

Как собрать бинарный deb пакет: подробное HowTo

Марк Вартанян :: 13.12.2009

Сегодня я расскажу на абстрактном примере как правильно создать *.deb пакет для Ubuntu/Debian. Пакет мы будем делать бинарный. Пакеты, компилирующие бинарники из исходников здесь не рассматриваются: осилив изложенные ниже знания, в дальнейшем по готовым примерам можно понять суть и действовать по аналогии :)

В статье не будет никакой лишней возни «вручную»: формат пакета эволюционировал в достаточно простую, а главное — логичную структуру, и всё делается буквально на коленке, с применением пары специализированных утилит.

В качестве бонуса в конце статьи будет пример быстрого создания собственного локального репозитория: установка пакетов из репозитория позволяет автоматически отслеживать зависимости, и конечно же! — устанавливать всё одной консольной командой на нескольких машинах :)

Для тех, кто не хочет вдаваться в мощную систему установки софта в Linux, рекомендую посетить [сайт проги CheckInstall](#): она автоматически создаёт deb-пакет из команды «make install» ;) А мы вместе с любопытными —

Источники

Информация надёргана из многих мест, но вот два основных:

- Opennet:L [Введение в создание пакетов для дистрибутива GNU Debian/Linux](#)
- Debian.org: [Debian Policy Manual](#) — официальная
- Ubuntu Wiki: [PackagingGuide/Basic](#)

В статье подробно изложены основы создания пакетов, достаточные для получения достаточно мощного управления установкой приложений. Более продвинутые фишки опущены, но предложены прямые ссылки на документацию для интересующихся.

Статья не является копией или переводом какой-либо документации: это — коллекция знаний, валявшихся в виде заметок, а теперь оформленная в виде статьи. Для ясности везде есть примеры, разъяснения на пальцах, найденные мной удобные фишки и некоторые типичные ошибки, которые можно совершить по незнанию.

Подготовка

Зачем это всё?

Да, CheckInstall умеет создавать рабочий пакет, но он не поддерживает все вкусности, на которые способны deb пакеты :) А именно:

- Скрипты, выполняющиеся до, после ~~и вместе~~ установки пакета :)
- Автоматическое управление конфигурационными файлами: пакет не позволит затереть старые конфиги новыми без спроса
- Работа с шаблонами: возможность задавать пользователю вопросы при установке (!!!)
- Изменение файлов других пакетов

Что потребуется

Конечно, для создания полноценного пакета хватит архиваторов tar, gz, ar, но можно исключить лишнюю возню, и воспользоваться инструментами, созданными для облегчения жизни :)

Ставим:

```
$ sudo apt-get install dpkg debconf debhelper lintian
```

Что мы будем делать

Для примера будет рассмотрен некий скрипт /usr/bin/super.sh. Не важно что внутри, главное — как он появится на правильном месте :)

Подготовка папки

В домашнем каталоге (или где удобно) создаём папку, в которой будут лежать все файлы будущего пакета: `mkdir ~/supersh`. Далее будем называть её **корень пакета**.

В корне пакета создаём папку «DEBIAN» (заглавными буквами, это важно!). Эта папка содержит управляющую генерацией пакета информацию, и не копируется на диск при установке пакета.

Также корневая папка пакета содержит будущий «корень диска»: при установке пакета все файлы (кроме папки «debian») распаковываются в корень /. поэтому наш скрипт должен лежать по такому пути, относительно корня пакета: «usr/bin/super.sh»

Белым по чёрному:

```
mkdir -p ~/supersh/DEBIAN # управляющая папка
```

```
mkdir -p ~/supersh/usr/bin # путь к скрипту
```

```
cp super.sh ~/supersh/usr/bin/ # копируем наш скрипт в нужное место
```

В итоге имеем:

```
supersh/DEBIAN/
```

```
supersh/usr/
```

```
supersh/usr/bin/
```

```
supersh/usr/bin/super.sh
```

Создание пакета: DEBIAN/*

Как я уже сказал, папка DEBIAN содержит файлы, используемые при установке. Здесь я опишу (с примерами) каждый файл.

