Buildroot用户手册

前言

- 从git版本8a94ff12d2于2018-03-04 21:31:16 UTC生成的Buildroot 2018.02手册
- 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许可的。
- 有关此许可证的全文，请参考Buildroot源中的COPYING文件。
-

第一部分-入门

第1章 关于Buildroot

Buildroot是一个工具，它可以使用交叉编译来简化和自动化为嵌入式系统构建完整的Linux系统的过程。

为了实现这一目标，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可以独立地用于这些选项的任何组合（例如，您可以使用现有的交叉编译工具链，并仅通过Buildroot来构建根文件系统）。

Buildroot主要对使用嵌入式系统的人有用。 嵌入式系统通常使用的处理器不是每个人都已经习惯在其PC中使用的常规x86处理器。 它们可以是PowerPC处理器，MIPS处理器，ARM处理器等。

Buildroot支持众多处理器及其变体。 它还提供了一些现成可用板卡的默认配置。 除此之外，许多第三方项目都基于Buildroot或基于其开发BSP 1或SDK 2。

> 1.BSP：板级支持包
> 2.SDK：软件开发套件

第2章 系统要求

Buildroot旨在在Linux系统上运行。

尽管Buildroot会自行构建编译所需的大多数主机软件包，但某些标准Linux实用程序预计已安装在主机系统上。 您将在下面找到强制性软件包和可选软件包的概述（请注意，软件包名称在发行版之间可能有所不同）。

强制安装的包

构建工具：

- which
- sed
- make (version 3.81 or any later)
- binutils
- build-essential (only for Debian based systems)
- gcc (version 2.95 or any later)
- g++ (version 2.95 or any later)
- bash
- patch
- gzip
- bzip2
- perl (version 5.8.7 or any later)
- tar
- cpio
- python (version 2.6 or any later)
- unzip
- rsync
- file (must be in /usr/bin/file) – bc

源获取工具：

- wget

可选的包

- 配置接口依赖性：
  对于这些库，您需要同时安装运行时和开发数据，在许多发行版中，这些数据是单独打包的。
  开发软件包通常具有-dev或-devel后缀。
  – ncurses5使用menuconfig界面
  – qt4使用xconfig接口
  – glib2，gtk2和glade2使用gconfig接口
- 源获取工具：
  在官方树中，大多数软件包源都是使用wget从ftp，http或https位置检索的。 只有版本控制系统才能提供一些软件包。 此外，Buildroot能够通过其他工具（例如rsync或scp）下载源代码（有关更多详细信息，请参见第19章）。 如果使用以下任何一种方法启用软件包，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有关的软件包，如果需要为目标系统构建Java Classpath：
  – javac编译器
  – jar工具
- 文档生成工具：
  – asciidoc，版本8.6.3或更高版本
  – w3m
  –带有argparse模块的python（在2.7+和3.2+中自动存在）
  – dblatex（仅适用于pdf手册）
- 图形生成工具：
  – graphviz使用图依赖和`<pkg> -graph-`依赖
  – python-matplotlib使用图构建

第3章 获取Buildroot

Buildroot版本每2个月，2月，5月，8月和11月发布一次。 版本号的格式为YYYY.MM，例如2013.02、2014.08。

可从http://buildroot.org/downloads/获得发行压缩包。

为方便起见，Buildroot源树中的support / misc / Vagrantfile中提供了Vagrantfile，可快速设置具有所需依赖关系的虚拟机以开始使用。

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

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

如果您使用的是Windows，请将其粘贴到您的Powershell中：

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

如果要关注开发，可以使用每日快照或克隆Git存储库。 有关更多详细信息，请参考Buildroot网站的“下载”页面。

第4章 Buildroot快速入门

重要提示：您可以并且应该以普通用户身份来构建所有内容。 无需root用户即可配置和使用Buildroot。 通过以常规用户身份运行所有命令，可以保护系统免受在编译和安装过程中表现异常的软件包的侵害。

使用Buildroot的第一步是创建配置。 Buildroot有一个不错的配置工具，类似于在Linux内核或BusyBox中可以找到的工具。

从buildroot目录运行

```
$ make menuconfig
```

对于原始的基于curses的配置器，或者

```
$ make nconfig
```

对于新的基于curses的配置器，或者

```
$ make xconfig
```

基于Qt的配置器，或

$ make gconfig

用于基于GTK的配置器。

所有这些“ make”命令都将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”包。 有关更多详细信息，请参见第2章，尤其是第2.2节中的可选要求，以获取您喜欢的接口的依赖性。

对于配置工具中的每个菜单条目，您都可以找到描述条目目的的相关帮助。 有关某些特定配置方面的详细信息，请参见第6章。

完成所有配置后，配置工具将生成一个包含整个配置的.config文件。 该文件将由顶级Makefile读取。

要开始构建过程，只需运行：

```
$ make
```

永远不要将make -jN与Buildroot一起使用：当前不支持顶级并行make。 而是使用BR2_JL EVEL选项告诉Buildroot使用make -jN运行每个软件包的编译。
make命令通常将执行以下步骤：

- 下载源文件（根据需要）；
- 配置，构建和安装交叉编译工具链，或仅导入外部工具链；
- 配置，构建和安装选定的目标软件包；
- 构建内核映像（如果选择）；
- 构建引导加载程序映像（如果已选择）；
- 以选定的格式创建根文件系统。

Buildroot的输出存储在单个目录output /中。 该目录包含几个子目录：

- image：存储所有映像（内核映像，引导加载程序和根文件系统映像）的位置。这些是您需要放在目标系统上的文件。
- build：构建所有组件的位置（包括主机上Buildroot所需的工具以及为目标编译的软件包）。该目录包含每个组件的一个子目录。
- staging：包含与根文件系统层次结构相似的层次结构。该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，该目录并非旨在作为目标的根文件系统：它包含许多开发文件，未剥离的二进制文件和库，这些文件对于嵌入式系统而言太大了。这些开发文件用于为依赖于其他库的目标编译库和应用程序。
- target：几乎包含目标的完整根文件系统：除了设备外，所有需要的东西都存在/ dev /中的文件（Buildroot无法创建文件，因为Buildroot不能以root身份运行，也不想以root身份运行）。另外，它没有正确的权限（例如，busybox二进制文件的setuid）。因此，该目录不应在您的目标上使用。相反，您应该使用images /目录中内置的映像之一。如果您需要提取根文件系统的映像以通过NFS引导，请使用images /中生成的tarball映像并将其提取为root。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：不存在开发文件（标头等），二进制文件被剥离。
- host：包含为正确执行Buildroot所需的为主机编译的工具的安装，包括交叉编译工具链。

这些命令，make menuconfig | nconfig | gconfig | xconfig和make是基本的命令，它们使您可以轻松，快速地生成符合您需求的图像，并启用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具有在社区内和外部共享信息的不同方式。
如果您正在寻找帮助，想了解Buildroot或为项目做贡献，那么每种方式都可能使您感兴趣。

- 邮件列表
  Buildroot有一个讨论和开发邮件列表。 它是Buildroot用户和开发人员进行交互的主要方法。
  仅Buildroot邮件列表的订阅者可以发布到该列表。 您可以通过邮件列表信息页面进行订阅。
  发送到邮件列表的邮件也可以在邮件列表档案中找到，也可以通过Gmane在gmane.comp.lib.uclibc.buildroot中获得。 在问问题之前，请先搜索邮件列表档案，因为很有可能其他人之前也曾问过同样的问题。
- IRC

  Buildroot IRC频道#buildroot托管在Freenode上。 这是一个提出快速问题或讨论某些主题的有用地方。
  在IRC上寻求帮助时，请使用代码共享网站（例如http://code.bulix.org）共享相关日志或代码段。
  请注意，对于某些问题，将其发布到邮件列表可能会更好，因为它将吸引更多的开发人员和用户。
- 错误追踪器
  可以通过邮件列表或通过Buildroot Bugtracker报告Buildroot中的错误。 创建错误报告之前，请参阅第21.6节。
- 维基
  Buildroot Wiki页面位于eLinux Wiki上。 它包含一些有用的链接，过去和即将发生的事件的概述以及TODO列表。
- patchwork
  Patchwork是一个基于Web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补丁被系统“捕获”，并显示在网页上。
  发布的任何引用该修补程序的注释也将附加到修补程序页面。 有关Patchwork的更多信息，请参见http://jk.ozlabs.org/projects/patchwork/。
  Buildroot的Patchwork网站主要供Buildroot的维护者使用，以确保不会丢失补丁。 Buildroot修补程序审阅者也使用它（另请参阅第21.3.1节）。 但是，由于该网站在简洁明了的Web界面中公开了补丁及其相应的评论，因此它对所有Buildroot开发人员都是有用的。
  Buildroot修补程序管理界面可从http://patchwork.buildroot.org获得。

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，其中提供有关该选项的详细信息。
make * config命令还提供了搜索工具。 阅读不同的前端菜单中的帮助消息以了解如何使用它：

- 在menuconfig中，通过按/调用搜索工具
- 在xconfig中，通过按Ctrl + f调用搜索工具

搜索结果显示匹配项的帮助消息。 在menuconfig中，左栏中的数字提供了相应条目的快捷方式。 只需键入此数字即可直接跳至该条目，或在由于缺少相关性而无法选择该条目的情况下跳至包含菜单。
尽管条目的菜单结构和帮助文本应该很容易解释，但是许多主题需要额外的说明，这些内容很难在帮助文本中介绍，因此在以下各节中进行介绍。

交叉编译工具链

编译工具链是允许您为系统编译代码的一组工具。 它由一个编译器（在我们的情况下为gcc），二进制utils（例如汇编程序和链接程序）（在我们的情况下为binutils）和一个C标准库（例如GNU Libc，uClibc-ng）组成。

开发工作站上安装的系统肯定已经具有一个编译工具链，可用于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系统中，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”。 在其上运行并在其上工作的计算机称为“主机系统”。此术语不同于GNU configure所使用的术语，GNU configure所使用的主机是运行应用程序的计算机（通常与目标计算机相同）。

