。

•可以将 FOO_MODULE_MAKE_OPTS 设置为包含额外的变量定义，以传递给 Linux 构建系统。

你也可以参考（但你可能不设置！）这些变量：

•LINUX_DIR 包含 Linux 内核提取和构建的路径。

•LINUX_VERSION 包含用户配置的版本字符串。

•LINUX_VERSION_PROBED 包含使用运行 make -C $（LINUX_DI）检索的内核的真实版本字符串

•KERNEL_ARCH 包含当前架构的名称，如 arm，mips。 。 。

17.16 asciidoc 文件的基础设施

您正在阅读的 Buildroot 手册完全使用 AsciiDoc 标记语法编写。然后将手册呈现为多种格式：

•HTML
•split-HTML
•PDF
•EPUB
•text

虽然 Buildroot 只包含一个 AsciiDoc 编写的文档，但是对于包来说，使用 AsciiDoc 语法呈现文档的基础架构。

另外对于包，AsciiDoc 基础设施可从 br2 外部树第 9.2 节获取。这允许 br2 外部树的文档与 Buildroot 文档匹配，因为它将被渲染为相同的格式并使用相同的布局和主题。

17.16.1 asciidoc 文档教程

而包基础设施后缀为-package，文档基础结构后缀为-document。 因此，AsciiDoc 基础结构被命名为 asciidoc 文档。

以下是一个简单的 AsciiDoc 文档的例子。

```
1:  ################################################################################
2:  #
3:  # foo-document
4:  #
5:  ################################################################################
6:
7:  FOO_SOURCES = $(sort $(wildcard $(pkgdir)/*))
8:  $(eval $(call asciidoc-document))
```

在第 7 行，Makefile 声明文档的来源。目前，预计该文件的来源只是本地的; Buildroot 不会尝试下载任何东西来呈现文档。因此，您必须指出来源在哪里。通常，上面的字符串对于没有子目录结构的文档是足够的。

在第 8 行，我们称之为 asciidoc-document 函数，它生成渲染文档所需的所有 Makefile 代码。

17.16.2 asciidoc 文件参考

可以在.mk 文件中设置以提供元数据信息的变量列表（假设文档名称为 foo）：

•FOO_SOURCES，强制性，定义文档的源文件。

•FOO_RESOURCES（可选）可能包含一个空格分隔的一个或多个包含所谓资源（如 CSS 或图像）的目录的路径列表。默认为空。

•FOO_DEPENDENCIES，可选，在构建此文档之前必须构建的软件包列表（最可能是主机包）。如果您的文档的钩子需要访问 Kconfig 结构，则可以将 prepare-kconfig 添加到依赖关系列表中。还有其他钩子（关于钩子的一般信息，请参见第 17.18 节），文档可能设置为在各个步骤中定义额外的操作：

•在 Buildroot 复制源之后，FOO_POST_RSYNC_HOOKS 运行其他命令。这可以例如用于使用从树中提取的信息来生成手册的一部分。例如，Buildroot 使用此钩子来生成附录中的表。

•FOO_CHECK_DEPENDENCIES_HOOKS，以对所需组件运行附加测试来生成文档。在 Asci-iDoc 中，可以调用过滤器，即将解析 AsciiDoc 块并适当地呈现它的程序（例如，dita 或 aafigure）。

•FOO_CHECK_DEPENDENCIES_ <FMT> _HOOKS，为指定格式<FMT>运行附加测试（请参阅上面的 ren-dered 格式列表）。

这是一个使用所有变量和所有钩子的完整示例：

```
1:  ################################################################################
2:  #
3:  # foo-document
4:  #
5:  ################################################################################
6:
7:  FOO_SOURCES = $(sort $(wildcard $(pkgdir)/*))
8:  FOO_RESOURCES = $(sort $(wildcard $(pkgdir)/ressources))
```

```
9:
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))
```

17.17 特定于 Linux 内核包的基础设施

Linux 内核软件包可以使用一些基于包钩子的特定基础架构来构建 Linux 内核工具或/或构建 Linux 内核扩展。

17.17.1 linux-kernel-tools

Buildroot 提供了一个帮助基础设施，可以为 Linux 内核源中的可用目标构建一些用户空间工具。 由于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，并重新使用在目标上运行的 Linux 内核的源代码。

我们来看一个 Linux 工具的例子。 对于名为 foo 的新型 Linux 工具，请在现有 package/linux-tools / Config.in 中创建一个新的菜单项。 该文件将包含与将在配置工具中使用和显示的

每个内核工具相关的选项说明。 它基本上看起来像:

```
1: config BR2_PACKAGE_LINUX_TOOLS_FOO
2:         bool "foo"
3:         select BR2_PACKAGE_LINUX_TOOLS
4:         help
5:           This is a comment that explains what foo kernel tool is.
6:
7:           http://foosoftware.org/foo/
```

该选项的名称以前缀 BR2_PACKAGE_LINUX_TOOLS_开头，后跟工具的大写名称（就像为包完成）。

注意与其他软件包不同，linux-tools 软件包选项显示在 Linux 内核菜单中的 Linux Kernel Tools 子菜单下，而不是 Target 软件包主菜单。

然后对于每个 linux 工具，添加一个名为 package/linux-tools/linux-tool-foo.mk.in 的新的.mk.in 文件。 它基本上看起来像:

```
1:  ################################################################################
2:  #
3: # foo
4: #
5:  ################################################################################
6:
7:  LINUX_TOOLS += foo
8:
9:  FOO_DEPENDENCIES = libbbb
10:
11: define FOO_BUILD_CMDS
12:       $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools foo
13: endef
14:
15: define FOO_INSTALL_STAGING_CMDS
16:       $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
17:                 DESTDIR=$(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
```