Для создания полноценного пакета достаточно контрольного файла «control», все остальные используются либо для прикрепления текстовой информации (changelog, лицензия), либо для управления расширенными возможностями установки приложений.

Из описанных ниже файлов в папке DEBIAN/* выбираем необходимые, и заполняем согласно инструкции :)

В наше примере реально используется только **обязательный DEBIAN/control**.

DEBIAN/control: Основная информация

control — центральный файл пакета, описывающего все основные свойства. Файл — текстовый, состоящий из пар «Атрибут: значение». Можно использовать комментарии: символ «#» в начале строки (возможность была добавлена в версии dpkg >= 1.10.11, надеяться на комментарии не стоит :).

В таблице приведены все поля, определённые для контрольного файла. Обязательные поля выделены **жирным**: без них пакет не будет считаться составленным верно.

Атрибут	Описание — основные —	Примеры
Package:	Имя пакета: [a-zA-Z0-9-] — только латиница, цифры, и дефис. Имя используется при установке: apt-get install <package>	Package: supersh
Version:	Версия пакета (и проги внутри). Используется для определения «обновлять ли». Формат принят такой: <версия_программы>-<версия_пакета>. Рекомендую всегда указывать версию пакета: при изменении структуры пакета цифра увеличивается на единицу. Допустимые символы достаточно вольные: можно использовать дату и буквы. Примеры смотрите сегодня в своём репозитории :) Имя приложения (возможно, виртуальное), регистрируемое в системе в результате установки этого пакета.	Version: 1.0-1 Version: 2009.12.12-1
Provides	Используется редко: в основном, если нужно изменить имя пакета, или если более одного пакета предлагают одинаковый функционал. Например, пакеты Apache и nginx предоставляют возможность демона httpd: Provides: httpd Вы наверняка сталкивались с ошибкой при попытке установки: «is a virtual package». Это оно и есть :)	Provides: supersh
Maintainer	Имя и почта мэйнтейнера пакета: человека, который «дебианизировал» приложение. Формат произвольный, но принято имя <e-mail>	Maintainer: o_O Tync <o-o-tync.habrahabr.ru>
Architecture	Архитектура процессора, для которой предназначен пакет. Допустимые значения: i386, amd64, all, source all используется для скриптов: они же портативные, верно? :)	Architecture: all

`source` используется для компилируемых пакетов с исходниками

Определяет задачу, для которой приложение обычно используется (группа приложений).

Section

Возможные значения: admin, base, comm, contrib, devel, doc, editors, electronics, embedded, games, gnome, graphics, hamradio, interpreters, kde, libs, libdevel, mail, math, misc, net, news, non-free, oldlibs, otherosfs, perl, python, science, shells, sound, tex, text, utils, web, x11

Section: misc

Описание пакета.

Description

Описание состоит из двух частей: короткое описание (70 символов) на той же строке, и длинное описание на последующих строках, **начинающихся с пробела**.

В расширенном описании все переводы строки игнорируются. Для вставки `\n` используется одиночная точка.

Description:
Short.
Long

goes here.

.
New
line.

— СВЯЗИ И ЗАВИСИМОСТИ —

Depends

Список пакетов через запятую, которые требуются для установки этого пакета.

После имени пакета можно в круглых скобках указать ограничение на версию, используя операторы: `<`, `=`, `>`, `<=`, `>=`. Если оператор не указан — используется `>=`

Depends: dpkg,
libz (>= 1.2.3),
jpeg (= 6b), png
(< 2.0)

Список пакетов, которые требуются в процессе установки этого пакета.

Pre-Depends

Эти зависимости могут потребоваться для скриптов установки пакета: например, пакет `flash-installer` требует `wget`

Можно использовать ограничения на версию (см. Depends).

