第 6 章 构建软件包

目录

现在我们已经为构建软件包做好了准备。

6.1. 完整的(重)构建

为保证完整的软件包(重)构建能顺利进行，你必须保证系统中已经安装

- build-essential 软件包；
- 列于 Build-Depends 域的软件包(参看 第 4.1 节 “control”)；
- 列于 Build-Depends-indep 域的软件包(参看 第 4.1 节 “control”)。

然后在源代码目录中执行以下命令：

```
$ dpkg-buildpackage -us -uc
```

这样会自动完成所有从源代码包构建二进制包的工作，包括：

- 清理源代码树(debian/rules clean)
- 构建源代码包(dpkg-source -b)
- 构建程序(debian/rules build)
- 构建二进制包(fakeroot debian/rules binary)
- 制作 .dsc 文件
- 用 dpkg-genchanges 命令制作 .changes 文件。

如果构建结果令人满意，那就用 debsign 命令以你的私有 GPG 密钥签署 .dsc 文件和 .changes 文件。你需要输入密码两次。[63]

对于非本土 Debian 软件包，比如 gentoo，构建软件包之后，你将会在上一级目录(~/gentoo) 中看到下列文件：

- gentoo_0.9.12.orig.tar.gz

  这是原始的源代码 tarball，最初由 dh_make -f ../gentoo-0.9.12.tar.gz 命令创建，它的内容与上游 tarball 相同，仅被重命名以符合 Debian 的标准。

- gentoo_0.9.12-1.dsc

  这是一个从 control 文件生成的源代码概要，可被 dpkg-source(1) 程序解包。

- gentoo_0.9.12-1.debian.tar.gz

  这个压缩的 Tar 归档包含你的 debian 目录内容。其他所有对于源代码的修改都由 quilt 补丁存储于 debian/patches 中。

  如果其他人想要重新构建你的软件包，他们可以使用以上三个文件很容易地完成。只需复制三个文件，再运行 dpkg-source -x gentoo_0.9.12-1.dsc。[64]

- gentoo_0.9.12-1_i386.deb

  这是你的二进制包，可以使用 dpkg 程序安装或卸载它，就像其他软件包一样。

- gentoo_0.9.12-1_i386.changes

  这个文件描述了当前修订版本软件包中的全部变更，它被 Debian FTP 仓库维护程序用于安装二进制和源代码包。它是部分从 changelog 和 .dsc 文件生成的。

  随着你不断完善这个软件包，程序的行为会发生变化，也会有更多新特性添加进来。下载你软件包的人可以查看这个文件来快速找到有哪些变化，Debian 仓库维护程序还会把它的内容发表至 debian-devel-changes@lists.debian.org 邮件列表。

在上传到 Debian FTP 仓库中前，gentoo_0.9.12-1.dsc 文件和 gentoo_0.9.12-1_i386.changes 文件必须用 debsign 命令签署，其中使用你自己存放在 ~/.gnupg/ 目录中的 GPG 私钥。用你的公钥，可以令 GPG 签名证明这些文件真的是你的。

debsign 命令可以用来以指定 ID 的 GPG 密钥进行签署（这方便了赞助(sponsor)软件包），只要照着下边在 ~/.devscripts 中的内容：

```
DEBSIGN_KEYID=Your_GPG_keyID
```

.dsc 和 .changes 文件中很长的数字串是其中提及文件的 SHA1/SHA256 校验和。下载你软件包的人可以使用 sha1sum(1) 或 sha256sum(1) 来进行核对。如果校验和不符，则说明文件已被损坏或偷换。

6.2. 自动编译系统

Debian 支持非常多的 移植(port)，同时它有着 自动构建网络，这个网络在不同体系结构的计算机上运行着 buildd 守护进程。虽然你不需要自己做这件事情，你也应该知道在你的软件包身上发生了什么。让我们来简要地看看他们是如何为你给多个体系结构构建软件包的。[65]

对于 Architecture: any 的软件包，自动编译系统重构建它们。它确保系统中已经安装

- build-essential 软件包；
- 列于 Build-Depends 域的软件包(参看 第 4.1 节 “control”)。

然后在源代码目录中执行以下命令：

```
$ dpkg-buildpackage -B
```

这样会自动完成从源代码包构建平台依赖二进制包的工作，包括：

- 清理源代码树(debian/rules clean)
- 构建程序(debian/rules build)
- 构建平台依赖二进制包(fakeroot debian/rules binary-arch)
- 使用 gpg 签署 .dsc 文件
- 使用 dpkg-genchanges 和 gpg 创建并签署上传用的 .changes 文件