分发工具提供了编译工具链，并且Buildroot与它无关（除了使用它来构建交叉编译工具链和在开发主机上运行的其他工具之外）。

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

Buildroot为交叉编译工具链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”。

可以使用“工具链”菜单中的“工具链类型”选项在这两种解决方案之间进行选择。 选择一种解决方案后，将出现许多配置选项，以下各节将详细介绍这些选项。

内部工具链后端

内部工具链后端是Buildroot在构建目标嵌入式系统的用户空间应用程序和库之前，自行构建交叉编译工具链的后端。

该后端支持多个C库：uClibc-ng，glibc和musl。

选择此后端后，将显示许多选项。 最重要的允许：

- 更改用于构建工具链的Linux内核标头的版本。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。该库提供了用户空间应用程序和Linux内核之间的接口。为了知道如何与Linux内核“对话”，C库需要访问Linux内核头文件（即内核中的.h文件），这些头文件定义了用户空间和内核之间的接口（系统调用，数据结构等）。由于此接口是向后兼容的，因此用于构建工具链的Linux内核标头的版本不需要与打算在嵌入式系统上运行的Linux内核的版本完全匹配。他们只需要具有与要运行的Linux内核相同或更低的版本即可。如果使用的内核标头比嵌入式系统上运行的Linux内核更新，则C库可能正在使用Linux内核未提供的接口。
- 更改GCC编译器，binutils和C库的版本
- 选择许多工具链选项（仅uClibc）：工具链应具有RPC支持（主要用于NFS），宽字符支持，语言环境支持（用于国际化），C ++支持还是线程支持。 根据您选择的选项，在Buildroot菜单中可见的用户空间应用程序和库的数量将发生变化：许多应用程序和库需要启用某些工具链选项。 当需要某个工具链选项才能启用那些软件包时，大多数软件包都会显示注释。 如果需要，可以通过运行make uclibc-menuconfig进一步优化uClibc配置。 但是请注意，已针对Buildroot中捆绑的默认uClibc配置对Buildroot中的所有软件包进行了测试：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，无论何时修改这些选项之一，都必须重建整个工具链和系统。 请参阅第8.2节。

值得注意的是，无论何时修改这些选项之一，都必须重建整个工具链和系统。 请参阅第8.2节。

该后端的优点：

- 与Buildroot良好集成
- 快速，仅构建必要的内容

该后端的缺点：

- 做清理时需要重建工具链，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。

外部工具链后端

外部工具链后端允许使用现有的预先构建的交叉编译工具链。 Buildroot知道许多著名的交叉编译工具链（来自ARM的Linaro，ARM的Sourcery CodeBench，x86-64，PowerPC和MIPS，并且能够自动下载它们，或者可以指向自定义工具链。 ，可以下载或在本地安装。

然后，您有三种使用外部工具链的解决方案：

- 使用预定义的外部工具链配置文件，并让Buildroot下载，提取和安装工具链。 Buildroot已经知道一些CodeSourcery和Linaro工具链。 只需从可用的工具链中选择工具链配置文件。 这绝对是最简单的解决方案。
- 使用预定义的外部工具链配置文件，而不是让Buildroot下载并解压缩工具链，而是可以告诉Buildroot您的工具链已在系统上安装的位置。 只需通过可用工具栏中的工具链中选择工具链配置文件，取消选择自动下载工具链，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。 您需要填写工具链路径，工具链前缀和外部工具链C库选项。 然后，您必须告诉Buildroot您的外部工具链支持什么。 如果您的外部工具链使用glibc库，则只需告诉您工具链是否支持C ++，以及它是否具有内置的RPC支持。 如果您的外部工具链使用uClibc库，那么您必须告诉Buildroot它是否支持RPC，宽字符，语言环境，程序调用，线程和C ++。 在执行开始时，Buildroot会告诉您所选的选项是否与工具链配置不匹配。

我们的外部工具链支持已通过CodeSourcery和Linaro的工具链，crosstool NG生成的工具链以及Buildroot本身生成的工具链进行了测试。 通常，所有支持sysroot功能的工具链都应该起作用。 如果没有，请立即与开发人员联系。

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（即仅编译器，binutils，C和C ++库）。 相反，这些工具链带有大量预编译的库和程序。 因此，Buildroot无法导入工具链的sysroot，因为它将包含通常由Buildroot生成的数百兆字节的预编译库。

我们也不支持使用发行版工具链（即您的发行版中安装的gcc / binutils / C库）作为为目标构建软件的工具链。 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，也必须使用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，可以将其用作Buildroot中的外部工具链，则我们的建议绝对是使用crosstool-NG进行构建。 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。

该后端的优点：

- 允许使用众所周知且经过测试的交叉编译工具链。
- 避免了交叉编译工具链的构建时间，这在嵌入式Linux系统的总体构建时间中通常非常重要。
  该后端的缺点：
- 如果您预先构建的外部工具链有错误，则可能很难从工具链供应商处获得修复，除非您自己使用Crosstool-NG来构建外部工具链。

外部工具链包装器

使用外部工具链时，Buildroot会生成包装程序，该程序将适当的选项（根据配置）透明地传递给外部工具链程序。 如果您需要调试此包装器以检查传递的参数是正确的，则可以将环境变量BR2_DEBUG_WRAPPER设置为以下任意一个：

- 0，为空或未设置：无调试
- 1：单行跟踪所有参数
- 2：每行跟踪一个参数

/ dev管理

在Linux系统上，/ dev目录包含称为设备文件的特殊文件，这些文件允许用户空间应用程序访问Linux内核管理的硬件设备。 没有这些设备文件，即使Linux内核正确识别了硬件设备，您的用户空间应用程序也将无法使用它们。 在系统配置/ dev管理下，Buildroot提供了四种不同的解决方案来处理/ dev目录：

- 第一个解决方案是使用设备表的静态。这是在Linux中处理设备文件的传统方法。使用这种方法，设备文件会永久存储在根文件系统中（即它们在重新引导后仍然存在），并且在从系统中添加或删除硬件设备时，没有什么东西会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，默认文件存储在Buildroot源代码的system / device_table_dev.txt中。当Buildroot生成最终的根文件系统映像时，将处理此文件，因此在输出/目标目录中看不到设备文件。 BR2_ROOTFS_STATIC_DEVICE_TABLE选项允许更改Buildroot使用的默认设备表，或添加其他设备表，以便Buildroot在构建过程中创建其他设备文件。因此，如果您使用此方法，而系统中缺少设备文件，则可以例如创建包含其他设备文件描述的`board / <yourcompany> / <yourproject> /device_table_dev.txt`文件，然后您可以将BR2_ROOTFS_STATIC_DEVICE_TABLE设置为`system / dev ice_table_dev.txt板/ <您的公司> / <您的项目> /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。
- 第二种解决方案是仅使用devtmpfs进行动态处理。 devtmpfs是Linux内核中的一个虚拟文件系统，已在内核2.6.32中引入（如果使用较旧的内核，则无法使用此选项）。 在/ dev中挂载时，此虚拟文件系统将自动使设备文件在添加和从系统中删除硬件设备时显示和消失。 该文件系统在重新启动后并不持久：它由内核动态填充。 使用devtmpfs要求启用以下内核配置选项：CONFIG_DEVTMPFS和CONFIG_DEVTMPFS_MOUNT。 当Buildroot负责为嵌入式设备构建Linux内核时，请确保启用了这两个选项。 但是，如果在Buildroot之外构建Linux内核，则有责任启用这两个选项（如果失败，则Buildroot系统将不会启动）。
- 第三种解决方案是使用devtmpfs + mdev进行动态处理。此方法还依赖于上面详述的devtmpfs虚拟文件系统（因此仍然需要在内核配置中启用CONFIG_DEVTMPFS和CONFIG_DEVTMPFS_MOUNT的要求），但是在其之上添加了mdev用户空间实用程序。 mdev是BusyBox的程序部分，内核在每次添加或删除设备时都会调用。借助/etc/mdev.conf配置文件，可以将mdev配置为例如在设备文件上设置特定的权限或所有权，在设备出现或消失时调用脚本或应用程序，等等。基本上，它允许用户空间执行以下操作：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，可以使用mdev自动加载内核模块。如果您的设备需要固件，则mdev也很重要，因为它将负责将固件内容推送到内核。 mdev是udev的轻量级实现（功能较少）。有关mdev及其配置文件的语法的更多详细信息，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介绍的devtmpfs虚拟文件系统，但在其上面添加了eudev用户空间守护程序。 eudev是一个后台运行的守护程序，当从系统中添加或删除设备时，内核会调用它。 与mdev相比，它是重量级的解决方案，但具有更高的灵活性。 eudev是udev的独立版本，udev是大多数桌面Linux发行版中使用的原始用户空间守护程序，现已成为Systemd的一部分。 有关更多详细信息，请参见`http://en.wikipedia.org/wiki/Udev`

Buildroot开发人员的建议是从仅使用devtmpfs的Dynamic解决方案开始，直到需要在添加/删除设备或需要固件时通知用户空间为止，在这种情况下，通常最好使用devtmpfs + mdev进行Dynamic处理。解。

请注意，如果选择systemd作为初始化系统，则/ dev管理将由systemd提供的udev程序执行。
Buildroot开发人员的建议是从仅使用devtmpfs的Dynamic解决方案开始，直到需要在添加/删除设备或需要固件时通知用户空间为止，在这种情况下，通常最好使用devtmpfs + mdev进行Dynamic处理，请注意，如果选择systemd作为初始化系统，则/ dev管理将由systemd提供的udev程序执行。

初始化系统

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

Buildroot允许使用三种不同类型的初始化系统，可以从系统配置（初始化系统）中进行选择：

- 第一个解决方案是BusyBox。 在许多程序中，BusyBox都有一个基本的初始化程序的实现，对于大多数嵌入式系统而言，这已经足够了。 启用BR2_INIT_BUSYBOX将确保BusyBox将生成并安装其init程序。 这是Buildroot中的默认解决方案。 BusyBox初始化程序将在引导时读取/ etc / inittab文件，以了解操作方法。 可以在`http://git.busybox.net/busybox/tree/examples/inittab`中找到此文件的语法（请注意，BusyBox inittab语法很特殊：请勿使用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。 除了安装一些重要的文件系统之外，默认的inittab的主要工作是启动/etc/init.d/rcS shell脚本，并启动一个getty程序（提供登录提示）。
- 第二种解决方案是systemV。 该解决方案使用旧的传统sysvinit程序，该程序打包在Buildroot中，位于package / sysvinit中。 这是大多数桌面Linux发行版中使用的解决方案，直到他们切换到更新的替代版本（如Upstart或Systemd）为止。 sysvinit也可用于inittab文件（其语法与BusyBox中的语法略有不同）。 随此init解决方案一起安装的默认inittab位于package / sysvinit / inittab中。
- 第三种解决方案是systemd。 systemd是用于Linux的新一代init系统。 它的功能远远超过传统的init程序：积极的并行化功能，使用套接字和D-Bus激活来启动服务，按需启动守护进程，使用Linux控制组跟踪进程，支持快照和恢复系统状态， systemd在相对复杂的嵌入式系统上很有用，例如需要D-Bus和服务之间相互通信的系统。 值得注意的是，systemd带来了大量的大型依赖项：dbus，udev等。 有关systemd的更多详细信息，请参见`http://www.freedesktop.org/wiki/Software/systemd`。

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

第7章 其他组件的配置

在尝试修改下面的任何组件之前，请确保您已经配置了Buildroot本身，并启用了相应的软件包。

BusyBox
如果您已有BusyBox配置文件，则可以使用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，尽管不建议这样做。 有关更多详细信息，请参见第8.6节。

uClibc
uClibc的配置与BusyBox的配置相同。 用于指定现有配置文件的配置变量为BR2_UCLIBC_CONFIG。 进行后续更改的命令是make uclibc-menuconfig。

