Руководство начинающего разработчика Debian

COLLABORATORS

	TITLE : Руководство начинающего разработчика Debian	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Джосип Родин, Осаму Аоки, L10N-russian	25 мая 2017 г.	
Russian Translation		25 мая 2017 г.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Оглавление

Глава 1

Хорошее начало —половина дела

В этом документе описан процесс создания пакета Debian с точки зрения обычного пользователя и начинающего разработчика. Он написан простым языком и содержит работающие примеры. В этом руководстве мы пытаемся следовать старой латинской поговорке: *Longum iter est per praecepta, breve et efficax perexempla!* (Путь длинен, если изучать правила, но короток и эффективен, если пользоваться примерами!).

Этот документ был адаптирован для выпуска Debian `jessie` [1].

Одна из сильных, по сравнению с другими дистрибутивами, сторон Debian —это система управления пакетами. Несмотря на то, что для Debian уже существует очень много пакетов, может случиться так, что вам понадобится установить программу, для которой не существует соответствующего пакета. Это может заставить вас задуматься о том, как создать свой собственный пакет. Для тех, кто делает первые шаги в Linux, это сложно, но вы к ним не относитесь, если сейчас читаете этот документ :-) Вам понадобятся некоторые знания о программировании под Unix, но ни в коем случае вы не обязаны быть гуру [2].

Одно можно сказать определённо: создание и сопровождение пакетов Debian занимает много времени. Несомненно, чтобы наша система работала, сопровождающие должны быть технически грамотными и прилежными.

Если вам нужна помощь в пакетировании, прочтите Раздел 1.4.

Самые новые версии этого документа всегда доступны на странице http://www.debian.org/doc/maint-guide/ и в пакете `maint-guide`. Переводы доступны в отдельных пакетах, например `maint-guide-es`. Заметим, что данный документ может быть слегка устаревшим.

Так как это учебное пособие, каждый важный вопрос будет объясняться последовательно, шаг за шагом. Некоторые из них могут показаться вам ненужными. Будьте терпеливее. Также, для упрощения документа были намеренно опущены некоторые крайние случаи и приведены только ссылки.

1.1 Социальная динамика Debian

Вот некоторые наблюдения за социальной динамикой Debian, представленные в надежде, что это подготовит вас к взаимодействию с Debian:

- Все занимаются Debian на добровольной основе.
  - Вы не можете указывать другим что делать.
  - Вы сами должны быть заинтересованы что-то делать.

[1] В документе предполагается, что вы используете `jessie` или более новую версию. Если у вас старая версия (включая старые выпуски Ubuntu и т.д.), установите современные версии пакетов `dpkg` и `debhelper` из специального репозитория (backports).

[2] О том, как работать с системой Debian, можно найти в справочнике Debian (http://www.debian.org/doc/manuals/debian-reference/) . В нём также содержатся ссылки на материалы по программированию в системах Unix.

- Движущая сила —дружественное сотрудничество.
  - Ваш вклад не должен перенапрягать остальных.
  - Ваш вклад полезен, если так посчитают остальные.
- Debian —это не школа, где вы автоматически получите внимание учителей.
  - Вы должны быть способны учиться самостоятельно.
  - Внимание других добровольцев —очень дефицитный ресурс.
- Debian постоянно улучшается.
  - От вас ожидают высококачественных пакетов.
  - Вы сами должны адаптироваться к изменениям.

Есть несколько групп людей, взаимодействующих в Debian друг с другом в различных качествах:

- **автор программы (upstream author)** —человек, который создал программу.
- **сопровождающий программы (upstream maintainer)** —человек, который сопровождает программу в настоящее время.
- **сопровождающий (maintainer)** —человек, который создал для программы пакет Debian.
- **поручитель (sponsor)** —человек, который помогает сопровождающим помещать пакеты в официальный архив пакетов Debian (после проверки их содержимого).
- **наставник (mentor)** —человек, который помогает новым сопровождающим в пакетировании и т.п.
- **разработчик Debian** (DD) —человек, являющийся участником проекта Debian. У него есть право на размещение пакетов в официальном архиве пакетов Debian.
- **сопровождающий Debian** (DM) —человек, обладающий ограниченными правами на размещение пакетов в официальном архиве пакетов Debian.

Вы не можете стать официальным **разработчиком Debian** за вечер, так как для этого требуются не только технические знания. Но не унывайте. Если ваш пакет полезен другим, вы можете предложить его будучи **сопровождающим** через **поручителя** или как **сопровождающий Debian**.

Заметим, что вам не нужно обязательно создавать новый пакет, чтобы стать официальным разработчиком Debian, для этого достаточно поддерживать существующие пакеты. Есть много пакетов, которые ждут хороших сопровождающих (смотрите Раздел 2.2).

Этот документ описывает технические моменты пакетирования. О том, как работает Debian, и как вы можете помочь, обратите внимание на следующие страницы:

- Debian: 17 лет Свободного ПО, «дело-кратия» и демократия (http://upsilon.cc/~zack/talks/2011/20110321-taipei.pdf) (вступительные слайды)
- Как помочь Debian? (http://www.debian.org/intro/help) (официальная страница)
- Часто задаваемые вопросы по Debian GNU/Linux, глава 13: «Содействие проекту Debian» (http://www.debian.org/doc/-FAQ/ch-contributing) (полуофициальная страница)
- Страница HelpDebian в Debian Wiki (http://wiki.debian.org/HelpDebian) (дополнительно)
- Сайт нового сопровождающего Debian (https://nm.debian.org/) (официальный)
- Список ответов на часто задаваемые вопросы наставникам Debian (http://wiki.debian.org/DebianMentorsFaq) (дополнительно)

1.2 Программы, необходимые для разработки

Перед тем как начать, нужно убедиться, что установлены все необходимые для разработки пакеты. Обратите внимание, что приведённый ниже список не содержит пакеты, помеченные как `обязательные` (essential) или `требуемые` (required) —считается, что эти пакеты уже установлены на вашей машине.

Приведённые ниже пакеты присутствуют в стандартной установке Debian, то есть, скорее всего, они уже установлены на вашей машине (как, впрочем, и пакеты, которые им нужны для работы). Несмотря на это, мы рекомендуем проверить их наличие с помощью команды `aptitude show` *`пакет`* или `dpkg -s` *`пакет`*.

Самый важный пакет в системе разработчика —`build-essential`. Его установка повлечёт за собой *загрузку* других пакетов, требуемых для основы среды сборки.

Кроме пакетов, требуемых для сборки любого пакета, есть пакеты, которые нужны только для некоторых пакетов; установите их, они могут пригодиться именно для вашего пакета:

- `autoconf`, `automake` и `autotools-dev` —данные утилиты (смотрите `info autoconf`, `info automake`) используются во многих современных программах для создания сценариев настройки и файла `Makefile`. В пакете `autotools-dev` содержатся самые новые версии некоторых файлов auto- и документация по их применению.
- `debhelper` и `dh-make` —пакет `dh-make` необходим для создания скелета нашего будущего пакета. Для этого он будет использовать некоторые инструменты из пакета `debhelper`. Использовать их необязательно, но мы *очень* рекомендуем их начинающим разработчикам. Они сильно упрощают процесс создания и поддержки пакетов (смотрите dh_make(8), debhelper(1)) [3].

  The new `debmake` may be used as the alternative to the standard `dh-make`. It does more and comes with HTML documentation with extensive packaging examples in `debmake-doc`.
- `devscripts` —данный пакет содержит сценарии, полезные для сопровождающих, но так же не являющиеся необходимыми для сборки пакетов. Стоит обратить внимание на рекомендуемые и предлагаемые им пакеты (смотрите `/usr/share/doc/devscripts/README.gz`).
- `fakeroot` —данная утилита позволяет эмулировать наличие прав прав пользователя root, которые необходимы на некоторых этапах процесса сборки (смотрите fakeroot(1)).
- `file` —данная программа позволяет определить тип файла (смотрите file(1)).
- `gfortran` —пакет содержит компилятор GNU Fortran; требуется, если программа написана на Fortran (смотрите gfortran(1)).
- `git` —данный пакет предоставляет популярную систему контроля версий, разработанную для быстрого и эффективного сопровождения очень больших проектов; она используется во многих известных проектах с открытым кодом, наиболее заметным из которых является ядро Linux (смотрите git(1), руководство по git (`/usr/share/doc/git-doc/index.html`)).
- `gnupg` —данный инструмент позволяет *подписывать* пакеты. Это особенно важно, если вы хотите распространять их, и вы точно будете делать это, если хотите, чтобы ваш пакет был включён в дистрибутив Debian (смотрите gpg(1)).
- `gpc` —пакет содержит компилятор GNU Pascal, который требуется при работе с программами, написанными на Pascal. Для этой задачи также хорошо подходит `fp-compiler`, Free Pascal Compiler (смотрите gpc(1), ppc386(1)).
- `lintian` —данная программа предназначена для проверки пакетов Debian. Если вы допустили одну из распространённых ошибок, она сообщит вам об этом после сборки пакета и покажет пояснение по каждой найденной ошибке (смотрите lintian(1), руководство пользователя lintian (https://lintian.debian.org/manual/index.html)).
- `patch` —данная утилита изменяет исходный файл в соответствии со списком различий между файлами, полученным при помощи программы **diff** (смотрите patch(1)).
- `patchutils` —данный пакет содержит несколько утилит для работы с заплатами, например **lsdiff**, **interdiff** и **filterdiff**.

[3] Существуют также похожие, более специализированные пакеты, такие как `dh-make-perl`, `dh-make-php` и т.д.

- `pbuilder` —пакет содержит программы, которые используются для создания и сопровождения окружения **chroot**. Сборка пакета Debian в окружении **chroot** позволяет проверить правильность указанных сборочных зависимостей и избежать ошибок FTBFS (ошибки при сборке из исходного кода) (смотрите pbuilder(8) и pdebuild(1)).
- `perl` —один из наиболее используемых интерпретируемых языков в Unix-системах. Его часто называют «Unix's Swiss Army Chainsaw» (швейцарской армейской пилой) (смотрите perl(1)).
- `python` —ещё один из наиболее используемых интерпретируемых языков в Debian, который объединяет необычайную мощь с очень понятным синтаксисом (смотрите python(1)).
- `quilt` —пакет помогает управлять большими наборами заплат, отслеживая каждое сделанное изменение. Заплаты логически организуются в стек, и вы можете накладывать их, откатывать изменения, обновлять их и т.д. (смотрите quilt(1), `/usr/share/doc/quilt/quilt.pdf.gz`).
- `xutils-dev` —пакет содержит программы, которые используются при сборке пакетов для X11, например с их помощью генерируется `Makefile` из набора макрофункций (смотрите imake(1), xmkmf(1)).

Краткие описания, показанные выше, даны лишь для того, чтобы у вас сложилось общее представление о том, для чего предназначен каждый пакет. Прежде чем продолжить, полностью прочитайте документацию к каждой программе (в том числе по установленным согласно зависимостям пакетам, например **make**), по крайней мере, по основам работы. Сейчас это может оказаться слишком трудным, но позже вы будете *очень* довольны, что сделали это. Если позднее у вас возникнут конкретные вопросы, перечитайте документацию, упомянутую выше.

1.3 Документация, необходимая для разработки

Кроме этого документа также *очень важно* прочитать следующую документацию:

- `debian-policy` —в руководстве по политике Debian (http://www.debian.org/doc/devel-manuals#policy) содержится описание структуры и содержимого архива Debian, некоторых проблем при разработке операционной системы, стандарт иерархии файловой системы (http://www.debian.org/doc/packaging-manuals/fhs/fhs-2.3.html) (FHS, в котором оговаривается расположение каждого файла и каталога) и т.д. Также (что для вас важнее всего), в пакете указаны требования, которым должен удовлетворять каждый пакет Debian для того, чтобы он мог быть включён в дистрибутив (смотрите локальные файлы `/usr/share/doc/debian-policy/policy.pdf.gz` и `/usr/share/doc/debian-policy/fhs/fhs-2.3.pdf.gz`).
- `developers-reference` —в справочнике разработчика Debian (http://www.debian.org/doc/devel-manuals#devref) содержится информация, не относящаяся непосредственно к техническим вопросам создания пакетов. Здесь содержится информация о структуре архива, о том, как переименовывать, переводить в брошенное состояние или подбирать брошенные пакеты, как обновить пакет, не являясь его разработчиком (NMU), как управлять ошибками, когда и как обновлять пакеты и т.п. (смотрите локальный файл `/usr/share/doc/developers-reference/developers-reference.pdf`).

Кроме этого документа также *важно* прочитать следующую документацию:

- В учебнике по Autotools (http://www.lrde.epita.fr/~adl/autotools.html) представлено очень хорошее руководство по системе сборки GNU (GNU Autotools), наиболее важными компонентами которой являются Autoconf, Automake, Libtool и gettext.
- `gnu-standards` —в этом пакете содержатся две части документации проекта GNU: стандарты написания кода GNU (http://www.gnu.org/prep/standards/html_node/index.html) и информация для сопровождающих ПО GNU (http://www.gnu.org/prep/maintain/html_node/index.html) . Хотя в Debian не требуется их соблюдения, они всё равно полезны для общего понимания (смотрите локальные файлы `/usr/share/doc/gnu-standards/standards.pdf.gz` и `/usr/share/doc/gnu-standards/maintain.pdf.gz`).

Если этот документ в чём-то противоречит документам, упомянутым выше, это считается ошибкой. Отправьте сообщение об ошибке в пакете `maint-guide` с помощью **reportbug**.

Также, вместе с этим документом можно почитать следующую документацию:

- Учебник Debian по пакетированию (http://www.debian.org/doc/packaging-manuals/packaging-tutorial/packaging-tutorial)

1.4 Где искать помощь

Прежде всего, перед тем как задавать вопрос, внимательно прочитайте документацию:

- файлы в `/usr/share/doc/`*`пакет`* для всех используемых пакетов
- содержимое **`man`** *`команда`* для всех используемых команд
- содержимое **`info`** *`команда`* для всех используемых команд
- содержимое архива списка рассылки debian-mentors@lists.debian.org (http://lists.debian.org/debian-mentors/)
- содержимое архива списка рассылки debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/)

Для более эффективного поиска с помощью поисковых машин добавьте в строку поиска `site:lists.debian.org` для ограничения по домену поиска.

Создание маленького тестового пакета —хороший способ научиться пакетированию. Изучая устройство тщательно сопровождаемых пакетов, можно узнать ещё больше.

Если вы не смогли найти ответы на свои вопросы в документации и веб, то можете задать их интерактивно:

- в список рассылки debian-mentors@lists.debian.org (http://lists.debian.org/debian-mentors/) (для новичков).
- в список рассылки debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) (для опытных разработчиков).
- на канале IRC (http://www.debian.org/support#irc) , например в `#debian-mentors`.
- Команды, работающие над определённым набором пакетов (полный список https://wiki.debian.org/Teams (https://wiki.debian.org
Teams)).
- Списки рассылки на родном языке, например debian-devel-{french,italian,portuguese,spanish}@lists.debian.org или debian-devel@debian.or.jp (полный список https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) и https://lists.debian.org/-users.html (https://lists.debian.org/users.html)).

Более опытные разработчики Debian будут рады помочь вам, если на правильно зададите вопрос после попыток разобраться самостоятельно.

Когда вы получите сообщение об ошибке (да, сообщения о реальных ошибках!), то знайте, что пришло время разобраться с системой отслеживания ошибок Debian (http://www.debian.org/Bugs/) и прочитать имеющуюся там документацию, чтобы эффективно работать с сообщениями об ошибках. Настоятельно рекомендуем прочитать справочник разработчика Debian, раздел 5.8. «Работа с ошибками» (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling)
.

Даже если всё правильно работало, настало время молиться. Почему? Потому что через несколько часов или дней пользователи по всему миру начнут использовать ваш пакет и, если вы допустили какую-нибудь критическую ошибку, многочисленные разгневанные пользователи Debian устроят атаку на почтовый ящик···Шутка, шутка. :-)

Отдохните и приготовьтесь получать сообщения об ошибках, так как много чего ещё нужно сделать для того, чтобы пакет полностью соответствовал политике Debian (ещё раз, прочитайте *имеющуюся документацию*). Успехов!

Глава 2

Первые шаги

Давайте попробуем создать свой собственный пакет (или, ещё лучше, адаптировать существующий).

2.1 Порядок сборки пакета Debian

При создании пакета Debian из исходного кода программы в процессе сборки на каждом этапе генерируется несколько файлов со специальными именами:

- получение копии исходного ПО, обычно в формате сжатого tar
  - *пакет*`-`*версия*`.tar.gz`
- добавление специальных изменений Debian в исходную программу в каталог `debian` и создание неродного (non-native) пакета с исходным кодом (то есть, набора входных файлов, используемых для сборки пакета Debian) в формате `3.0 (quilt)`
  - *пакет*`_`*версия*`.orig.tar.gz`
  - *пакет*`_`*версия*`-`*редакция*`.debian.tar.gz`[1]
  - *пакет*`_`*версия*`-`*редакция*`.dsc`
- сборка двоичных пакетов Debian для получения обычных файлов пакетов для установки в формате `.deb` (или в формате `.udeb`, который используется Debian Installer) из пакета Debian с исходным кодом
  - *пакет*`_`*версия*`-`*редакция*`_`*архитектура*`.deb`

Заметим, что в именах файлов пакетов Debian для разделения *пакета* и *версии* используется символ _ (подчёркивание), а не `-` (дефис), как в имени оригинального tar-файла.

В именах файлов, представленных выше, замените часть *пакет* на **имя пакета**, часть *версия* на **версию исходной программы**, часть *редакция* на **редакцию Debian** и часть *архитектура* на **архитектуру, для которой предназначен пакет**, в соответствии с политикой Debian [2].

Каждый шаг, показанный ранее, детально описан на примерах в последующих разделах.

[1] В старом формате `1.0` для неродных пакетов Debian с исходным кодом использовалось имя *пакет*`_`*версия*`-`*редакция*`.diff.gz`.

[2] Смотрите описание полей 5.6.1 «Source» (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source) , 5.6.7 «Package» (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Package) и 5.6.12 «Version» (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version) . Значение **архитектуры пакета** задаётся согласно политике Debian в разделе 5.6.8, «Architecture» (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) и автоматически назначается в процессе сборки пакета.

2.2 Выбор программы

Вы, вероятно, уже выбрали пакет, который хотите создать. Первое, что вам необходимо сделать, это проверить, нет ли уже этого пакета в архиве дистрибутива, используя:

- команду **aptitude**
- веб-страницу пакетов Debian (http://www.debian.org/distrib/packages)
- веб-страницу системы отслеживания пакетов Debian (http://packages.qa.debian.org/common/index.html)

Если пакет уже есть, то просто установите его! :-) Если случится так, что он окажется **брошенным** (orphaned) —то есть сопровождающий передал его Debian QA Group (http://qa.debian.org/) , то вы сможете подобрать его, если он ещё будет доступен. Также вы можете взять пакет, если его сопровождающий послал «запрос об усыновлении» (Request for Adoption, **RFA**) [3].

Есть несколько ресурсов для получения информации о состоянии владения пакетом:

- Команда **wnpp-alert** из пакета `devscripts`
- Требующие доработки и будущие пакеты (http://www.debian.org/devel/wnpp/)
- Журналы сообщений об ошибках Debian: ошибки в псевдо-пакете `wnpp` в `unstable` (http://bugs.debian.org/wnpp)
- Пакеты Debian, которым нужна забота (http://wnpp.debian.net/)
- Просмотр ошибок `wnpp` на основе категорий debtag (http://wnpp-by-tags.debian.net/)

Попутно отметим, что в Debian уже есть пакеты для большинства типов программ, и их количество в архиве Debian намного больше, чем участников с правом загрузки. Поэтому работа над пакетами, которые уже включены в архив, намного предпочтительней (и с большей вероятностью получит поручительство) с точки зрения других разработчиков [4]. Этого можно достичь различными путями:

- подобрать брошенный, но пока активно используемый пакет
- войти в одну из команд по пакетированию (http://wiki.debian.org/Teams)
- исправлять ошибки в очень популярных пакетах
- подготовить пакет для QA или NMU загрузки (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload)

Если вы способны усыновить (adopt) пакет, возьмите пакет с исходным кодом (например, с помощью `apt-get source` *`имя_пакета`*) и просмотрите его. К сожалению, этот документ не содержит полной информации об усыновлении пакетов. Но вам не потребуется много времени, чтобы выяснить, как работает пакет, так как кто-то уже выполнил первоначальную работу. Тем не менее продолжайте читать этот документ, многое из него пригодится и в вашем случае.