这就是你看到你的软件包在其他平台上可用的原因。

尽管通常的打包工作中 Build-Depends-indep 字段中列出的软件包都需要安装(参看 第 6.1 节 “完整的(重)构建”)，但是在编译平台依赖二进制包时它们无需在自动编译系统上安装。[66]通常打包和自动编译系统的这种不同为你指出如何考虑必须的软件包应如何放在 debian/control 文件的 Build-Depends 或 Build-Depends-indep 域中(参看 第 4.1 节 “control”)。

6.3. debuild 命令

你可以使用 debuild 命令来进一步自动化 dpkg-buildpackage 的构建过程。参看 debuild(1)

debuild 命令会执行 lintian 命令，以在 Debian 软件包构建结束之后进行静态检查。lintian 命令可以用下边出现在 ~/.devscripts 文件中的项来定制：

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
```

在普通用户帐号中可以使用以下这样简单的命令清理源代码并重构建软件包：

```
$ debuild
```

还可以这样简单地清理源代码树：

```
$ debuild -- clean
```

6.4. pbuilder 软件包

对于使用净室(chroot)编译环境来验证编译依赖而言，**pbuilder** 软件包是非常有用的。[67]它确保了软件包在不同构架上的 **sid** 发行版环境中的自动编译器中能干净地编译，避免了总是被归类于 RC (Release Critical，影响发布)的严重 FTBFS (Fails To Build From Source，从源代码编译失败) Bug。[68]

我们通过以下操作来定制 **pbuilder** 软件包。

- 设置 **/var/cache/pbuilder/result** 对当前用户可写。
- 创建一个对用户可写的目录保存钩子脚本，例如 ***/var/cache/pbuilder/hooks***
- 在 **~/.pbuilderrc** 或 **/etc/pbuilderrc** 中添加以下内容：

```
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
```

首先使用以下命令初始化本地 **pbuilder chroot** 系统：

```
$ sudo pbuilder create
```

如果你已经创建了源代码包，在包含 ***foo*.orig.tar.gz**、***foo*.debian.tar.gz** 和 ***foo*.dsc** 文件的目录中执行下面的命令来更新 **pbuilder chroot** 系统以便在其中执行构建：

```
$ sudo pbuilder --update
$ sudo pbuilder --build foo_version.dsc
```

新构建的无 GPG 签名的软件包会被以非 root 属主放置于 **/var/cache/pbuilder/result/**。

.dsc 文件和 **.changes** 文件中的 GPG 签名可以用如下方法生成：

```
$ cd /var/cache/pbuilder/result/
$ debsign foo_version_arch.changes
```

如果你已经更新了源代码树但没有生成对应的源代码包，在存放 **debian** 目录的源码目录里执行：

```
$ sudo pbuilder --update
$ pdebuild
```

你可以使用 **pbuilder --login --save-after-login** 命令登录到这个 **chroot** 环境中并按照需要对其进行配置。通过 **^D** (Control-D)离开这个 shell 时环境会被保存。

最新版的 **lintian** 命令可以通过设置钩子脚本 ***/var/cache/pbuilder/hooks*/B90lintian** 在 **chroot** 环境中运行。脚本内容如下：[69]

```
#!/bin/sh
set -e
install_packages() {
        apt-get -y --allow-downgrades install "$@"
        }
install_packages lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes" - pbuilder
# use this version if you don't want lintian to fail the build
#su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" - pbuilder
echo "+++ end of lintian output +++"
```

为 **sid** 编译软件包需要使用 **sid** 环境。在现实中 **sid** 存在很多问题以至于你不愿意将整个系统都迁移到其上。**pbuilder** 软件包可以在这种情况下很好地解决问题。

你可能需要在 **stable** 软件包发布后为 **stable-proposed-updates**、**stable/updates** 等仓库升级你维护的软件包。[70]对于这类情况，“我正在运行 **sid** 系统”并不是你不为它们进行升级的充分理由。**pbuilder** 软件包可以帮助你使用到相同 CPU 体系结构下几乎所有 Debian 和 Debian 衍生版系统环境。

参见 http://www.netfort.gr.jp/~dancer/software/pbuilder.html, pdebuild(1), pbuilderrc(5), 和 pbuilder(8).

6.5. git-buildpackage 及其相似命令

如果你的上游对源代码使用版本控制系统(VCS)[71]，你也应该考虑使用它们。这会使得合并和提取上游补丁更加简单。有多个为不同 VCS 设计的包装脚本包来协助 Debian 软件包构建。

- **git-buildpackage**：帮助维护 Git 仓库中 Debian 软件包的套件。
- **svn-buildpackage**：帮助维护 Subversion 仓库中 Debian 软件包的程序。
- **cvs-buildpackage**：为 CVS 源代码树设计的 Debian 软件包脚本集。

git-buildpackage 的使用 在 Debian 开发者之中非常流行，它可以使用 alioth.debian.org 上的 Git 服务器来管理 Debian 软件包。[72] 这个软件包提供了许多命令来 *自动化* 打包操作：

- gbp-import-dsc(1): 将一个早先的 Debian 软件包导入到 Git 仓库中。
- gbp-import-orig(1): 将上游 tar 文件导入到 Git 仓库中。
- gbp-dch(1): 用 Git commit 信息来生成 Debian changelog。
- git-buildpackage(1): 从 Git 仓库中构建 Debian 软件包。
- git-pbuilder(1): 用 **pbuilder**/**cowbuilder** 从 Git 仓库来构建 Debian 软件包。

这些命令使用 3 个分支来跟踪打包操作：

- **main** 用于 Debian 软件源码树。
- **upstream** 用于上游源码树。
- 由 **--pristine-tar** 为上游 tarball 生成的**pristine-tar** 。[73]

你可以设置 **git-buildpackage**，通过修改 **~/.gbp.conf** 文件。参见 gbp.conf(5). [74]

6.6. 快速重构建

对于很大的软件包，在调试 **debian/rules** 的过程中你可能不想每次都对整个软件包进行重构建。仅用于测试目的，你可以不重新构建源代码包而使用以下的方法创建 **.deb** 文件[75]：