Linux内核
如果已经有了内核配置文件，则可以使用BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG在Buildroot配置中直接指定此文件。 如果您还没有内核配置文件，则可以使用BR2_LINUX_KERNEL_USE_DEFCONFIG在Buildroot配置中指定一个defconfig开始，或者使用BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG创建一个空文件并将其指定为自定义配置文件开始。 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的配置与Linux内核的配置相同。 相应的配置变量是BR2_TARGET_BAREBOX_USE_CUSTOM_CONFIG和BR2_TARGET_BAREBOX_USE_DEFCONFIG。 要打开配置编辑器，请使用make nudebox-menuconfig

U-Boot
U-Boot（版本2015.04或更高版本）的配置与Linux内核的配置相同。 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 Buildroot的一般用法

make 的一些技巧

这是一系列技巧，可帮助您充分利用Buildroot。
显示make执行的所有命令：

> $ make V=1 `<target>`

显示带有defconfig的板列表：

> $ make list-defconfigs

显示所有可用的目标：

> $ make help

并非所有目标都始终可用，.config文件中的某些设置可能会隐藏一些目标：

- busybox-menuconfig仅在启用busybox时有效；
- linux-menuconfig和linux-savedefconfig仅在启用linux时起作用；
- 仅当在内部工具链后端中选择了uClibc C库时，uclibc-menuconfig才可用。
- 仅在启用了裸箱引导加载程序时，beardbox-menuconfig和beardbox-savedefconfig才起作用。
- uboot-menuconfig和uboot-savedefconfig仅在启用U-Boot引导加载程序时起作用。

清理：更改任何体系结构或工具链配置选项时，都需要显式清洁。 要删除所有构建产品（包括构建目录，主机，暂存和目标树，映像和工具链）：

> $ make clean

生成手册：当前的手册源位于docs / manual目录中。 生成手册：

> $ make manual-clean
> $ make manual

手动输出将在output / docs / manual中生成。
注意：需要一些工具来构建文档（请参阅：2.2节）

为新目标重置Buildroot：删除所有构建产品和配置：

> $ make distclean

注意如果启用了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，请参阅第8.12.3节。

转储内部make变量：可以转储已知的所有make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会将列表限制为名称与指定的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）将打印未扩展的值

例如：

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

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

$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES RAW_VARS=YES
BUSYBOX_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_ALL_DEPENDENCIES=$(sort $(BUSYBOX_FINAL_DEPENDENCIES)
$(BUSYBOX_FINAL_PATCH_DEPENDENCIES))
BUSYBOX_FINAL_DEPENDENCIES=$(sort $(BUSYBOX_DEPENDENCIES))
BUSYBOX_FINAL_PATCH_DEPENDENCIES=$(sort $(BUSYBOX_PATCH_DEPENDENCIES))
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

带引号的变量的输出可以在shell脚本中重用，例如：

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

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不会尝试检测应重建系统的哪些部分。 在某些情况下，Buildroot应该重建整个系统，在某些情况下，仅应重建软件包的特定子集。 但是以完全可靠的方式检测到这一点非常困难，因此Buildroot开发人员已决定不尝试这样做。

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

- 更改目标体系结构配置时，需要完全重建。 例如，更改体系结构变体，二进制格式或浮点策略会对整个系统产生影响。
- 更改工具链配置时，通常需要完全重建。 更改工具链配置通常涉及更改编译器版本，C库的类型或其配置或其他一些基本配置项，这些更改会影响整个系统。
- 将其他程序包添加到配置时，不一定需要完全重建。 Buildroot将检测到此软件包从未构建过，并将对其进行构建。 但是，如果此程序包是可以选择供已构建的程序包使用的库，则Buildroot将不会自动重建它们。 您或者知道应该重建哪些软件包，或者可以手动重建它们，或者应该进行完全重建。 例如，假设您使用ctorrent软件包构建了一个系统，但没有openssl。 您的系统可以运行，但是您意识到要在ctorrent中获得SSL支持，因此在Buildroot配置中启用openssl软件包并重新启动构建。 Buildroot将检测到应该构建并构建openssl，但是它不会检测到应该重新构建ctorrent以从openssl中受益，以添加OpenSSL支持。 您将必须进行完全重建，或重建ctorrent本身
- 从配置中删除软件包时，Buildroot不会执行任何特殊操作。 它不会从目标根文件系统或工具链sysroot中删除此软件包安装的文件。 需要完全重建才能摆脱此软件包。 但是，通常您不必立即删除此软件包：您可以等待下一个午餐时间以从头开始重新构建。

- 更改软件包的子选项时，不会自动重建软件包。 进行此类更改后，仅重建该软件包通常就足够了，除非启用package子选项，否则将某些功能添加到该软件包中，这些功能对于已经构建的另一个软件包有用。 同样，Buildroot不会跟踪何时应该重新构建软件包：一旦构建了软件包，就不会对其进行重新构建，除非明确要求这样做。
- 更改根文件系统框架后，需要完全重建。 但是，在更改根文件系统覆盖图，进行后构建脚本或后映像脚本时，无需完全重建：简单的make调用将考虑这些更改。

一般来说，当您遇到构建错误并且不确定所做的配置更改可能带来的后果时，请进行完全重建。 如果您遇到相同的构建错误，则可以确定该错误与软件包的部分重建无关，并且如果官方Buildroot的软件包发生此错误，请立即报告该问题！ 随着您对Buildroot的了解的发展，您将逐步了解何时真正需要进行完全重建，并且可以节省越来越多的时间。

供参考，可以通过运行来完全重建

```
$ make clean all
```

了解如何重建软件包

Buildroot用户提出的最常见问题之一是如何重建给定的软件包或如何删除软件包而不重新构建所有内容。

Buildroot不支持删除软件包，而无需从头开始重建。 这是因为Buildroot不会跟踪哪个软件包安装了output / staging和output / target目录中的哪些文件，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将从头开始重新提取，重新配置，重新编译和重新安装此软件包。 您可以要求makeroot使用`make <package> -dirclean`命令执行此操作。

另一方面，如果只想从编译步骤重新开始软件包的构建过程，则可以运行`make <package> -rebuild`，然后运行make或`make <package>`。 它将重新启动软件包的编译和安装，但不会从头开始：它基本上在软件包内部重新执行make和make install，因此它将仅重建已更改的文件。

如果要从配置步骤重新启动软件包的构建过程，可以运行`make <package> -reconfigure`，然后运行make或`make <package>`。
它将重新启动软件包的配置，编译和安装。
在内部，Buildroot创建所谓的图章文件以跟踪每个软件包已完成的构建步骤。 它们存储在软件包的构建目录下，即`output / build / <package>-<version> /`，并命名为`.stamp_ <step name>`。 上面详细介绍的命令仅操作这些标记文件即可强制Buildroot重新启动软件包构建过程的一组特定步骤。

有关包装特殊制造目标的更多详细信息，请参见第8.12.5节。

离线版本

如果打算进行脱机构建，而只想下载以前在配置器中选择的所有源（menuconfig，nconfig，xconfig或gconfig），则输入：

```
make source
```

现在，您可以断开dl目录的内容或将其复制到构建主机。

建立树外

默认情况下，Buildroot生成的所有内容都存储在Buildroot树的目录输出中。
Buildroot还支持使用类似于Linux内核的语法从树中构建。 要使用它，请将`O = <directory>`添加到make命令行：

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

或者

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

所有输出文件将位于/ tmp / build下。 如果O路径不存在，则Buildroot将创建它。
注意：O路径可以是绝对路径，也可以是相对路径，但是如果以相对路径传递，则需要注意的是，它是相对于Buildroot主目录而不是当前工作目录进行解释的。

使用树外版本时，Buildroot .config和临时文件也存储在输出目录中。 这意味着只要它们使用唯一的输出目录，就可以使用同一源代码树安全地并行运行多个构建。
为了易于使用，Buildroot在输出目录中生成一个Makefile包装器-因此，在第一次运行后，您不再需要传递`O = <...>`和`-C <...>`，只需在输出目录中运行）

> $ `make <target>`

环境变量

当传递给环境中的make或set时，Buildroot还尊重一些环境变量：

- HOSTCXX，要使用的主机C ++编译器
- HOSTCC，要使用的主机C编译器
- UCLIBC_CONFIG_FILE = <path / to / .config>，是要构建内部工具链时用于编译uClibc的uClibc配置文件的路径。注意，还可以从Buildroot .config文件从配置界面设置uClibc配置文件。 这是建议的设置方式。
- BUSYBOX_CONFIG_FILE = <路径/至/.config>，即BusyBox配置文件的路径。
  请注意，也可以从Buildroot .config文件从配置界面设置BusyBox配置文件。 这是建议的设置方式。
- BR2_CCACHE_DIR会覆盖使用ccache时Buildroot存储缓存文件的目录。
- BR2_DL_DIR覆盖Buildroot存储/检索下载文件的目录
  请注意，还可以从Buildroot .config文件从配置界面设置Buildroot下载目录。 有关如何设置下载目录的更多详细信息，请参见第8.12.4节。
- BR2_GRAPH_ALT（如果已设置且为非空值），则在构建时图表中使用备用颜色方案
- BR2_GRAPH_OUT设置生成的图形的文件类型，为pdf（默认值）或png。
- BR2_GRAPH_DEPS_OPTS将额外的选项传递给依赖图； 有关可接受的选项，请参见第8.8节
- 将BR2_GRAPH_DOT_OPTS作为选项逐字传递给点实用程序以绘制依赖关系图。

使用位于顶层目录和$ HOME中的配置文件的示例：

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

如果要使用默认gcc或g ++以外的其他编译器在主机上构建辅助二进制文件，请执行

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

有效处理文件系统映像

文件系统映像可能会变得很大，这取决于您选择的文件系统，程序包数，是否已配置可用空间。 。 。
但是，文件系统映像中的某些位置可能只是空的（例如，长时间为零）； 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，并且只会存储或写入稀疏文件中不为空的部分。
例如：

- tar接受-S选项来告诉它仅存储稀疏文件的非零块：
  – tar cf archive.tar -S [files ...]将有效地将稀疏文件存储在tarball中
  – tar xf archive.tar -S将有效地存储从tarball提取的稀疏文件
- cp接受--sparse = WHEN选项（WHEN是自动，永不或总是自动之一）：
  – cp --sparse =总是source.file dest.file将使dest.file成为稀疏文件（如果source.file长期运行为零）

其他工具可能具有类似的选项。 请查阅各自的手册页。
如果您需要存储文件系统映像（例如，从一台计算机传输到另一台计算机），或者需要将它们发送（例如，发送给问答团队），则可以使用稀疏文件。
但是请注意，在使用dd的稀疏模式时将文件系统映像刷新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，如果文件系统中的文件稀疏，则这些部分可能会损坏）。 读回时不是全零）。 您仅应在构建计算机上处理文件时使用稀疏文件，而不是在将文件传输到将在目标计算机上使用的实际设备时使用稀疏文件。