在第 7 行，我们将 Linux 工具 foo 注册到可用的 Linux 工具列表中。

在第 9 行，我们指定此工具依赖的依赖关系列表。仅当选择了 foo 工具时，这些依赖项才会添加到 Linux 包依赖关系列表中。

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做什么，如通用包 17.5.1 节。只有当选择了 foo 工具时，它们才会被使用。唯一支持的命令是_BUILD_CMDS，_INSTALL_STAGING_CMDS 和_INSTALL_TARGET_CMDS。

注意一个不能调用$（eval $（generic-package））或任何其他包基础架构！ Linux 工具本身不是软件包，它们是 linux-tools 软件包的一部分。

17.17.2 Linux 内核的扩展

一些包提供了新的功能，需要修改 Linux 内核树。这可以是要在内核树上应用的补丁的形式，或以要添加到树中的新文件的形式。 Buildroot 的 Linux 内核扩展基础架构提供了一个简单的解决方案，可以自动执行此操作，只需在内核源解压缩之后并在应用内核补丁之前。使用此机制打包的扩展的示例是实时扩展 Xenomai 和 RTAI，以及一组 out-of-tree LCD 屏幕驱动程序 fbtft。

我们来看一个关于如何添加一个新的 Linux 扩展 foo 的例子。

首先，创建提供扩展的包 foo：这个包是一个标准包;请参阅前几章，介绍如何创建这样一个包。此软件包负责下载源存档，检查哈希，定义许可证信息和构建用户空间工具（如果有的话）。

然后正确创建 Linux 扩展：在现有的 linux / Config.ext.in 中创建一个新的菜单条目。该文件包含与将在配置工具中使用和显示的每个内核扩展相关的选项说明。它基本上看起来像：

```
1:  config BR2_LINUX_KERNEL_EXT_FOO
2:          bool "foo"
3:          help
4:              This is a comment that explains what foo kernel extension is.
5:
6:              http://foosoftware.org/foo/
```

然后对于每个 linux 扩展，添加一个名为 linux / linux-ext-foo.mk 的新的.mk 文件。 它应该基本上包含：

```
1:  ################################################################################
2:  #
3:  # foo
4:  #
5:  ################################################################################
```

```
6:
7:    LINUX_EXTENSIONS += foo
8:
9:    define FOO_PREPARE_KERNEL
10:           $(FOO_DIR)/prepare-kernel-tree.sh --linux-dir=$(@D)
11:   endef
```

在第 7 行，我们将 Linux 扩展名 foo 添加到可用的 Linux 扩展列表中。

在第 9-11 行，我们定义扩展来做什么来修改 Linux 内核树；这是特定于 linux 扩展，可以使用由 foo 包定义的变量，如：$（FOO_DIR）或$（FOO_VERSION）。。。 以及所有的 Linux 变量，如：$（LINUX_VERSION）或$（LINUX_VERSION_PROBED），$（KERNEL_ARCH）。。。 看到这些内核变量的定义[simpara]。

17.18 钩子可用于各种构建步骤

通用基础架构（以及衍生的自动工具和 cmake 基础结构）也允许程序包指定钩子。 这些定义了在现有步骤后执行的进一步操作。 大多数钩子对于通用程序包并不是非常有用，因为.mk 文件已经完全控制了在程序包构造的每个步骤中执行的操作。

以下钩点可用：

- 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

这些变量是包含要在此钩点处执行的操作的变量名称列表。 这允许在给定的钩点注册几个钩子。

这是一个例子:

```
define LIBFOO_POST_PATCH_FIXUP
        action1
        action2
endef
LIBFOO_POST_PATCH_HOOKS += LIBFOO_POST_PATCH_FIXUP
```

17.18.1 使用 POST_RSYNC 挂钩

POST_RSYNC 钩子仅对使用本地源的软件包运行，通过本地站点方法或 OVERRI DE_SRCDIR 机制运行。在这种情况下，使用 rsync 将包源从本地位置复制到 buildroot 构建目录中。 rsync 命令不会从源目录复制所有文件。不会复制属于版本控制系统的文件，如目录.git，.hg 等。对于大多数软件包来说，这是足够的，但给定的软件包可以使用 POST_RSYNC 钩子执行其他操作。

原则上，钩子可以包含你想要的任何命令。一个具体的用例是使用 rsync 有意的复制版本控制目录。在钩子中使用的 rsync 命令可以使用以下变量:

•$（SRCDIR）：覆盖源目录的路径

•$（@ D）：构建目录的路径

17.18.2 目标确定钩

软件包也可以在 LIBFOO_TARGET_FINALIZE_HOOKS 中注册钩子。所有这些钩子在构建所有包之后，但在生成文件系统映像之前运行。他们很少使用，你的包可能不需要它们。

17.19 Gettext 集成和与包的交互

许多支持国际化的包使用 gettext 库。这个图书馆的依赖性相当复杂，因此值得一些解释。

glibc C 库集成了一个完整的 gettext 实现，支持翻译。原始语言支持是由 glibc 内置的。

另一方面，uClibc 和 musC C 库只提供了 gettext 功能的存根实现，它允许使用 gettext 函数来编译库和程序，但不提供完整的 gettext 实现的翻译功能。使用这样的 C 库，如果需要真正的 Native Language Support，那么可以由 gettext 包的 libintl 库提供。