Если пакет новый и вы решили, что он нужен в Debian, то сделайте следующее:

- Во-первых, вы должны знать, что программа работает и вы, поработав с ней некоторое время, сочли её полезной.
- Проверьте, нет ли её в списке пакетов, над которыми уже ведётся работа (http://www.debian.org/devel/wnpp/) . Если нет, то пошлите сообщение об ошибке типа ITP (Intent To Package, намерение создать пакет) на псевдо-пакет `wnpp` с помощью **reportbug**. Если работа уже ведётся, свяжитесь с сопровождающим и спросите, не нужно ли помочь. Если помощь не требуется —поищите другую интересную программу, которую ещё никто не сопровождает.
- У программы **обязательно должна быть лицензия**.

[3] Смотрите справочник разработчика Debian 5.9.5. «Адаптация пакета» (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#adopting) .

[4] С другой стороны, всегда будут появляться новые программы, которые хотелось бы иметь в виде пакета.

  - Для секции `main` политика Debian требует, чтобы программа **удовлетворяла критериям Debian по определению Свободного ПО** (DFSG (http://www.debian.org/social_contract#guidelines)) и **не зависела от пакетов вне `main`** для компиляции или выполнения. Это предпочтительный вариант.
  - Для секции `contrib` она должна удовлетворять DFSG, но может зависеть от пакетов вне `main` для компиляции или выполнения.
  - Для секции `non-free` она может не удовлетворять DFSG, но **должна быть распространяема**.
  - Если вы не уверены в том, где должна размещаться программа, отправьте текст лицензии в debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) и попросите совета.
- Программа **не** должна создавать проблем с безопасностью и сопровождением системы Debian.
  - Программа должна быть хорошо документирована или, по крайней мере, понятна (то есть не содержать специально запутанного кода).
  - Вы должны связаться с авторами программы, чтобы убедиться, что они не против создания пакета Debian с их программой. Возможность консультироваться с авторами программы по поводу тех или иных проблем обычно очень важна, поэтому лучше не пытайтесь создавать пакеты для неподдерживаемых программ.
  - Программа определённо **не должна** требовать запуска с помощью setuid root, а ещё лучше —вообще не требовать прав доступа setuid или setgid к чему-либо.
  - Программа не должна работать как служба, размещаться в каталогах `*/sbin` или открывать порты с правами root.

Разумеется, последние —это всего лишь меры предосторожности, которые спасут вас от разъяренных пользователей, если вы сделали что-то не так со службой, использующей setuid. Как только вы приобретёте определённый опыт, то сможете создавать пакеты с таким ПО.

Как начинающий сопровождающий, не беритесь за сложные пакеты, пока не научитесь пакетировать самые простые.

- Простые пакеты
  - одиночный двоичный пакет, архитектура = all (набор данных, например обои для рабочего стола)
  - одиночный двоичный пакет, архитектура = all (исполняемые файлы, написанные на интерпретируемых языках, таких как оболочка POSIX)
- Пакеты средней сложности
  - одиночный двоичный пакет, архитектура = any (исполняемые двоичные файлы ELF, скомпилированные из исходного кода на C и C++)
  - несколько двоичных пакетов, архитектура = any + all (пакеты с исполняемыми двоичными файлами ELF + документация)
  - исходный код в формате, отличном от `tar.gz` или `tar.bz2`
  - исходный код с содержимым, не подлежащим распространению
- Пакеты повышенной сложности
  - пакет для интерпретируемого модуля, используемого другими пакетами
  - пакет для обычной библиотеки ELF, используемого другими пакетами
  - несколько двоичных пакетов, включающих пакет с библиотекой ELF
  - исходный код, получаемый из нескольких источников
  - пакеты с модулями ядра
  - пакеты с заплатами к ядру
  - любой пакет со сложными сценариями сопровождающего

Пакетирование пакетов повышенной сложности не слишком трудно, но требует больше знаний. Вы должны найти нужное руководство для каждой сложной функции. Например, для некоторых языков есть своя документация с политикой:

- Политика для Perl (http://www.debian.org/doc/packaging-manuals/perl-policy/)
- Политика для Python (http://www.debian.org/doc/packaging-manuals/python-policy/)
- Политика для Java (http://www.debian.org/doc/packaging-manuals/java-policy/)

Есть ещё одно старое латинское выражение: *fabricando fit faber* (мастер создаётся трудом). *Настоятельно* рекомендуем вам выполнять и экспериментировать над всеми шагами пакетирования Debian с простыми пакетами при чтении руководства. Создайте простейший архив с исходным кодом `hello-sh-1.0.tar.gz` следующим образом [5]:

```
$ mkdir -p hello-sh/hello-sh-1.0; cd hello-sh/hello-sh-1.0
$ cat > hello <<EOF
#!/bin/sh
# (C) 2011 Foo Bar, GPL2+
echo "Hello!"
EOF
$ chmod 755 hello
$ cd ..
$ tar -cvzf hello-sh-1.0.tar.gz hello-sh-1.0
```

2.3 Получение программы и ознакомление со сборкой

Итак, первое, что вы должны сделать —найти и скачать исходный код программы. Предполагается, что вы уже взяли файл с домашней страницы автора. Исходный код программ для Unix обычно предоставляется в виде архива в формате **tar**+**gzip** с расширением `.tar.gz`, **tar**+**bzip2** с расширением `.tar.bz2` или **tar**+**xz** с расширением `.tar.xz`. Внутри архива обычно есть каталог с именем *`пакет-версия`*, в котором находятся все файлы исходного кода программы.

Если самая новая версия кода доступна из VCS, например из репозитория Git, Subversion или CVS, то вам нужно получить её с помощью команд `git clone`, `svn co` или `cvs co` и перепаковать в формат **tar**+**gzip** с параметром `--exclude-vcs`.

Если исходный код выбранной программы поставляется в другом виде (например, имя файла оканчивается на `.Z` или `.zip`[6]), распакуйте его соответствующими средствами и перепакуйте его.

Если исходный код выбранной программы поставляется с содержимым, не удовлетворяющим DFSG, то вам также нужно распаковать его и удалить это содержимое и перепаковать его, добавив в версию исходного кода пометку `dfsg`.

В качестве примера взята программа **gentoo** —менеджер файлов, использующий GTK+ [7].

В своём домашнем каталоге создайте каталог с именем `debian`, `deb` или с любым удобным (например, в нашем случае можно было бы использовать `~/gentoo`). Поместите скачанный архив в этот каталог и распакуйте его (`tar xzf gentoo-0.9.12.tar.gz`). Убедитесь, что при этом не возникло никаких, *даже, казалось бы, не относящихся к делу* ошибок, так как их появление означает, что они могут возникнуть и на машинах других людей, у которых их инструменты распаковки посчитают это за реальную ошибку. В командной строке оболочки вы должны увидеть следующее:

```
$ mkdir ~/gentoo ; cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xvzf gentoo-0.9.12.tar.gz
$ ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
```

В результате вы получите подкаталог `gentoo-0.9.12`. Перейдите в этот каталог и *внимательно* прочитайте имеющуюся документацию. Обычно, полезными оказываются файлы `README*`, `INSTALL*`, `*.lsm` или `*.html`. Вы должны

[5] Не беспокойтесь об отсутствии файла `Makefile`. Вы можете установить команду **hello** просто с помощью **debhelper**, как показано в Раздел 5.11, или изменить исходный код, добавив новый `Makefile` с целью `install`, как показано в Глава 3.

[6] Вы можете определить формат архива с помощью команды **file**, если по расширению это непонятно.

[7] Для этой программы пакет уже создан. В текущей версии (http://packages.qa.debian.org/g/gentoo.html) в качестве структуры сборки используется Autotools и, следовательно, есть различия с приводимыми примерами, которые были написаны для версии 0.9.12.

найти инструкции, которые позволят вам правильно скомпилировать и установить программу (скорее всего, в каталог `/usr/local/bin`; вы должны будете установить программу в другой каталог, подробнее об этом в Раздел 3.3).

Вы должны начинать процедуру пакетирования в полностью оригинальном (ещё неизменённым) каталоге с исходным кодом (для этого, например, можно заново распаковать архив с исходным кодом).

2.4 Простые системы сборки

В простых программах обычно используется файл `Makefile` и для компиляции достаточно выполнить команду `make`[8]. Некоторые из них поддерживают команду `make check`, по которой выполняется самопроверка. Установка в каталог назначения обычно выполняется с помощью `make install`.

Теперь скомпилируйте программу и попробуйте её запустить, чтобы убедиться, что она правильно работает и что при установке и запуске ничто другое не было испорчено.

Вы также можете попытаться воспользоваться командой `make clean` (или лучше `make distclean`) для очистки каталога сборки. Иногда есть даже команда `make uninstall`, которая позволит удалить все установленные файлы.

2.5 Популярные переносимые системы сборки

Многие свободные программы написаны на языках C и C++. В некоторых из них для переносимости между платформами используются Autotools или CMake. Эти инструменты используются для генерации `Makefile` и других необходимых исходных файлов. Такие программы обычно собираются с помощью `make; make install`.

Autotools —это система сборки GNU, состоящая из Autoconf, Automake, Libtool и gettext. Её использование можно определить по включённым в исходный архив файлам `configure.ac`, `Makefile.am` и `Makefile.in` [9].

Первый шаг автоматизации с помощью Autotools обычно выполняет автор программы. Он запускает команду `autoreconf -i -f` в каталоге с исходным кодом и затем распространяет сгенерированные файлы вместе с исходным кодом.

```
configure.ac-----+-> autoreconf -+-> configure
Makefile.am -----+        |      +-> Makefile.in
src/Makefile.am -+        |      +-> src/Makefile.in
                          |      +-> config.h.in
                      automake
                      aclocal
                      aclocal.m4
                      autoheader
```

Для правки файлов `configure.ac` и `Makefile.am` требуется знание инструментов **autoconf** и **automake**. Смотрите `info autoconf` и `info automake`.

Второй шаг автоматизации с помощью Autotools обычно выполняет пользователь. Он получает распространяемый код и запускает `./configure && make` для компиляции программы в исполняемый ***двоичный файл***.

```
Makefile.in -----+               +-> Makefile -----+-> make -> двоичный файл
src/Makefile.in -+-> ./configure -+-> src/Makefile -+
config.h.in -----+               +-> config.h -----+
                 |
  config.status -+
  config.guess --+
```

[8] Со многими современными программами поставляется сценарий `configure`, который при запуске создаёт файл `Makefile`, изменённый специально под вашу систему.

[9] Autotools —слишком большой инструмент для описания его работы здесь. В этом разделе описаны только ключевые понятия и приведены ссылки. Прочитайте руководство по Autotools (http://www.lrde.epita.fr/~adl/autotools.html) и локальную копию `/usr/share/doc/autotools-dev/README.Debian.gz`, если вам потребуется его использовать.

Вы можете изменять различные переменные в файле `Makefile`. Например каталог установки по умолчанию изменяется с помощью параметра в командной строке: `./configure --prefix=/usr`.

Хотя это не обязательно, обновление `configure` и других файлов с помощью `autoreconf -i -f` может улучшить совместимость исходного кода [10].

Альтернативной системой сборки является CMake. Её можно определить по наличию файла `CMakeLists.txt`.

2.6 Имя и версия пакета

Если оригинальный исходный код программы содержится в файле `gentoo-0.9.12.tar.gz`, то в качестве (исходного) **имени пакета** можно взять `gentoo`, а в качестве **версии исходной программы** —`0.9.12`. Имя и версия используются в файле `debian/changelog`, который описан далее в Раздел 4.3.

Хотя такой простой способ выбора имени почти всегда срабатывает, вам может потребоваться привести **имя пакета** и **версию исходной программы** в соответствие с политикой Debian и существующим соглашением.

Имя пакета может содержать только строчные буквы (`a-z`), цифры (`0-9`), знаки плюс (`+`) и минус (`-`) и точки (`.`). Оно должно быть не короче двух символов, должно начинаться с буквы или цифры и не должно быть уже использовано для другого пакета. Рекомендуем ограничиться длиной до 30 символов [11].

Если для имени автор программы использовал какие-то общие слова, например `test-suite`, то лучше назначить другое имя, которое явно описывает содержимое и не засоряет пространство имён [12].

Вы должны выбрать **версию исходной программы** так, чтобы она содержала только буквы или цифры (`0-9A-Za-z`), плюс (`+`), тильду (`~`) и точку (`.`). Она должна начинаться с цифры (`0-9`) [13]. Если возможно, лучше ограничиться длиной до 8 символов [14].

Если автор программы не применяет обычную схему ведения версий, такую как `2.30.32`, а использует какую-то зависимость от даты, такую как `11Apr29`, произвольную строку с именем или хэш-значение из VCS для части версии, убедитесь, что удалили это из **версии исходной программы**. Эту информацию можно сохранить в файле `debian/changelog`. Если вам нужно придумать строку для версии исходной программы, используйте формат `ГГГГММДД` (`20110429`). Это позволит **dpkg** правильно учитывать номер версии при обновлениях. Если в будущем вам предстоит выполнить плавный переход на обычную схему версий, такую как `0.1`, используйте формат `0~YYMMDD` (`0~110429`) для версии исходной программы.

Строки версий [15] можно сравнить с помощь dpkg(1) следующим образом:

```
$ dpkg --compare-versions версия1 операция версия2
```

Правило сравнения версий вкратце можно описать так:

- Строки сравниваются от начала к концу.
- Буквы больше, чем цифры.
- Числа сравниваются как целые.
- Буквы сравниваются согласно порядку кодов ASCII.
- Для символов точки (`.`), плюса (`+`) и тильды (`~`) правила следующие:
  `0.0` < `0.5` < `0.10` < `0.99` < `1` < `1.0~rc1` < `1.0` < `1.0+b1` < `1.0+nmu1` < `1.1` < `2.0`

[10] Вы можете это автоматизировать с помощью пакета `dh-autoreconf`. Смотрите Раздел 4.4.3.

[11] В **aptitude** длина поля имени пакет по умолчанию равна 30. Длина имён более чем 90% пакетов менее 24 символов.

[12] Если вы посмотрите справочник разработчика Debian, раздел 5.1. «Новые пакеты» (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage) , то увидите, что в процессе ITP обычно выявляются проблемы с именованием.

[13] Это жёсткое правило должно помочь избежать путаницы в именах файлов.

[14] В **aptitude** длина поля версии по умолчанию равна 10. Редакция Debian с символом переноса обычно занимает 2 символа. В более 80% пакетов версия исходной программы —менее 8 символов, а редакция Debian —менее 2 символов. В более 90% пакетов версия исходной программы —менее 10 символов, а редакция Debian —менее 3 символов.

[15] Строками версий могут быть **версия исходной программы** (*версия*), **редакция Debian** (*редакция*) или **версия** (*версия*-*редакция*). Смотрите в Раздел 8.1 как увеличивается значение **редакции Debian**.

Иногда бывает, что автор выпускает предварительную версию `gentoo-0.9.12.tar.gz` с именем `gentoo-0.9.12-ReleaseCandidate-99.tar.gz`. Чтобы правильно сработало обновление пакета, вам нужно переименовать файл с исходным кодом в `gentoo-0.9.12~rc99.tar.gz`.

2.7 Настройка dh_make

Для указания вашего адреса электронной почты и имени, которые будут использоваться в пакетах инструментами сопровождения Debian, настройте переменные окружения `$DEBEMAIL` и `$DEBFULLNAME` [16]:

```
$ cat >>~/.bashrc <<EOF
DEBEMAIL="ваш.адрес.эл.почты@example.org"
DEBFULLNAME="Имя Фамилия"
export DEBEMAIL DEBFULLNAME
EOF
$ . ~/.bashrc
```

2.8 Начальный неродной пакет Debian

Обычно, пакеты Debian являются неродными (non-native) пакетами Debian, создаваемыми из внешнего исходного кода. Если вы хотите создать неродной пакет Debian из исходного кода `gentoo-0.9.12.tar.gz`, то можете сгенерировать начальный неродной пакет Debian с помощью команды **dh_make** следующим образом:

```
$ cd ~/gentoo
$ wget http://example.org/gentoo-0.9.12.tar.gz
$ tar -xvzf gentoo-0.9.12.tar.gz
$ cd gentoo-0.9.12
$ dh_make -f ../gentoo-0.9.12.tar.gz
```

Здесь замените имя файла именем вашего архива с исходным кодом [17]. Подробней смотрите dh_make(8).

После запуска вам будет предложено указать тип создаваемого пакета. Для gentoo —одиночный двоичный пакет —из него создаётся только один двоичный пакет, т.е, один файл `.deb`, поэтому выберите первый вариант (клавишей `s`), проверьте информацию на экране и подтвердите выбор нажатием *`ENTER`* [18].

После выполнения **dh_make** в родительском каталоге создался исходный архив tar с именем `gentoo_0.9.12.orig.tar.gz`, который в дальнейшем будет использоваться для создания неродного пакета с исходным кодом Debian с именем `debian.tar.gz`:

```
$ cd ~/gentoo ; ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
gentoo_0.9.12.orig.tar.gz
```

Отметьте два ключевых момента в имени файла `gentoo_0.9.12.orig.tar.gz`:

- Имя пакета и версия разделены символом _ (подчёркивание).

[16] В примере предполагается, что в качестве регистрационной оболочки у вас используется Bash. Если вы используете другую регистрационную оболочку (например, Z shell), используйте другие соответствующие файлы настройки вместо `~/.bashrc`.

[17] Если автор исходного кода включил каталог `debian` в исходный код, то программу **dh_make** нужно запускать с дополнительным параметром `--addmissing`. Новый формат исходного кода `3.0 (quilt)` позволяет ничего не сломать даже в подобных пакетах. Вам может потребоваться обновить содержимое, предоставленное автором программы для создания пакета Debian.

[18] Здесь предлагается несколько вариантов: `s` —одиночный двоичный пакет, `i` —пакет, независящий от архитектуры, `m` —несколько двоичных пакетов, `l` —пакет для библиотеки, `k` —пакет для модуля ядра, `n` —пакет с заплатами к ядру и `b` —пакет, применяющий `cdbs`. В этом документе, в основном, описывается использование команды **dh** (из пакета `debhelper`) для создания одиночного двоичного пакета, но также затрагиваются варианты с пакетами, независящими от архитектуры, и когда из одной программы создаётся несколько двоичных пакетов. Пакет `cdbs` предоставляет инфраструктуру пакетирования альтернативную команде **dh** и не описан в этом документе.

- Перед `.tar.gz` есть часть `.orig`.

Вы, наверное, уже заметили, что в исходном коде в каталоге `debian` было создано множество файлов шаблонов. Их назначение описано в Глава 4 и Глава 5. Как вы уже догадались, процесс пакетирования не может быть полностью автоматическим. Вам нужно изменить исходный код программы для Debian (Глава 3). После этого вам нужно правильно собрать пакет Debian (Глава 6), проверить его (Глава 7) и отослать в архив (Глава 9). Далее будут рассмотрены все эти шаги.

Если вы случайно стёрли какой-то файл шаблона при работе, то можете восстановить его, повторно запустив **dh_make** с параметром `--addmissing` в дереве исходного кода пакета Debian.

Обновлять существующий пакет сложнее, так как для его создания могли использоваться старые методы. Поэтому на время обучения пока беритесь за современные пакеты. Мы вернёмся к этому позднее в Глава 8.

Заметим, что в файле с исходным кодом необязательно должна использоваться одна из систем сборки, описанная в Раздел 2.4 и Раздел 2.5. Он может содержать просто набор графических данных и т.п. В этом случае установку файлов можно выполнить только с помощью файлов настройки `debhelper`, таких как `debian/install` (смотрите Раздел 5.11).

Глава 3

Изменение исходного кода

Заметим, что в данный документ невозможно поместить *всю* информацию по исправлению исходного кода программы, однако здесь приведены основные шаги и проблемы, с которыми часто встречаются люди.

3.1 Настройка quilt

Программа **quilt** предлагает простой способ записи изменений, произведённых в исходном коде для пакетирования Debian. Для удобства использования слегка изменим настройки по умолчанию, добавив для пакетирования псевдоним **dquilt** в файле `~/.bashrc`. Вторая строка служит для включения автодополнения к **dquilt**, такого же как у команды **quilt**:

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F _quilt_completion -o filenames dquilt
```

Затем создадим файл `~/.quiltrc-dpkg` со следующим содержимым:

```
d=. ; while [ ! -d $d/debian -a `readlink -e $d` != / ]; do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
    # если в пакетируемом дереве Debian не задана $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_ctx=35: ↩
        diff_cctx=33"
    if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

О том, как использовать **quilt**, читайте в quilt(1) и `/usr/share/doc/quilt/quilt.pdf.gz`.