绘制软件包之间的依赖关系

Buildroot的工作之一就是了解软件包之间的依赖关系，并确保它们以正确的顺序构建。 这些依赖有时可能非常复杂，对于给定的系统，通常不容易理解为什么Buildroot将这样的软件包引入了构建。 为了帮助理解依赖性，从而更好地理解嵌入式Linux系统中不同组件的作用，Buildroot能够生成依赖性图。
要生成已编译的整个系统的依赖关系图，只需运行：

```
make graph-depends
```

您可以在output / graphs / graph-depends.pdf中找到生成的图形。 如果您的系统很大，则依赖关系图可能太复杂且难以阅读。 因此，可以仅针对给定的包生成依赖关系图：

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

您可以在`output / graph / <pkg> -graph-depends.pdf`中找到生成的图形。
请注意，依赖关系图是使用Graphviz项目中的点工具生成的，您必须已将其安装在系统上才能使用此功能。 在大多数发行版中，它可以作为graphviz软件包使用。 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，可以切换到其他输出格式，例如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
```

可以通过设置BR2_GRAPH_DEPS_OPTS环境变量中的选项来控制依赖图形的行为。
可接受的选项是：

- --depth N，-d N，用于将依赖深度限制为N个级别。 默认值为0，表示没有限制。
- --stop-on PKG，-s PKG，以停止包装PKG上的图形。 PKG可以是实际的软件包名称，全局名称，关键字virtual（在虚拟软件包上停止）或关键字host（在主机软件包上停止）。 该程序包仍然存在于图上，但其依赖性不存在。
- -排除PKG，-x PKG，例如--stop-on，但也从图中省略了PKG。
- --transitive，--no-transitive，用于绘制（或不绘制）传递相关性。 默认为不绘制传递依赖。
- --colours R，T，H，以逗号分隔的颜色列表，用于绘制根包（R），目标包（T）和主机包（H）。 默认为：浅蓝色，灰色，gainsboro

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

绘制构建持续时间

当系统构建花费很长时间时，有时能够了解最长构建哪些软件包，查看是否可以采取任何措施来加快构建速度有时会很有用。 为了帮助进行此类构建时间分析，Buildroot收集每个软件包每个步骤的构建时间，并允许从该数据生成图形。 要在构建后生成构建时间图，请运行：

```
make graph-build
```

这将在输出/图形中生成一组文件：

- build.hist-build.pdf，每个软件包的构建时间的直方图，按构建顺序排序。

- build.hist-duration.pdf，每个软件包的构建时间直方图，按持续时间排序（最长的时间）
- build.hist-name.pdf，每个软件包的构建时间直方图，按软件包名称排序。
- build.pie-packages.pdf，每个包的构建时间的饼图
- build.pie-steps.pdf，这是在软件包构建过程的每个步骤中花费的全球时间的饼图。

此图形构建目标需要安装Python Matplotlib和Numpy库（在大多数发行版中均安装python-matplotlib和python-numpy），如果要使用的Python版本早于2.7，则还需要安装argparse模块（在大多数情况下，使用python-argparse 分布）。

默认情况下，图形的输出格式为PDF，但是可以使用BR2_GRAPH_OUT环境变量选择其他格式。 支持的唯一其他格式是PNG：

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

用图形表示软件包的文件系统大小

当目标系统增长时，了解每个Buildroot软件包对整个根文件系统大小的贡献有时会很有用。 为了帮助进行此类分析，Buildroot收集有关每个软件包安装的文件的数据，并使用此数据，生成图形和CSV文件，详细说明不同软件包的大小。

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，每个包对根文件系统总大小的贡献的饼图
- output / graphs / package-size-stats.csv，一个CSV文件，提供每个程序包对总根文件系统大小的贡献
- output / graphs / file-size-stats.csv，一个CSV文件，提供每个已安装文件对其所属软件包的大小以及整个文件系统大小的贡献。

此图形大小的目标需要安装Python Matplotlib库（在大多数发行版中都需要安装python-matplotlib），并且如果您使用的Python版本早于2.7（在大多数发行版中则是python-argparse），则还需要安装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，请参见第8.8节。

注意仅在完全干净的重建之后，收集的文件系统大小数据才有意义。 使用make graph-size之前，请确保全部运行make clean。

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

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

与Eclipse集成

嵌入式Linux开发人员的一部分，例如Vim或Emacs等经典文本编辑器，以及基于命令行的界面，而其他许多嵌入式Linux开发人员，例如更丰富的图形界面，来完成他们的开发工作。 Eclipse是最受欢迎的集成开发环境之一，Buildroot与Eclipse集成在一起以简化Eclipse用户的开发工作。

我们与Eclipse的集成简化了在Buildroot系统上构建的应用程序和库的编译，远程执行和远程调试。 它不集成Buildroot配置，也不使用Eclipse构建自身的进程。 因此，我们的Eclipse集成的典型用法模型为：
使用make menuconfig，make xconfig或Buildroot提供的任何其他配置界面来配置Buildroot系统。

- 通过运行make构建Buildroot系统。
- 启动Eclipse以开发，执行和调试您自己的自定义应用程序和库，它们将依赖于Buildroot构建和安装的库。

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

高级用法

在Buildroot外部使用生成的工具链

您可能希望针对自己的目标，编译自己的程序或其他未打包在Buildroot中的软件。 为此，您可以使用Buildroot生成的工具链。

默认情况下，由Buildroot生成的工具链位于output / host /中。 最简单的使用方法是将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，这可以将工具链分发给其他开发人员以为您的目标构建应用程序。 为达到这个：

- 运行make sdk，以准备可重定位的工具链；
- 压缩输出/主机目录的内容；
- 分发生成的tarball。
  将工具链安装到新位置后，用户必须运行relocate-sdk.sh脚本以确保所有路径均已使用新位置更新。

在Buildroot中使用gdb

Buildroot允许进行交叉调试，其中调试器在构建计算机上运行，并与目标上的gdbserver通信以控制程序的执行。

为达到这个：

- 如果使用内部工具链（由Buildroot构建），则必须启用BR2_PACKAGE_HOST_GDB，BR2_PACKAGE_GDB和BR2_PACKAGE_GDB_SERVER。 这样可以确保交叉gdb和gdbserver均已构建，并且gdbserver已安装至目标。
- 如果使用的是外部工具链，则应启用BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY，这会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跨gdb或gdbserver，也可以通过启用与内部工具链后端相同的选项，让Buildroot来构建它们。
  现在，要开始调试名为foo的程序，应在目标计算机上运行：

> gdbserver :2345 foo

这将导致gdbserver在TCP端口2345上侦听来自交叉gdb的连接。
然后，在主机上，应该使用以下命令行启动交叉gdb：

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

当然，foo必须在使用调试符号构建的当前目录中可用。 通常，您从构建foo的目录（而不是从output / target /）中启动此命令，因为该目录中的二进制文件被剥离了。
`<buildroot> / output / staging / usr / share / buildroot / gdbinit`文件将告诉交叉gdb在哪里找到目标库。最后，从交叉gdb连接到目标：

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

在Buildroot中使用ccache

ccache是编译器缓存。 它存储每个编译过程产生的目标文件，并能够通过使用预先存在的目标文件来跳过将来对同一源文件（具有相同的编译器和相同的参数）的编译。 从头开始进行几乎相同的构建多次时，它可以很好地加快构建过程。

ccache支持集成在Buildroot中。 您只需要在“生成”选项中启用“启用编译器缓存”即可。 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，以便可以由单独的Buildroot构建共享。 如果要摆脱缓存，只需删除该目录。

您可以通过运行make ccache-stats来获取有关缓存的统计信息（其大小，命中次数，未命中次数等）。maketarget ccache-options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对源文件和编译器选项进行哈希处理。 如果编译器选项不同，则将不使用缓存的目标文件。 但是，许多编译器选项都包含登台目录的绝对路径。 因此，在不同的输出目录中构建将导致许多高速缓存未命中。

为避免此问题，buildroot具有使用相对路径选项（BR2_CCACHE_USE_BASEDIR）。 这会将指向输出目录内部的所有绝对路径重写为相对路径。 因此，更改输出目录不再导致
缓存未命中。

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

有关重写绝对路径的更多详细信息，请参见ccache手册的“在不同目录中编译”部分。

下载包的位置

Buildroot下载的各种压缩文件都存储在BR2_DL_DIR中，默认情况下为dl目录。 如果您想保留一个完整的Buildroot版本，该版本可以与相关的tarball一起使用，则可以复制此目录。 这将允许您使用完全相同的版本重新生成工具链和目标文件系统。

如果维护多个Buildroot树，则最好具有共享的下载位置。 这可以通过将BR2_DL_DIR环境变量指向目录来实现。 如果设置了此设置，则Buildroot配置中的BR2_DL_DIR的值将被覆盖。 将以下行添加到`<~/.bashrc>`中。 `export BR2_DL_DIR = <共享下载位置>`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，此值被BR2_DL_DIR环境变量覆盖。

特定于包装的目标

运行`make <package>`将构建并安装该特定的软件包及其依赖项。
对于依赖Buildroot基础结构的软件包，有许多特殊的make目标，可以像这样单独调用：

> `make <package>-<target>`

软件包构建目标是（按照它们执行的顺序）：

命令/目标	描述
资源	获取源代码（下载压缩包，克隆源代码仓库等）
依赖	构建并安装构建程序包所需的所有依赖项
解压	将源代码放入包构建目录中（解压缩tarball，复制源代码等）
补丁	应用补丁（如果有）
配置	运行configure命令（如果有）
构建	运行编译命令
安装阶段	目标软件包：如果需要，请在登台目录中运行软件包的安装
安装目标	目标软件包：如有必要，在目标目录中运行软件包的安装
安装	目标软件包：运行之前的2个安装命令 主机软件包：在主机目录中运行软件包的安装

此外，还有一些其他有用的make目标：

命令/目标	描述
显示依赖	显示构建程序包所需的依赖关系
图形化依赖	在当前Buildroot配置的上下文中，生成程序包的依赖关系图。 有关依赖关系图的更多详细信息，请参见第8.8节。
完全清理	删除整个软件包构建目录
重新安装	重新运行安装命令
重建	重新运行编译命令-仅在使用OVERRIDE_SRCDIR功能或直接在构建目录中修改文件时，这才有意义
重新配置	重新运行configure命令，然后重新构建-仅在使用OVERRIDE_SRCDIR功能或直接在构建目录中修改文件时，这才有意义

在开发过程中使用Buildroot

Buildroot的正常操作是下载tarball，解压缩，配置，编译和安装该tarball中的软件组件。 源代码被提取到`output / build / <package>-<version>`中，这是一个临时目录：每当使用make clean时，此目录将被完全删除，并在下一次make调用时重新创建。 即使将Git或Subversion存储库用作包源代码的输入，Buildroot也会从中创建一个tarball，然后像对待tarball一样正常工作。

当Buildroot主要用作集成工具来构建和集成嵌入式Linux系统的所有组件时，此行为非常适合。 但是，如果在系统的某些组件的开发过程中使用Buildroot，则此行为不是很方便：而是希望对一个软件包的源代码进行少量更改，并能够使用Buildroot快速重建系统。

直接在`output / build / <package>-<version>`中进行更改是不合适的解决方案，因为在make clean时会删除此目录。

因此，Buildroot针对此用例提供了一种特定的机制：`<pkg> _OVERRIDE_SRCDIR`机制。 Buildroot读取一个覆盖文件，该文件允许用户告诉Buildroot某些软件包的源位置。 默认情况下，此替代文件名为local.mk，位于Buildroot源树的顶层目录中，但是可以通过BR2_PACKAGE_OVERRIDE_FILE配置选项指定其他位置。

在此替代文件中，Buildroot希望找到以下形式的行：

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

For example:

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

当Buildroot发现给定软件包时，已经定义了`<pkg> _OVERRIDE_SRCDIR`，它将不再尝试下载，提取和修补软件包。 而是直接使用指定目录中可用的源代码，make clean不会触摸此目录。 这允许将Buildroot指向您自己的目录，该目录可以由Git，Subversion或任何其他版本控制系统进行管理。 为此，Buildroot将使用rsync将组件的源代码从指定的`<pkg> _OVERRIDE_SRCDIR`复制到`output / build / <package> -custom /`。

此机制最好与`make <pkg> -rebuild`和`make <pkg> -reconfigure`目标结合使用。
一个`make <pkg> -rebuild all`序列将把源代码从`<pkg> _OVERRIDE_SRCDIR`同步到`output / build / <package> -custom`（由于rsync，仅复制修改过的文件），并重新开始构建过程包。

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