因此，为了确保本地语言支持得到妥善处理，Buildroot 中可以使用 NLS 支持的软件包应该：

1.确保 BR2_SYSTEM_ENABLE_NLS = y 时启用了 NLS 支持。这对于自动工具包自动完成，因此应该仅对使用其他包基础架构的包完成。

2.将$（TARGET_NLS_DEPENDENCIES）添加到包<pkg> _DEPENDENCIES 包中。这个添加应该无条件地进行：这个变量的值由核心基础设施自动调整，以包含相关的程序包列表。如果禁用 NLS 支持，则此变量为空。如果启用了 NLS 支持，则该变量包含 host-gettext，以便主机上可以使用编译翻译文件所需的工具。另外，如果使用 uClib 或 musl，这个变量也包含了 gettext，以便获得完整的 gettext 实现。

3.如果需要，将$（TARGET_NLS_LIBS）添加到链接器标记中，以便包与 libintl 链接。自动工具包通常不需要它们，因为它们通常会自动检测到它们应该与 libintl 链接。但是，使用其他构建系统的软件包或基于自动工具的问题的软件包可能需要此功能。 $（TARGET_NL S_LIBS）应无条件添加到链接器标志中，因为核心会根据配置自动将其设置为空或定义为 lintl。

不要对 Config.in 文件进行更改，以支持 NLS。

最后，某些包在目标上需要一些gettext实用程序，例如允许从命令行检索转换的字符串的gettext程序本身。在这种情况下，包应该：

•在其 Config.in 文件中使用选择 BR2_PACKAGE_GETTEXT，在上面的注释中指出它只是一个运行时依赖关系。

•在其.mk 文件的 DEPENDENCIES 变量中不添加任何 gettext 依赖关系。

17.20 提示和技巧

17.20.1 包名，配置条目名和 makefile 变量关系

在 Buildroot 中，有一些关系：

•软件包名称，即软件包目录名称（和* .mk 文件的名称）；

•在 Config.in 文件中声明的配置条目名称;

•makefile 变量前缀。

使用以下规则，必须保持这些元素之间的一致性：

•包目录和* .mk 名称是包名本身（例如：package / foo-bar_boo / foo-bar_boo。mk）；

•make 目标名称是包名本身（例如：foo-bar_boo）；

•配置条目是大写包名称。和 - 以_替换的字符，前缀为 BR2_PACKAGE_（例如：BR2_PACKAGE_FOO_BAR_BOO）；

•* .mk 文件变量前缀是大写的包名称。和 - 用_（例如：FOO_BAR_BO O_VERSION）替换的字符。

17.20.2 如何测试包装

一旦添加了新的包，重要的是您可以在各种条件下进行测试：它是为所有架构构建的吗？它是用不同的 C 库构建的？ 是否需要线程，NPTL？ 等等。。。

Buildroot 运行自动建模器，它连续测试随机配置。 然而，这些只是构建了 git 树的主分支，而你新的花哨的包还没有在那里。

Buildroot 在 utils / test-pkg 中提供了脚本，它使用与自动构建器相同的基本配置，以便您可以在相同的条件下测试包。

首先，创建一个配置代码段，其中包含启用包所需的所有必需选项，但没有任何架构或工具链选项。 例如，让我们创建一个只需启用 libcurl 的配置代码段，而不需要任何 TLS 后端：

```
$cat libcurl.config BR2_PACKAGE_LIBCURL=y
```

如果您的软件包需要更多配置选项，可以将它们添加到配置代码段中。 例如，以下将介绍

如何使用 openssl 作为 TLS 后端和卷曲程序来测试 libcurl：

```
$cat libcurl.config
BR2_PACKAGE_LIBCURL=y
BR2_PACKAGE_CURL=y
BR2_PACKAGE_OPENSSL=y
```

然后运行 test-pkg 脚本，告诉它要使用什么配置代码片段以及要测试的软件包：

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
```

这将尝试针对自动构建器使用的所有工具链（除了内部工具链，因为需要太长时间）来构建软件包。 输出列出所有工具链和相应的结果（摘录，结果是假的）：

```
$ ./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
```

结果意味着：

•OK：构建成功。

•SKIPPED：配置片段中列出的一个或多个配置选项在最终配置中不存在。这是由于具有工具链不满足的依赖性的选项，例如依赖于具有 noMMU 工具链的 BR2_U SE_MMU 的包。缺少的选项在输出构建目录（∽/ br-test-pkg / TOOLCHAIN_NAME /默认）中的 missing.config 中报告。

•FAILED：构建失败。检查输出构建目录中的日志文件，看看出了什么问题：

- 实际构建失败，
- 法律信息失败，
- 其中一个初步步骤（下载配置文件，应用配置，为包运行 dirclean）失败。

当出现故障时，您可以重新运行脚本，具有相同的选项（修复包后）;脚本将尝试重新构建使用-p 指定的所有工具链的包，而无需重新构建该包的所有依赖关系。

test-pkg 脚本接受几个选项，您可以通过运行以下方法获得一些帮助：

```
$ ./utils/test-pkg -h
```

17.20.3 如何从 GitHub 添加包

GitHub 上的软件包通常没有释放 tarball 的下载区域。 但是，可以直接从 GitHub 上的存储库下载 tarball。 由于 GitHub 以前已经改变了下载机制，所以应该使用 github 帮助函数，如下所示。