3.2 Исправление ошибок в исходной программе

Предположим, вы нашли ошибку в файле `Makefile` из архива программы —вместо `install: gentoo` должно быть `install: gentoo-target`:

```
install: gentoo
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Внесём исправление и запишем его при помощи команды **dquilt** в файл `fix-gentoo-target.patch` [1]:

```
$ mkdir debian/patches
$ dquilt new fix-gentoo-target.patch
$ dquilt add Makefile
```

Изменим файл `Makefile` следующим образом:

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Запустите **dquilt** для генерации и записи заплаты в файл `debian/patches/fix-gentoo-target.patch` и добавьте к ней описание в соответствии с DEP-3: Руководство по маркировке заплат (http://dep.debian.net/deps/dep3/) :

```
$ dquilt refresh
$ dquilt header -e
... описание заплаты
```

3.3 Установка файлов в их каталоги назначения

Большинство стороннего ПО устанавливает себя в иерархию каталогов `/usr/local`. В Debian это место зарезервировано для использования администратором по своему усмотрению, поэтому пакеты не должны использовать такие каталоги как `/usr/local/bin`, а должны устанавливаться в системные каталоги, такие как `/usr/bin`, согласно стандарту иерархии файловой системы (http://www.debian.org/doc/packaging-manuals/fhs/fhs-2.3.html) (FHS).

Обычно, для автоматизации сборки программы используется make(1), а по команде `make install` выполняется установка программ в желаемый каталог, назначенный для цели `install` в файле `Makefile`. Для создания двоичных, готовых к установке пакетов Debian, требуется изменить систему сборки таким образом, чтобы она устанавливала программы в образ файловой системы, развёрнутый во временном каталоге, а не в реальное место назначения.

Эти два различия между нормальной установкой программы, с одной стороны, и системой пакетирования Debian с другой, могут быть прозрачно переданы пакетом `debhelper` с помощью команд **dh_auto_configure** и **dh_auto_install**, если соблюдаются следующие условия:

- Файл `Makefile` соответствует соглашениям GNU и поддерживает переменную `$(DESTDIR)` [2].
- Исходный код следует FHS.

Программы, использующие пакет GNU **autoconf**, автоматически следуют соглашениям GNU, и их легко пакетировать. На основе этого и эвристики можно сделать вывод, что пакет `debhelper` будет работать с почти 90% пакетов без внесения серьёзных изменений в их системы сборки. Поэтому процесс пакетирования не так сложен, как кажется.

Если вам необходимо внести изменения в файл `Makefile`, убедитесь, что он поддерживает переменную `$(DESTDIR)`. Значение переменной `$(DESTDIR)` явно в нём не задаётся, но указывается в начале каждого файлового пути, который используется для установки программы. Сценарий пакетирования присвоит `$(DESTDIR)` значение временного каталога.

При создании из исходного кода одиночного пакета временный каталог, используемый командой **dh_auto_install**, будет установлен в `debian/`*`пакет`* [3]. Всё, что содержится во временном каталоге, будет помещено в систему пользователя при

[1] Каталог `debian/patches` уже должен существовать, если вы запускали **dh_make**, как это описывалось ранее. В этом примере каталог создаётся вручную, на случай если обновляется существующий пакет.

[2] Смотрите Стандарты программирования GNU: 7.2.4 DESTDIR: Поддержка поэтапной установки (http://www.gnu.org/prep/standards/html_node/-DESTDIR.html#DESTDIR) .

[3] При создании из исходного кода нескольких двоичных пакетов команда **dh_auto_install** использует `debian/tmp` в качестве временного каталога, а команда **dh_install** с помощью файлов `debian/`*`пакет-1`*`.install` и `debian/`*`пакет-2`*`.install` разнесёт содержимое `debian/tmp` по временным каталогам `debian/`*`пакет-1`* и `debian/`*`пакет-2`* для создания двоичных пакетов *`пакет-1`*`_*.deb` и *`пакет-2`*`_*.deb`.

установке пакета. Различие в том, что **dpkg** установит файлы в систему относительно корневого каталога, а не вашего рабочего каталога.

Помните: даже если ваша программа устанавливается в `debian/`*`пакет`*, нужно внимательно следить за правильностью её установки из пакета `.deb` в корневой каталог. Поэтому вы не должны разрешать системе сборки записывать неизменяемые строки вроде `/home/moero/deb/`*`пакет-версия`*`/usr/share/`*`пакет`* в файлы пакета.

Вот соответствующая часть файла `Makefile`, взятого из пакета `gentoo`[4]:

```
# Куда помещать исполняемые файлы по команде «make install»?
BIN     = /usr/local/bin
# Куда помещать значки по команде «make install»?
ICONS   = /usr/local/share/gentoo
```

Данные настройки предполагают установку в `/usr/local`. Как указывалось ранее, эта иерархия каталогов зарезервирована для локального использования в Debian, поэтому измените этот путь на:

```
# Куда помещать исполняемые файлы по команде «make install»?
BIN     = $(DESTDIR)/usr/bin

# Куда помещать значки по команде «make install»?
ICONS   = $(DESTDIR)/usr/share/gentoo
```

Где именно должны располагаться исполняемые файлы, значки, документация и т.д. описывает FHS —стандарт иерархии файловой системы. Рекомендуется обратить внимание на разделы, которые касаются вашего пакета.

Итак, исполняемые файлы следует устанавливать в `/usr/bin` вместо `/usr/local/bin`, справочные страницы в `/usr/share/man/man1` вместо `/usr/local/man/man1` и т.д. Обратите внимание, что хотя в `Makefile` из пакета `gentoo` отсутствует упоминание о справочной странице, Debian Policy требует её наличия для каждой программы, поэтому позднее мы создадим такую страницу и установим её в `/usr/share/man/man1`.

Некоторые программы не используют переменные в `Makefile` для подобного указания путей. Это означает, что, возможно, вам придётся редактировать исходные файлы, написанные на языке C, для указания правильного расположения. Но где и как искать эти пути? Для этого вы можете воспользоваться следующей командой:

```
$ grep -nr --include='*.[c|h]' -e 'usr/local/lib' .
```

Команда **grep** рекурсивно обходит всё дерево исходного кода и при обнаружении совпадений сообщает вам имя соответствующего файла и номер строки.

Отредактируйте указанные строки в этих файлах, заменив `usr/local/lib` на `usr/lib`. Это можно сделать автоматически с помощью команды:

```
$ sed -i -e 's#usr/local/lib#usr/lib#g' \
       $(find . -type f -name '*.[c|h]')
```

Если вы хотите подтверждать каждую замену, запустите следующую команду:

```
$ vim '+argdo %s#usr/local/lib#usr/lib#gce|update' +q \
       $(find . -type f -name '*.[c|h]')
```

После этого вам нужно найти цель `install` (обычно достаточно найти строку, начинающуюся с `install:`) и заменить все ссылки на каталоги, которые отличаются от указанных в начале файла `Makefile`.

Первоначально, цель `install` в `gentoo` была такой:

```
install: gentoo-target
       install ./gentoo $(BIN)
       install icons/* $(ICONS)
       install gentoorc-example $(HOME)/.gentoorc
```

[4] Это просто пример, который показывает как должен выглядеть `Makefile`. Если `Makefile` создан командой **./configure**, то исправлять файл `Makefile` предпочтительно вызовом **./configure** из **dh_auto_configure** с параметрами по умолчанию и добавленным параметром `--prefix=/usr`.

Давайте исправим ошибку с путями в исходной программе и запишем её при помощи команды **dquilt** в файл `debian/patches/install.patch`.

```
$ dquilt new install.patch
$ dquilt add Makefile
```

Воспользуемся редактором для внесения в пакет Debian следующих изменений:

```
install: gentoo-target
        install -d $(BIN) $(ICONS) $(DESTDIR)/etc
        install ./gentoo $(BIN)
        install -m644 icons/* $(ICONS)
        install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

Разумеется вы заметили, что в начале цели появилась команда `install -d`. В исходном `Makefile` её не было, так как обычно `/usr/local/bin` и другие каталоги уже существуют в системе на момент запуска `make install`. Однако, так как мы будем проводить установку в создаваемый каждый раз свой каталог, нам следует создать в нём все необходимые каталоги.

В конец правила установки мы можем добавить что-нибудь ещё, например, установку дополнительной документации, которую не включили авторы программы:

```
        install -d $(DESTDIR)/usr/share/doc/gentoo/html
        cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

Убедившись, что всё сделано правильно, с помощью **dquilt** сгенерируем заплату `debian/patches/install.patch` и добавим её описание:

```
$ dquilt refresh
$ dquilt header -e
... описание заплаты
```

Теперь у вас есть несколько заплат:

1. Заплата для исправления ошибки в программе: `debian/patches/fix-gentoo-target.patch`
2. Изменения, необходимые для создания пакета Debian: `debian/patches/install.patch`

Всякий раз внося изменения, которые не относятся к пакету Debian, как те, что содержатся в `debian/patches/fix-gentoo-patch`, не забывайте отправлять их сопровождающему программы, чтобы он мог включить эти правки в следующую версию и они стали доступны всем. Перед отправкой ещё раз убедитесь, что ваши исправления не относятся только к Debian или Linux (или даже вообще Unix) —это требуется для переносимости. Такие исправления легче применять.

Не отправляйте разработчикам программы файлы `debian/*`.

3.4 Несовпадение библиотек

Нередко можно столкнуться с проблемой несовпадения библиотек на различных платформах. Например, файл `Makefile` может содержать ссылки на библиотеку, которой нет в Debian. В этом случае вы должны изменить ссылку, указав библиотеку, служащую тем же самым целям, но присутствующую в Debian.

Предположим, что в программе есть `Makefile` (или `Makefile.in`) со строкой следующего вида:

```
LIBS = -lfoo -lbar
```

Если ваша программа не компилируется из-за отсутствия библиотеки `foo` и в Debian для неё есть замена в виде `foo2`, то вы можете исправить проблему со сборкой с помощью `debian/patches/foo2.patch`, который заменяет `foo` на `foo2`:[5]

[5] Если изменяется программный интерфейс (API) (то есть, вместо библиотеки `foo` начинают использовать библиотеку `foo2`), то требуется изменить исходный код так, чтобы он использовал новый API.

```
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e ’s/-lfoo/-lfoo2/g’ Makefile
$ dquilt refresh
$ dquilt header -e
... описание заплаты
```

Глава 4

Обязательные файлы в каталоге `debian`

В каталоге с исходным кодом программы появился новый подкаталог `debian`. В нём содержится несколько файлов, которые нужно отредактировать для изменения свойств пакета. Наиболее важные из них: `control`, `changelog`, `copyright` и `rules`; они обязательны для всех пакетов [1].

4.1 Файл `control`

Этот файл содержит информацию, которая используется программами **dpkg**, **dselect**, **apt-get**, **apt-cache**, **aptitude** и некоторыми другими инструментами для работы с пакетами. Он описан в руководстве по политике Debian, в главе 5 «Управляющие файлы и их поля» (http://www.debian.org/doc/debian-policy/ch-controlfields.html) .

Вот, например, файл `control`, который был создан для нас программой **dh_make**:

```
 1 Source: gentoo
 2 Section: unknown
 3 Priority: extra
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=9)
 6 Standards-Version: 3.9.4
 7 Homepage: <insert the upstream URL, if relevant>
 8
 9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Description: <insert up to 60 chars description>
13  <insert long description, indented with spaces>
```

(номера строк добавлены для наглядности)

Строки 1-7 содержат управляющую информацию для пакета с исходным кодом. Строки 9-13 содержат управляющую информацию для двоичного пакета.

В строке 1 содержится название пакета с исходным кодом.

В строке 2 содержится название раздела в дистрибутиве, к которому относится пакет с исходным кодом.

Возможно вы уже заметили, что архив Debian разбит на несколько областей: `main` (свободное ПО), `non-free` (не совсем свободное ПО) и `contrib` (свободное ПО, зависящее от несвободного ПО). Каждая из них делится на разделы, чтобы приближённо разделить пакеты по категориям. Так, в `admin` содержатся программы только для администратора, в `devel` хранятся инструменты разработки ПО, в `doc` содержится документация, в `libs` содержатся библиотеки,

[1] Для простоты в этой главе файлы в каталоге `debian` указаны без начального `debian/`.

в `mail` содержатся почтовые серверы и программы чтения почты, в `net` содержатся сетевые приложения и службы, в `x11` содержатся программы для X11, которые не вошли куда-то ещё, и так далее [2].

В нашем случае мы должны указать x11 (префикс `main/` указывать не обязательно —он подставляется по умолчанию).

В строке 3 указывается насколько важен данный пакет [3]:

- Приоритет `optional`, обычно, назначается новым пакетам, которые не конфликтуют с другими, имеющими приоритет `required`, `important` или `standard`.
- Приоритет `extra`, обычно, назначается новым пакетам, которые конфликтуют с другими пакетами, имеющими приоритета не `extra`.

Значения раздела и приоритета учитываются в интерфейсах управления пакетами (например, в **aptitude**) при сортировке и выборе пакетов. При размещении пакета в архиве Debian значения этих полей могут быть изменены администратором архива, о чём вам будет сообщено по электронной почте.

Так как наш пакет имеет обычный приоритет и не конфликтует с другими, мы укажем значение приоритета `optional`.

В строке 4 указано имя и адрес электронной почты сопровождающего. Убедитесь, что это значение пригодно для поля `To` заголовка электронной почты, потому что после загрузки пакета в архив это значение будет использовано системой отслеживания ошибок для связи с вами. Не используйте в этой строке запятые, скобки и амперсанд.

В строке 5 содержится список пакетов, необходимых для сборки вашего пакета; это значение присваивается полю `Build-Depends`. Также, здесь можно вставить дополнительную строку с полем `Build-Depends-Indep` [4]. Некоторые пакеты (например, `gcc` и `make`), требуемые пакетом `build-essential`, будут подставлены автоматически в список зависимостей. Если вам требуются дополнительные инструменты сборки —добавьте их в эту строку. В качестве разделителя элементов используется запятая. Для более подробного ознакомления с синтаксисом этой строки обратитесь к материалам по зависимостям двоичных пакетов.

- Для всех пакетов, упаковываемых с помощью команды **dh**, в файле `debian/rules` вам потребуется указать `debhelper (>=9)` в поле `Build-Depends` для соответствия требованиям политики Debian, касающимся цели `clean`.
- Пакеты с исходным кодом, в двоичных пакетах которых указано `Architecture: any`, пересобираются с помощью autobuilder. Так как данная процедура autobuilder перед запуском `debian/rules build` устанавливает только пакеты, перечисленные в поле `Build-Depends` (смотрите Раздел 6.2), то это поле должно содержать практически все необходимые пакеты, в отличие от `Build-Depends-Indep`, которое используется редко.
- В пакетах с исходным кодом, в двоичных пакетах которых указано `Architecture: all`, поле `Build-Depends-Indep` может содержать все требуемые пакеты, не перечисленные в `Build-Depends`, для соответствия требованиям политики Debian, касающимся цели `clean`.

Если вы не знаете, какое из двух полей использовать —остановитесь на поле `Build-Depends` [5].

Для выяснения того, какие пакеты требуются для сборки, выполните команду:

```
$ dpkg-depcheck -d ./configure
```

Для ручного, более точного поиска зависимостей программы ***/usr/bin/foo*** выполните:

```
$ objdump -p /usr/bin/foo | grep NEEDED
```

[2] Смотрите руководство по политике Debian, раздел 2.4 «Разделы» (http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections) и список разделов в `sid` (http://packages.debian.org/unstable/) .

[3] Смотрите руководство по политике Debian, раздел 2.5 «Приоритеты» (http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities) .

[4] Смотрите руководство по политике Debian, раздел 7.7 «Связи между пакетами с исходным кодом и двоичными пакетами —Build-Depends, Build-Depends-Indep, Build-Conflicts, Build-Conflicts-Indep» (http://www.debian.org/doc/debian-policy/ch-relationships.html#s-sourcebinarydeps) .

[5] Эта несколько странная ситуация является хорошо документированной возможностью, описанной в руководстве по политике Debian, сноска 55 (http://www.debian.org/doc/debian-policy/footnotes.html#f55) . Причина не в использовании команды **dh** в файле `debian/rules`, а в специфике работы **dpkg-buildpackage**. Аналогичный случай встречается в системе автоматической сборки Ubuntu (https://bugs.launchpad.net/launchpad-buildd/+bug/-238141) .

и для каждой найденной библиотеки, например, **libfoo.so.6**, выполните:

```
$ dpkg -S libfoo.so.6
```

Затем укажите `-dev` версию каждого пакета в поле `Build-Depends`. Если для этой цели воспользоваться **ldd**, то вы, помимо прочего, узнаете неявные библиотечные зависимости библиотеки и столкнётесь с проблемой избыточных сборочных зависимостей.

Кроме того, пакет `gentoo` требует для сборки пакеты `xlibs-dev`, `libgtk1.2-dev` и `libglib1.2-dev`. Укажите их после пакета `debhelper`.

В строке 6 указывается версия документа руководства по политике Debian (http://www.debian.org/doc/devel-manuals#policy) , стандартам которого следует данный пакет. Прочитайте его при создании пакета.

В строке 7 можно указать URL домашней страницы программы.

В строке 9 содержится имя двоичного пакета. Обычно, оно совпадает с именем пакета с исходным кодом, но это не является обязательным требованием.

В строке 10 перечислены архитектуры двоичного пакета, для которых он может быть скомпилирован. Обычно, здесь указывает одно из следующих значений, в зависимости от типа двоичного пакета [6]:

- `Architecture: any`
  - Генерируемый двоичный пакет зависит от архитектуры, обычно определяемой компилируемым языком.
- `Architecture: all`
  - Генерируемый двоичный пакет не зависит от архитектуры, обычно в нём содержится текст, изображения или сценарии интерпретируемого языка.

Мы не будет менять строку 10, так как программа написана на C. Утилита dpkg-gencontrol(1) подставит соответствующее значение архитектуры машины, на которой будет скомпилирован пакет с исходным кодом.

Если ваш пакет не зависит от архитектуры (например, это документ, сценарий оболочки или Perl), укажите значение `all` и прочитайте Раздел 4.4, в котором описаны правила использования `binary-indep` вместо `binary-arch`.

В строке 11 показана одна из мощнейших сторон пакетной системы Debian. Пакеты могут быть связаны друг с другом различными способами. Кроме поля `Depends` за отношения между пакетами отвечают `Recommends`, `Suggests`, `Pre-Depends`, `Breaks`, `Conflicts`, `Provides` и `Replaces`.

Инструменты управления пакетами, обычно, одинаковым образом обрабатывают эти связи; если это так, то будет приведён комментарий (смотрите dpkg(8), dselect(8), apt(8), aptitude(1) и т.д.).

Вот упрощённое описание связей между пакетами [7]:

- `Depends`

  Пакет не будет установлен, пока не установлены пакеты, от которых он зависит. Используйте этот тип зависимости, если ваша программа гарантировано не будет работать (или вызовет какие-нибудь серьезные проблемы) при отсутствии какого-то пакета.

- `Recommends`

  Используйте это поле для пакетов, которые не обязательны, но обычно используются с вашей программой. При установке программы все интерфейсы управления пакетами предложат пользователю установить и рекомендуемые пакеты. Утилиты **aptitude** и **apt-get** по умолчанию (которое можно изменить) автоматически устанавливают рекомендуемые пакеты вместе с пакетом. Утилита **dpkg** игнорирует это поле.

[6] Подробней об этом смотрите в руководстве по политике Debian, раздел 5.6.8 «Architecture» (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) .

[7] Смотрите руководство по политике Debian, главу 7 «Объявление связей между пакетами» (http://www.debian.org/doc/debian-policy/ch-relationships.html) .

- `Suggests`

  Используйте это поле для пакетов, которые отлично дополнят вашу программу, но не требуются для её работы. При установке программы интерфейсы управления пакетами, обычно, не предлагают пользователю установить такие пакеты. В **aptitude** можно включить автоматическую установку этих предлагаемых (suggested) пакетов (по умолчанию не выполняется). Программы **dpkg** и **apt-get** игнорируют это поле.

- `Pre-Depends`

  В этом поле указываются более важные зависимости, чем в `Depends`. Пакет не будет установлен, если какие-либо пакеты из числа таких зависимостей не установлены, либо *не правильно настроены*. Используйте это поле *очень* осторожно и только после обсуждения в рассылке debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) . Ещё лучше —не используйте его вообще :-)

- `Conflicts`

  Пакет не будет установлен, пока все перечисленные в этом поле пакеты не будут удалены. Используйте это поле только, если ваша программа не будет запускаться, либо возникнут серьёзные проблемы при наличии этих пакетов.

- `Breaks`

  Если пакет будет установлен, то работоспособность всех перечисленных пакетов будет нарушена. Чаще всего, в поле `Breaks` указываются пакеты с уточнением версии типа «старее чем». Утилиты управления пакетами, обычно, предлагают обновить перечисленные пакеты до более новых версий.

- `Provides`

  В случае, когда для какого-то типа пакетов существует несколько альтернатив, вводятся виртуальные имена. Полный список виртуальных пакетов приведён в файле virtual-package-names-list.txt.gz (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) . Вы должны использовать данное поле, если ваша программа выполняет функции существующего виртуального пакета.

- `Replaces`

  Используйте данное поле, если ваш пакет заменяет файлы из другого пакета или же полностью заменяет другой пакет (в этом случае вы также должны использовать поле `Conflicts`). В этом случае файлы из указанного пакета будут перезаписаны файлами из вашего.

Формат всех этих полей одинаков. Он представляет собой список имён пакетов, разделённых запятыми. Здесь также могут быть указаны имена альтернативных пакетов, разделённые вертикальной чертой `|` (символ канала).

Для каждого пакета в списке можно указать версии, которых касается данная зависимость. Версии указываются в круглых скобках после имени пакета и должны состоять из символа сравнения, за которым следует номер версии. Допустимыми символами сравнения являются: `<<`, `<=`, `=`, `>=` и `>>` для «строго раньше», «раньше или равно», «в точности равно», «равно или позже» и «строго позже», соответственно. Например:

```
Depends: foo (>= 1.2), libbar1 (= 1.3.4)
Conflicts: baz
Recommends: libbaz4 (>> 4.0.7)
Suggests: quux
Replaces: quux (<< 5), quux-foo (<= 7.6)
```

В завершение рассмотрим конструкции `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}` и прочие.

Утилита dh_shlibdeps(1) вычисляет зависимости двоичного пакета от общих библиотек. Она генерирует список исполняемых файлов ELF и общих библиотек, которые находит для каждого двоичного пакета. Этот список подставляется вместо `${shlibs:Depends}`.

Утилита dh_perl(1) вычисляет зависимости Perl. Она генерирует список зависимостей от `perl` или `perlapi` для каждого двоичного пакета. Этот список подставляется вместо `${perl:Depends}`.

Некоторые команды пакета `debhelper` могут добавлять зависимости к вашему генерируемому пакету. Каждая команда генерирует список необходимых пакетов для каждого двоичного пакета. Этот список подставляется вместо `${misc:Depends}`.

Утилита dh_gencontrol(1) генерирует файл `DEBIAN/control` для каждого двоичного пакета, заменяя `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}` и т.д на полученные значения.

В нашем случае, мы можем оставить поле `Depends` без изменений и добавить после него строчку `Suggests: file`, так как пакет `gentoo` использует некоторые возможности пакета `file`.

В строке 9 указан URL домашней страницы программы. Предположим, что это будет http://www.obsession.se/gentoo/.

В строке 12 содержится краткое описание пакета. Размер экрана у большинства пользователей имеет 80 столбцов, поэтому постарайтесь ограничить описание шестьюдесятью символами. В нашем случае, заполним поле следующим текстом: `fully GUI-configurable, two-pane X file manager`.

В строке 13 указывается длинное описание. Это должен быть абзац, содержащий более полную информацию о пакете. Каждая строка должна начинаться с пробела. В тексте не должно быть пустых строк. Если пустая строка всё же требуется, то поставьте точку (`.`) в первом столбце. Не выводите более одной пустой строки после длинного описания [8].

Давайте вставим поля `Vcs-*` для указания местонахождения системы контроля версий между строкой 6 и 7 [9]. Предположим, что пакет `gentoo` в качестве VCS использует сервис Debian Alioth Git по адресу `git://git.debian.org/git/collab-maint/gentoo.git`.

Вот обновлённый файл `control`:

```
 1 Source: gentoo
 2 Section: x11
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=9), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
 6 Standards-Version: 3.9.4
 7 Vcs-Git: git://git.debian.org/git/collab-maint/gentoo.git
 8 Vcs-browser: http://git.debian.org/?p=collab-maint/gentoo.git
 9 Homepage: http://www.obsession.se/gentoo/
10
11 Package: gentoo
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Suggests: file
15 Description: fully GUI-configurable, two-pane X file manager
16  gentoo is a two-pane file manager for the X Window System. gentoo lets the
17  user do (almost) all of the configuration and customizing from within the
18  program itself. If you still prefer to hand-edit configuration files,
19  they’re fairly easy to work with since they are written in an XML format.
20  .
21  gentoo features a fairly complex and powerful file identification system,
22  coupled to an object-oriented style system, which together give you a lot
23  of control over how files of different types are displayed and acted upon.
24  Additionally, over a hundred pixmap images are available for use in file
25  type descriptions.
26  .
29  gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
30  for its interface.
```

(номера строк добавлены для наглядности)

4.2 Файл `copyright`

Этот файл содержит информацию об авторских правах и лицензионное соглашение исходной программы. Его содержание определяется в руководстве по политике Debian, разделе 12.5 «Информация об авторских правах» (http://www.debian.org/-

[8] Эти описания составляются на английском языке. Переводы описаний выполняются через проект переводов описаний Debian —DDTP (http://www.debian.org/intl/l10n/ddtp) .

[9] Смотрите справочник разработчика Debian, раздел 6.2.5. «Местонахождение системы контроля версий» (http://www.debian.org/doc/manuals/-developers-reference/best-pkging-practices.html#bpp-vcs) .

doc/debian-policy/ch-docs.html#s-copyrightfile) , а формат описан в DEP-5: автоматизируемый `debian/copyright` (http://dep.de
deps/dep5/) .

Шаблон файла `copyright` можно создать с помощью **dh_make**. Укажем параметр `--copyright gpl2`, чтобы получить файл шаблона для пакета `gentoo`, выпущенного с лицензией GPL-2.

В этом файле вы должны указать отсутствующую информацию, например, откуда был получен пакет, действующее уведомление об авторском праве и лицензию. Список распространённых лицензий на свободное ПО: GNU GPL-1, GNU GPL-2, GNU GPL-3, LGPL-2, LGPL-2.1, LGPL-3, GNU FDL-1.2, GNU FDL-1.3, Apache-2.0 или Artistic. Их тексты можно найти в соответствующих файлах в каталоге `/usr/share/common-licenses/`, который есть в каждой системе Debian. В противном случае вы должны включить файл с полным текстом лицензии.

Вкратце, вот как должен выглядеть файл `copyright` для пакета `gentoo`:

```
 1 Format-Specification: http://svn.debian.org/wsvn/dep/web/deps/dep5.mdwn?op=file&rev=135
 2 Name: gentoo
 3 Maintainer: Josip Rodin <joy-mg@debian.org>
 4 Source: http://sourceforge.net/projects/gentoo/files/
 5
 6 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
 7 License: GPL-2+
 8
 9 Files: icons/*
10 Copyright: 1998 Johan Hanson <johan@tiq.com>
11 License: GPL-2+
12
13 Files: debian/*
14 Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
15 License: GPL-2+
16
17 License: GPL-2+
18  This program is free software; you can redistribute it and/or modify
19  it under the terms of the GNU General Public License as published by
20  the Free Software Foundation; either version 2 of the License, or
21  (at your option) any later version.
22  .
23  This program is distributed in the hope that it will be useful,
24  but WITHOUT ANY WARRANTY; without even the implied warranty of
25  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
26  GNU General Public License for more details.
27  .
28  You should have received a copy of the GNU General Public License along
29  with this program; if not, write to the Free Software Foundation, Inc.,
30  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
31  .
32  On Debian systems, the full text of the GNU General Public
33  License version 2 can be found in the file
34  ‘/usr/share/common-licenses/GPL-2’.
```

(номера строк добавлены для наглядности)

Следуйте HOWTO, написанному ftpmaster-ами, и пошлите анонс в debian-devel-announce: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

4.3 Файл `changelog`

Это обязательный файл, его специальный формат описан в руководстве по политике Debian, раздел 4.4 «debian/changelog» (http://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog) . Этот формат используется программой **dpkg** и другими для получения информации о номере версии, редакции, разделе и срочности пакета.

Для вас он также важен, так как является хорошим местом для описания всех изменений, которые вы сделали. Это поможет людям, скачавшим ваш пакет, определить, какие выпуски есть у пакета, о которых они должны знать. Он будет сохранён как `/usr/share/doc/gentoo/changelog.Debian.gz` в двоичном пакете.

Программа **dh_make** создает файл по умолчанию, похожий на этот:

```
1  gentoo (0.9.12-1) unstable; urgency=low
2
3   * Initial release (Closes: #nnnn)  <nnnn is the bug number of your ITP>
4
5  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
6
```

(номера строк добавлены для наглядности)

В строке 1 указано название пакета, версия, редакция, дистрибутив и срочность. Имя должно совпадать с названием пакета с исходным кодом. Дистрибутив должен быть `unstable` (нестабильный) и срочность не должна заменяться на что-либо большее, чем `low` (низкая) :-)

В строках 3-5 содержится заметка, в которой описывается сделанное изменение в этой версии пакета (не изменение в самой программе —для этой цели есть специальный файл, созданный автором программы, который будет установлен позднее как `/usr/share/doc/gentoo/changelog.gz`). Предположим, что номер вашего сообщения об ошибке ITP (Intent To Package) был `12345`. Новые строки вставляются как и самая верхняя строка, со звёздочками `*`. Вы можете сделать это с помощью dch(1)или вручную в текстовом редакторе.

Чтобы предотвратить случайную заливку пакета при обновлении пакета, желательно изменить имя выпуска на некорректное, например на `UNRELEASED`.

Теперь файл будет выглядеть так:

```
1  gentoo (0.9.12-1) UNRELEASED; urgency=low
2
3   * Initial Release. Closes: #12345
4   * This is my first Debian package.
5   * Adjusted the Makefile to fix $(DESTDIR) problems.
6
7  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
8
```

(номера строк добавлены для наглядности)

После проверки правильности всех изменений и их описания в `changelog`, измените имя выпуска с `UNRELEASED` на значение целевого дистрибутива `unstable` (или даже на `experimental`). [10]

Дополнительную информацию об обновлении файла `changelog` можно найти далее в Глава 8.

4.4 Файл `rules`

Теперь взглянем на то, какие же правила выполняются программой dpkg-buildpackage(1) для создания пакета. Обычно, файл `rules` это просто ещё один файл `Makefile`, но не тот, что прилагается вместе с исходным кодом. В отличие от остальных файлов в каталоге `debian`, он помечен как исполняемый.

4.4.1 Цели из файла `rules`

Файл `rules`, как и любой `Makefile`, состоит из нескольких правил, каждое из которых описывает цель и способ её достижения [11]. Новое правило начинается с объявления цели в первой колонке. Следующие за ним строки начинаются с

[10] Если для выполнения изменения вы используете команду `dch -r`, то убедитесь, что записали файл `changelog` именно редактором.

[11] Начать учиться написанию `Makefile` можно со справочника Debian, раздел 12.2. «Make» (http://www.debian.org/doc/manuals/debian-reference/-ch12#_make) . Полная документация доступна в http://www.gnu.org/software/make/manual/html_node/index.html или в пакете `make-doc` в разделе архива `non-free`.

кода TAB (ASCII 9); в них описывается команды достижения цели. Пустые строки и строки, начинающиеся с # (решётка), считаются комментариями и игнорируются [12].

Правило, которое вы хотите выполнить, вызывается по имени цели, указанному в аргументе командной строки. Например, при вызове `debian/rules build` и `fakeroot make -f debian/rules binary` выполняются правила для цели *build* и *binary*, соответственно.

Упрощённое объяснение целей:

- Цель `clean` служит для удаления всех скомпилированных, сгенерированных и бесполезных файлов в дереве сборки (требуемая).
- Цель `build` служит для сборки исходного кода в скомпилированные программы и отформатированные документы в дереве сборки (требуемая).
- Цель `build-arch` служит для сборки исходного кода в независящие от архитектуры скомпилированные программы в дереве сборки (требуемая).
- Цель `build-indep` служит для сборки исходного кода в независящие от архитектуры отформатированные документы в дереве сборки (требуемая).
- Цель `install` служит для установки файлов в дерево файлов для каждого двоичного пакета из каталога `debian` (необязательная). Цели `binary*` непосредственно зависят от этой цели.
- Цель `binary` служит для создания всех двоичных пакетов (комбинация целей `binary-arch` и `binary-indep`) (требуемая) [13].
- Цель `binary-arch` служит для создания двоичных пакетов, зависящих от архитектуры (`Architecture: any`), в родительском каталоге (требуемая) [14].
- Цель `binary-indep` служит для создания двоичных пакетов, независящих от архитектуры (`Architecture: all`), в родительском каталоге (требуемая) [15].
- Цель `get-orig-source` служит для получения самой новой версии пакета с оригинальным исходным кодом с сайта разработчика (необязательная).

Возможное непонимание должно уйти после того, как вы посмотрите на содержимое файла `rules` по умолчанию, который создаётся программой **dh_make**.

4.4.2 Файл `rules` по умолчанию

Новая версия **dh_make** генерирует очень простой, но эффективный файл `rules` по умолчанию, использующий команду **dh**:

```
 1 #!/usr/bin/make -f
 2 # смотрите debhelper(7) (раскомментируйте для включения)
 3 # вывод каждой команды, которая изменяет файлы в системе сборки
 4 #DH_VERBOSE = 1
 5
 6 # смотрите ПРИМЕРЫ в dpkg-buildflags(1) и прочтите /usr/share/dpkg/*
 7 DPKG_EXPORT_BUILDFLAGS = 1
 8 include /usr/share/dpkg/default.mk
 9
10 # смотрите FEATURE AREAS в dpkg-buildflags(1)
11 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
12
```

[12] В руководстве по политике Debian, раздел 4.9 «Главный сценарий сборки: debian/rules» (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules) этот файл описан подробно.
[13] Эта цель используется `dpkg-buildpackage` как описано в Раздел 6.1.
[14] Эта цель используется `dpkg-buildpackage -B`, как описано в Раздел 6.2.
[15] Эта цель используется `dpkg-buildpackage -A`.

```
13 # смотрите ENVIRONMENT в dpkg-buildflags(1)
14 # как сопровождающему добавить CFLAGS
15 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
16 # как сопровождающему добавить LDFLAGS
17 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
18
19 # главный сценарий упаковки на основе синтаксиса dh7
20 %:
21         dh $@
```

(Для наглядности добавлены номера строк и удалены некоторые комментарии. В реальном файле `rules` начальные пробельные символы имеют код TAB.)

Вероятно, вы знакомы с назначением строки 1 по сценариям оболочки и Perl. Она указывает операционной системе, что этот файл нужно обработать с помощью `/usr/bin/make`.

Строку 4 можно раскомментировать, установив значение переменной `DH_VERBOSE` равным 1. При этом команда **dh** будет выводить команды **dh_***, которые она выполняет. Также, здесь вы можете добавить строку `export DH_OPTIONS=-v`. В этом случае каждая команда **dh_*** будет выводить все команды, которые она выполняет. Это помогает понять что в действительности происходит в этом простом файле `rules` и при решении проблем. Новая команда **dh** является основной инструментов `debhelper` и не скрывает своих действий от вас.

Вся работа выполняется строками 20 и 21 с помощью неявного правила, в котором используется шаблонное правило. Знак процента означает «любую цель», что приводит к вызову программы **dh** с именем цели в качестве аргумента [16]. Команда **dh** представляет собой обёрточный сценарий, который запускает соответствующие последовательности программ **dh_*** в зависимости от аргумента [17].

- При запуске `debian/rules clean` выполняется команда `dh clean`, которая запускает другие:

```
dh_testdir
dh_auto_clean
dh_clean
```

- При запуске `debian/rules build` выполняется команда `dh build`, которая запускает другие:

```
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
```

- При запуске `fakeroot debian/rules binary` выполняется команда `fakeroot dh binary`, которая запускает другие[18]:

```
dh_testroot
dh_prep
dh_installdirs
dh_auto_install
dh_install
dh_installdocs
dh_installchangelogs
dh_installexamples
```

[16] Здесь используются новые возможности `debhelper` v7+. Принципы его проектирования можно найти в Not Your Grandpa's Debhelper (http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf) ; он представлен на Debconf9 разработчиком `debhelper`. В `lenny`, **dh_make** создаёт намного более сложный файл `rules` с указанием сценариев **dh_*** для всех перечисленных ранее целей, большинство из которых теперь ненужны (и это показывает возраст пакета). Новая команда **dh** более проста и избавляет вас от выполнения работы вручную. Вы по прежнему всё можете дорабатывать с помощью целей `override_dh_*`. Смотрите Раздел 4.4.3. Данное описание основано только на использовании пакета `debhelper` и не усложняет понимание процесса создания пакета использованием таких программ как `cdbs`.

[17] Вы можете проверить какие программы **dh_*** запускаются без реального выполнения действий, указав нужную *цель* как `dh --no-act` *цель* или `debian/rules -- '--no-act` *цель*`'`.

[18] В следующем примере предполагается, что ваш `debian/compat` содержит значение, равное 9 или более, что позволяет избежать автоматического вызова команд поддержки python.

```
dh_installman
dh_installcatalogs
dh_installcron
dh_installdebconf
dh_installemacsen
dh_installifupdown
dh_installinfo
dh_installinit
dh_installmenu
dh_installmime
dh_installmodules
dh_installlogcheck
dh_installlogrotate
dh_installpam
dh_installppp
dh_installudev
dh_installwm
dh_installxfonts
dh_bugfiles
dh_lintian
dh_gconf
dh_icons
dh_perl
dh_usrlocal
dh_link
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
```

- При запуске `fakeroot debian/rules binary-arch` выполняется команда `fakeroot dh binary-arch`, которая запускает те же команды что и `fakeroot dh binary`, но с параметром `-a`, добавляемым к каждой команде.
- При запуске `fakeroot debian/rules binary-indep` выполняется команда `fakeroot dh binary-indep`, которая запускает все команды что и `fakeroot dh binary`, исключая **dh_strip**, **dh_makeshlibs** и **dh_shlibdeps** и добавляя параметр `-i` к каждой оставшейся команде.

Назначение команд **dh_*** практически полностью совпадает с их именами [19]. Вот несколько наиболее примечательных команд с упрощённым описанием, предполагающим использование типичной среды сборки на основе `Makefile` [20]:

- Если существует `Makefile` с целью `distclean`, то **dh_auto_clean**, обычно, выполняет следующее [21]:

```
make distclean
```

- Команда **dh_auto_configure**, обычно, выполняет следующее (если существует `./configure`; список аргументов сокращён для повышения читаемости):

```
./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var ...
```

[19] Всю информацию о том, что делают сценарии **dh_*** и какие они имеют параметры, можно найти в их справочных страницах и документации к `debhelper`.

[20] Эти команды поддерживаются в других сборочных средах, таких как `setup.py`; их список можно получить, выполнив `dh_auto_build --list` в каталоге пакета с исходным кодом.

[21] В действительности, в `Makefile` ищется и выполняется первая из доступных целей: `distclean`, `realclean` или `clean`.

- Команда **dh_auto_build**, обычно, выполняет следующее для первой цели в `Makefile` (если он существует):

```
make
```

- Команда **dh_auto_test**, обычно, выполняет следующее (если существует `Makefile` с целью `test`; строка сокращена для повышения читаемости) [22]:

```
make test
```

- Команда **dh_auto_install**, обычно, выполняет следующее (если существует `Makefile` с целью `install`; строка сокращена для повышения читаемости):