```
make linux-rebuild all
```

并在几秒钟内在输出/图像中获取更新的Linux内核图像。 同样，可以在/ home / bob / busybox中以及之后对BusyBox源代码进行更改：

```
make busybox-rebuild all
```

输出/映像中的根文件系统映像包含更新的BusyBox。

第9章 特定于项目的定制

对于给定的项目，您可能需要执行的典型操作是：

- 配置Buildroot（包括构建选项和工具链，引导加载程序，内核，软件包和文件系统映像类型选择）
- 配置其他组件，例如Linux内核和BusyBox
- 自定义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）
- 在生成文件系统映像后运行任意命令（使用BR2_ROOTFS_POST_IMAGE_SCRIPT）
- 向某些程序包添加特定于项目的补丁程序（使用BR2_GLOBAL_PATCH_DIR）
- 添加特定于项目的软件包

有关此类特定于项目的自定义项的重要说明：请仔细考虑哪些更改确实是特定于项目的，哪些更改对项目外的开发人员也有用。 Buildroot社区强烈建议并鼓励上游改进，软件包和董事会对Buildroot官方项目的支持。 当然，由于更改是高度特定的或专有的，因此有时上游不可能或不希望这样做。

本章介绍了如何在Buildroot中进行此类特定于项目的自定义，以及如何以一种可重现的方式构建同一映像的方式来存储它们，即使在运行make clean之后也是如此。 通过遵循推荐的策略，您甚至可以使用同一Buildroot树来构建多个不同的项目！

推荐目录结构

为项目自定义Buildroot时，将创建一个或多个需要存储在某个地方的特定于项目的文件。 尽管大多数这些文件可以放置在任何位置，因为它们的路径将在Buildroot配置中指定，但是Buildroot开发人员建议使用本节中描述的特定目录结构。

与该目录结构正交，您可以选择将其本身放置在何处：在Buildroot树中还是在外部使用br2-external树。 这两个选项均有效，选择取决于您。

```
+-- 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)
```

本章将进一步介绍上面显示的文件的详细信息。

注意：如果您选择将此结构放置在Buildroot树之外但在br2-external树中，则`<company>`以及可能的`<boardname>`组件可能是多余的，可以省去。

实施分层定制

用户拥有一些需要部分自定义项的相关项目是很常见的。 建议不要使用分层的自定义方法，而不是为每个项目复制这些自定义，如本节中所述。

Buildroot中几乎所有可用的自定义方法（例如，生成后脚本和根文件系统覆盖）都接受以空格分隔的项目列表。 始终按从左到右的顺序处理指定的项目。 通过创建多个这样的项目，一个用于通用的自定义项，另一个用于真正的特定于项目的自定义项，可以避免不必要的重复。 每层通常由`board / <company> /`中的单独目录体现。 根据您的项目，您甚至可以引入两个以上的层。

用户具有两个通用和fooboard自定义层的目录结构示例是：

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

> BR2_GLOBAL_PATCH_DIR=`"board/<company>/common/patches board/<company>/fooboard/patches"`
> 然后首先将应用来自公共层的补丁，然后是来自fooboard层的补丁。

将定制保留在Buildroot之外

如9.1节所述，您可以在两个位置放置特定于项目的自定义项：

- 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，以便轻松升级到较新的Buildroot版本。
- 使用br2-external机制在Buildroot树之外。 这种机制允许将软件包配方，电路板支持和配置文件保留在Buildroot树之外，同时仍将它们很好地集成到构建逻辑中。 我们将此位置称为br2外部树。 本节说明如何使用br2外部机制以及在br2外部树中提供的内容。

通过将BR2_EXTERNAL生成变量设置为要使用的br2外部树的路径，可以告诉Buildroot使用一棵或多棵br2外部树。 可以将其传递给任何Buildroot make调用。 它会自动保存在输出目录中隐藏的.br-external.mk文件中。 因此，无需在每次make调用时都传递BR2_EXTERNAL。 但是，可以随时通过传递新值来更改它，也可以通过传递空值来删除它。

注意br2外部树的路径可以是绝对路径或相对路径。 如果将其作为相对路径传递，则必须注意，它是相对于主Buildroot源目录而不是相对于Buildroot输出目录进行解释的。

注意：如果使用Buildroot 2016.11之前的br2-external树，则需要先对其进行转换，然后才能将其用于Buildroot 2016.11及更高版本。 请参阅第25.1节以获取帮助。

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树中的定义：

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

我们可以随时切换到另一棵br2-external树：

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

我们还可以使用多个br2外部树：

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

或禁用任何br2-external树的用法：

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

br2-外部树的布局

br2外部树必须至少包含这三个文件，如以下各章所述：

- external.desc
- external.mk
- 配置
  除了那些强制性文件之外，br2外部树中可能还存在其他可选内容，例如configs /目录。 以下各章也对它们进行了描述。
  稍后还将介绍完整的br2-external树布局示例。

external.desc文件

该文件描述了br2-external树：该br2-external树的名称和描述。
该文件的格式基于行，每行以关键字开头，后跟冒号和一个或多个空格，然后是分配给该关键字的值。
当前识别出两个关键字：

- 名称（必填）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；禁止使用其他任何字符。 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br2外部树。该变量在Kconfig中都可用，因此您可以使用它来为您的Kconfig文件（请参见下文）和Makefile提供源，以便可以使用它来包含其他Makefile（请参见下文）或引用其他文件（如数据文件） ）从您的br2外部树中。
  注意：由于可以一次使用多个br2外部树，因此Buildroot使用此名称为每个树生成变量。该名称用于标识您的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，以使其相对唯一，以免与其他br2的其他名称冲突。 -外部树，尤其是当您计划以某种方式与第三方共享br2外部树或使用第三方的br2外部树时。
- desc（可选）为该br2外部树提供简短描述。 它应该放在一行上，并且大部分是自由格式的（见下文），并且在显示有关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）。 该描述在`BR2_EXTERNAL _ $（NAME）_DESC`变量中可用。

名称和相应的`BR2_EXTERNAL _ $（NAME）_PATH`变量的示例：

- FOO→BR2_EXTERNAL_FOO_PATH
- BAR_42→BR2_EXTERNAL_BAR_42_PATH
  在以下示例中，假定名称设置为BAR_42。
  注意：Kconfig文件和Makefile文件中都提供了`BR2_EXTERNAL _ $（NAME）_PATH`和`BR2_EXTERNAL _ $（NAME）_DESC`。 它们也可以在环境中导出，因此可以在构建后，图像后和fakeroot脚本中使用。

Config.in和external.mk文件

这些文件（可能每个都为空）可用于定义程序包配方（即foo / Config.in和foo / foo.mk，就像Buildroot本身捆绑的程序包一样）或其他自定义配置选项或make逻辑。

Buildroot自动从每个br2-external树中包含Config.in，使其出现在顶层配置菜单中，并从每个br2-external树中包含external.mk以及其余的makefile逻辑。

此方法的主要用途是存储包装配方。 推荐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，创建一个external.mk文件，如下所示：

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

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

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。

configs /目录

可以将Buildroot defconfigs存储在br2-external树的configs子目录中。 Buildroot将在make list-defconfigs的输出中自动显示它们，并允许使用常规的make <name> _defconfig命令加载它们。 它们将显示在make list-defconfigs输出中的“外部配置”标签下，该标签包含定义它们的br2-external树的名称。

注意：如果多个br2外部树中存在defconfig文件，则使用最后一个br2外部树中的一个。 因此，可以覆盖Buildroot或另一个br2-external树中捆绑的defconfig。

自由格式的内容

可以在其中存储所有特定于主板的配置文件，例如内核配置，根文件系统覆盖或Buildroot允许为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，您可以按以下方式将路径设置为全局补丁目录，rootfs覆盖图和内核配置文件的路径（例如，通过运行make menuconfig并填写以下选项）：

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

示例布局

这是一个使用br2-external的所有功能的示例布局（示例内容显示在其上方的文件中，当与br2外部树的解释相关时；当然，所有这些都是为了说明而完全组成的 ）

```
/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_PATH)/board/my-
board/kernel.config"
|   `----
|
|- patches/linux/0001-some-change.patch
|- patches/linux/0002-some-other-change.patch
|- patches/busybox/0001-fix-something.patch
|
|- board/my-board/kernel.config
|- board/my-board/overlay/var/www/index.html
|- board/my-board/overlay/var/www/my.css
|- board/my-board/flash-image
`- board/my-board/post-image.sh
    |#!/bin/sh
```