```
#  Use a tag or a full
commit ID
FOO_VERSION =
v1.0
FOO_SITE = $(call github,<user>,<package>[,<version>])
```

提示：

•FOO_VERSION 可以是标签或提交 ID。

•github 生成的 tarball 名称与 Buildroot 中的默认值相匹配（例如：foo-f6fb6654af62045239caed5950bc6c7971965e60.tar.gz），因此无需在.mk 文件中指定。

•使用提交 ID 作为版本时，应使用完整的 40 个十六进制字符。

•版本参数是可选的，只有在需要使用 FOO_VERSION 以外的任何其他功能时才应指定。当此功能用于指定自定义 Linux 或 U-Boot tarball 时。

如果您希望添加的软件包在 GitHub 上有一个发行版，维护者可能已经上传了一个 release tarball，或者这个 release 可能只是从 git 标签指向自动生成的 tarball。如果维护者上传了一个 release tarball，我们更喜欢使用它，因为它可能稍有不同（例如，它包含一个配置脚本，因此我们不需要做 AUTORECONF）。

如果是上传的压缩包或 git 标签，则可以在发行页面上看到：

github_hash_mongrel2.png

•如果它看起来像上面的图像，那么它是由维护者上传的，你应该使用该链接（例如：mongrel2-v1.9.2.tar.bz2）来指定 FOO_SITE，而不使用 github 帮助器。

•另一方面，如果只有“源代码”链接，那么它是一个自动生成的 tarball，您应该使用 github 帮助

函数。

17.21 结论

您可以看到，向 Buildroot 添加软件包只是使用现有示例编写 Makefile 并根据程序包所需的编译过程对其进行修改的问题。

如果您打包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）！

第 18 章 修补包装

在集成新软件包或更新现有软件包或更新现有软件包时，可能需要对软件源进行修补，以便在 Buildroot 内进行交叉编译。

Buildroot 提供了一个基础设施，可以在构建过程中自动处理。它支持三种应用补丁集的方法：

下载的修补程序，buildroot 中提供的修补程序和位于用户定义的全局修补程序目录中的修补程序。

18.1 提供补丁

18.1.1 下载

如果需要应用可供下载的补丁，则将其添加到<packagename> _PATCH 变量。如果一个条目包含：//，那么 Buildroot 将假定它是一个完整的 URL，并从该位置下载修补程序。否则，Buildroot 将假定补丁应该从<packagename> _SITE 下载。它可以是单个补丁，也可以是包含补丁系列的 tarball。

像所有下载一样，哈希应该添加到<packagename> .hash 文件中。

该方法通常用于 Debian 软件包。

18.1.2 Buildroot 内

大部分修补程序都在 Buildroot 中，在包目录中提供;这些通常旨在修复交叉编译，libc 支持或其他此类问题。

这些补丁文件应该命名为<number> - <description> .patch。

请注意：

•Buildroot 附带的补丁文件不应包含其文件名中的任何软件包版本引用。

•补丁文件名中的字段<number>是指应用顺序，从 1 开始;最好将数字最多填入 4 位，如 git-

format-patch。例如：0001-foobar-the-buz.patch

•以前，修补程序必须以包的名称为前缀，如<package> - <number> - <desc_ription> .patch，但不一定是这样。随着时间的过去，现有的包将被修复。不要在包名称前缀补丁。

•以前，被套件使用的系列文件也可以添加到包目录中。在这种情况下，系列文件定义了补丁应用程序顺序。这是不推荐的，将来会被删除。不要使用系列文件。

18.1.3 全局补丁目录

BR2_GLOBAL_PATCH_DIR 配置文件选项可用于指定包含全局包修补程序的一个或多个目录的空格分隔列表。详见 9.8 节。

18.2 如何应用补丁

1.如果定义，运行<packagename> _PRE_PATCH_HOOKS 命令;

2.清理构建目录，删除任何现有的* .rej 文件;

3.如果定义了<packagename> _PATCH，则应用这些 tarball 的补丁;

4.如果包的 Buildroot 目录或名为<packageve rsion>的包子目录中有一些* .patch 文件，那么：

•如果包目录中存在一系列文件，则根据系列文件应用修补程序;

•否则，匹配* .patch 的修补程序文件按字母顺序应用。因此，为了确保按照正确的顺序进行应用，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枚举目录。按照上一步骤所述应用补丁。

6.如果定义，运行<packagename> _POST_PATCH_HOOKS 命令。

如果步骤 3 或 4 出现问题，则构建失败。

18.3 包补丁的格式和许可

补丁按照与其适用的软件相同的许可证发布（见第 12.2 节）。

应该在补丁的头部注释中添加一个消息，说明补丁的作用以及为什么需要它。

您应该在每个修补程序的标题中添加一个 Signed-off-the 语句，以帮助跟踪更改，并证明补丁在与修改的软件相同的许可证下发布。

如果软件处于版本控制下，建议使用上游 SCM 软件生成补丁集。

否则，将头与 diff -purN package-version.orig / package-version /命令的输出连接起来。

如果您更新现有的修补程序（例如，当碰撞包版本时），请确保现有的 From 标头和 Signed-off-by 标签未被删除，但是在适当的时候更新补丁注释的其余部分。

最后，补丁应该是：

```
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],
        The Buildroot user manual        94 / 109
        +       [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 Web 上找到的修补程序

当您整合您不是作者的补丁时，您必须在修补程序本身的标题中添加一些内容。

根据补丁是否已从项目存储库本身或从 Web 的某个地方获取，请添加以下标签之一：

```
Backported from: <some commit id>
or
Fetch from: <some url>
```

添加关于可能已经需要的修补程序的任何更改的几个字也是明智的。

第 19 章 下载基础设施

第 20 章 调试 buildroot

可以对 Buildroot 在构建程序包时执行的步骤进行调试。定义变量 BR2_INSTRUMENTATION_SCRIPTS 以包含一个或多个脚本（或其他可执行文件）的路径，在空格分隔的列表中，您希望在每个步骤之前和之后调用。脚本按顺序调用，具有三个参数：