```
make install \
  DESTDIR=/путь/к/пакету_версия-редакция/debian/пакет
```

Цели, которым требуется команда **fakeroot**, содержат **dh_testroot**. Если вы не притворитесь root с помощью этой команды, то выполнение завершится с ошибкой.

Важно учесть, что файл `rules`, созданный **dh_make**, это только предложение. Он будет работать для большинства пакетов, но в более сложных случаях не бойтесь изменять его под ваши нужды.

Хотя цель `install` не является обязательной, она всё равно поддерживается. Команда `fakeroot dh install` работает также как `fakeroot dh binary`, но останавливается после **dh_fixperms**.

4.4.3 Доработка файла `rules`

Есть много способов доработать файл `rules`, созданный новой программой **dh**.

Работу команды `dh $@` можно изменить следующим образом [23]:

- Добавление поддержки команды **dh_python2** (лучший выбор для Python) [24]:
  - В `Build-Depends` укажите пакет `python`.
  - Используйте `dh $@ --with python2`.
  - Это позволяет работать с модулями Python, использующими инфраструктуру `python`.
- Добавление поддержки команды **dh_pysupport** (устарела):
  - В `Build-Depends` укажите пакет `python-support`.
  - Используйте `dh $@ --with pysupport`.
  - Это позволяет работать с модулями Python, использующими инфраструктуру `python-support`.
- Добавление поддержки команды **dh_pycentral** (устарела):
  - В `Build-Depends` укажите пакет `python-central`.
  - Вместо `dh $@` используйте `dh $@ --with python-central`.
  - При этом отключается команда **dh_pysupport**.
  - Это позволяет работать с модулями Python, использующими инфраструктуру `python-central`.
- Добавление поддержки команды **dh_installtex**:
  - В `Build-Depends` укажите пакет `tex-common`.

[22] В действительности, в `Makefile` ищется и выполняется первая из доступных целей: `test` или `check`.

[23] Если с пакетом устанавливается файл `/usr/share/perl5/Debian/Debhelper/Sequence/`*`своё_имя`*`.pm`, то вам нужно активировать его функцию доработки командой `dh $@ --with` *`своё_имя`*.

[24] Команда **dh_python2** предпочтительнее, чем **dh_pysupport** или **dh_pycentral**. Не используйте команду **dh_python**.

  - Вместо `dh $@` используйте `dh $@ --with tex`.
  - Она регистрирует шрифты в формате Type 1, правила переносов или форматы в TeX.

- Добавление поддержки команд **dh_quilt_patch** и **dh_quilt_unpatch**:
  - В `Build-Depends` укажите пакет `quilt`.
  - Вместо `dh $@` используйте `dh $@ --with quilt`.
  - Эта команда накладывает и откатывает заплаты на исходный код из файлов каталога `debian/patches` для пакетов с исходным кодом версии `1.0`.
  - Она не требуется, если вы используете пакеты с исходным кодом новой версии `3.0 (quilt)`.

- Добавление поддержки команды **dh_dkms**:
  - В `Build-Depends` укажите пакет `dkms`.
  - Вместо `dh $@` используйте `dh $@ --with dkms`.
  - Это команда позволяет корректно использовать DKMS для пакетов с модулями ядра.

- Добавление поддержки команд **dh_autotools-dev_updateconfig** и **dh_autotools-dev_restoreconfig**:
  - В `Build-Depends` укажите пакет `autotools-dev`.
  - Вместо `dh $@` используйте `dh $@ --with autotools-dev`.
  - Эта команда обновляет и восстанавливает `config.sub` и `config.guess`.

- Добавление поддержки команд **dh_autoreconf** и **dh_autoreconf_clean**:
  - В `Build-Depends` укажите пакет `dh-autoreconf`.
  - Вместо `dh $@` используйте `dh $@ --with autoreconf`.
  - Эта команда обновляет файлы GNU Build System и восстанавливает их после сборки.

- Добавление поддержки команды **dh_girepository**:
  - Укажите пакет `gobject-introspection` в `Build-Depends`.
  - Вместо `dh $@` используйте `dh $@ --with gir`.
  - Эта команда вычислит зависимости пакетов, содержащих описательные (introspection) данные GObject и сгенерирует переменную подстановки `${gir:Depends}` для пакетной зависимости.

- Добавление поддержки свойства автодополнения **bash**:
  - Укажите пакет `bash-completion` в `Build-Depends`.
  - Вместо `dh $@` используйте `dh $@ --with bash-completion`.
  - Эта команда установит дополнения **bash** согласно файлу настройки `debian/`*`пакет`*`.bash-completion`.

Многие команды **dh_***, вызываемые новой командой **dh**, могут быть настроены в соответствующих конфигурационных файлах в каталоге `debian`. Смотрите Глава 5 и справочную страницу к каждой команде для настройки этих параметров.

Некоторые команды **dh_***, вызванные новой командой **dh**, могут потребовать от вас запустить их с некоторыми аргументами, выполнить вместе с ними дополнительные команды или пропустить их. В таких случаях надо создать цель `override_dh_`*`foo`* с правилами в файле `rules` только для той команды **dh_*foo***, которую вы собираетесь изменить. Она воспринимается как *запусти меня вместо той* [25].

Заметим, что команды **dh_auto_*** пытаются делать больше, чем было описано (очень) поверхностно ранее, для того, чтобы учесть все возможные ситуации. Использование упрощённых эквивалентов команд вместо целей `override_dh_*` —плохая идея (за исключением цели `override_dh_auto_clean`), так как это может привести к удалению важных функций `debhelper`.

Так, например, если вы хотите хранить системные файлы настройки пакета `gentoo` (использующего Autotools) в каталоге `/etc/gentoo` вместо обычного `/etc`, то можете переопределить аргумент `--sysconfig=/etc`, заданный по умолчанию, в команде **dh_auto_configure**, которая передаст его **./configure**:

[25] В `lenny`, если вы хотите изменить поведение сценария **dh_***, нужно найти соответствующую строку в файле `rules` правил и изменить её.

```
override_dh_auto_configure:
        dh_auto_configure -- --sysconfig=/etc/gentoo
```

Аргументы, указываемые после `--`, добавляются к аргументам автоматически выполняемой программы по умолчанию для их замены. Использование в данном случае команды **dh_auto_configure** более предпочтительно, чем вызов **./configure**, так как она всего лишь заменит аргумент `--sysconfig`, сохранив при этом другие безвредные аргументы, переданные **./configure**.

Если `Makefile` из исходного кода `gentoo` требует от вас указания `build` в качестве цели для сборки [26], то для этого можно создать цель `override_dh_auto_build`.

```
override_dh_auto_build:
        dh_auto_build -- build
```

Это гарантирует выполнение `$(MAKE)` со всеми аргументами по умолчанию, заданными в командах **dh_auto_build** и аргументе `build`.

Если `Makefile` из исходного кода `gentoo` требует от вас указания цели `packageclean` для очистки пакета Debian, а не `distclean` или `clean`, то для этого можно создать цель `override_dh_auto_clean`.

```
override_dh_auto_clean:
        $(MAKE) packageclean
```

Если `Makefile` из исходного кода `gentoo` содержит цель `test`, которую вы не хотите выполнять для процесса сборки пакета Debian, то можно использовать пустую цель `override_dh_auto_test`, чтобы пропустить это действие.

```
override_dh_auto_test:
```

Если в пакете `gentoo` используется необычный журнальный файл с именем `FIXES`, то по умолчанию **dh_installchangelogs** не установит этот файл. Для его установки укажите команде **dh_installchangelogs** имя `FIXES` в качестве аргумента [27].

```
override_dh_installchangelogs:
        dh_installchangelogs FIXES
```

При работе с новой командой **dh**, используйте явные цели, перечисленные в Раздел 4.4.1, остальные же (кроме `get-orig-source`) могут привести к сложностям понимания их конечного эффекта. Если возможно, ограничьтесь явно заданными целями `override_dh_*` и полностью независимыми целями.

[26] Программа **dh_auto_build** без аргументов выполняет правила первой цели в `Makefile`:

[27] Файлы `debian/changelog` и `debian/NEWS` всегда устанавливаются автоматически. Файл журнала ищется сопоставлением имён файлов, приведённых к нижнему регистру и совпадением их с именами `changelog`, `changes`, `changelog.txt` и `changes.txt`.

Глава 5

Другие файлы в каталоге `debian/`

Для контроля большей части того, что делает `debhelper` при сборке пакетов, служат необязательные файлы настройки в каталоге `debian`. В этой главе будет рассмотрено, для чего нужен каждый из них и его формат. В руководстве по политике Debian (http://www.debian.org/doc/devel-manuals#policy) и справочнике разработчика Debian (http://www.debian.org/doc/-devel-manuals#devref) приведены общие принципы пакетирования.

При запуске команда **dh_make** в каталоге `debian` создаёт несколько шаблонов файлов настройки. Большинство из них имеет расширение `.ex`. Некоторые имена файлов начинаются именем двоичного пакета. Рассмотрим каждый из них [1].

Некоторые шаблоны файлов настройки `debhelper` не могут быть созданы командой **dh_make**. В таких случаях вам необходимо создать их с помощью текстового редактора.

Если вы хотите активировать некоторые из них, выполните следующие действия:

- переименуйте файлы шаблонов, убирая расширение `.ex` или `.EX`, если они есть;
- переименуйте файлы настройки, используя имя реального двоичного пакета;
- измените содержимое файла шаблона согласно вашим нуждам;
- удалите файлы шаблонов, в которых нет необходимости;
- при необходимости, измените файл `control` (смотрите Раздел 4.1);
- при необходимости, измените файл `rules` (смотрите Раздел 4.4);

Любые файлы настройки `debhelper` без префикса *`имя пакета`* как, например, `install`, относятся к первому двоичному пакету. Если есть несколько двоичных пакетов, их файлы настройки могут быть созданы с указанием их имени в имени файла настройки, например, *`пакет-1`*`.install`, *`пакет-2`*`.install` и т.д.

5.1 Файл `README.Debian`

В этот файл записывается любая дополнительная информация, а также различия между программой из пакета Debian и исходной программой.

Программа **dh_make** создала файл, похожий на этот:

```
gentoo for Debian
-----------------
<possible notes regarding this package - if none, delete this file>
 -- Josip Rodin <joy-mg@debian.org>, Wed, 11 Nov 1998 21:02:14 +0100
```

Если вам нечего сюда написать, удалите этот файл. Смотрите dh_installdocs(1).

[1] Для простоты в этой главе файлы в каталоге `debian` указаны без начального `debian/`.

5.2 Файл `compat`

Файл `compat` определяет уровень совместимости `debhelper`. В настоящее время вы должны указывать уровень `debhelper` v9 следующим образом:

```
$ echo 9 > debian/compat
```

5.3 Файл `conffiles`

Бывает очень обидно, когда тратишь много времени и усилий на настройку программы, а при очередном обновлении все настройки исчезают. Debian предлагает решение этой проблемы через механизм пометки файлов как настроечных — conffiles [2]. Для таких файлов при обновлении пакета вам будет задан вопрос, нужно ли заменить старые файлы теми, что включены в новый пакет.

Программа dh_installdeb(1) *автоматически* помечает все файлы в каталоге `/etc` как файлы conffiles, так что, если у вашей программы есть файлы conffiles только там, вам не обязательно указывать их в этом файле. Для большинства типов пакетов единственное место (и для них так должно быть всегда), в котором содержатся файлы conffiles, это `/etc` и, таким образом, в этом файле нет необходимости.

Если ваша программа не только использует файлы настройки, но и изменяет их, будет лучше не отмечать их как conffiles, так как в этом случае **dpkg** каждый раз будет просить пользователя проверить изменения.

Если программа, которую вы упаковываете, требует от пользователя изменить файлы настройки в каталоге `/etc`, существует два популярных способа не отмечать их как conffiles, чтобы подавить вопросы со стороны **dpkg**:

- Создать символьную ссылку в каталоге `/etc`, указывающую на файл в каталоге `/var`, генерируемый сценариями сопровождающего.
- Сгенерировать файл в каталоге `/etc` с помощью сценариев сопровождающего.

Информацию о сценариях сопровождающего смотрите в Раздел 5.18.

5.4 Файлы *`пакет`*`.cron.*`

Эти файлы используются для планирования регулярно выполняемых задач. Вы можете назначить список задач, выполняемых ежечасно, ежедневно, еженедельно, ежемесячно или с любым произвольным интервалом. Вот имена этих файлов:

- *`пакет`*`.cron.hourly` —устанавливается как `/etc/cron.hourly/`*`пакет`*; выполняется один раз в час.
- *`пакет`*`.cron.daily` —устанавливается как `/etc/cron.daily/`*`пакет`*; выполняется один раз в день.
- *`пакет`*`.cron.weekly` —устанавливается как `/etc/cron.weekly/`*`пакет`*; выполняется один раз в неделю.
- *`пакет`*`.cron.monthly` —устанавливается как `/etc/cron.monthly/`*`пакет`*; выполняется один раз в месяц.
- *`пакет`*`.cron.d` —устанавливается как `/etc/cron.d/`*`пакет`*; выполняется в любое другое время.

Большинство этих файлов являются сценариями оболочки, за исключением *`пакет`*`.cron.d`, который должен иметь формат crontab(5).

Для архивирования журнальных файлов никаких файлов `cron.*` задавать ненужно —для этого есть другие средства (смотрите dh_installlogrotate(1) и logrotate(8)).

[2] Смотрите dpkg(1) и руководство по политике Debian, раздел «D.2.5 Conffiles» (http://www.debian.org/doc/debian-policy/ap-pkg-controlfields.html#s-pkg-f-Conffiles) .

5.5 Файл `dirs`

В этом файле указываются каталоги, которые необходимы для обычной установки (`make install DESTDIR=...`, вызываемая `dh_auto_install`), но которые автоматически не создаются. Обычно, это указывает на проблему в `Makefile`.

Файлы, указанные в файле `install`, не нуждаются в предварительном создании каталогов. Смотрите Раздел 5.11.

Для начала лучше попробовать запустить установку и использовать этот файл, только если есть проблемы. В начале имён каталогов, перечисляемых в файле `dirs`, косая черта не указывается.

5.6 Файл *`пакет`*`.doc-base`

Если ваш пакет содержит документацию, отличную от справочных страниц или файлов в формате info, то для её регистрации в системе вы должны воспользоваться файлом `doc-base`; это позволит пользователю найти её при помощи, например, dhelp(1), dwww(1) или doccentral(1).

Обычно, к таким файлам относятся файлы в форматах HTML, PS и PDF, помещаемые в `/usr/share/doc/`*`имя_пакета`*`/`.

Вот как выглядит файл `gentoo`, входящий в пакет `gentoo.doc-base`:

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: File Management
Format: HTML
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

Формат этого файла описан в справочной странице install-docs(8), а также в локальной копии руководства Debian по doc-base (/usr/share/doc/doc-base/doc-base.html/index.html) из пакета `doc-base`.

Подробная информация по установке дополнительной документации приведена в Раздел 3.3.

5.7 Файл `docs`

В этом файле указаны имена файлов документации, которые можно установить во временный каталог с помощью dh_installdocs(1

По умолчанию будут включены все файлы, имеющиеся в самом верхнем каталоге с исходным кодом, а именно `BUGS`, `README*`, `TODO` и т.д.

Для пакета `gentoo` также включаются несколько других файлов:

```
BUGS
CONFIG-CHANGES
CREDITS
NEWS
README
README.gtkrc
TODO
```

5.8 Файлы `emacsen-*`

В этом файле указываются файлы для Emacs, которые могут быть скомпилированы во время установки пакета.

Они устанавливаются во временный каталог при помощи dh_installemacsen(1).

Если они вам не нужны, удалите их.

5.9 Файл *пакет*.examples

Команда dh_installexamples(1) устанавливает файлы и каталоги, указанные в этом файле, как файлы примеров.

5.10 Файлы *пакет*.init и *пакет*.default

Если ваш пакет содержит службу и её необходимо запускать при старте системы, вероятнее всего, вы не обратили внимания на мой изначальный совет, не так ли? :-)

Файл *пакет*.init устанавливается как сценарий /etc/init.d/*пакет* для запуска и остановки службы. Довольно общий его шаблон создаётся командой **dh_make** в файле init.d.ex. Вам придётся его переименовать и отредактировать, обеспечив соблюдение стандарта заголовка согласно Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb) (LSB). Он устанавливается во временный каталог с помощью dh_installinit(1).

Файл *пакет*.default будет установлен в /etc/default/*пакет*. В этом файле хранятся значения по умолчанию, используемые сценарием инициализации. В большинстве случаев, файл *пакет*.default используется для отключения запуска службы, установки некоторых флагов по умолчанию или времени ожидания. Если ваш сценарий инициализации имеет какие-либо настраиваемые параметры, вам надо поместить их в файл *пакет*.default, а не в сам сценарий запуска.

Если с оригинальной программой поставляется сценарий инициализации, то вы можете использовать и его. Если он вам не подходит, то создайте свой в файле с именем *пакет*.init. Однако, если оригинальный сценарий инициализации хорошо написан и устанавливается в правильное место, вам всё ещё нужно установить символьные ссылки в rc*. Для этого замените **dh_installinit** в файле rules на следующие строки:

```
override_dh_installinit:
        dh_installinit --onlyscripts
```

Если сценарий инициализации не требуется, удалите эти файлы.

5.11 Файл install

Если есть файлы, которые необходимо установить в пакет, но стандартная команда make install не делает этого, поместите их имена и пути назначения в файл install. Они устанавливаются при помощи dh_install(1)[3]. В первую очередь, вы должны проверить, нет ли более подходящего способа это сделать. Например, файлы документации должны находиться в файле docs, а не в этом.

В файле install для каждого устанавливаемого файла отводится одна строка; в ней задаётся имя файла (относительно верхнего каталога сборки), потом пробел, потом установочный каталог (относительно каталога установки). Например, если при установке не устанавливается src/*bar*, то файл install будет выглядеть так:

```
src/bar usr/bin
```

В результате после установки пакета в системе появится исполняемая команда /usr/bin/*bar*.

Также, в файле install можно указывать только имя файла без каталога установки, если не менялся относительный путь. Такой формат, обычно, используется для большого пакета, в котором результат сборки разбивается на несколько двоичных пакетов; список устанавливаемых файлов помещается в соответствующий файл *пакет-1*.install, *пакет-2*.inst
и т.д.

Если команда **dh_install** не находит файлы в текущем каталоге, то она ищет их в debian/tmp (или в любом другом месте, указанном в --sourcedir).

[3] Этот файл заменяет устаревшую команду dh_movefiles(1), которая настраивается с помощью файла files.

5.12 Файл *пакет*.info

Если у вашего пакета есть страницы в формате info, то их нужно устанавливать при помощи dh_installinfo(1), перечислив их в файле *пакет*.info.

5.13 Файл *пакет*.links

Если вам, как сопровождающему пакета, нужно создать дополнительные символьные ссылки в каталогах сборки пакета, установите их с помощью dh_link(1), перечислив полные пути файлов источника и назначения в файле *пакет*.links.

5.14 Файлы {*пакет*.,source/}lintian-overrides

Если lintian находит ошибки, хотя для этого случая в политике Debian разрешены исключения, то можно использовать файл *пакет*.lintian-overrides или source/lintian-overrides, чтобы убрать подобные сообщения. Прочтите руководство пользователя Lintian (https://lintian.debian.org/manual/index.html) и не сообщайте о некорректности работы программы проверки.

Файл *пакет*.lintian-overrides предназначен для двоичного *пакета* и устанавливается в usr/share/lintian/overrides/*пакет* с помощью команды **dh_lintian**.

Файл source/lintian-overrides предназначен для пакета с исходным кодом. Он не устанавливается.

5.15 Файлы manpage.*

Для каждой программы должна быть справочная страница. Если её нет, её необходимо создать. Команда **dh_make** создаёт несколько шаблонов справочных страниц. Они должны быть скопированы и отредактированы для каждой команды, не имеющей собственной справочной страницы. Проверьте, что удалили неиспользованные шаблоны.

5.15.1 Файл manpage.1.ex

Справочные страницы, как правило, пишутся на языке разметки nroff(1). Файл шаблона manpage.1.ex также написан на **nroff**. Смотрите справочную страницу man(7), в которой приведено краткое описание действий по редактированию подобных файлов.

Окончательное имя файла справочной страницы должно включать имя документируемой программы, поэтому мы переименуем её с manpage на gentoo. Имя файла также включает .1 в качестве первого суффикса, который означает, что эта справочная страница описывает пользовательскую команду. Убедитесь, что используется правильный раздел. Вот короткий список разделов справочных страниц:

Раздел	Описание	Примечания
1	Пользовательские команды	Запускаемые команды или сценарии
2	Системные вызовы	Функции, предоставляемые ядром
3	Библиотечные вызовы	Функции, предоставляемые системными библиотеками
4	Специальные файлы	Обычно находятся в каталоге /dev
5	Форматы файлов	Например, формат /etc/passwd
6	Игры	Игры и другие несерьёзные программы
7	Пакеты макросов	Например, макросы **man**
8	Системное администрирование	Программы, обычно запускаемые только суперпользователем
9	Функции ядра	Нестандартные вызовы и внутреннее устройство

Таким образом, справочная страница для `gentoo` должна называться `gentoo.1`. Если в исходном коде нет справочной страницы `gentoo.1`, то её нужно создать путём переименовывания шаблона `manpage.1.ex` в `gentoo.1` и затем изменить его, используя информацию из примера и документации к исходной программе.

Также, вы можете использовать команду **help2man** для генерации справочной страницы, которая для создания использует результат запуска команды с параметрами `--help` и `--version` [4].

5.15.2 Файл `manpage.sgml.ex`

Если вы предпочитаете использовать SGML вместо **nroff**, то можете использовать шаблон `manpage.sgml.ex`. Для этого необходимо:

- переименовать файл в `gentoo.sgml`;
- установить пакет `docbook-to-man`;
- в файл `control` добавить `docbook-to-man` в строку `Build-Depends`;
- добавить цель `override_dh_auto_build` в файл `rules`:

```
override_dh_auto_build:
        docbook-to-man debian/gentoo.sgml > debian/gentoo.1
        dh_auto_build
```

5.15.3 Файл `manpage.xml.ex`

Если вы предпочитаете использовать XML вместо SGML, то можете использовать шаблон `manpage.xml.ex`. Для этого необходимо:

- переименовать исходный файл в `gentoo.1.xml`;
- установить пакет `docbook-xsl` и обработчик XSLT (рекомендуется `xsltproc`);
- в файл `control` добавить пакеты `docbook-xsl`, `docbook-xml` и `xsltproc` в строку `Build-Depends`;
- добавить цель `override_dh_auto_build` в файл `rules`:

```
override_dh_auto_build:
        xsltproc --nonet \
         --param make.year.ranges 1 \
         --param make.single.year.ranges 1 \
         --param man.charmap.use.subset 0 \
         -o debian/ \
http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.xsl\
        debian/gentoo.1.xml
        dh_auto_build
```

5.16 Файл *`пакет`*`.manpages`

Если ваш пакет содержит справочные страницы, то их следует устанавливать с помощью dh_installman(1), перечислив в файле *`пакет`*`.manpages`.

Чтобы установить справочную страницу `doc/gentoo.1` для пакета `gentoo`, создайте следующий файл `gentoo.manpages`:

```
docs/gentoo.1
```

[4] Заметим, что в шаблоне справочной страницы из **help2man** утверждается, что более подробная документация приведена в справочной системе info. Если для команды нет страницы **info**, то вам нужно изменить справочную страницу, созданную командой **help2man**.

5.17 Файл `NEWS`

Этот файл устанавливается командой dh_installchangelogs(1).

5.18 Файлы `{pre|post}{inst|rm}`

Файлы `postinst`, `preinst`, `postrm` и `prerm` [5] называются сценариями сопровождающего. Эти сценарии находятся в управляющей области пакета и запускаются программой **dpkg** при установке, обновлении или удалении пакета.

Как начинающему сопровождающему вам следует избегать ручной правки сценариев сопровождающего, так как они имеют тенденцию усложняться. Более подробную информацию смотрите в руководстве по политике Debian, глава 6 «Сценарии сопровождающего пакета и процесс установки» (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) , и взгляните на файлы примеров, предоставляемые **dh_make**.

Если вы не послушали меня и сделали собственные сценарии сопровождающего для пакета, вам следует убедиться, что вы сделали всё правильно, протестировав их не только для **установки** и **обновления**, но и для **удаления** и **вычистки**.

При обновлении пакета до новой версии на экран ничего не должно выводиться и это должен быть ненавязчивый процесс (пользователи не должны обращать внимание на обновление, за исключением обнаружения того, что старые ошибки были исправлены и, возможно, появились новые возможности).

Когда для обновления требуются какие-то действия (например, файлы настройки разбросаны по различным каталогам в домашнем каталоге с абсолютно разной структурой), вы можете перевести пакет в безопасное состояние (например, отключив службу) и предоставить соответствующую документацию, требуемую политикой (`README.Debian` и `NEWS.Debian`). Не беспокойте пользователя меню **debconf**, вызываемым из сценариев сопровождающего для обновления.

Пакет `ucf` предоставляет *conffile-подобную* инфраструктуру для сохранения пользовательских изменений в файлах, которые не могут быть помечены как *conffiles*, например файлы, управляемые сценариями сопровождающего. Это должно минимизировать возможные проблемы, связанные с ними.

Сценарии сопровождающего являются сильной стороной Debian, из-за них **люди выбирают Debian**. Вы должны быть очень осторожны, чтобы не превратить их в источник раздражения.

5.19 Файл *`пакет`*`.symbols`

Новичкам лучше не браться за пакетирование библиотек, так как это не просто. Но всё же, если вы упаковываете библиотеки, у вас должны быть файлы `debian/`*`пакет`*`.symbols`. Смотрите Раздел A.2.

5.20 Файл `TODO`

Этот файл устанавливается командой dh_installdocs(1).

5.21 Файл `watch`

Формат файла `watch` описан в справочной странице uscan(1). Файл `watch` настраивает программу **uscan** (предоставляется пакетом `devscripts`) для слежения за сайтами, с которых вы скачали исходный код. Он также используется службой Debian для слежения за состоянием внешних источников (DEHS) (http://wiki.debian.org/DEHS) .

Вот его содержимое:

[5] Несмотря на то, что здесь используется сокращённое выражение **bash** для обозначения этих файлов в виде `{pre|post}{inst|rm}`, в сценариях сопровождающего рекомендуется использовать только синтаксис POSIX для лучшей совместимости с системной оболочкой **dash**.

```
# управляющий файл watch для uscan
version=3
http://sf.net/gentoo/gentoo-(.+)\.tar\.gz debian uupdate
```

Обычно, с указанного в файле `watch` URL `http://sf.net/gentoo` скачивается страница и в ней ищутся ссылки вида `<a href=...>`. Базовое имя (часть за последним `/`) каждой найденной ссылки сравнивается с шаблоном регулярного выражения Perl (смотрите perlre(1)) `gentoo-(.+)\.tar\.gz`. Из совпавших файлов загружается файл с наибольшим номером версии и запускается программа **uupdate**, которая создаёт из них обновлённое дерево исходного кода.

Хотя это верно для других сайтов, служба загрузки SourceForge http://sf.net является исключением. Когда файл `watch` содержит URL, совпадающий с регулярным выражением Perl `^http://sf\.net/`, программа **uscan** заменяет его на `http://qa.debian.org/watch/sf.php/` и затем применяет это правило. Служба перенаправлений URL http://qa.debian.o создана для обеспечения стабильности перенаправлений на требующийся файл для любого файла `watch`, который содержит `http://sf.net/`*`проект/имя-tar`*`-(.+)\.tar\.gz`. Это решает проблемы, связанные с периодическими изменениями URL на SourceForge.

Если автор программы подписывает tarball ключом, то рекомендуется проверять его подпись с помощью параметра `pgp sigurlmangle` как описано в uscan(1).

5.22 Файл `source/format`

В файле `debian/source/format` должна содержаться единственная строка, указывающая на желаемый формат пакета с исходным кодом (полный список смотрите в dpkg-source(1)). После `squeeze` он должен содержать либо:

- `3.0 (native)` —для родных пакетов Debian;
- `3.0 (quilt)` —для всего остального.

В новом формате пакетов с исходным кодом `3.0 (quilt)` изменения записываются в виде серии заплат **quilt** в каталог `debian/patches`. Эти изменения автоматически накладываются в ходе распаковки пакета исходного кода [6]. Изменения Debian хранятся в архиве `debian.tar.gz`, содержащем все файлы из каталога `debian`. Новый формат позволяет сопровождающему включать двоичные файлы, такие как значки в формате PNG, без дополнительных ухищрений [7].

Когда программа **dpkg-source** распаковывает пакет с исходным кодом в формате `3.0 (quilt)`, она автоматически накладывает все заплаты, перечисленные в файле `debian/patches/series`. Вы можете избежать наложения заплат в конце распаковки, указав параметр `--skip-patches`.

5.23 Файл `source/local-options`

При ведении работы по пакетированию Debian в VCS, как правило, создаётся две ветви: одна (например, `upstream`) для отслеживания исходного кода программы и другая (обычно, `master` в Git) для отслеживания пакета Debian. В последней, обычно, хочется иметь неизменённый исходный код программы и свои файлы `debian/*` для пакетирования Debian, чтобы легко производить слияние с новой версией авторского исходного кода.

После того, как вы собрали пакет, исходный код, обычно, остаётся изменённым. Вам необходимо вручную убрать изменения, внесённые заплатами, с помощью запуска `dquilt pop -a` перед сохранением в ветку `master`. Вы можете автоматизировать это, добавив дополнительный файл `debian/source/local-options`, содержащий `unapply-patches`. Этот файл не включается в генерируемый пакет с исходным кодом и изменяет только поведение локальной сборки. Также, он может содержать `abort-on-upstream-changes` (смотрите dpkg-source(1)).

```
unapply-patches
abort-on-upstream-changes
```

[6] Обзор перехода на новые форматы пакетов `3.0 (quilt)` и `3.0 (native)` с исходным кодом смотрите в DebSrc3.0 (http://wiki.debian.org/Projects/DebSrc3.0) .

[7] Фактически, новый формат также позволяет использовать несколько tar-архивов исходной программы и дополнительные методы сжатия. Описание этого выходит за рамки данного документа.

5.24 Файл `source/options`

Автоматически генерируемые файлы в дереве исходного кода могут немного раздражать при пакетировании, так как из-за них генерируются бесполезные большие файлы заплат. Для решения этой проблемы есть специальные модули, такие как **dh_autoreconf**, описанные в Раздел 4.4.3.

При создании пакета с исходным кодом вы можете указать регулярное выражение Perl в параметре `--extend-diff-ignore` для dpkg-source(1), чтобы игнорировать изменения в автоматически генерируемых файлах.

Общим решением этой проблемы с автоматически создаваемыми файлами является сохранение аргумента **dpkg-source** в файле пакета с исходным кодом `source/options`. Благодаря следующим настройкам в файлы заплат не будут включены `config.sub`, `config.guess` и `Makefile`.

```
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

5.25 Файлы `patches/*`

При использовании старого формата пакета с исходным кодом `1.0` создавался один большой файл `diff.gz`, который содержал файлы сопровождения в `debian` и заплаты к исходному коду. Такой пакет немного громоздок для проверки и понимания каждого изменения дерева исходного кода. Это не так уж хорошо.

В новом формате `3.0 (quilt)` заплаты хранятся в файлах `debian/patches/*`, для создания которых применяется команда **quilt**. Эти заплаты и другие данные пакета —всё содержимое каталога `debian` —упаковывается в файл `debian.tar.gz`. Так как команда **dpkg-source** может работать с данными **quilt** в формате `3.0 (quilt)` без пакета `quilt`, то его не нужно включать в зависимости `Build-Depends` [8].

Работа с командой **quilt** описана в quilt(1). Она записывает изменения исходного кода в виде набора заплат `-p1` в каталоге `debian/patches`, а дерево исходного кода вне каталога `debian` остаётся нетронутым. Порядок применения этих заплат указывается в файле `debian/patches/series`. Вы можете легко наложить (=push), откатить (=pop) или обновить заплаты [9].

В Глава 3 мы создали 3 заплаты в `debian/patches`.

Поскольку заплаты Debian расположены в `debian/patches`, убедитесь, что программа **dquilt** настроена правильно, как было описано в Раздел 3.1.

Позднее, когда кто-то (включая вас самих) предоставляет заплату *`foo`*`.patch` к исходному коду, изменить пакет исходного кода `3.0 (quilt)` очень просто:

```
$ dpkg-source -x gentoo_0.9.12.dsc
$ cd gentoo-0.9.12
$ dquilt import ../foo.patch
$ dquilt push
$ dquilt refresh
$ dquilt header -e
... описание заплаты
```

Заплаты, хранимые в новом формате исходного кода `3.0 (quilt)`, должны быть *без ненужного мусора*. Вы должны убедиться в этом, запустив: `dquilt pop -a; while dquilt push; do dquilt refresh; done`.

[8] Для управления набором заплат в пакетировании Debian были предложены и применяется несколько методов. Система **quilt** —предпочтительная система сопровождения. Также есть **dpatch**, **dbs**, **cdbs** и другие. Многие из них хранят заплаты в файлах `debian/patches/*`.

[9] Если вы просите поручителя загрузить ваш пакет, такое чёткое разделение и документирование ваших изменений очень важно для ускорения проверки пакета поручителем.

Глава 6

Сборка пакета

Теперь мы готовы собрать пакет.

6.1 Полная (пере)сборка

Для полной (пере)сборки пакета, необходимо убедиться в установке следующего:

- пакета `build-essential`,
- пакетов, перечисленных в поле `Build-Depends` (смотрите Раздел 4.1) и
- пакетов, перечисленных в поле `Build-Depends-indep` (смотрите Раздел 4.1).

Затем выполните следующую команду в каталоге с исходным кодом:

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

Она сделает всё, что необходимо для создания всех двоичных пакетов и пакета с исходным кодом. Она выполняет:

- очистку дерева исходного кода (`debian/rules clean`)
- сборку пакета с исходным кодом (`dpkg-source -b`)
- сборку программы (`debian/rules build`)
- сборку двоичных пакетов (`fakeroot debian/rules binary`)
- создаёт файл `.dsc`
- создаёт файл `.changes` с помощью **dpkg-genchanges**

Если результат сборки устраивает, подпишите файл `.dsc` и `.changes` вашим закрытым ключом GPG с помощью команды **debsign**. Вам потребуется ввести секретную фразу дважды. [1]

Для неродного пакета Debian, например, `gentoo`, после сборки пакетов вы увидите следующие файлы в родительском каталоге (`~/gentoo`):

[1] Для подключения к доверенному веб (web of trust) этот ключ GPG должен быть подписан разработчиком Debian и должен быть зарегистрирован в связке ключей Debian (http://keyring.debian.org) . Это позволяет закачивать пакеты для приёма в архивы Debian. Смотрите Создание нового ключа GPG (http://keyring.debian.org/creating-key.html) и вики Debian по подписыванию ключами (http://wiki.debian.org/Keysigning) .

- `gentoo_0.9.12.orig.tar.gz`

  Это оригинальный архив tar с исходным кодом, только переименованный для соответствия стандартам Debian. Обратите внимание, что первоначально он был создан с помощью `dh_make -f ../gentoo-0.9.12.tar.gz`.

- `gentoo_0.9.12-1.dsc`

  Этот файл содержит информацию о содержимом исходного кода. Он файл генерируется из вашего файла `control` и используется при распаковке пакета с исходным кодом с помощью dpkg-source(1).

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

  В этом сжатом архиве tar находится содержимое вашего каталога `debian`. Все дополнения, вносимые вами в исходный код, хранятся в виде заплат **quilt** в каталоге `debian/patches`.

  Если кто-либо захочет пересоздать ваш пакет с нуля, они легко смогут сделать это, используя перечисленные выше три файла. Процедура извлечения тривиальна: просто скопируйте куда-нибудь три файла и выполните `dpkg-source -x gentoo_0.9.12-1.dsc` [2].

- `gentoo_0.9.12-1_i386.deb`

  Это ваш собранный двоичный пакет. Вы можете использовать **dpkg** для его установки и удаления, как любой другой пакет.

- `gentoo_0.9.12-1_i386.changes`

  В этом файле описываются все изменения, сделанные в данной редакции пакета, и он используется программами поддержки FTP-архива Debian для внесения пакетов (как двоичных, так и с исходным кодом) в архив. Он частично генерируется из файлов `changelog` и `.dsc`.

  По мере того, как вы будете работать над пакетом, будет меняться его поведение и будет добавляться новая функциональность. Люди, загружающие ваш пакет, могут посмотреть этот файл и сразу понять, что было изменено. Программы поддержки архива Debian будут также отправлять содержимое этого файла в список рассылки debian-devel-changes@lists.debian.org (http://lists.debian.org/debian-devel-changes/) .

Файлы `gentoo_0.9.12-1.dsc` и `gentoo_0.9.12-1_i386.changes` должны быть подписаны с помощью команды **debsign** вашим закрытым ключом GPG из каталога `~/.gnupg/` до отправки их в архив Debian FTP. С помощью вашего открытого ключа GPG можно проверить, что эти файлы подписали действительно вы.

Команде **debsign** можно указать ID секретного ключа GPG, которым нужно подписывать (удобно для спонсирования) в `~/.devscripts`:

```
DEBSIGN_KEYID=ID_ключа_GPG
```

Длинные строки цифр в файлах `.dsc` и `.changes` —это контрольные суммы SHA1/SHA256 упомянутых файлов. Человек, загружающий ваши файлы, может проверить их с помощью sha1sum(1) или sha256sum(1) и, если числа не совпадают, он поймёт, что файл был повреждён или подделан.

6.2 Autobuilder

Debian поддерживает много переносов (http://www.debian.org/ports/) с помощью сети autobuilder (http://www.debian.org/-devel/buildd/) , в которой работает служба **buildd** на компьютерах различных архитектур. Хотя вам и не придётся этим заниматься, но всё же следует узнать, что будет происходить с вашими пакетами. Рассмотрим в общих чертах как ваш пакет пересобирается для различных архитектур [3].

Для пакетов с `Architecture: any` система autobuilder выполняет пересборку. Она убеждается, что установлен

- пакет `build-essential` и

[2] Вы можете избежать наложения заплат **quilt** в формате исходного кода `3.0 (quilt)` в конце извлечения, указав параметр `--skip-patches`. Или же вы можете выполнить `quilt pop -a` после обычной распаковки.

[3] На самом деле, работа системы autobuilder гораздо сложнее, однако её детальное описание не для этого документа.

- пакеты, перечисленные в поле `Build-Depends` (смотрите Раздел 4.1).

Затем в каталоге с исходным кодом вызывается команда

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

Она делает всё необходимое для сборки пакетов для данной архитектуры. А именно:

- очистку дерева исходного кода (`debian/rules clean`)
- сборку программы (`debian/rules build`)
- сборку архитектурно-зависимых двоичных пакетов (`fakeroot debian/rules binary-arch`)
- подпись файла исходного кода `.dsc` с помощью **gpg**
- создание и подпись загружаемого файла `.changes` с помощью **dpkg-genchanges** и **gpg**

Так ваш пакет собирается для различных архитектур.

Хотя пакеты, перечисленные в поле `Build-Depends-indep`, должны быть установлены при ручном пакетировании (смотрите Раздел 6.1), их не нужно устанавливать при использовании системы autobuilder, так как она занимается исключительно архитектурно-зависимыми двоичными пакетами [4]. Это различие между обычным пакетированием и использованием autobuilder определяет, где должны быть перечислены необходимые пакеты: в поле `Build-Depends` или `Build-Depends-indep` файла `debian/control` (смотрите Раздел 4.1).

6.3 Команда debuild

Для автоматизации действий по сборки с помощью **dpkg-buildpackage** можно использовать команду **debuild**. Смотрите debuild(1).

Команда **debuild** запускает команду **lintian** для выполнения статической проверки после сборки пакета Debian. Команда **lintian** можно настроить через файл `~/.devscripts`:

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

Для очистки каталога исходного кода и пересборки пакета выполните:

```
$ debuild
```

Для очистки древа исходного кода выполните:

```
$ debuild clean
```

6.4 Пакет `pbuilder`

В чистой среде сборки (**chroot**) для проверки пакетных зависимостей очень полезен пакет `pbuilder` [5]. Он поможет убедиться в чистоте сборки под управлением auto-builder из `sid` для различных архитектур и избежать опасной ошибки FTBFS (ошибка сборки из исходного кода), которая всегда относится к категории RC (критична для данного выпуска) [6].

Давайте настроим пакет `pbuilder` следующим образом:

[4] В отличие от пакета `pbuilder`,окружение **chroot** из пакета `sbuild`, который используется системой autobuilder, не требует минимальной системы и допускает наличие установленных пакетов, не являющихся необходимыми.

[5] Так как пакет `pbuilder` всё ещё развивается, за подробностями настройки обратитесь к его официальной документации.

[6] Более полная информация по автоматической сборке пакетов Debian приведена в http://buildd.debian.org/.

- дадим пользователю право записи в каталог `/var/cache/pbuilder/result`
- создадим каталог для размещения сценариев, например, */var/cache/pbuilder/hooks*, доступный пользователю для записи.
- добавим в файл `~/.pbuilderrc` или в `/etc/pbuilderrc` следующие строки:

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

Для начальной инициализации локальной системы **chroot** пакета `pbuilder` выполним:

```
$ sudo pbuilder create
```

Если у вас имеется готовый пакет исходного кода, то в каталоге, где расположены файлы *foo*`.orig.tar.gz`, *foo*`.debian.tar.gz` и *foo*`.dsc`, для обновления локальной системы **chroot** пакета `pbuilder` и сборки соответствующих двоичных пакетов выполните следующие команды:

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

Только что собранные пакеты без подписи GPG будут помещены в каталог `/var/cache/pbuilder/result/`, их владельцем будет обычный пользователь (не суперпользователь).

Подписи GPG для файлов `.dsc` и `.changes` можно сгенерировать следующим образом:

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

Если у вас есть обновлённое древо исходного кода, но нет собранных пакетов с исходным кодом, выполните в каталоге, где размещён подкаталог `debian`, другие команды:

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

Вы можете войти внутрь окружения **chroot** и настроить его командой `pbuilder --login --save-after-login`. Изменения будут сохранены, когда вы покинете оболочку, нажав `^D` (Control-D).

Последняя версия команды **lintian** может быть вызвана в окружении `chroot`, используя сценарий */var/cache/pbuilder/hooks/*`/B90lintian`, настроенный следующим образом [7]:

```
#!/bin/sh
set -e
install_packages() {
        apt-get -y --force-yes 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 +++"
```

Для правильной сборки пакета для `sid` вам потребуется самое свежее окружение этой ветви, однако миграция на `sid` всей системы может быть нежелательна. Пакет `pbuilder` поможет вам в этой ситуации.