Pre-Depends: wget
(>= 1.0)

Conflicts

Список пакетов, которые не могут быть установлены одновременно с этим.

Установка не удастся, если хоть один из перечисленных пакетов уже будет установлен.

Conflicts:
crapscript

Список пакетов, файлы которых модифицируются этим пакетом.

Replaces

Требуется в случае создания «пакета-патча», изменяющего что-либо: в противном случае при замене файлов чужого пакета возникнет ошибка при установке. У меня, например, такой пакет патчит UT2004 и убирает звук наводящейся ракетницы :)

Replaces: ut2004

Recommends

Список пакетов, рекомендуемых к установке

Эти пакеты не обязательны, но обычно используются вместе с текущим

Recommends:
superplatform

Suggests

Список пакетов, предлагаемых к установке.

Эти пакеты не обязательны, но с ними прога работает ещё лучше :) По идее, менеджер пакетов должен предлагать установить их.

Suggests: supersh-
modules

Build-Depends

(Только для Architecture: source)

Список пакетов, требуемых для компиляции исходников. То же, что и Depends, но логически отделено.

Build-Depends:
cmake

— экстра —

Installed-Size

Размер файлов пакета в килобайтах.

Просто цифра, округлённая до ближайшего целого.

Installed-Size: 3

	Используется менеджером пакетов для определения суммарного требуемого объёма на диске.	
Priority	Приоритет пакета: насколько он важен в системе Возможные значения: extra, optional, standard, important, required (такие пакеты не удаляются вообще!).	Priority: optional
Esssential	Если установить этот атрибут в значение «yes», пакет нельзя будет удалить.	Esssential: yes
Origin	Строка: откуда получены программы в пакете. Обычно используется URL сайта автора, почта или имя.	Origin: brain
X-Source	Полная ссылка на *.tar.gz архив с исходниками	X-Source: ...*.tgz

Да, вот такие солидные возможности у контрольного файла :)

А в нашем примере он выглядит так:

```
Package: supersh
Version: 1.0-1
Section: misc
Architecture: all
Depends: bash, sed (>= 3.02-8)
Maintainer: o_o Tync <o-o-tync.habrahabr.ru>
Description: Super Shell Script
  A super example script.
  .
  It does nothing :)
```

DEBIAN/copyright: © / лицензия

Текст лицензии. Файл не обязателен, но лучше подчеркнуть своё авторство ;)

DEBIAN/changelog: история изменений

Changelog в специальном формате: используется dpkg для получения номера версии, ревизии, дистрибутива и важности пакета. Лучше посмотреть в [официальной документации](#) ;) а я лишь приведу пример:

```
supersh (1.0-1) stable; urgency=medium

* Testing.

-- o_o Tync <o-o-tync.habrahabr.ru> Sun, 13 Dec 2009 00:11:46 +0300
```

DEBIAN/rules: правила компиляции

Используется для управления компиляцией пакета: это когда Architecture: source :)

См. [официальную документацию](#)

DEBIAN/conffiles: список файлов конфигурации

Обычно пакеты содержат болванки конфигурационных файлов, например, размещаемых в /etc. Очевидно, что если конфиг в пакете обновляется, пользователь потеряет свой отредактированный конфиг. Эта проблема легко решается использованием папок типа «config.d», содержимое которых включается в основной конфиг, заменяя собой повторяющиеся опции.

Файл «DEBIAN/conffiles» позволяет решить проблему иначе: он содержит список файлов конфигурации (по одному на строке). Если в текущей версии пакета один из этих файлов обновляется, то пользователь получает предупреждение о конфликте версий конфигов, и может выбрать: удалить, заменить, или сделать merge.

С этой ситуацией наверняка сталкивался каждый линуксоид, копавшийся в конфигах :) А ноги растут отсюда.

На каждой строке должен быть полный абсолютный путь до каждого конфига. Например:

```
/etc/supersh/init.conf  
/etc/supersh/actions.conf
```

DEBIAN/dirs: список папок для создания

«Список абсолютных путей к папкам, которые требуются программе, но по каким-либо причинам не создаются.» — гласит официальная документация. На практике — здесь перечисляются все папки, так или иначе используемые программой: и где лежат бинарники, и которые используются программой.