```
$ fakeroot debian/rules binary
```

或者可以通过以下方法验证它是否能通过编译：

```
$ fakeroot debian/rules build
```

一旦完成了调试，记住要按照前面所给出的正常过程重构建你的软件包。你可能无法正常上传用此种方法构建的 **.deb** 文件。

6.7. 命令层级

这里有一个简练的总结，关于在命令层次结构中有多少用于构建软件包的命令能够组合到一起。做这件事情的方法非常多。

- **debian/rules** = 软件包构建过程的专用 maintainer script
- **dpkg-buildpackage** = 软件包构建之核心工具
- **debuild** = **dpkg-buildpackage** + **lintian** (在干净的环境变量下构建)
- **pbuilder** = Debian chroot 环境核心工具
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (在chroot环境中构建)
- **cowbuilder** = 加速 **pbuilder** 执行
- **git-pbuilder** = 命令行语法友好**pdebuild** (被 **gbp buildpackage** 使用)
- **gbp** = 在 Git 仓库中管理 Debian 源码
- **gbp buildpackage** = **pbuilder** + **dpkg-buildpackage** + **gbp**

即便像 **gbp buildpackage** 和 **pbuilder** 这样的高级命令的应用可以确保完美的软件包 构建环境，了解像 **debian/rules** 和 **dpkg-buildpackage** 这样的低级命令行工具如何被它们执行也是至关重要的。

[63] 为了连接至信任网络，本 GPG 密钥必须被一名 Debian 开发者签署，而且必须注册到 the Debian keyring（Debian密钥环）中。这样你上传的软件包就能被接受到 Debian 归档中了。参见 Creating a new GPG key 以及 Debian Wiki on Keysigning。

[64] 你可以使用 `--skip-patches` 选项来在正常的提取操作后避免应用 `3.0 (quilt)` 源代码包格式中的 **quilt** 补丁。你也可以在正常解压后使用 `quilt pop -a` 还原这些补丁对源码的修改。

[65] 实际中的自动构建系统包含了极为复杂(比这里说明的还要复杂)的体制。不过这类细节已经超出本文档范围。

[66] 和在 `pbuilder` 中不同，自动编译系统使用的 `sbuild` 软件包所维护的 **chroot** 不强制要求最小化的编译系统，并可能保持很多软件包始终安装在其中。

[67] 由于 `pbuilder` 软件包仍然在进化，你应当查阅最新的官方文档来检查实际的配置状况。

[68] 参见 http://buildd.debian.org/ 以获取更多关于 Debian 软件包 auto-building 的信息。

[69] 此处默认 `HOOKDIR=/var/cache/pbuilder/hooks`。你可以在 `/usr/share/doc/pbuilder/examples` 目录中找到很多钩子脚本的例子。

[70] 升级你的 `stable` 软件包有一些规定的限制。

[71] 参见 Version control systems 以获取更多信息。

[72] Debian wiki Alioth 这里说明了 如何使用 alioth.debian.org 服务。

[73] `--pristine-tar` 选项会调用 **pristine-tar** 命令，它能利用一个很小的二进制差量文件以及在版本控制系统中某个分支内保存的上游文件内容(分支名通常为 `upstream`)重新生成与上游 tarball 完全相同的一份副本。

[74] 这里有一些给高级读者参考的网页资源。

- 用 git-buildpackage (`/usr/share/doc/git-buildpackage/manual-html/gbp.html`) 构建 Debian 软件包
- debian packages in git
- Using Git for Debian Packaging
- git-dpm: Debian packages in Git manager

[75] 常规情形下被配置好的环境变量在此时不会被自动设置。永远不要将使用这个 **快速** 方法构建的软件包上传到任何地方。