附录 A. 高级打包

目录

这里有一些关于你可能遇到的高级打包问题的提示。如果有需要的话，本教程强烈建议阅读这里引用和建议的文档。

你可能需要手工编辑由 **dh_make** 命令生成的打包模板文件，以此来解决本章中所讨论的问题。新的 **debmake** 命令应该能更好地解决这些问题。

A.1. 共享库

在打包 共享库 之前，你 应该阅读以下的主要参考资料：

- Debian Policy Manual, 8 "Shared libraries"
- Debian Policy Manual, 9.1.1 "File System Structure"
- Debian Policy Manual, 10.2 "Libraries"

以下是帮助你开始的极简解释：

- 共享库均为 elf 对象文件，其包含编译好的机器码。
- 共享库以 ***.so** 文件的形式发放。(既非 ***.a** 文件也非 ***.la** 文件)
- 共享库主要用于在不同的二进制可执行程序之间共享代码，这背后使用了 **ld** (译注：链接)机制。
- 共享库有时会为一个可执行程序提供多个插件，这背后使用了 **dlopen** 机制。
- 共享库能导出代表着变量，函数和类的 symbols(符号)；并允许链接到它的可执行文件访问之。
- 共享库 **libfoo.so.*1*** 中的 SONAME：**objdump -p libfoo.so.*1* | grep SONAME** [88]
- 共享库的 SONAME 常常与库文件自身文件名一致(不过有特例)。
- 链接到 ***/usr/bin/foo*** 的共享库的 SONAME：**objdump -p */usr/bin/foo* | grep NEEDED** [89]
- **libfoo*1***: 共享库 **libfoo.so.*1*** 的库文件包，其 SONAME ABI 版本为 ***1***.[90]
- 在某些情况下，库软件包的 maintainer scripts 必须调用 **ldconfig** 来为 SONAME 创建 必要的符号链接。[91]
- **libfoo*1*-dbg**: 包含了调试共享库包用的调试符号的软件包 **libfoo*1***.
- **libfoo-dev**: 包含了头文件等内容的开发包。用于 **libfoo.so.*1***.[92]
- 一般而言，Debian 软件包不应当包含 ***.la** Libtool 归档文件。[93]
- 一般来说，Debian 软件包不应当使用 RPATH。[94]
- 虽然这有点过时，而且是第二参考，Debian Library Packaging Guide 可能仍然对你有用。

A.2. 管理 debian/*package*.symbols

当你给共享库打包时，你应当创建 **debian/*package*.symbols** 文件来管理在共享库名称不变，在同一个 SONAME 下又要提供 ABI 向后兼容性的情况下每个符号关联到的最小版本号。[95] 你可以阅读下边的主要参考以获知细节：

- Debian Policy Manual, 8.6.3 "The symbols system"[96]
- dh_makeshlibs(1)
- dpkg-gensymbols(1)
- dpkg-shlibdeps(1)
- deb-symbols(5)

这是个粗略的例子，用来来演示创建 **libfoo1** 软件包的方法，此时使用上游 **1.3** 版本，有着妥当的**debian/libfoo1.symbols** 文件：

- 使用上游提供的 **libfoo-1.3.tar.gz** 文件来准备 Debian化 的源码骨架。
  - 如果这是库软件包 **libfoo1** 的第一次打包，那么以空内容创建 **debian/libfoo1.symbols** 文件。
  - 如果之前的上游版本 **1.2** 已经被 **libfoo1** 软件包打包了，并且其源码包中有妥当的 **debian/libfoo1.symbols**，再用它一次。
  - 如果前一个上游 **1.2** 版本打包时没有 **debian/libfoo1.symbols**，那就从具有相同库 SONAME 的同一个共享库包的所有可用的二进制软件包中创建它并命名为 **symbols** 文件。比如 **1.1-1** 和 **1.2-1**。[97]

```
$ dpkg-deb -x libfoo1_1.1-1.deb libfoo1_1.1-1
$ dpkg-deb -x libfoo1_1.2-1.deb libfoo1_1.2-1
$ : > symbols
$ dpkg-gensymbols -v1.1 -plibfoo1 -Plibfoo1_1.1-1 -Osymbols
$ dpkg-gensymbols -v1.2 -plibfoo1 -Plibfoo1_1.2-1 -Osymbols
```

- 尝试用像 **debuild** 和 **pdebuild** 这样的工具来对源码树进行试构建。(如果这因为缺失符号之类原因而失败，那么这里就有一些不向后兼容的 ABI 改变，这就需要你转移(bump)共享库的名称到诸如 **libfoo1a**，并重新开始一次。)

```
$ cd libfoo-1.3
$ debuild
...
dpkg-gensymbols: warning: some new symbols appeared in the symbols file: ...
 see diff output below
--- debian/libfoo1.symbols (libfoo1_1.3-1_amd64)
+++ dpkg-gensymbolsFE5gzx        2012-11-11 02:24:53.609667789 +0900
@@ -127,6 +127,7 @@
  foo_get_name@Base 1.1
  foo_get_longname@Base 1.2
  foo_get_type@Base 1.1
+ foo_get_longtype@Base 1.3-1
  foo_get_symbol@Base 1.1
  foo_get_rank@Base 1.1
  foo_new@Base 1.1
...
```