По одной на строке. Например:

```
/var/log/supersh  
/var/lib/supersh
```

Удобно использовать для создания нескольких пустых папок.

DEBIAN/menu: создание пунктов меню

Хитрый файл для создания пунктов меню. У меня он так и не заработал :) Складывается ощущение, что его содержимое используется либо в необычных оконных менеджерах, либо в каком-то консольном меню... или же использовалось ранее и было забыто :)

Пример:

```
?package(supersh):needs="text" section="Applications/Programming"  
title="Super Shell Script" command="/usr/bin/super.sh"
```

TODO: узнать зачем нужно. Об этом написано в man5 menufile, честно говоря я не вникал :)

UPD: Правильный способ добавления пункта меню

Файл /DEBIAN/menu создаёт неизвестно что и непонятно где: элементы графического меню всё равно не создаются. Поэтому будем делать правильно :)

В `/usr/share/applications` видим кучку `*.desktop` файлов: это и есть пункты меню. Они представляют собой текстовые файлы с синтаксисом наподобие `ini`-файла. Открываем, учимся, делаем так же и кладем получившийся `*.desktop` файл в `usr/share/applications/`. Иконка для него должна лежать в `usr/share/pixmaps`.

После этого в `postinst` скрипт нужно добавить выполнение команды обновления меню `update-menus`:

```
if [ "$1" = "configure" ] && [ -x "`which update-menus 2>/dev/null`" ] ; then
update-menus
fi
```

Работа со скриптами установки пакета будет рассмотрена далее.

Спасибо [Condorious](#) за наводку :)

DEBIAN/md5sums: контрольные суммы файлов

Используется для проверки целостности пакета. Важный файл.

Заполняется так (`cwd`=корень пакета):

```
$ md5deep -r usr > DEBIAN/md5sums
```

DEBIAN/watch: мониторинг сайта, откуда была скачана прога

Функция полезна, если Вы мэйнтейните от нескольких десятков пакетов, и уследить за всеми обновлениями сложно.

Файл содержит инструкции для программ `uscan` и `uupdate`. Используя эту возможность, можно следить за сайтом, откуда были получены исходники пакета, и обеспечивать контроль качества дистрибутива в целом.

Пример:

```
# Site Directory Pattern Version Script
ftp.obsession.se /gentoo gentoo-(.*)\.tar\.gz debian uupdate
```

И ещё пример для `uscan(1)`:

```
version=3
madwimax.googlecode.com/files/madwimax-(.*)\.tar\.gz
```

Лучше почитайте [официальную документацию](#), такие мощные вещи нечасто требуются простым смертным :)

DEBIAN/cron.d: инсталляция заданий cron

Файл содержит кусок `crontab` для инсталляции. Пример объясняет всё:

```
#
```

```
# Launches super.sh periodically
#
0 4 * * * root [ -x /usr/bin/super.sh ] && usr/bin/super.sh
```

DEBIAN/inid.d: init-скрипт

В этот файл пишется содержимое init-скрипта. О написании init-скриптов в инете можно найти

Скриптинг

Мы подошли к самому интересному: встраиванию скриптов в deb пакеты. Скрипты позволяют управлять установкой, переустановкой и удалением пакета, выполняя действия, которые нельзя сделать простым копированием файлов в правильные места. Это может быть скачивание дополнительных файлов (как это делает flash-installer), изменение существующих, а также — вывод интерактивных (GUI или ncurses) диалогов, позволяющих пользователю сконфигурировать пакет под себя: например, mysql спрашивает какой установить пароль для root. Все скрипты выполняются от пользователя root (а как же ещё :). Также они получают аргументы (которые обрабатывать не обязательно), конкретизирующие на каком именно этапе находится установка. Подробнее об этом [здесь](#).