```
    |generate-my-binary-image \
    |   --root ${BINARIES_DIR}/rootfs.tar \
    |   --kernel ${BINARIES_DIR}/zImage \
    |   --dtb ${BINARIES_DIR}/my-board.dtb \
    |   --output ${BINARIES_DIR}/image.bin
    `----
```

br2-external树然后将在menuconfig中可见（布局已展开）：

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

如果您使用了不止一棵br2-external树，则它看起来像（扩展了布局，第二棵树的名称为FOO_27，但external.desc中没有desc：字段）：

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

存储Buildroot配置

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

通过删除默认值的配置选项，可以简化Buildroot配置。 结果存储在名为defconfig的文件中。 如果要将其保存在其他位置，请在Buildroot配置本身中更改BR2_DEFCONFIG选项，或使用`make savedefconfig BR2_DEFCONFIG = <path-to-defconfig>`调用make。

推荐的存储该defconfig的位置是`configs / <boardname> _defconfig`。 如果遵循此建议，则配置将列在make帮助中，并且可以通过运行`make <boardname> _defconfi`g进行重新设置。

或者，您可以将文件复制到任何其他位置，并使用`make defconfig BR2_DEFCONFIG = <defconfig-file的路径>`进行重建。

存储其他组件的配置

如果已更改，还应该存储BusyBox，Linux内核，Barebox，U-Boot和uClibc的配置文件。 对于这些组件中的每一个，都存在Buildroot配置选项以指向输入配置文件，例如BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE。 要存储它们的配置，请将这些配置选项设置为要保存配置文件的路径，然后使用下面描述的帮助器目标实际存储配置。

如9.1节所述，建议存储这些配置文件的路径是`board / <company> / <boardname> /foo.config`。

更改BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE等选项之前，请确保已创建配置文件。 否则，Buildroot将尝试访问该配置文件，该配置文件尚不存在，并且将失败。 您可以通过运行make linux menuconfig等来创建配置文件。

Buildroot提供了一些帮助程序目标，以简化配置文件的保存。

- make linux-update-defconfig将linux配置保存到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。
- make busybox-update-config将busybox配置保存到BR2_PACKAGE_BUSYBOX_CONFIG指定的路径。
- make uclibc-update-config将uClibc配置保存到BR2_UCLIBC_CONFIG指定的路径。
- make palebox-update-defconfig将裸箱配置保存到BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE指定的路径。
- make uboot-update-defconfig将U-Boot配置保存到BR2_TARGET_UBOOT_CUSTOM_CONFIG_FILE指定的路径。
- 对于at91bootstrap3，不存在任何帮助程序，因此您必须手动将配置文件复制到BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE。

自定义生成的目标文件系统

除了通过make * config更改配置之外，还有其他几种方法可以自定义生成的目标文件系统。
可以共存的两种推荐方法是根文件系统覆盖和后构建脚本。

根文件系统覆盖（BR2_ROOTFS_OVERLAY）
文件系统覆盖是一棵文件树，在构建目标文件系统后，将其直接复制到目标文件系统上。 要启用此功能，请将配置选项BR2_ROOTFS_OVERLAY（在“系统配置”菜单中）设置为覆盖的根。 您甚至可以指定多个以空格分隔的叠加层。 如果指定相对路径，则它将相对于Buildroot树的根。 版本控制系统的隐藏目录（如.git，.svn，.hg等），名为.empty的文件和以∽结尾的文件将从副本中排除。

如9.1节所示，此覆盖的推荐路径是`board / <company> / <boardname> / rootfs`覆盖。

生成后脚本（BR2_ROOTFS_POST_BUILD_SCRIPT
生成后脚本是在Buildroot生成所有选定软件之后但在组装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_POST_BUILD_SCRIPT中（在系统配置菜单中）指定以空格分隔的后生成脚本列表。如果指定相对路径，则它将相对于Buildroot树的根。

使用构建后脚本，可以删除或修改目标文件系统中的任何文件。但是，您应谨慎使用此功能。每当发现某个程序包生成错误或不需要的文件时，都应修复该程序包，而不要使用某些生成后的清理脚本来解决它。

如9.1节所示，此脚本的推荐路径为`board / <company> / <boardname> /post_build.sh`。

构建后脚本以Buildroot主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，则这些参数也将传递给脚本。所有脚本都将传递完全相同的参数集，不可能将不同的参数集传递给每个脚本。

此外，您还可以使用以下环境变量：

- 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：所有二进制文件（也称为图像）的存储位置
- BASE_DIR：基本输出目录
  下面介绍了三种自定义目标文件系统的方法，但不建议使用。

直接修改目标文件系统
对于临时修改，您可以直接修改目标文件系统并重建映像。 目标文件系统在output / target /下可用。进行更改后，运行make来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，但是如果您需要使用make clean清理Buildroot树，这些更改将丢失。 在某些情况下，必须进行此类清洁，有关详细信息，请参阅第8.2节。 因此，该解决方案仅对快速测试有用：更改无法在make clean命令中保留。 确认更改后，您应确保使用根文件系统覆盖或构建后脚本在清理后仍将其保留。

自定义目标框架（BR2_ROOTFS_SKELETON_CUSTOM）
根文件系统映像是从目标框架创建的，所有软件包都在其上面安装其文件。在构建和安装任何软件包之前，将框架复制到目标目录输出/目标。默认的目标框架提供了标准的Unix文件系统布局以及一些基本的初始化脚本和配置文件。

如果默认框架（在system / skeleton下可用）不符合您的需求，则通常会使用根文件系统覆盖或构建后脚本来适应它。但是，如果默认框架与您所需的框架完全不同，则使用自定义框架可能更合适。

要启用此功能，请启用配置选项BR2_ROOTFS_SKELETON_CUSTOM并将BR2_ROOTFS_SKELETON_CUSTOM_PATH设置为自定义框架的路径。这两个选项在系统配置菜单中均可用。如果指定相对路径，则它将相对于Buildroot树的根。

不建议使用此方法，因为它会复制整个框架，从而阻止利用Buildroot更高版本中对默认框架带来的修复或改进。

仿制后根脚本（BR2_ROOTFS_POST_FAKEROOT_SCRIPT）
汇总最终映像时，该过程的某些部分需要root权限：在/ dev中创建设备节点，设置文件和目录的权限或所有权。 。 。为了避免需要实际的root用户权限，Buildroot使用fakeroot模拟root用户权限。这并不能完全代替实际成为root用户，但足以满足Buildroot的需求。

假根目录后脚本是外壳脚本，它们在fakeroot阶段结束时即在调用文件系统映像生成器之前被调用。因此，它们在fakeroot上下文中被调用。

如果需要调整文件系统以执行通常仅对root用户可用的修改，则后fakeroot脚本可能会很有用。

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

注意：生成后脚本（上面）和fakeroot脚本之间的区别在于，在fakeroot上下文中不调用生成后脚本。

注意;使用fakeroot并不能绝对替代实际成为root。 fakeroot只伪造文件访问权限和类型（常规，块或字符设备...）和uid / gid；这些是在内存中模拟的。

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的权限或所有权。例如，某些文件可能需要由root拥有。由于构建后脚本不是作为root用户运行的，因此除非您在构建后脚本中使用显式的falseroot，否则您无法从那里进行此类更改。

而是，Buildroot为所谓的权限表提供支持。要使用此功能，请将配置选项BR2_ROOTFS_DEVICE_TABLE设置为权限表的空格分隔列表，这些权限表是遵循makedev语法第23章的常规文本文件。

如果您使用的是静态设备表（即未使用devtmpfs，mdev或（e）udev），则可以使用相同的语法在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设置为设备表的空格分隔列表。

如9.1节所示，此类文件的推荐位置是`board / <company> / <boardname> /`。

请注意，如果特定的权限或设备节点与特定的应用程序相关，则应在程序包的.mk文件中设置变量FOO_PERMISSIONS和FOO_DEVICES（请参见第17.5.2节）。

添加自定义用户帐户

有时需要在目标系统中添加特定用户。 为了满足此要求，Buildroot为所谓的用户表提供支持。 要使用此功能，请将配置选项BR2_ROOTFS_USERS_TABLES设置为以空格分隔的用户表列表，这是遵循makeusers语法第24章的常规文本文件。

如9.1节所示，此类文件的推荐位置是`board / <company> / <boardname> /`。

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

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构建后脚本（第9.5节）时，在创建所有图像之后，可以使用后映像脚本执行某些特定操作。

例如，映像后脚本可用于在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，请在配置选项BR2_ROOTFS_POST_IMAGE_SCRIPT中（在系统配置菜单中）指定以空格分隔的后映像脚本列表。如果指定相对路径，则它将相对于Buildroot树的根。

就像生成后脚本一样，运行后映像脚本时会将Buildroot主树作为当前工作目录。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_ARGS不为空，则这些参数也将传递给脚本。所有脚本都将传递完全相同的参数集，不可能将不同的参数集传递给每个脚本。

同样，就像生成后脚本一样，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。

映像后脚本将以执行Buildroot的用户身份执行，该用户通常不应该是root用户。因此，在这些脚本之一中需要root权限的任何操作都将需要特殊处理（使用fakeroot或sudo），该处理留给脚本开发人员。

添加特定于项目的补丁

在Buildroot中提供的补丁程序之上，有时将额外的补丁程序应用于软件包很有用。 例如，这可能用于支持项目中的自定义功能，或者在使用新体系结构时。

BR2_GLOBAL_PATCH_DIR配置选项可用于指定一个或多个包含软件包补丁程序的目录的空格分隔列表。

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

1. 对于存在于BR2_GLOBAL_PATCH_DIR中的每个目录`-<global-patch-dir>`，将按以下方式确定`<package-patch dir>`：
    - `<global-patch-dir> / <packagename> / <packageversion> /`（如果目录存在）。
    - 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：
    - 如果程序包目录中存在系列文件，则根据系列文件应用补丁；
    - 否则，匹配`* .patch`的补丁文件将按字母顺序应用。 因此，为确保以正确的顺序应用它们，强烈建议使用以下名称来命名补丁文件：`<number>-<description> .patch`，其中`<number>`是指应用顺序。

有关如何将补丁应用于软件包的信息，请参见第18.2节。

BR2_GLOBAL_PATCH_DIR选项是为软件包指定自定义补丁目录的首选方法。 它可用于为buildroot中的任何软件包指定补丁程序目录。 还应使用它代替可用于软件包（例如U-Boot和Barebox）的自定义补丁程序目录选项。 这样，它将允许用户从一个顶级目录管理其补丁程序。

BR2_GLOBAL_PATCH_DIR是指定自定义补丁的首选方法，其例外是BR2_LINUX_KERNEL_PATCH。 应该使用BR2_LINUX_KERNEL_PATCH来指定URL上可用的内核补丁。 注意：BR2_LINUX_KERNEL_PATCH指定在BR2_GLOBAL_PATCH_DIR中可用的补丁之后应用的内核补丁，因为它是从Linux软件包的补丁后挂钩完成的。

添加特定于项目的软件包

通常，任何新软件包都应直接添加到软件包目录中，然后提交给Buildroot上游项目。 通常，如何在Buildroot中添加软件包的方法将在第17章中详细介绍，在此不再赘述。 但是，您的项目可能需要一些无法上游的专有软件包。 本节将说明如何将此类特定于项目的软件包保存在特定于项目的目录中。

如9.1节所示，针对特定项目的软件包的推荐位置为`package / <company> /`。 如果您使用br2-external树功能（请参见9.2节），建议的位置是将其放在br2-external树中名为package /的子目录中。

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

Buildroot将自动在包目录的第一级子目录中包含.mk文件（使用模式package / * / *。mk）。 如果我们希望Buildroot从更深的子目录（如`package / <company> / package1 /`）中包含.mk文件，则只需在包含这些附加.mk文件的第一级子目录中添加.mk文件。 因此，使用以下内容创建文件`package / <company> / <company> .mk`（假定`package / <company> /`下只有一个额外的目录级别）：

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

对于Config.in文件，创建一个文件包`/<company>/Config.in`，其中包括所有包的Config.in文件。 由于kconfig的source命令不支持通配符，因此必须提供详尽的列表。 例如：

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

从package / Config.in中包含此新文件`package / <company> /Config.in`，最好包含在公司特定的菜单中，以使与以后的Buildroot版本的合并更加容易。

如果使用br2-external树，请参见第9.2节以了解如何填写这些文件

存储特定于项目的自定义的快速指南

在本章的前面，已经描述了进行特定于项目的自定义的不同方法。 现在，本节将通过提供有关存储特定于项目的自定义设置的分步说明来总结所有这一切。 显然，可以跳过与您的项目无关的步骤。

1. make menuconfig配置工具链，软件包和内核。
2. 使linux-menuconfig更新内核配置，类似于其他配置，例如busybox，uclibc 、。 。 。
3. `mkdir -p board / <制造商> / <boardname>`
4. 将以下选项设置为`board / <制造商> / <boardname> / <package> .config`（只要相关）。
    - BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE
    - BR2_PACKAGE_BUSYBOX_CONFIG
    - BR2_UCLIBC_CONFIG
    - BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE
    - BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE
    - BR2_TARGET_UBOOT_CUSTOM_CONFIG_FILE

5. 编写配置文件：
    - 使linux-update-defconfig
    - 使busybox-update-config
    - 使uclibc-update-config
    - `cp <输出> / build / at91bootstrap3-* /。configboard / <制造商> / <板名> /at91bootstrap3.config`
    - 使barebox-update-defconfig
    - 使uboot-update-defconfig
6. 创建`board / <制造商> / <boardname> / rootfs-overlay /`，并在rootfs上填充所需的其他文件，例如`board / <制造商> / <boardname> / rootfs-overlay / etc / inittab`。将BR2_ROOTFS_OVERLAY设置为`board / <制造商> / <boardname> / rootfs-overla`。
7. 创建一个构建后脚本`board / <制造商> / <boardname> /post_build.sh`。将BR2_ROOTFS_POST_BUILD_SCRIPT设置为`board / <制造商> / <boardname> /post_build.sh`
8. 如果必须设置其他setuid权限或必须创建设备节点，请创建`board / <制造商> / <boardname> /device_table.txt`并将该路径添加到BR2_ROOTFS_DEVICE_TABLE。
9. 如果必须创建其他用户帐户，请创建`board / <制造商> / <boardname> / users_table`。txt，然后将该路径添加到BR2_ROOTFS_USERS_TABLES。
10. 要将自定义补丁添加到某些软件包，请将BR2_GLOBAL_PATCH_DIR设置为`board / <manufacturer> / <boardname> / patches /`，然后将每个软件包的补丁添加到以该软件包命名的子目录中。每个修补程序应称为`<程序包名称>-<num>-<说明> .patch`。
11. 特别针对Linux内核，还存在选项BR2_LINUX_KERNEL_PATCH，其主要优点是它还可以从URL下载补丁。如果不需要它，则首选BR2_GLOBAL_PATCH_DIR。 U Boot，Barebox，at91bootstrap和at91bootstrap3也具有单独的选项，但与BR2_GLOBAL_PATCH_DIR相比，它们没有任何优势，将来可能会被删除。
12. 如果需要添加特定于项目的软件包，请创建`package / <manufacturer> /`，然后将软件包放在该目录中。创建一个整体的`<manufacturer> .mk`文件，其中包括所有软件包的.mk文件。创建一个整体Config.in文件，该文件将为所有软件包提供Config.in文件。包括Buildroot的package / Config.in文件中的Config.in文件。
13. make savedefconfig保存buildroot配置。
14. `cp defconfig configs / <板名> _defconfig`

第10章 常见问题和故障排除

启动网络后，引导挂起

如果引导过程似乎在以下消息之后挂起（消息不一定完全相似，具体取决于所选软件包的列表）：

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

则表示您的系统正在运行，但没有在串行控制台上启动外壳程序。 为了使系统在串行控制台上启动外壳，您必须进入Buildroot配置，在系统配置中，在启动后修改Run a getty（登录提示符），并在getty options子菜单中设置适当的端口和波特率。 。 这将自动调整生成的系统的/ etc / inittab文件，以便外壳程序在正确的串行端口上启动。

为什么目标上没有编译器?

已决定从Buildroot-2012.11版本中停止对目标上的本机编译器的支持，因为：

- 此功能既未维护也未测试，并且经常损坏；
- 此功能仅适用于Buildroot工具链；
- Buildroot主要针对板载资源有限（CPU，内存，大容量存储）的小型或小型目标硬件，在目标上进行编译没有太大意义；
- Buildroot旨在简化交叉编译，从而无需在目标上进行本机编译。 如果仍然需要在目标计算机上使用编译器，则Buildroot不适合您的目的。 在这种情况下，您需要一个实际的发行版，并且应该选择以下内容：
    - openembedded
    - yocto
    - emdebian
    - Fedora
    - openSUSE ARM
    - Arch Linux ARM

为什么目标上没有开发文件?

由于目标上没有可用的编译器（请参见第10.2节），因此浪费头文件或静态库的空间是没有意义的。 因此，自Buildroot-2012.11发行以来，这些文件总是从目标中删除。

为什么目标上没有文档?

由于Buildroot主要针对具有有限板载资源（CPU，内存，大容量存储）的小型或小型目标硬件，因此浪费文档数据空间是没有意义的。 如果仍然需要有关目标的文档数据，则Buildroot不适合您的目的，您应该寻找一个真正的发行版（请参阅：10.2节）

为什么有些软件包在Buildroot配置菜单中不可见?

如果某个软件包存在于Buildroot树中，并且没有出现在config菜单中，则最有可能意味着未满足某些软件包的依赖关系。

要了解有关软件包依赖性的更多信息，请在配置菜单中搜索软件包符号（请参见第8.1节）。

然后，您可能必须递归启用几个选项（与未满足的依赖项相对应）才能最终选择包。

如果由于某些未满足的工具链选项而导致该包不可见，那么您当然应该运行完整的重建（有关更多说明，请参见第8.1节）。

为什么不将目标目录用作chroot目录?

有很多原因不使用chroot目标目录，其中包括：

- 在目标目录中未正确设置文件所有权，模式和权限；
- 设备节点未在目标目录中创建。

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

如果要在chroot中或作为NFS根目录运行目标文件系统，请使用images /中生成的tarball图像并将其提取为root。

为什么Buildroot不生成二进制包（.deb，.ipkg ...）?

Buildroot列表上经常讨论的一项功能是“程序包管理”的一般主题。 总而言之，该想法是增加对哪个Buildroot软件包安装哪些文件的跟踪，目标是：

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

通常，大多数人认为这很容易做到：只跟踪安装了哪个软件包，然后在取消选择该软件包时将其删除。但是，它比这复杂得多：

- 它不仅与目标/目录有关，而且还与host / / sysroot中的sysroot和host /目录本身有关。必须跟踪各种软件包在这些目录中安装的所有文件。
- 从配置中取消选择软件包时，仅删除其安装的文件是不够的。还必须删除其所有反向依赖关系（即依赖它的程序包）并重建所有这些程序包。例如，程序包A可选地依赖于OpenSSL库。两者都被选中，并且构建了Buildroot。程序包A使用OpenSSL进行加密支持。稍后，从配置中取消选择OpenSSL，但保留了程序包A（因为OpenSSL是可选的依赖项，所以可能）。如果仅删除OpenSSL文件，则由程序包A安装的文件将被破坏：它们使用的库是不再出现在目标上。尽管从技术上讲这是可行的，但它为Buildroot增加了很多复杂性，这与我们一直坚持的简单性背道而驰。
- 除了先前的问题外，Buildroot甚至还不了解可选依赖项。例如，版本1.0中的程序包A从未使用过OpenSSL，但在版本2.0中，它会自动使用OpenSSL（如果有）。如果尚未更新Buildroot .mk文件以将其考虑在内，则程序包A将不属于OpenSSL的反向依赖项，并且在删除OpenSSL时也不会删除和重建。当然，应该修复包A的.mk文件，以提及此可选依赖项，但与此同时，您可以具有不可复制的行为。
- 该请求还允许将menuconfig中的更改应用于输出目录，而不必从头开始重新构建所有内容。但是，以可靠的方式很难做到这一点：更改包的子选项时会发生什么（我们必须检测到这一点，并从头开始构建包，并可能重新构建所有的依赖关系），如果使用工具链选项会发生什么目前，Buildroot所做的工作既清晰又简单，因此其行为非常可靠，并且易于支持用户。如果在下一个make之后应用menuconfig中所做的配置更改，则它必须在所有情况下都能正确正确地工作，并且不会出现一些异常情况。风险在于获取如下错误报告：“我启用了程序包A，B和C，然后运行了make，然后禁用了程序包C，然后启用了程序包D并运行了make，然后重新启用了程序包C和启用了程序包E，然后有一个构建失败”。或更糟糕的是，“我先进行一些配置，然后构建，然后进行一些更改，然后进行构建，再进行一些更改，然后再进行构建，但现在失败了，但是我不记得我所做的所有更改以及更改顺序”。这将不可能得到支持。

由于所有这些原因，得出的结论是，添加跟踪已安装文件以在未选择软件包时将其删除，或生成二进制软件包的存储库是很难可靠实现的，并且会增加很多复杂性。在此问题上，Buildroot开发人员发表以下立场声明：

- Buildroot努力使生成根文件系统变得容易（顺便说一下，因此得名）。这就是我们要使Buildroot擅长的：构建根文件系统。
- Buildroot并不是要成为发行版（或更确切地说，是发行版生成器。）大多数Buildroot开发人员认为这不是我们应该追求的目标。我们相信，还有比Buildroot更适合生成发行版的其他工具。例如，Open Embedded或openWRT就是这样的工具。
- 我们倾向于将Buildroot推向易于（甚至更容易）生成完整根文件系统的方向。这就是Buildroot在人群中脱颖而出的原因（当然还有其他事情！）
- 我们认为，对于大多数嵌入式Linux系统，二进制软件包不是必需的，并且可能有害。使用二进制软件包时，这意味着可以部分升级系统，这会产生大量可能的软件包版本组合，应在嵌入式设备上进行升级之前进行测试。另一方面，通过一次升级整个根文件系统映像来进行完整的系统升级，可以保证部署到嵌入式系统的映像确实是经过测试和验证的映像。

如何加快构建过程?

由于Buildroot通常涉及对整个系统进行完整的重建，这可能会很长，因此我们在下面提供了一些技巧来帮助减少构建时间：

- 使用预构建的外部工具链，而不是默认的Buildroot内部工具链。 通过使用预构建的Linaro工具链（在ARM上）或Sourcery CodeBench工具链（用于ARM，x86，x86-64，MIPS等），您将在每次完全重建时节省该工具链的构建时间，大约为15 20分钟。 请注意，在系统的其余部分正常工作后，临时使用外部工具链并不能防止您切换回内部工具链（这可能提供更高级别的自定义）。
- 使用ccache编译器缓存（请参阅第8.12.3节）；
- 了解有关仅重建您实际关心的少数软件包的信息（请参阅第8.3节），但请注意，有时仍然需要完全重建（请参阅第8.2节）；
- 确保您没有将虚拟机用于运行Buildroot的Linux系统。 众所周知，大多数虚拟机技术都会对I / O产生重大的性能影响，这对于构建源代码确实非常重要。
- 确保仅使用本地文件：请勿尝试通过NFS进行构建，这会显着降低构建速度。 在本地拥有Buildroot下载文件夹也有帮助。
- 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

- 如果此类选项包含$符号，则无法通过BR2_TARGET_LDFLAGS传递额外的链接器选项。 例如，以下内容被打破：BR2_TARGET_LDFLAGS =“-Wl，rpath ='$ ORIGIN /../ lib'”
- SuperH 2和ARC体系结构不支持libffi软件包。
- prboom软件包使用Sourcery CodeBench 2012.09版的SuperH 4编译器触发了编译器故障。

第12章 法律声明和许可

符合开源许可证

Buildroot的所有最终产品（工具链，根文件系统，内核，引导加载程序）均包含开源软件，并已获得各种许可。

使用开放源代码软件，您可以自由地构建丰富的嵌入式系统，可以从各种程序包中进行选择，但还必须承担一些必须了解和兑现的义务。 某些许可证要求您在产品文档中发布许可证文本。 其他要求您将软件的源代码重新分发给接收产品的软件。

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