•开始或结束以表示步骤的开始（或结束）；

•即将开始或即将结束的步骤的名称;

•包的名称。

例如 ：

使 BR2_INSTRUMENTATION_SCRIPTS =“/ path / to / my / script1 / path / to / my / script2”

步骤列表是：

•提取物

•补丁

• 配置

•构建

•安装主机，当在$（HOST_DIR）中安装主机包时，

•安装目标，当目标包安装在$（TARGET_DIR）中时，

•安装阶段，当$（STAGING_DIR）中安装目标包时，

•安装映像，当目标包安装$（BINARIES_DIR）中的文件时

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：请参见第 17.5.2 节

•BUILD_DIR：提取和构建软件包的目录

•BINARIES_DIR：存储所有二进制文件（也称为图像）的位置

•BASE_DIR：基本输出目录

第 21 章 有助于建立根基

有很多方法可以为 Buildroot 做出贡献：分析和修复错误，分析和修复自动建模器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 TODO 列表中的项目并发送自己的改进到 Buildroot 或其手册。以下部分将详细介绍这些项目。

如果您有兴趣为 Buildroot 贡献，您首先应该订阅 Buildroot 邮件列表。该列表是与其他 Buildroot 开发人员进行交互并发送贡献的主要方式。如果您尚未订阅，请参阅第 5 章订阅链接。

如果您要触摸代码，强烈建议您使用 Buildroot 的 git 仓库，而不是从提取的源代码压缩包开始。 Git 是最简单的开发方式，并直接将您的补丁发送到邮件列表。有关获取 Buildroot git 树的更多信息，请参阅第 3 章。

21.1 重现，分析和修复错误

第一种贡献方式是查看 Buildroot 错误跟踪器中的开放错误报告。当我们努力使错误数量尽可能小时，所有帮助复制，分析和修复报告的错误是非常受欢迎的。不要犹豫，添加评论报告你的发现的错误报告，即使你还没有看到完整的图片。

21.2 分析和修复自动生成故障

Buildroot 自动构建器是一组基于随机配置连续运行 Buildroot 构建的构建机器。这是由 Buildroot 支持的所有架构，各种工具链以及随机选择的软件包完成的。随着 Buildroot 的大量提交活动，这些自动构建器在提交后很早就能很好地检测问题。

所有构建结果可在 http://autobuild.buildroot.org 获取，统计数据位于 http://autobuild.buildroot.org/stats.php 。每天都会将所有失败包的概述发送到邮件列表。

检测问题是很大的，但显然这些问题也必须加以修正。您的贡献非常受欢迎！

基本上可以做两件事情：

•分析问题。每日摘要邮件不包含有关实际故障的详细信息：为了看到发生了什么，您必须打开

构建日志并检查最后的输出。有人为邮件中的所有包执行此操作对其他开发人员非常有用，因为它们可以基于此输出单独进行快速初始分析。

•解决问题。 修复 autobuild 失败时，您应该按照下列步骤操作：

1.检查您是否可以通过使用相同的配置构建来重现问题。 您可以手动执行此操作，也可以使用将自动克隆 Buildroot git 存储库，检出正确修订版本，下载并设置正确配置并启动构建的 br-reproduction-build 脚本。

2.分析问题并创建修复。

3.通过从干净的 Buildroot 树开始并仅应用您的修复程序，验证问题是否真的被修复。

4.将修复程序发送到 Buildroot 邮件列表（参见第 21.5 节）。 如果您针对软件包源创建了修补程序，则还应该向上发送修补程序，以便在以后的版本中修复问题，并且可以删除 Buildroot 中的修补程序。 在修补 autobuild 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 查看和测试补丁

随着每天发送到邮件列表的修补程序的数量，维护者非常难以判断哪些补丁已准备好应用，哪些补丁没有被应用。参与者可以通过查看和测试这些补丁来大大帮助他们。

在审查过程中，请不要犹豫，回复补丁提交的意见，建议或任何有助于大家了解补丁并使之更好的任何内容。在回复补丁提交时，请使用纯文本电子邮件中的互联网风格的回复。

要指示补丁的批准，有三个正式标签跟踪此批准。要将您的标签添加到修补程序中，请使用原作者签名的下一行的批准标签来回复。这些标签将通过拼接自动获取（参见第 21.3.1 节），并且将在修补程序被接受时成为提交日志的一部分。

经测试通过

表示修补程序已成功测试。鼓励您指定您执行的测试类型（架构 X 和 Y 上的编译测试，目标 A 上的运行时测试，...）。此附加信息有助于其他测试人员和维护人员。

代码审查

表示您对代码进行了代码审查，并尽最大努力发现问题，但您对于提供 Acked-by 标签所触及的区域不够熟悉。这意味着补丁中可能存在将在该领域有更多经验的人员发现的问题。如果检测到这些问题，您的已审阅标签仍然适用，您不能被指责。

答疑

表示您对代码进行了修改，并且熟悉该区域，感觉该补丁可以按原样提交（无需额外更改）。如果稍后证明补丁有问题，您的 Acked-by 可能被认为是不合适的。因此，Acked-by 和 Checking-by 之间的区别主要是因为您准备对 Acked 补丁负责，而不是评论者。

如果您查看了修补程序并对其进行了评论，那么您应该仅仅回复补丁说明这些注释，而不需要提供经过审查的或者 Acked-by 的标签。只有当您将补丁视为好时，才应提供这些标签。