DEBIAN/(preinst|postinst|prerm|postrm): скрипты установки

Всего можно создать до четырёх скриптов в одном пакете:

Скрипт	Назначение
DEBIAN/preinst	Выполняется перед установкой пакета: он может подготовить что-либо для успешной установки
DEBIAN/postinst	Выполняется сразу после установки пакета: он настраивает установленный пакет так, чтоб он был готов к работе. Здесь также выполняется интерактивная конфигурация пакета: это делается при помощи dh_input и файла DEBIAN/templates
DEBIAN/prerm	Выполняется непосредственно перед удалением пакета: обычно этот скрипт подчищает установочные пути пакета так, чтоб ничего лишнего не завалялось :)
DEBIAN/postrm	Выполняется сразу после удаления пакета: вычищает остатки

Обратите внимание, что ошибки, возникающие в этих скриптах **никак не логируются**: ничего интереснее кода возврата скрипта нигде не сохраняется, и логирование необходимо делать вручную! Пользователи одного моего пакета терпели неудачу при установке на Linux Mint, и не было даже возможности попросить у них лог ошибок (которого нету) чтобы выдебагать причину :) Рекомендую использовать в начале каждого скрипта следующую болванку: она будет сохранять в syslog все возникающие ошибки.


```
#!/bin/bash
set -e # fail on any error
set -u # treat unset variables as errors

# =====[ Trap Errors ]=====#
set -E # let shell functions inherit ERR trap

# Trap non-normal exit signals:
# 1/HUP, 2/INT, 3/QUIT, 15/TERM, ERR
trap err_handler 1 2 3 15 ERR
function err_handler {
local exit_status=${1:-$?}
logger -s -p "syslog.err" -t "ootync.deb" "supersh.deb" script '$0' error code
$exit_status (line $BASH_LINENO: '$BASH_COMMAND')"
exit $exit_status
}

... Ваш код установочного скрипта ...

exit 0
```

WARNING: болванка пока не тестировалась широко, проверьте лишний раз! На невозможность отладки наткнулся совсем недавно :)

DEBIAN/templates: шаблоны для диалогов

Как уже было сказано, в скрипте DEBIAN/config можно задавать пользователю вопросы: ввести строку, выбрать один из вариантов, поставить галочку,... Этим занимается «библиотека» bash функций debhelper пакета debconf, умеющая кроме этого ещё массу полезных вещей. Здесь их не рассматриваю :)

Файл DEBIAN/templates содержит данные, используемые при выводе диалоговых окон (GUI или ncurses). Файл содержит блоки, разделённые пустой строкой. Каждый блок определяет ресурсы, используемые в одном конкретном диалоговом окне.

Шапка для всех типов диалогов стандартная:

```
Template: supersh/template-name
Type: string
Default: Default-value
Description: Dialog-title
└Dialog-text
```

Template — уникальный (в пределах одного пакета) идентификатор шаблона. Если в скрипте нужно вызвать определённый диалог — используется именно это имя.

Type — тип шаблона. Определены такие типы: string, password, boolean, select, multiselect, text, note, error.

Default-value — значение по умолчанию: пользователь может просто согласиться с ним.

Description — как и в контрольном файле, состоит из двух полей: короткое описание, и длинный текст. Первое — это заголовок «окна», второе — более развёрнутое описание того, что требуется от пользователя. Рекомендуется не использовать слов вроде «введите», а сразу суть: «Приветствие скрипта», «Точка монтирования»,...

Тип	Описание шаблона
string	Приглашение на ввод текстовой строки
password	Приглашение на ввод пароля. Для этого типа шаблона нет значения Default по понятным причинам :)
boolean	Галочка :) Имеет строковое значение «true» или «false»
select	Возможность выбора одного из нескольких вариантов. Варианты предлагаются в дополнительном атрибуте шаблона: Choices: yes, no, maybe
multiselect	Возможность выбора нескольких вариантов галочками. Варианты предлагаются в дополнительном атрибуте шаблона: Choices: sex, drugs, rock-n-roll
text	Выводит на экран текст: некоторая не очень важная информация
note	Выводит на экран текст: важная информация
error	Выводит на экран текст: очень важная информация, критическая.