- 如果你如上述看见由 **dpkg-gensymbols** 命令打印出来的差异，那就从生成的二进制库包中抽取妥当更新的 **symbols** 文件。[98]

```
$ cd ..
$ dpkg-deb -R  libfoo1_1.3_amd64.deb libfoo1-tmp
$ sed -e 's/1\.3-1/1\.3/' libfoo1-tmp/DEBIAN/symbols \
        >libfoo-1.3/debian/libfoo1.symbols
```

- 使用像 **debuild** 和 **pdebuild** 这样的工具来构建发行软件包。

```
$ cd libfoo-1.3
$ debuild -- clean
$ debuild
...
```

对上边这个例子补充一点，我们需要进一步检查 ABI (应用程序二进制接口) 兼容性 并在需要的时候手动更新一些符号的版本。[99]

虽然这只是第二参考，Debian wiki UsingSymbolsFiles 和它指向的页面可能会有所帮助。

A.3. 多体系结构

Debian wheezy 引入的多体系结构特性，集成了对 二进制包跨体系结构安装的支持 (尤其是 **i386<->amd64**，其他的组合也有) 于 **dpkg** 和 **apt** 中。你可以阅读下边的参考：

- Ubuntu wiki MultiarchSpec (上游)
- Debian wiki Multiarch/Implementation (Debian 的局势)

它为每个共享库的安装路径使用了类似 `i386-linux-gnu` 和 `x86_64-linux-gnu` 这样的三元名字。实际上每个二进制软件包构建的三元路径是被动态设置到 `$(DEB_HOST_MULTIARCH)` 变量中的，经由 dpkg-architecture(1) 命令。举个例子，安装多体系结构库文件的路径被按照下表进行了修改：[100]

旧路径	i386 多体系结构路径	amd64 多体系结构路径
`/lib/`	`/lib/i386-linux-gnu/`	`/lib/x86_64-linux-gnu/`
`/usr/lib/`	`/usr/lib/i386-linux-gnu/`	`/usr/lib/x86_64-linux-gnu/`

下面是一些典型的多体系结构软件包分离情景：

- 库源码 `libfoo-1.tar.gz`
- 一个用编译型语言编写的工具的源码 `bar-1.tar.gz`
- 一个用解释型语言编写的工具的源码 `bar-1.tar.gz`

软件包	体系结构：	多体系结构：	软件包内容
`libfoo1`	任何	相同	共享库，可共同安装
`libfoo1-dbg`	任何	相同	共享库调试符号，可共同安装
`libfoo-dev`	任何	相同	共享库头文件之类，可共同安装
`libfoo-tools`	任何	外来	运行时支持程序，不可共同安装
`libfoo-doc`	全部	外来	共享库文档
`bar`	任何	外来	编译好的程序文件，不可共同安装
`bar-doc`	全部	外来	程序的配套文档文件
`baz`	全部	外来	解释型程序文件

请注意，开发软件包应该包含一个指向共享库的符号链接并且 **不带有版本号**。比如：`/usr/lib/x86_64-linux-gnu/libfoo.so -> libfoo.so.1`

A.4. 构建共享库包

你可以用 dh(1) 通过以下方法构建一个支持多体系结构的 Debian 库软件包：

- 更新 `debian/control`。
  - 为源码包部分添加 `Build-Depends: debhelper (>=10)` 部分。
  - 为每个二进制库软件包添加 `Pre-Depends: ${misc:Pre-Depends}`
  - 在每个二进制包的段中添加 `Multi-Arch:` 小节。
- 设置 `debian/compat` 为"10"。
- 将所有打包脚本中的路径从普通的 `/usr/lib/` 调整到多体系结构的 `/usr/lib/$(DEB_HOST_MULTIARCH)/`。
  - 首先，在 `debian/rules` 中调用 `DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)` 以设置 `DEB_HOST_MULTIARCH` 变量。
  - 在 `debian/rules` 中用 `/usr/lib/$(DEB_HOST_MULTIARCH)/` 替换 `/usr/lib/`。
  - 如果 `debian/rules` 文件中的 `override_dh_auto_configure` 目标使用了 `./configure` 文件，那么请确认用 `dh_auto_configure --` 来替换它。[101]
  - 在`debian/foo.install` 文件中将所有 `/usr/lib/` 的事件替换为 `/usr/lib/*/` 。
  - 如需从 `debian/foo.links.in` 动态地生成像 `debian/foo.links` 这样的文件，可以添加一个脚本到 `debian/rules` 文件的 `override_dh_auto_configure` 目标中。

```
override_dh_auto_configure:
        dh_auto_configure
        sed 's/@DEB_HOST_MULTIARCH@/$(DEB_HOST_MULTIARCH)/g' \
                debian/foo.links.in > debian/foo.links
```