重要的是要注意，既没有被审查，也没有 Acked 表示测试已被执行。为了表明您同时审查并测试了修补程序，请提供两个单独的标签（已审核/确认和测试）。

还要注意，任何开发人员都可以提供经过测试/审查/确认的标签，无一例外，我们鼓励大家做到这一点。 Buildroot 没有定义的核心开发人员组，只是发生在一些开发人员比其他开发人员更为活跃。维护者将根据其提交者的记录来对标签进行评估。标签由常规贡献者提供的标签当然是比新手提供的标签更受信任的。当您更经常地提供标签时，您的可信度（在维护者的眼中）会上升，但是提供的任何标签都是有价值的。

Buildroot 的 Patchwork 网站可用于提供补丁以进行测试。有关使用 Buildroot 的 Patchwork 网站应用补丁的更多信息，请参阅第 21.3.1 节。

21.3.1 从拼布中应用补丁

Buildroot 的 Patchwork 网站为开发人员的主要用途是将补丁引入其本地 git 仓库进行测试。在拼贴管理界面中浏览修补程序时，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
```

应用补丁的另一个选择是创建一个 bundle。捆绑是一组可以使用拼接接口组合在一起的补丁。 创建捆绑包并将捆绑包公开后，您可以复制捆绑包的 mbox 链接，并使用上述命令应用捆绑包。

21.4 从 TODO 列表中处理项目

如果您想为 Buildroot 贡献但不知道从哪里开始，并且您不喜欢任何上述主题，您可以随时从 Buildroot TODO 列表中处理项目。 不要犹豫，在邮件列表或 IRC 上首先讨论一个项目。 编辑维基来指明何时开始处理项目，所以我们避免重复的工作。

21.5 提交补丁

注意

请不要附加修补程序到错误，而是将它们发送到邮件列表。

如果您对 Buildroot 进行了一些更改，并且您希望将其贡献给 Buildroot 项目，请按照以下步骤操作。

21.5.1 补丁的格式化

我们希望补丁以特定方式进行格式化。这是必要的，以便于查看补丁，以便能够轻松地将它们应用到 git 仓库，以便在历史记录中轻松查找事情发生变化以及为什么发生变化，并且可以使用 git bisect 找出问题的根源。

首先，补丁有一个很好的提交信息是至关重要的。提交消息应以单独的行开头，并以变更的简要摘要为起点，以受影响的包的名称开头。提交消息的正文应该描述为什么需要进行这种更

改，如有必要，还可以详细介绍它的完成情况。在撰写提交信息时，请考虑评论者如何阅读，但还要考虑如何在几年之后再看一下这个变化。

其次，补丁本身应该只做一个改变，但是要做到这一点。两个不相关或弱相关的更改通常应该在两个单独的补丁中完成。这通常意味着修补程序仅影响单个包。如果有几个更改相关，通常仍然可以将它们以小补丁分开，并以特定的顺序应用它们。小修补程序可以更容易地进行审查，并且往往使之后更容易理解为什么要进行更改。但是，每个补丁必须完整。当只应用第一个但不是第二个补丁时，不允许该构建中断。这是必要的，以后能够使用 git 平分。

当然，当你正在做你的开发工作时，你可能会在软件包之间来回走动，而且肯定不会以足够干净的方式提交事情。因此，大多数开发人员重写提交的历史记录，以生成适合提交的一组干净的提交。为此，您需要使用交互式版本。您可以在 Pro Git 书中了解它。有时候，使用 git reset --soft origin / master 丢弃你的历史记录，并使用 git add -i 或 git add -p 来选择个别的更改。

最后，补丁应该被签名。这是通过添加已签名的：您的真实姓名<您的@电子邮件来完成的。地址>在提交消息的结尾。 git commit -s 对于您，如果配置正确。 Signed-off 标签表示您根据 Buildroot 许可证（即，GPL-2.0 +，除了具有上游许可证的软件包修补程序），并允许您这样做，发布修补程序。有关详细信息，请参阅开发商证书原件。

添加新软件包时，您应该将每个软件包都提交到单独的软件包中。此修补程序应具有 package / Config.in，软件包 Config.in 文件，.mk 文件，.hash 文件，任何 init 脚本和所有软件包修补程序的更新。如果包装有许多子选项，则有时会更好地添加这些子选项作为单独的后续修补程序。摘要行应该是像<packagename>：新的包。提交消息的主体对于简单的包可以是空的，或者它可以包含包的描述（如 Config.in 帮助文本）。如果需要做任何特殊的工作来构建软件包，这也应该在提交消息正文中明确说明。

当您将软件包颠覆到新版本时，您还应该为每个软件包提交一个单独的补丁。不要忘记更新.hash 文件，或者如果尚不存在则将其添加。还不要忘记检查_LICENSE 和_LICENSE_FILES 是否仍然有效。摘要行应该像<packagename>：bump to version <new version>。如果新版本仅包含与现有版本相比的安全更新，则摘要应为<packagename>：安全性崩溃到版本<新版本>，提交消息体应显示已修复的 CVE 号。如果新版本中可以删除某些软件包修补程序，那么应该明确说明为什么可以删除它们，最好使用上游提交 ID。还应明确说明任何其他必需的更改，例如不再存在或不再需要的配置选项。

如果您有兴趣收到构建失败的通知以及您添加或修改的软件包的进一步更改，请将自己添加到 DEVELOPERS 文件中。这应该在系列的单独补丁中完成。有关详细信息，请参阅开发人员

文件第 22 章。

21.5.2 准备补丁系列

从本地 git 视图中提交的更改开始，在生成修补程序集之前，先将开发分支重新上传到上游树。

要这样做，请运行：

```
$ git fetch --all --tags
$ git rebase origin/master
```

现在，您已准备好生成然后提交您的修补程序集。

要生成它，运行：