Возможно, вам потребуется обновить ваши пакеты из ветви `stable` их версиями из архивов `stable-proposed-updates`, `stable/updates` и других [8]. В таких случаях утверждения «Я использовал окружение `sid`» недостаточно, если обновление не было проведено своевременно. Пакет `pbuilder` поможет вам получить доступ к окружению практически любого производного от Debian дистрибутива той же архитектуры.

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

[7] Здесь предполагается, что `HOOKDIR=/var/cache/pbuilder/hooks`. Примеры вызываемых (hook) сценариев можно найти в каталоге `/usr/share/doc/pbuilder/examples`.

[8] Есть некоторые ограничения для такого обновления пакета из `stable`.

6.5 Команда git-buildpackage и подобные ей

Если автор программы применяет систему управления кодом [9], то и вы можете использовать её возможности. Это упрощает слияние и выборку заплат для исходного кода. Для каждой VCS существуют специализированные инструменты, позволяющие производить сборку пакетов Debian:

- `git-buildpackage`: инструменты для пакетирования в репозиториях Git.
- `svn-buildpackage`: вспомогательные программы для сопровождения пакетов Debian в Subversion.
- `cvs-buildpackage`: набор сценариев для пакетирования в CVS.

Среди разработчиков Debian становится популярным использовать `git-buildpackage` для управления пакетами Debian на сервере Git alioth.debian.org (http://alioth.debian.org/) . [10] В этот пакет включено много команд для *автоматизации* процесса упаковки:

- git-import-dsc(1): импорт существующего пакета Debian в репозитории Git.
- git-import-orig(1): импорт нового tar с исходным кодом в репозиторий Git.
- git-dch(1): генерация changelog Debian на основе сообщений о записи в Git.
- git-buildpackage(1): сборка пакетов Debian из репозитория Git.
- git-pbuilder(1): сборка пакетов Debian из репозитория Git с помощью **pbuilder**/**cowbuilder**.

Для ведения пакетирования в этих командах используются 3 ветви:

- `main` —дерево родных пакетов Debian.
- `upstream` —дерево авторского исходного кода.
- `pristine-tar` для авторского tar, создаваемый при использовании параметра `--pristine-tar`.[11]

Для настройки `git-buildpackage` используйте файл `~/.gbp.conf`. Смотрите gbp.conf(5). [12]

6.6 Быстрая пересборка

При работе над большим пакетом, возможно, вы не захотите каждый раз полностью пересобирать его из исходного кода при изменении настроек в файле `debian/rules`. Исключительно для тестовых целей вы можете создать deb-файл без пересборки кода следующим образом [13]:

[9] Смотрите Системы управления версиями (http://www.debian.org/doc/manuals/debian-reference/ch10#_version_control_systems) .

[10] В вики Debian Alioth (http://wiki.debian.org/Alioth) описано как использовать службу alioth.debian.org (http://alioth.debian.org/) .

[11] При указании параметра `--pristine-tar` вызывается команда **pristine-tar**, которая может регенерировать точную копию первоначального авторского файла tar на основе только маленького двоичного файла различий и содержимого tar, который обычно хранится в ветви `upstream` в VCS.

[12] Несколько веб-ресурсов для опытных пользователей:

- Сборка пакетов Debian с помощью git-buildpackage (`/usr/share/doc/git-buildpackage/manual-html/gbp.html`)
- Пакеты debian в git (https://honk.sigxcpu.org/piki/development/debian_packages_in_git/)
- Использование Git для пакетирования Debian (http://www.eyrie.org/~eagle/notes/debian/git.html)
- git-dpm: пакеты Debian в Git (http://git-dpm.alioth.debian.org/)
- Использование TopGit для генерации заплат quilt при пакетировании Debian (http://git.debian.org/?p=collab-maint/topgit.git;a=blob_plain;f=debian/-HOWTO-tg2quilt;hb=HEAD)

[13] В данном случае не выполняется настройка переменных окружения, как это происходит при обычной сборке. Никогда не загружайте в архив пакеты, собранные таким **быстрым** способом.

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

Или просто выполните для выяснения возможна ли сборка пакета:

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

После выбора оптимальных настроек не забудьте пересобрать пакет стандартными средствами. Полученные таким способом файлы `.deb` могут быть загружены в архив некорректно, если вы попытаетесь это сделать.

6.7 Иерархия команд

Вот краткая сводка команд, которые объединяются в единую систему для сборки пакетов. Есть много способов сделать одно и тоже.

- `debian/rules` = сценарий сопровождающего для сборки пакета
- **dpkg-buildpackage** = ключевой инструмент сборки пакета
- **debuild** = **dpkg-buildpackage** + **lintian** (сборка при проверенных переменных окружение)
- **pbuilder** = ключевой инструмент chroot-окружения Debian
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (сборка в chroot)
- **cowbuilder** = ускорение выполнения **pbuilder**
- **git-pbuilder** = простой синтаксис командной строки для **pdebuild** (используется в **gbp buildpackge**)
- **gbp** = управление исходным кодом Debian в репозитории git
- **gbp buildpackge** = **pbuilder** + **dpkg-buildpackage** + **gbp**

Хотя высокоуровневые команды, такие как **gbp buildpackge** и **pbuilder**, предоставляют замечательную среду сборки пакета, необходимо понимать какие нижележащие команды, такие как `debian/rules` и **dpkg-buildpackage**, в них используются.

Глава 7

Проверка пакета на наличие ошибок

Есть несколько стандартных процедур для самостоятельной проверки пакета на наличие ошибок перед его загрузкой в публичный архив, которые вам следует знать.

Лучше проверять пакет на другой машине (не на той, на которой он собирался). Обращайте пристальное внимание на предупреждения и сообщения об ошибках, получаемые в результате описываемых тестов.

7.1 Подозрительные изменения

Если вы обнаружите новые автоматически сгенерированные файлы заплат `debian-changes-*` в каталоге `debian/patches` после сборки своего неродного пакета Debian в формате `3.0 (quilt)`, то, вероятнее всего, вы неумышленно изменили какие-то файлы или это сделал авторский сценарий сборки. Если это ваша ошибка, исправьте её. Если это сценарий, то исправьте источник ошибки с помощью **dh-autoreconf**, как это описано в Раздел 4.4.3, или обойдите её с помощью `source/options`, который описан в Раздел 5.24.

7.2 Проверка установки пакета

Вы должны убедиться, что пакет устанавливается. Команда debi(1) поможет протестировать установку всех сгенерированных двоичных пакетов.

```
$ sudo debi gentoo_0.9.12-1_i386.changes
```

Используя взятый из архива Debian файл `Contents-`*`i386`*, убедитесь, что в собранном пакете нет файлов, которые существуют в других пакетах, что может приводить к проблемам установки. Для этой задачи может пригодиться команда **apt-file**. Если совпадения есть, то решите эту проблему либо переименовав файл в своём пакете, либо выделив общий файл в отдельный пакет, от которого будут зависеть конфликтующие пакеты, или воспользуйтесь механизмом альтернатив (смотрите update-alternatives(1)) совместно с другими сопровождающими других пакетов, либо объявите параметр `Conflicts` с нужным значением в файле `debian/control`.

7.3 Проверка сценариев сопровождающего пакета

Все сценарии сопровождающего (`preinst`, `prerm`, `postinst` и `postrm`) сложны в написании, если только для их автоматической генерации не применялись программы из пакета `debhelper`. Поэтому не пользуйтесь этими сценариями, если вы начинающий сопровождающий (смотрите Раздел 5.18).

Если ваш пакет использует эти нетривиальные сценарии сопровождающего, убедитесь, что не только установка, но и удаление, вычистка и обновление пакета также проходят успешно. Многие ошибки в таких сценариях проявляются при удалении или вычистке. Для проверки используйте команду **dpkg**:

```
$ sudo dpkg -r gentoo
$ sudo dpkg -P gentoo
$ sudo dpkg -i gentoo_версия-редакция_i386.deb
```

Следует выполнить следующие шаги:

- установите предыдущую версию (если необходимо)
- обновите пакет с предыдущей версии
- откатитесь на предыдущую версию (по желанию)
- вычистите пакет
- установите новый пакет
- удалите его
- установите опять
- вычистите пакет

Если это ваш первый пакет, то для тестирования вам понадобятся ещё пакеты-пустышки различных версий.

Не забудьте проверить наличие в Debian предыдущей версии программы, которую вы пакетируете. В этом случае пользователи, у которых установлена предыдущая версия, могут захотеть обновить пакет и вам следует убедиться в отсутствии проблем при таком обновлении. Также протестируйте обновления и с этой версии.

Хотя откат к предыдущей версии официально не поддерживается, будет здорово обеспечить такую возможность.

7.4 Использование `lintian`

Запустите lintian(1), передав ей параметром файл `.changes`. Команда **lintian** выполняет множество тестовых сценариев, проверяющих наличие типичных ошибок пакетирования [1].

```
$ lintian -i -I --show-overrides gentoo_0.9.12-1_i386.changes
```

Разумеется, следует заменить имя файла `.changes` на то, которое было сгенерировано для вашего пакета. В результатах команды **lintian** используются следующие метки:

- `E:` —ошибка; нарушение политики или ошибка пакетирования.
- `W:` —предупреждение; возможное нарушение политики или ошибка пакетирования.
- `I:` —для информации; сведения о некоторых аспектах пакетирования.
- `N:` —замечание; уточнение, помогающее при отладке.
- `O:` —скрытые сообщения; информация, скрываемая на основе файла `lintian-overrides`, но показываемая при указании параметра `--show-overrides`.

Если вы видите предупреждения —исправьте пакет, чтобы их не было или убедитесь, что это нормально. Если предупреждения излишни —настройте файл `lintian-overrides`, как описано в Раздел 5.14.

Заметим, что команда debuild(1) или pdebuild(1) позволяет собрать пакет с помощью **dpkg-buildpackage** и сразу проверить его **lintian**.

[1] Вам не потребуется указывать параметр **lintian** `-i -I --show-overrides`, если вы настроили файл настройки `/etc/devscripts.conf` или `~/.devscripts`, как это было описано в Раздел 6.3.

7.5 Команда debc

Вы можете просмотреть список файлов в двоичном пакете Debian с помощью команды debc(1).

```
$ debc пакет.changes
```

7.6 Команда debdiff

Вы можете сравнить содержимое файлов двух пакетов исходного кода Debian с помощью команды debdiff(1).

```
$ debdiff старый-пакет.dsc новый-пакет.dsc
```

Также с помощью команды debdiff(1) вы можете сравнить списки файлов двух двоичных пакетов Debian.

```
$ debdiff старый-пакет.changes новый-пакет.changes
```

Это полезно для определения того, что изменилось в пакетах исходного кода, и что не произошло никаких непреднамеренных изменений при обновлении двоичных пакетов, например неправильного перемещения или удаления файлов.

7.7 Команда interdiff

Команда interdiff(1) позволяет сравнить два файла `diff.gz`. Это полезно для проверки отсутствия сделанных сопровождающим нечаянных правок исходного кода при обновлении пакетов в старом формате `1.0`.

```
$ interdiff -z старый-пакет.diff.gz новый-пакет.diff.gz
```

В новой версии формата пакетов с исходным кодом `3.0` изменения хранятся в нескольких файлах заплат (описано в Раздел 5.25). Вы можете отследить изменения каждого файла `debian/patches/*` также с помощью **interdiff**.

7.8 Команда mc

Большинство этих файловых проверок могут быть сделаны интуитивно понятным образом с помощью файлового менеджера типа mc(1), который позволяет просматривать содержимое не только файлов пакетов `*.deb`, но и таких файлов как `*.udeb`, `*.debian.tar.gz`, `*.diff.gz` и `*.orig.tar.gz`.

Внимательно следите за лишними ненужными файлами или файлами нулевой длины как в двоичном пакете, так и в пакете с исходным кодом. Зачастую, мусор не вычищается должным образом; подправьте ваш файл `rules`, чтобы исправить это.

Глава 8

Обновление пакета

Вскоре после выпуска пакета, вам понадобится его обновить.

8.1 Новая редакция Debian

Предположим, что в вашем пакете нашли ошибку (номер `#654321`), и описываемую там проблему вы можете решить. Для того, чтобы создать новую редакцию пакета, нужно:

- Если исправление должно быть записано в виде новой заплаты, сделайте следующее:
  - запустите `dquilt new` *`название-ошибки.patch`* для присвоения имени заплате;
  - запустите `dquilt add` *`файл-с-ошибкой`* для объявления файла, который должен быть изменён;
  - исправьте ошибку в пакете исходного кода;
  - запустите `dquilt refresh` для записи исправления в файл *`название-ошибки.patch`*;
  - запустите `dquilt header -e` для добавления её описания;
- Если для исправления требуется обновление существующей заплаты, сделайте следующее:
  - запустите `dquilt pop` *`foo.patch`* для того, чтобы откатить наложенную заплату *`foo.patch`*;
  - исправьте проблему в старой заплате *`foo.patch`*;
  - запустите `dquilt refresh` для обновления заплаты *`foo.patch`*;
  - запустите `dquilt header -e` для обновления её описания;
  - запустите `while dquilt push; do dquilt refresh; done` для применения всех заплат при удалении *шероховатостей*;
- Добавьте новую редакцию в начало файла Debian `changelog`, например, с помощью `dch -i` или вручную с помощью `dch -v` *`версия-редакция`*, а затем добавьте комментарии с помощью текстового редактора [1].
- Включите краткое описание ошибки и её решение в список изменений (changelog), сопроводив текстом `Closes: #654321`. Это позволит *автомагически* закрыть сообщение об ошибке с помощью программного обеспечения обслуживания архива в тот момент, когда ваш пакет будет принят в архив Debian.
- Повторите то, что делали выше, для исправления других ошибок, обновляя файл Debian `changelog` с помощью `dch` по мере надобности.
- Повторите всё из Раздел 6.1 и Глава 7.

[1] Дату в нужном формате можно получить с помощью команды `LANG=C date -R`.

- После проверки правильности, измените в `changelog` имя выпуска с `UNRELEASED` на значение целевого дистрибутива `unstable` (или даже на `experimental`). [2]
- Закачайте пакет, следуя Глава 9. На этот раз разница в том, что не будет включён оригинальный архив исходного кода, поскольку он не изменён и уже присутствует в архиве Debian.

Стоит упомянуть одну хитрость на случай, когда вы делаете локальный пакет для эксперимента с пакетированием и не отправляете эту версию в официальный архив, например, *`1.0.1-1`*. Для плавного обновления рекомендуется создать запись в `changelog` со строкой версии вида *`1.0.1-1~rc1`*. Вы можете не перегружать `changelog` записями о локальных изменениях, объединяя их в одну для официального пакета. Об упорядочивании версий строк смотрите Раздел 2.6.

8.2 Изучение нового авторского выпуска

При подготовке пакетов нового авторского выпуска для архива Debian, вы должны сперва проверить новый авторский выпуск.

Начните с чтения файлов `changelog`, `NEWS` и всей остальной документации, которая может поставляться с новой версией.

Потом проверьте изменения между старым и новым исходным кодом программы, как описано ниже, чтобы найти что-нибудь подозрительное:

```
$ diff -urN foo-старая-версия foo-новая-версия
```

На изменения в некоторых автоматически сгенерированных файлах Autotools, таких как `missing`, `aclocal.m4`, `config.guess`, `config.h.in`, `config.sub`, `configure`, `depcomp`, `install-sh`, `ltmain.sh` и `Makefile.in`, можно не обращать внимания. Вы можете удалить их перед запуском **diff** для проверки исходного кода.

8.3 Новый авторский выпуск

Если пакет *`foo`* правильно собран в новом формате `3.0 (native)` или `3.0 (quilt)`, то для подготовки пакета новой версии программы достаточно переместить старый каталог `debian` в новый исходный код. Это можно сделать запуском `tar xvzf /путь/к/foo_старая-версия.debian.tar.gz` в каталоге с новым исходным кодом [3]. Конечно, потребуется сделать несколько очевидных рутинных операций:

- Скопируйте авторский исходный код в файл `foo_`*`новая-версия`*`.orig.tar.gz`.
- Обновите файл Debian `changelog` с помощью `dch -v` *`новая-версия-1`*.
  - Добавьте пометку `New upstream release`.
  - Лаконично опишите изменения *в новом авторском выпуске*, которые исправляют найденные ошибки, и закройте эти ошибки, добавляя `Closes:` *`#номер_ошибки`*.
  - Лаконично опишите изменения *в новом авторском выпуске*, сделанные сопровождающим, которые исправляют найденные ошибки, и закройте эти ошибки, добавляя `Closes:` *`#номер_ошибки`*.
- запустите `while dquilt push; do qduilt refresh; done` для применения всех заплат при удалении *шероховатостей*.

Если наложение/слияние произошло с ошибками, изучите ситуацию (сведения есть в файлах `.rej`).

- Если применяемая заплата к исходному коду была интегрирована в авторский исходный код, то

[2] Если для выполнения изменения вы используете команду `dch -r`, то убедитесь, что записали файл `changelog` именно редактором.

[3] Если пакет *`foo`* собран в старом формате `1.0`, то вместо этого можно запустить `zcat /путь/к/foo_старая-версия.diff.gz|patch -p1` в каталоге с новым исходным кодом.

  - выполните `dquilt delete` для её удаления.

- Если применяемая заплата к исходному коду конфликтует с новыми изменениями в авторском исходном коде, то
  - выполните `dquilt push -f` для наложения старых заплат с отбрасыванием конфликтующих *`baz`*`.rej`.
  - Исправьте файл *`baz`* ручным копированием нужных строки из *`baz`*`.rej`.
  - запустите `dquilt refresh` для обновления заплаты.
- Продолжайте, как обычно, командой `while dquilt push; do dquilt refresh; done`.

Это может быть автоматизировано с помощью команды uupdate(1):

```
$ apt-get source foo
...
dpkg-source: info: extracting foo in foo-старая-версия
dpkg-source: info: unpacking foo_старая-версия.orig.tar.gz
dpkg-source: info: applying foo_старая-версия-1.debian.tar.gz
$ ls -F
foo-старая-версия/
foo_старая-версия-1.debian.tar.gz
foo_старая-версия-1.dsc
foo_старая-версия.orig.tar.gz
$ wget http://example.org/foo/foo-новая-версия.tar.gz
$ cd foo-старая-версия
$ uupdate -v новая-версия ../foo-новая-версия.tar.gz
$ cd ../foo-новая-версия
$ while dquilt push; do dquilt refresh; done
$ dch
... описание проведённых изменений
```

Если вы настроили файл `debian/watch` по описанию из Раздел 5.21, то можете пропустить команду **wget**. Просто запустите uscan(1) в каталоге *`foo-старая-версия`* вместо команды **uupdate**. Она *автомагически* найдёт обновления исходного кода, скачает его и запустит команду **uupdate** [4].

Вы можете выпустить этот обновлённый исходный код, повторив то, что делали в Раздел 6.1, Глава 7 и Глава 9.

8.4 Обновление стиля пакетирования

При обновлении пакета обновлять стиль пакетирования необязательно. Но сделав это, вы сможете полностью использовать возможности современной системы `debhelper` и формата исходного кода `3.0` [5].

- Если по какой-то причине требуется пересоздать удалённые шаблоны файлов, вы можете ещё раз запустить **dh_make** с параметром `--addmissing` в том же дереве исходного кода пакета Debian, а затем отредактировать их должным образом.
- Если в пакете файл `debian/rules` не переписан с использованием команды **dh** из пакета `debhelper` v7+, то сделайте это. Обновите файл `debian/control` соответствующим образом.
- Если вы хотите переписать файл `rules` с использованием **dh**, в котором сейчас используется механизм включения `Makefile` из Common Debian Build System (`cdbs`), то для понимания его переменных настройки `DEB_*` смотрите следующие документы:
  - локальная копия `/usr/share/doc/cdbs/cdbs-doc.pdf.gz`

[4] Если команда **uscan** скачает обновлённый исходный код, но не запустит команду **uupdate**, исправьте файл `debian/watch` таким образом, чтобы упоминание `debian uupdate` было в конце URL.

[5] Не стоит беспокоиться или спорить, если ваш поручитель или другие сопровождающие возражают против обновления существующего стиля пакетирования. Есть более важные вещи.

  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)

- Если у вас есть пакет исходного кода формата `1.0` без файла *`foo`*`.diff.gz`, вы можете обновить его до нового формата исходного кода `3.0 (native)`, создав файл `debian/source/format` с содержимым `3.0 (native)`. Остальные файлы `debian/*` могут быть просто скопированы.
- Если у вас есть пакет с исходным кодом формата `1.0` с файлом *`foo`*`.diff.gz`, вы можете обновить его до нового формата исходного кода `3.0 (quilt)`, создав файл `debian/source/format` с содержимым `3.0 (quilt)`. Остальные файлы `debian/*` могут быть просто скопированы. Если нужно, импортируйте файл `big.diff`, полученный командой `filterdiff -z -x '*/debian/*' `*`foo`*`.diff.gz > big.diff`, в вашу систему **quilt** [6].
- Если в пакете используется другая система заплат, например, `dpatch`, `dbs` или `cdbs` с параметром `-p0`, `-p1` или `-p2`, перейдите на `quilt`, используя `deb3`, как описано в http://bugs.debian.org/581186.
- Если пакет был собран командой **dh** с параметром `--with quilt` или командами **dh_quilt_patch** и **dh_quilt_unpatch**, уберите их и перейдите на использование нового формата пакетов исходного кода `3.0 (quilt)`.