```
make legal-info
```

Buildroot将在您的输出目录中legal-info /子目录下收集与法律相关的资料。在那里您会发现：

- README文件，该文件汇总了所产生的材料并包含有关Buildroot无法产生的材料的警告。
- buildroot.config：这是通常由make menuconfig生成的Buildroot配置文件，对于重新生成构建是必需的。
- 所有软件包的源代码；它分别保存在目标和主机程序包的source /和host-sources /子目录中。设置为`<PKG> _REDISTRIBUTE = NO`的软件包的源代码将不会保存。还保存了已应用的补丁以及名为series的文件，该文件按应用顺序列出了补丁。修补程序与其修改的文件具有相同的许可证。注意：Buildroot将附加补丁程序应用于基于自动工具的软件包的Libtool脚本。这些修补程序可以在Buildroot源文件中的support / libtool下找到，由于技术限制，它们不会与软件包源文件一起保存。您可能需要手动收集它们。
- 一个清单文件（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。其中某些信息可能未在Buildroot中定义；这些项目被标记为“未知”。
- 所有软件包的许可证文本，分别位于目标软件包和主机软件包的licenses /和host-licenses /子目录中。如果未在Buildroot中定义许可证文件，则不会生成该文件，并且自述文件中的警告指出了这一点。

请注意，Buildroot的legal-info功能的目的是生产与合法遵守软件包许可证相关的所有材料。 Buildroot不会尝试生成必须以某种方式公开的确切材料。当然，所生产的材料超过了严格遵守法律所需要的材料。例如，它为在类似BSD的许可证下发布的软件包生成源代码，您无需以源形式重新分发。

此外，由于技术限制，Buildroot不会生成您将需要或可能需要的某些材料，例如工具链源代码和Buildroot源代码本身（包括需要分发源代码的软件包的补丁程序）。当您运行make legal-info时，Buildroot会在README文件中生成警告，以通知您无法保存的相关材料。

最后，请记住，make legal-info的输出基于每个软件包配方中的声明性语句。 Buildroot开发人员会尽其所能，尽最大努力使这些声明性语句尽可能准确。但是，这些声明性陈述很可能不是全部完全准确也不是详尽无遗。您（或您的法律部门）必须先检查make legal-info的输出，然后才能将其用作自己的合规性交付。请参阅Buildroot发行版根目录下COPYING文件中的NO WARRANTY子句（第11和12条）。

符合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（可选）任何更高版本发布，以下详述的软件包修补程序除外。但是，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计算机上，而不存在于设备存储中。

但是，Buildroot开发人员的普遍看法是，在发布包含GPL许可软件的产品时，您应同时发布Buildroot源代码和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，加上任何相关的接口定义文件，以及用于控制可执行文件的编译和安装的脚本”。 Buildroot是用于控制可执行文件的编译和安装的脚本的一部分，因此，它被视为必须重新分发的材料的一部分。

请记住，这只是Buildroot开发人员的意见，如果有任何疑问，应咨询法律部门或律师。

打包补丁

Buildroot还捆绑了补丁文件，这些补丁文件应用于各种软件包的源。 这些修补程序不受Buildroot许可的保护。 相反，它们受应用了补丁的软件的许可保护。 当上述软件在多个许可证下均可用时，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 buildroot之外

引导生成的镜像

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
```