```
$ git format-patch -M -n -s -o outgoing origin/master
```

这将在传出子目录中生成补丁文件，自动添加 Signed-off-by 行。

生成补丁文件后，您可以使用自己喜欢的文本编辑器在提交补丁文件之前查看/编辑提交消息。

Buildroot 提供了一个方便的工具，可以知道应该向谁发送补丁，称为获取开发者（有关更多信息，请参阅第 22 章）。 此工具读取您的修补程序并输出适当的 git send-email 命令以使用：

```
$ ./utils/get-developers outgoing/*
```

使用 get-developers 的输出发送你的补丁：

```
git send-email --to buildroot@buildroot.org --cc bob --cc alice outgoing/*
```

请注意，git 应配置为使用您的邮件帐户。 要配置 git，请参阅 man git-send-email 或 google。

如果您不使用 git 发送电子邮件，请确保发布的修补程序不是线条包装，否则不能轻易应用。

在这种情况下，请修复您的电子邮件客户端，或者更好地学习使用 git send-email。

21.5.3 求职信

如果要在单独的邮件中显示整个修补程序集，请将-cover-letter 添加到 git format-patch 命令（有关更多信息，请参阅 man git-format-patch）。这将生成一个介绍电子邮件到您的补丁系列的模板。

在以下情况下，求职信可能有助于介绍您提出的更改：

•系列中的大量提交;

•项目其余部分变化的深刻影响;

•RFC;

•每当感觉到这将有助于展示您的工作，您的选择，审查过程等。

21.5.4 修补版本更新日志

当要求改进时，每个提交的新修订应包括每次提交之间的更改日志。请注意，当您的补丁系列由求职信引入时，除了单个提交中的更改日志之外，还可以在求职信中添加总体更改日志。修改补丁系列的最好的办法是通过交互式版本：git rebase -i origin / master。有关详细信息，请参阅 git 手册。

当添加到单独的提交时，编辑提交消息时会添加此更改日志。在 Signed-off-by 部分下方，添加---和您的更改日志。

虽然更改日志对于邮件线程中的审阅者以及拼凑都是可见的，但是 git 会自动忽略下面的行---当修补程序将被合并时。这是预期的行为：更改日志并不意味着永远保存在项目的 git 历史中。

以下推荐布局：

```
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
on 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
```

任何补丁修订应包括版本号。 版本号仅由字母 v 后跟大于或等于 2 的整数（即“PATCH v2”，“PATCH v3”...）组成。

通过使用选项--subject-prefix，可以使用 git format-patch 轻松处理：

```
$ git format-patch --subject-prefix "PATCH
    v4" \ -M -s -o outgoing origin/master
```

由于 git 版本 1.8.1，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，请将旧的补丁标记为拼写错误。您需要创建一个补丁帐户才能修改补丁的状态。请注意，您只能更改自己提交的修补程序的状态，这意味着您在拼写错误中注册的电子邮件地址应与用于向邮件列表发送修补程序的电子邮件地址相匹配。

在向邮件列表提交修补程序时，还可以添加--in-reply-to <message-id>选项。邮件的回复 ID 可以在拼贴上的“Message Id”标签下找到。回复的优点是，拼接将自动标记补丁的先前版本被取代。

21.6 报告问题/错误或获得帮助

在报告任何问题之前，请检查邮件列表归档第 5 章是否有人已经报告和/或修复了类似的问题。

但是，您可以通过在错误跟踪器第 5 章中打开错误或通过发送邮件到邮件列表来选择报告错误或获取帮助。第 5 章，提供了一些细节，以帮助人们重现和查找解决问题。

尝试想像你试图帮助别人;在这种情况下，你需要什么？

以下是在这种情况下提供的详细信息的简短列表：

•主机（OS / release）

•Buildroot 版本

•构建失败的目标

•构建失败的包

•失败的命令及其输出

•您认为可能相关的任何信息

此外，您应该添加.config 文件（或者如果知道 defconfig，请参见第 9.3 节）。

如果其中一些细节太大，请不要犹豫，使用 pastebin 服务。请注意，并非所有可用的粘贴剂服务都将在下载原始粘贴时保留 Unix 样式的行终止符。以下 pastebin 服务已知可正常工作：- https://gist.github.com/ - http://code.bulix.org/

第四部分 附录

第 23 章 Makedev 语法文档

在 Buildroot 的几个地方使用 makedev 语法来定义要为权限创建的更改，要创建哪些设备文件以及如何创建它们，以避免调用 mknod。

此语法派生自 makedev 实用程序，更多完整的文档可以在 package / makedevs / README 文件中找到。

它采用空格分隔的字段列表的形式，每行一个文件；领域是：

name	type	mode	uid	gid	major	minor	start	inc	count

有几个不平凡的块：

•name 是要创建/修改的文件的路径

•type 是文件的类型，是以下之一：

- f：常规文件

- d：一个目录

- r：一个目录递归

- c：一个字符设备文件

- b：块设备文件

- p：一个命名管道

•模式是通常的权限设置（仅允许数字值）

•uid 和 gid 是在此文件上设置的 UID 和 GID；可以是数值或实际名称

•主要和次要的都是设备文件，设置为 - 用于其他文件

•start，inc 和 count 是当您想要创建一批文件时，可以简化为循环，从开始开始，将其计数器递增，直到达到计数

假设您要更改给定文件的权限；使用这种语法，你需要写：

```
/usr/bin/foo f 755 0 0 - - - - -
/usr/bin/bar f 755 root root - - - - -
/data/buz f 644 buz-user buz-group - - - - -
```

或者，如果要以递归方式更改目录的所有者/权限，可以写入（将 UID 设置为 foo，将 GID 设置为禁用并访问 rwxr-x 的权限，目录/ usr / share / myapp 和所有文件 和下面的目录）：