请确认该共享库软件包仅仅包含预期中的文件，并且你的 -dev 软件包还奏效。

所有作为多体系结构软件包而同时安装到同一个文件路径的所有文件应当具有完全一致的文件内容。你必须小心由数据字节序和压缩算法造成的区别。

A.5. Debian 本土软件包

如果一个软件包是仅仅为 Debian 维护的，或者是可能的本地使用，那么它的源码可以容纳所有的 `debian/*` 于其中。这里有它的两种打包方式。

你可以将除 `debian/*` 文件之外的部分制作成上游 tarball，然后将其作为非本土 Debian 软件包来打包，正如 第 2.1 节 "Debian 软件包制作流程" 所述。这是一些人鼓励使用的普通方法。

另一种方法就是本土 Debian 软件包的打包工作流。

- 用包含所有文件的，单一压缩过的 tar 文件，以 `3.0 (native)` 格式来创建本土 Debian 源码包。
  - `package_version.tar.gz`
  - `package_version.dsc`
- 用 Debian 本土源码包构建二进制包。
  - `package_version_arch.deb`

比如说，如果你的源代码文件都存放在 `~/mypackage-1.0` 中，而且没有 `debian/*` 这些文件，那么你可以用它创建 一个本土 Debian 软件包，只要按照下边的方法使用 **dh_make** 命令：

```
$ cd ~/mypackage-1.0
$ dh_make --native
```

接下来 `debian` 目录和它的内容都会被创建，正如 第 2.8 节 "初始化外来 Debian 软件包" 中那样。这不会创建一个 tarball，因为这是个本土 Debian 软件包。不过这也是唯一的区别。剩下的打包操作就是完全一致的了。

在执行了 **dpkg-buildpackage** 命令后，你将会在上一级目录中看到这些文件：

- `mypackage_1.0.tar.gz`

  这是 **dpkg-source** 命令用 `mypackage-1.0` 目录创建出来的源代码 tarball 。(它的 文件名后缀不是 `orig.tar.gz`)

- `mypackage_1.0.dsc`

  这是对源码内容的简述，正如在非本土 Debian 软件包中那样。(没有 Debian 修订号。)

- `mypackage_1.0_i386.deb`

  这是完成的二进制包，正如在非本土 Debian 软件包中那样。(没有 Debian 修订号。)

- `mypackage_1.0_i386.changes`

  这个文件描述了这个软件作为外来 Debian 包，在当前版本所作出的所有更改。(没有 Debian 修订)

---

[88] 或者这样：`readelf -d libfoo.so.1 | grep SONAME`

[89] 或者这样：`readelf -d libfoo.so.1 | grep NEEDED`

[90] 参见 Debian Policy Manual, 8.1 "Run-time shared libraries".

[91] 参见 Debian Policy Manual, 8.1.1 "ldconfig".

[92] 参见 Debian Policy Manual, 8.3 "Static libraries" and Debian Policy Manual, 8.4 "Development files".

[93] 参见 Debian wiki ReleaseGoals/LAFileRemoval.

[94] 参见 Debian wiki RpathIssue.

[95] 向后不兼容的 ABI 变更常常需要你更新共享库的 SONAME，并把共享库名称换成新的。

[96] 对于 C++ 库和其他追踪单个符号过于困难的情况下，请遵循 Debian Policy Manual, 8.6.4 "The shlibs system"。

[97] 所有先前的 Debian 软件包版本都能在 http://snapshot.debian.org/ 找到。不过 Debian 修订号 被去掉了，以使软件包的 backport 更为容易：`1.1` << `1.1-1~bpo70+1` << `1.1-1` and `1.2` << `1.2-1~bpo70+1` << `1.2-1`

[98] Debian 修订号已被从版本中去掉，这能让软件包的 backport 更为容易：`1.3` << `1.3-1~bpo70+1` << `1.3-1`

[99] 参见 Debian Policy Manual, 8.6.2 "Shared library ABI changes".

[100] 老旧且具有特殊用途的库路径，比如 **/lib32/** 和 **/lib64/** 已不再使用。

[101] 作为替代，你可以添加 **--libdir=\$\${prefix}/lib/$(DEB_HOST_MULTIARCH)** 和 **--libexecdir=\$\${prefix}/lib/$(DEB_HOST_MULTIARCH)** 参数到 **./configure** 后头。请注意 **--libexecdir** 指定了 安装可执行程序(它们被其他程序使用，而更少是用户)的默认路径。它的 Autotools 默认设置为 **/usr/libexec/** 但是 Debian 的设置为 **/usr/lib/**。