Проверьте DEP - Debian Enhancement Proposals (http://dep.debian.net/) и учтите ПРИНЯТЫЕ предложения.

Также вам нужно выполнить остальные задачи, описанные в Раздел 8.3.

8.5 Преобразование в UTF-8

Если авторские документы поставляются в старых кодировках, лучше преобразовать их в UTF-8.

- Для перекодирования простых файлов используйте iconv(1).

```
iconv -f latin1 -t utf8 foo_in.txt > foo_out.txt
```

- Для перекодирования файлов HTML в простой текст в кодировке UTF-8 используйте w3m(1). Выполнение данной операции должно проводиться при включённой локали UTF-8.

```
LC_ALL=en_US.UTF-8 w3m -o display_charset=UTF-8 \
        -cols 70 -dump -no-graph -T text/html \
        < foo_in.html > foo_out.txt
```

8.6 Замечания по обновлению пакетов

Вот несколько замечаний по обновлению пакетов:

- Не удаляйте старые записи из `changelog` (на первый взгляд это очевидно, но были случаи случайного набора `dch` вместо `dch -i`).
- Существующие изменения Debian должны быть пересмотрены; выбросьте инструментарий, который включил автор (в той или иной форме) и не забудьте оставить инструментарий, который не был включён автором, пока не появится убедительной причины этого не делать.
- Если в систему для сборки были внесены изменения (к счастью, вы узнаете об этом при изучении авторских изменений), то при необходимости обновите сборочные зависимости в файлах `debian/rules` и `debian/control`.
- Проверьте систему отслеживания ошибок (BTS) (http://www.debian.org/Bugs/) на случай, если кто-нибудь предоставил заплаты для исправления незакрытых ошибок.
- Проверьте содержимое файла `.changes` и убедитесь, что вы выполняете отправку в правильный дистрибутив, закрываемые ошибки перечислены в поле `Closes`, поля `Maintainer` и `Changed-By` совпадают, файл подписан GPG и т. д.

[6] Вы можете разделить файл `big.diff` на много маленьких приращиваемых заплат с помощью команды **splitdiff**.

Глава 9

Отправка пакета

Теперь, после тщательного тестирования вашего нового пакета, вы хотите отправить его в публичный архив для использования.

9.1 Отправка в архив Debian

После того, как вы станете официальным разработчиком [1], то сможете отправлять пакеты в архив Debian [2]. Вы можете делать это вручную, но легче воспользоваться существующими инструментами автоматизации, такими как dupload(1) или dput(1). Здесь будет рассказано как это сделать с помощью **dupload** [3].

Сначала, вам нужно настроить конфигурационный файл для **dupload**. Вы можете отредактировать системный файл `/etc/dupload.conf`, либо создать свой собственный файл `~/.dupload.conf`, указав те настройки, которые нужно изменить.

Описание каждого параметра приведено в справочной странице dupload.conf(5).

Параметр `$default_host` определяет, какая из очередей отправки будет использована по умолчанию. Первичной является `anonymous-ftp-master`, но возможно, что вы захотите использовать другую [4].

Соединившись с Интернетом, вы можете отправить свой пакет следующим образом:

```
$ dupload gentoo_0.9.12-1_i386.changes
```

Команда **dupload** проверяет, что контрольные суммы SHA1/SHA256 ваших файлов совпадают с указанным в файле `.changes`. Если они не совпадают, она предложит пересобрать пакет (о том, как это правильно делать, смотрите раздел Раздел 6.1).

Если при отправке в ftp://ftp.upload.debian.org/pub/UploadQueue/ возникли проблемы, то вы можете исправить их вручную загрузив туда файл `*.commands`, подписанный GPG, с помощью **ftp** [5]. Например, используя `hello.commands`:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Uploader: Foo Bar <Foo.Bar@example.org>
Commands:
```

[1] Смотрите Раздел 1.1.

[2] Существуют публично доступные архивы, например http://mentors.debian.net/, которые работают почти также как архив Debian и предоставляют зону для отправки людям, не имеющим статуса разработчика Debian. Вы можете создать свой архив с помощью инструментов, перечисленных в http://wiki.debian.org/HowToSetupADebianRepository. Поэтому данный раздел также будет полезен не только разработчикам Debian.

[3] Сейчас, вероятно, пакет `dput` имеет больше возможностей и становится более популярным, чем `dupload`. Для его настройки используется системный файл `/etc/dput` и пользовательский `~/.dput.cf`. Также он поддерживается службами Ubuntu без дополнительной настройки.

[4] Смотрите Справочник разработчика Debian, 5.6. «Отправка пакета» (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#upload) .

[5] Смотрите ftp://ftp.upload.debian.org/pub/UploadQueue/README. Или же вы можете использовать команду **dcut** из пакета `dput`.

```
 rm hello_1.0-1_i386.deb
 mv hello_1.0-1.dsx hello_1.0-1.dsc
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

[...]
-----END PGP SIGNATURE-----
```

9.2 Включение файла `orig.tar.gz` для отправки

При самой первой отправке пакета в архив, вам также потребуется добавить к нему файл с исходным кодом `orig.tar.gz`. Если номер редакции Debian для данной версии программы не равен `1` или `0`, то вам следует указать команде **dpkg-buildpackage** параметр `-sa`.

Для команды **dpkg-buildpackage**:

```
$ dpkg-buildpackage -sa
```

Для команды **debuild**:

```
$ debuild -sa
```

Для команды **pdebuild**:

```
$ pdebuild --debbuildopts -sa
```

Противоположный по действию параметр `-sd` позволит исключить файл с исходным кодом `orig.tar.gz`.

9.3 Пропущенные отправки

Если вы сделали много изменений в `debian/changelog`, но не выполняли отправку соответствующих версий, то должны создать соответствующий файл `*_.changes`, в который войдут все изменения с последней отправки. Это можно сделать указанием команде **dpkg-buildpackage** параметра `-v` с номером версии, например, *`1.2`*.

Для команды **dpkg-buildpackage**:

```
$ dpkg-buildpackage -v1.2
```

Для команды **debuild**:

```
$ debuild -v1.2
```

Для команды **pdebuild**:

```
$ pdebuild --debbuildopts "-v1.2"
```

Приложение A

Углублённое пакетирование

Далее приводятся подсказки и ссылки при сложных случаях пакетирования. Настоятельно рекомендуется прочитать все предлагаемые материалы.

Вам может потребоваться вручную отредактировать шаблонные файлы пакета, сгенерированные командой **dh_make**, чтобы подогнать их под темы, затронутые в этой главе. Новая команда **debmake** больше подходит к этим темам.

A.1 Общие библиотеки

Перед пакетированием общих библиотек прочтите следующие основные документы:

- руководство по политике Debian, раздел 8 «Общие библиотеки» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html)
- руководство по политике Debian, раздел 9.1.1 «Структура файловой системы (http://www.debian.org/doc/debian-policy/-ch-opersys.html#s-fhs)
- руководство по политике Debian, раздел 10.2 «Библиотеки» (http://www.debian.org/doc/debian-policy/ch-files.html#s-libraries)

Вот упрощённое представление, для начала:

- Общие библиотеки —это объектные файлы в формате ELF, в которых содержится скомпилированный код.
- Общие библиотеки распространяются в виде файлов `*.so` (не в файлах `*.a` или `*.la`).
- Главным образом, общие библиотеки нужны для совместного использования общего кода в исполняемых файлах посредством механизма **ld**.
- Иногда общие библиотеки используются в качестве подключаемых модулей исполняемых файлов посредством механизма **dlopen**.
- Общие библиотеки экспортируют символы —скомпилированные объекты: переменные, функции и классы; и разрешают к ним доступ из скомпонованных исполняемых файлов.
- SONAME общей библиотеки `lib`*`foo`*`.so.`*`1`*: `objdump -p lib`*`foo`*`.so.`*`1` `| grep SONAME` [1]
- SONAME общей библиотеки обычно совпадает с именем файла библиотеки (но не всегда).
- SONAME общих библиотек, которые скомпонованы с */usr/bin/foo*: `objdump -p` */usr/bin/foo* `| grep NEEDED` [2]

[1] Либо: `readelf -d lib`*`foo`*`.so.`*`1` `| grep SONAME`
[2] Либо: `readelf -d lib`*`foo`*`.so.`*`1` `| grep NEEDED`

- `lib`*`foo1`*: библиотечный пакет общей библиотеки `lib`*`foo`*`.so.`*`1`* с ABI-версией SONAME, равной *`1`*.[3]
- Пакетные сценарии сопровождающего для библиотеки должны вызывать **ldconfig** для создания необходимых символьных ссылок для SONAME при определённых условиях.[4]
- `lib`*`foo1`*`-dbg`: пакет с отладочными символами, который содержит символы для отладки пакета с общей библиотекой `lib`*`foo1`*.
- `lib`*`foo`*`-dev`: пакет для разработчика, который содержит заголовочные файлы и т .д. общей библиотек `lib`*`foo`*`.so.`*`1`*.[5]
- Обычно, в пакете Debian не должно быть файлов архива Libtool `*.la`.[6]
- Обычно, в пакете Debian не должен использоваться RPATH.[7]
- Несмотря на некоторое устаревание и статус вторичности, следующая ссылка тоже может быть полезна Debian Library Packaging Guide (http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html) .

A.2 Управление `debian/`*`пакет`*`.symbols`

Если пакет содержит общую библиотеку, то вы должны создать файл `debian/`*`пакет`*`.symbols`, в котором отражена минимальная версия каждого символа обратно совместимых изменений ABI с единым SONAME библиотеки и единым именем пакета с библиотекой.[8] Подробная информация приведена в следующих документах:

- Смотрите руководство по политике Debian, раздел 8.6.3 «Система символов» (http://www.debian.org/doc/debian-policy/-ch-sharedlibs.html#s-sharedlibs-symbols) [9]
- dh_makeshlibs(1)
- dpkg-gensymbols(1)
- dpkg-shlibdeps(1)
- deb-symbols(5)

Вот примерный план создания пакета `libfoo1` для авторской версии `1.3` с подходящим файлом `debian/libfoo1.symbols`:

- Подготовьте основу исходного дерева из авторского файла `libfoo-1.3.tar.gz`.
  - Если пакетирование `libfoo1` производится впервые, создайте пустой файл `debian/libfoo1.symbols`.
  - Если была упакована предыдущая авторская версия `1.2` в пакет `libfoo1` с соответствующим файлом `debian/libfoo1.symbols` в пакете с исходным кодом, то используйте его и сейчас.
  - Если предыдущая авторская версия `1.2` не была упакована с `debian/libfoo1.symbols`, создайте его как файл `symbols` из всех доступных двоичных пакетов с единым именем пакета общей библиотеки, содержащих одинаковый SONAME библиотеки, например версии `1.1-1` и `1.2-1`. [10]

[3] Смотрите руководство по политике Debian, раздел 8.1 «Общие библиотеки времени выполнения» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-runtime) .

[4] Смотрите руководство по политике Debian, раздел 8.1.1 «ldconfig» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-ldconfig) .

[5] Смотрите руководство по политике Debian, раздел 8.3 «Статические библиотеки» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-static) и руководство по политике Debian, раздел 8.4 «Файлы для разработки» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-dev) .

[6] Смотрите Debian wiki ReleaseGoals/LAFileRemoval (http://wiki.debian.org/ReleaseGoals/LAFileRemoval) .

[7] Смотрите вики Debian RpathIssue (http://wiki.debian.org/RpathIssue) .

[8] При обратно несовместимых изменениях ABI обычно требуется обновить SONAME библиотеки и поменять имя пакета общей библиотеки на новое.

[9] Вместо указанного для библиотек C++ и в других случаях, где слежение за отдельными символами слишком сложно, прочтите руководство по политике Debian, раздел 8.6.4 «Система shlibs» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) .

[10] Все предыдущие версии пакетов Debian доступны по http://snapshot.debian.org/ (http://snapshot.debian.org/) . Для облегчения переноса пакета в старые выпуски часть, отвечающая за версию Debian, отбрасывается: `1.1` << `1.1-1~bpo70+1` << `1.1-1` и `1.2` << `1.2-1~bpo70+1` << `1.2-1`

```
$ 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
```

- Попробуйте выполнить сборку в исходном дереве с помощью таких инструментов как **debuild** и **pdebuild** (если возникли ошибки из-за отсутствующих символов и т. д., то это указывает на обратно несовместимые изменения ABI, для которых требуется изменить имя пакета общей библиотеки на что-нибудь вроде `libfoo1a` и повторить сборку).

```
$ 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 02:24:53.609667389 +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
...
```

- Если вы видите различие, выдаваемое **dpkg-gensymbols**, как показано выше, то извлеките обновлённый правильный файл `symbols` из сгенерированного двоичного пакета общей библиотеки. [11]

```
$ 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
```

- Соберите выпускаемые пакеты с помощью таких инструментов как **debuild** и **pdebuild**.

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

В дополнение к вышеупомянутым примерам мы должны проверить дальнейшую совместимость ABI и, если понадобится, увеличить версии некоторых символов вручную. [12]

Несмотря на статус вторичности, вики Debian UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) и содержащиеся на ней ссылки могут быть полезными.

A.3 Мультиархитектурность

Свойство мультиархитектурности, появившееся в Debian wheezy, встраивает поддержку кросс-платформенной установки двоичных пакетов (а именно `i386`<->`amd64`, но есть и другие комбинации) в `dpkg` и `apt`. Подробная информация приведена в следующих документах:

[11] Для облегчения переноса пакета в старые выпуски часть, отвечающая за версию Debian, отбрасывается: `1.3` << `1.3-1~bpo70+1` << `1.3-1`

[12] Смотрите руководство по политике Debian, раздел 8.6.2 «Изменения ABI общей библиотеки» (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-updates) .

- вики Ubuntu MultiarchSpec (https://wiki.ubuntu.com/MultiarchSpec) (авторский документ)
- вики Debian Multiarch/Implementation (http://wiki.debian.org/Multiarch/Implementation) (ситуация в Debian)

При установке общих библиотек в путях используются триплеты, например `i386-linux-gnu` и `x86_64-linux-gnu`. Актуальный триплет динамически задаётся в переменной `$(DEB_HOST_MULTIARCH)` с помощью dpkg-architecture(1) при каждой сборке. Например, путь установки мультиархитектурных библиотек изменяется следующим образом:[13]

Старый путь	мультиархитектурный путь для i386	мультиархитектурный путь для amd64
`/lib/`	`/lib/i386-linux-gnu/`	`/lib/x86_64-linux-gnu/`
`/usr/lib/`	`/usr/lib/i386-linux-gnu/`	`/usr/lib/x86_64-linux-gnu/`

Вот несколько примеров разделения типично мультиархитектурных пакетов:

- библиотека с исходным кодом `lib`*`foo`*`-1.tar.gz`
- инструмент с исходным кодом *`bar`*`-1.tar.gz`, написанный на компилируемом языке
- инструмент с исходным кодом *`baz`*`-1.tar.gz`, написанный на интерпретируемом языке

Пакет	Архитектура:	Мультиархитектурная поддержка:	Содержимое пакета
`lib`*`foo1`*	любая	такая же	общая библиотека, одновременная установка
`lib`*`foo1`*`-dbg`	любая	такая же	отладочные символы общей библиотеки, одновременная установка
`lib`*`foo`*`-dev`	любая	такая же	заголовочные файлы общей библиотеки, одновременная установка
`lib`*`foo`*`-tools`	любая	сторонняя	программы поддержки времени выполнения, не одновременная установка
`lib`*`foo`*`-doc`	все	сторонняя	файлы документации общей библиотеки
`bar`	любая	сторонняя	скомпилированные файлы программы, одновременная установка
`bar``-doc`	все	сторонняя	файлы документации программы
`baz`	все	сторонняя	интерпретируемые файлы программы

Заметим, что пакет для разработчика должен содержать символьную ссылку на соответствующую общую библиотеку **без номера версии**. Пример: `/usr/lib/x86_64-linux-gnu/libfoo.so` -> `libfoo.so.1`

A.4 Сборка пакета с общей библиотекой

Вы можете собрать пакет Debian с библиотекой, включающий поддержку мультиархитектурности с помощью команды dh(1):

- Обновите `debian/control`.
  - Добавьте `Build-Depends: debhelper (>=9)` в раздел исходного кода пакета.
  - Добавьте `Pre-Depends: ${misc:Pre-Depends}` для каждого двоичного пакета с общей библиотекой.
  - Добавьте строку `Multi-Arch:` в раздел каждого двоичного пакета.
- Установите `debian/compat` равным «9».
- Измените путь с обычного `/usr/lib/` на мультиархитектурный `/usr/lib/$(DEB_HOST_MULTIARCH)/` во всех сценариях пакета.

[13] Старые пути к библиотекам, служащие для этих целей, такие как `/lib32/` и `/lib64/`, больше не используются.

- Во-первых, вызовите `DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)` в `debian/rules` для настройки переменной `DEB_HOST_MULTIARCH`.
- Замените `/usr/lib/` на `/usr/lib/$(DEB_HOST_MULTIARCH)/` в `debian/rules`.
- Если `./configure` используется в части цели `override_dh_auto_configure` в `debian/rules`, замените её на `dh_auto_configure -- .`[14]
- Замените все появления `/usr/lib/` на `/usr/lib/*/` в файлах `debian/`*`foo`*`.install`.
- Сгенерируйте такие файлы как `debian/`*`foo`*`.links` и `debian/`*`foo`*`.links.in` динамически, добавив сценарий в цель`override_dh_auto_configure` в `debian/rules`.

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

Проверьте, что пакет с общей библиотекой содержит только ожидаемые файлы и что ваши пакеты -dev ещё работают.

Все файлы, устанавливаемые одновременно из мультиархитектурного пакета по одному пути, должны иметь одинаковое содержимое. Внимательно следите за различиями при генерации данных с другим порядком байт и алгоритмом сжатия.

A.5 Родной пакет Debian

Если пакет сопровождается только для Debian или, возможно, предназначен только для локального использования, то файлы `debian/*` можно хранить прямо его в исходном коде. Есть два способа его пакетирования.

Вы можете использовать авторский tarball, исключив из него файлы `debian/*` и упаковав его как неродной пакет Debian (Раздел 2.1. Это —обычный способ, которые предпочитают использовать некоторые люди.

Альтернативный способ сборки родного пакета Debian:

- создание родного пакета Debian с исходным кодом в формате `3.0 (native)`, состоящем из одного сжатого файла tar, в который включены все файлы
  - *`пакет_версия`*`.tar.gz`
  - *`пакет_версия`*`.dsc`
- сборка двоичных пакетов Debian из родного пакета Debian с исходным кодом
  - *`пакет_версия_архитектура`*`.deb`

Например, если у вас есть файлы исходного кода в `~/mypackage-1.0` и нет файлов `debian/*`,то можете создать родной пакет Debian с помощью команды **dh_make** следующим образом:

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

В результате будет создан каталог `debian` и его содержимое, как в Раздел 2.8. При этом архив tar не создаётся, так как это родной пакет Debian —в этом единственное отличие. Остальные действия по пакетированию практически ни чем не отличаются.

После выполнения команды **dpkg-buildpackage**, вы увидите следующие файлы в родительском каталоге:

[14] Или же вы можете добавить параметры `--libdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` и `--libexecdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` в `./configure`. Заметим, что в `--libexecdir` задаётся путь по умолчанию для установки исполняемых программ, запускаемых другими программами, а не пользователями. Значение Autotools по умолчанию равно `/usr/libexec/`, но значение Debian по умолчанию равно `/usr/lib/`.

- `mypackage_1.0.tar.gz`

  Это сжатый архив tar с исходным кодом, созданный из каталога `mypackage-1.0` командой **dpkg-source** (его суффикс не `orig.tar.gz`).

- `mypackage_1.0.dsc`

  Содержит описание содержимого пакета с исходным кодом, такой же как для неродного пакета Debian (в имени нет редакции Debian).

- `mypackage_1.0_i386.deb`

  Собранный двоичный пакет, такой же как для неродного пакета Debian (в имени нет редакции Debian).

- `mypackage_1.0_i386.changes`

  Содержит описание всех изменений, сделанных в текущей версии пакета, такой же как для неродного пакета Debian (в имени нет редакции Debian).