```
/usr/share/myapp r 750 foo bar - - - - -
```

另一方面，如果要创建/ dev / hda 的设备文件和分区的相应的 15 个文件，则需要/ dev / hda：

```
/dev/hda b 640 root root 3 0 0 0 -
```

然后对应于/ dev / hda / dev / hdaX 分区的设备文件，X 范围从 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。 现在，只有功能才能作为扩展属性来支持。

|xattr	capability

•xattr 是指示扩展属性的“标志”

•capability 是添加到以前的文件的能力

如果要将功能 cap_sys_admin 添加到二进制 foo，您将写入：

```
/usr/bin/foo f 755 root root - - - - -
|xattr cap_sys_admin+eip
```

您可以使用多个| xattr 行将多个功能添加到文件。 如果要将功能 cap_sys_admin 和 cap_net_admin 添加到二进制 foo 中，您将写入：

```
/usr/bin/foo f 755 root root - - - - -
|xattr cap_sys_admin+eip
|xattr cap_net_admin+eip
```

第 24 章 Makeusers 语法文档

创建用户的语法灵感来自上面的 makedev 语法，但是特定于 Buildroot。

添加用户的语法是一个空格分隔的字段列表，每行一个用户；领域是：

username	uid	group	gid	password	home	shell	groups	comment

•用户名是用户所需的用户名（也称为登录名）。它不能是根，并且必须是唯一的。如果设置为 - ，则只会创建一个组。

•uid 是用户所需的 UID。它必须是唯一的，而不是 0.如果设置为-1，则 Buildroot 将在[1000。]范围内计算唯一的 UID。 。 。 1999 年]

•组是用户主组的所需名称。它不能是根。如果组不存在，它将被创建。

•gid 是用户主组的所需 GID。它必须是唯一的，而不是 0.如果设置为-1，并且组不存在，则 Buildroot 将在[1000..1999]范围内计算唯一的 GID

•密码是密码（3） - 密码。如果前缀为!，则登录被禁用。如果前缀为=，则将其解释为明文，并将被隐藏编码（使用 MD5）。如果前缀为！=，则密码将被密码编码（使用 MD5），登录将被禁用。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 是用户所需的主目录。如果设置为 - ，则不会创建主目录，并且用户的家将是/。明确地设置为/不允许。

•shell 是用户所需的 shell。如果设置为 - ，则/ bin / false 设置为用户的 shell。

•组是用户应该加入的其他组的逗号分隔列表。如果设置为 - ，那么用户将是没有其他组的成员。将使用任意的 gid 创建缺少的组。

•评论（又名 GECOS 字段）是一个几乎免费的文本。

每个字段的内容都有一些限制：

•除了评论，所有字段都是强制性的。

•除了注释外，字段可能不包含空格。

•没有字段可能包含冒号（:）。

如果没有，那么主目录和下面的所有文件将属于用户及其主组。

例子：

```
foo -1 bar -1 !=blabla /home/foo /bin/sh alpha,bravo Foo user
```

这将创建此用户：

•用户名（又称登录名）是：foo

•uid 由 Buildroot 计算

•主组是：bar

•主组 gid 由 Buildroot 计算

•明文密码是：blabla，将被隐藏（3）编码，登录被禁用。

•home 是：/ home / foo

•shell 是：/ bin / sh

•foo 也是组合的成员：alpha 和 bravo

•评论是：Foo 用户

```
test 8000 wheel -1 = - /bin/sh - Test user
```

这将创建此用户：

•用户名（又称登录名）是：test

•uid 是：8000

•分组：wheel

•主组 gid 由 Buildroot 计算，并将使用 rootfs 骨架中定义的值

•密码为空（也称为密码）

•home 是 / 但不属于测试

•shell 是：/ bin / sh

•测试不是任何其他组的成员

•评论是：测试用户

第 25 章 从较旧的 Buildroot 版本迁移

一些版本引入了后向不兼容性。 本节介绍这些不兼容性，并为每一个说明如何完成迁移。

25.1 迁至 2016.11

Buildroot 2016.11 之前，可以一次仅使用一个 br2 外部树。 与 Buildroot 2016.11 有可能同时使用多个（详见 9.2 节）。

然而，这意味着较旧的 br2 外部树不能原样使用。 必须做一个小的改动：在你的 br2 外部树中添加一个名字。

这可以很容易地完成，只需几个步骤：

•首先，在您的 br2 外部树的根目录下创建一个名为 external.desc 的新文件，单行定义您的 br2 外部树的名称：

```
$ echo 'name: NAME_OF_YOUR_TREE' >external.desc
```

注意选择名称时要小心：必须是唯一的，只能使用[A-Za-z0-9_]中的 ASCII 字符。

•然后，使用新变量更改 br2 外部树中的每一次 BR2_EXTERNAL 的发生：

```
$ find . -type f | xargs sed -i 's/BR2_EXTERNAL/BR2_EXTERNAL_NAME_OF_YOUR_TREE_PATH/g'
```

现在，您的 br2 外部树可以与 Buildroot 2016.11 一起使用。

注意：此更改使您的 br2 外部树与 2016 年之前的 Buildroot 不兼容。

25.2 迁移到 2017.08

在 Buildroot 2017.08 之前，主机软件包安装在$（HOST_DIR）/usr 中（例如 autotools'--prefix = $（HOS T_DIR）/usr）。 使用 Buildroot 2017.08，它们现在直接安装在$（HOST_DIR）中。

每当软件包安装与$（HOST_DIR）/lib 中的库相链接的可执行文件时，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。