切记将此路径添加到/ etc / exports。 然后，您可以从目标执行NFS引导。

现场CD

要生成实时CD映像，请在“文件系统映像”菜单中启用iso映像选项。 注意，此选项仅在x86和x86-64体系结构上可用，并且如果您正在使用Buildroot构建内核。

您可以使用IsoLinux，Grub或Grub 2作为引导加载程序来构建实时CD映像，但是只有Isolinux支持将此映像同时用作实时CD和实时USB（通过“构建混合映像”选项）。

您可以使用QEMU测试现场CD映像：

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

如果是混合ISO，也可以将其用作硬盘映像：

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

可以使用dd轻松将其刷新到USB驱动器：

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

chroot

如果要在生成的映像中使用chroot，那么您应该注意的几件事：

- 您应该从tar根文件系统映像中设置新的根；
- 所选目标体系结构与您的主机兼容，或者您应该使用一些qemu- *二进制文件并在binfmt属性中正确设置它，以便能够在您的主机上运行为目标构建的二进制文件；
- Buildroot当前未提供正确构建和设置用于这种用途的host-qemu和binfmt

第三部分-开发指南

第14章 Buildroot如何工作

如上所述，Buildroot基本上是一组Makefile，可以使用正确的选项下载，配置和编译软件。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）中涉及的软件包。

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

- 工具链/目录包含与交叉编译工具链相关的所有软件的Makefile文件和相关文件：binutils，gcc，gdb，内核头文件和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Buildroot可以编译并添加到目标根文件系统的所有用户空间工具和库的Makefile和相关文件。每个软件包有一个子目录。
- linux /目录包含Linux内核的Makefile和相关文件。
- boot /目录包含Buildroot支持的Bootloader的Makefile和相关文件。
- system /目录包含对系统集成的支持，例如目标文件系统框架和初始化系统的选择。
- fs /目录包含与目标根文件系统映像的生成相关的软件的Makefile和相关文件。

每个目录至少包含2个文件：

- something.mk是用于下载，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。它描述了与软件包有关的选项。

主Makefile执行以下步骤（一旦完成配置）：

- 在输出目录中创建所有输出目录：暂存，目标，构建等（默认情况下为output /，可以使用O =指定其他值）
- 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。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Buildroot环境。
- 生成TARGETS变量中列出的所有目标。此变量由所有单个组件的Makefile填充。生成这些目标将触发用户空间包（库，程序），内核，引导加载程序的编译以及根文件系统映像的生成，具体取决于配置。

第15章 编码风格

总体而言，这些编码样式规则可以帮助您在Buildroot中添加新文件或重构现有文件。
如果您稍微修改一些现有文件，那么重要的是保持整个文件的一致性，因此您可以：

- 遵循此文件中可能弃用的编码样式，
- 或完全重做以使其符合这些规则。

.config配置文件

Config.in文件包含Buildroot中几乎所有可配置项目的条目。
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/`
```

- 布尔值，取决于，选择和帮助行以一个选项卡缩进。
- 帮助文本本身应缩进一个标签和两个空格。
- 帮助文本应被包装为适合72列，其中制表符为8，因此文本本身为62个字符。
  Config.in文件是Buildroot（常规的Kconfig）中使用的配置工具的输入。 有关Kconfig语言的更多详细信息，请参阅`http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt`。

.mk文件

- 标头：文件以标头开头。 它包含模块名称，最好用小写字母，并包含在由80个哈希组成的分隔符之间。 标头后必须有一个空白行：

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

- 分配：use =前面有一个空格：

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

不要将=符号对齐。

- 缩进：仅使用选项卡：

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

请注意，定义块内的命令应始终以制表符开头，因此make会将其识别为命令。

- 可选依赖项：

– Prefer multi-line syntax.
YES:

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

NO:

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

– 保持配置选项和依赖关系紧密靠近。

- 可选的挂钩：如果一个代码块将挂钩的定义和分配保持在一起。

YES:

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

NO:

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

文档

该文档使用asciidoc格式。
有关asciidoc语法的更多详细信息，请参见
`http://www.methods.co.nz/asciidoc/userguide.html`。

支持脚本

support /和utils /目录中的某些脚本是用Python编写的，应遵循PEP8 Style Guide for Python Code。

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，因此该板的用户可以轻松地构建已知可以正常工作的系统。也欢迎您为Buildroot添加对其他板的支持。

为此，您需要创建一个普通的Buildroot配置，该配置为硬件构建一个基本系统：工具链，内核，引导加载程序，文件系统和一个仅BusyBox专用的用户空间。不应选择特定的程序包：配置应尽可能少，并且应仅为目标平台构建可运行的基本BusyBox系统。当然，您可以为内部项目使用更复杂的配置，但是Buildroot项目将仅集成基本的电路板配置。这是因为软件包选择是高度特定于应用程序的。

有了已知的工作配置后，运行make savedefconfig。这将在Buildroot源树的根目录下生成一个最小的defconfig文件。将此文件移到configs /目录，并将其重命名为`<boardname> _defconfig`。

建议使用尽可能多的Linux内核和引导程序上游版本，并使用尽可能多的默认内核和引导程序配置。如果它们对于您的电路板不正确或不存在默认值，我们建议您将修复程序发送到相应的上游项目。

但是，与此同时，您可能需要存储特定于目标平台的内核或引导程序配置或补丁。为此，创建一个`目录板/ <制造商>`和一个`子目录板/ <制造商> / <板名>`。然后，您可以将修补程序和配置存储在这些目录中，并从Buildroot主配置中引用它们。有关更多详细信息，请参见第9章。