Для шаблонов text, note, error также нет значения Default, так как они лишь отображают информацию :)

Поиграемся с следующим шаблоном:

Template: supersh/greeting

Type: string

Description: Welcome message

└The message you wish the script to welcome you with.

Default: Greetings, my master!

Основы использования debconf и debhelper

Это лишь работоспособные наброски. В оригинале почитать о шаблонах и работе с ними можно здесь: `man 7 debconf-devel` :)

Чтобы использовать шаблоны в своём скрипте настройки DEBIAN/config, необходимо сначала подключить функции debhelper:

`. /usr/share/debconf/confmodule`. Также этот файл нужно подключить в скрипте postinst: иначе скрипт DEBIAN/config вообще не выполнится!

Эти функции доступны в пакете debconf, не забудьте включить его в зависимости!

Примитивный пример использования. Файл DEBIAN/config

```
#!/bin/bash -e
```

```
# Подключение команд debconf
```

```
. /usr/share/debconf/confmodule
```

```

case "$1" in
configure|reconfigure)
# Запрос
db_input medium "supersh/greeting" || true # инициализация
db_go || true # вывод запроса на экран
# Обработка ответа
db_get "supersh/greeting" # Получение значения в переменную $RET
greeting="$RET"
echo "$greeting" > /etc/supersh/greeting.txt
;;
*)
echo "config called with unknown argument \"\$1\"" >&2
exit 1
;;
esac
# Запрос
db_input medium "supersh/greeting" || true # инициализация
db_go || true # вывод запроса на экран

# Обработка ответа
db_get "supersh/greeting" # Получение значения в переменную $RET
greeting="$RET"
echo "$greeting" > /etc/supersh/greeting.txt

```

Здесь уже кроется неприятная засада: обратите внимание, что функции `db_input` передаётся приоритет диалога `medium`. Для `debconf` можно установить минимальный приоритет: диалоги с приоритетом ниже которого не отображаются, а берётся значение по умолчанию (Default шаблона)! Чтобы этого ТОЧНО не случилось — используем приоритет `critical` :) Кроме того, при установке из GUI порог вывода вопросов выше, и многие из них не отображаются вообще.

Возможные приоритеты: `low` — всегда используется `default`, `medium` — дефолт обычно вполне подходит, `high` — дефолт нежелателен, `critical` — внимание пользователя жизненно важно.

`|| true` используется чтобы скрипт не помер из-за ключика `"-e"` переданного `bash`.

В этом скрипте тоже рекомендуется использовать ту болванку для отлова ошибок, иначе с распространяемым пакетом могут возникнуть проблемы при отладке :)

Все тонкости использования `debconf` (функции, способы, параметры, коды ошибок) описаны в достаточно многословном мане: `man debconf-devel`.

И последнее: при удалении пакета командой `purge` — `debconf` должен также вычистить из своей базы сведения о пакете. Например, он сохраняет выбор пользователя при запросах `db_input`.

Чтобы вычистить эти данные, нужно в `postinst`-скрипт добавить следующее:

```

if [ "$1" == "purge" ] && [ -e /usr/share/debconf/confmodule ] ; then
. /usr/share/debconf/confmodule
db_purge
fi

```

Собираем пакет! :)

Ура! Все нужные файлы созданы, лежат по нужным папочкам. Теперь пора собирать пакет :) Первое, что нужно сделать — это рекурсивно выставить всем файлам в корне пакета пользователя и группу root:root (или другие, если потребуется). Это нужно затем, что файлы пакета упаковываются в tar.gz архив который сохраняет и права доступа к файлам, и владельца. Потому нужно выполнить:

```
$ sudo chown -R root:root .
```

Однако делать это не обязательно. Есть отличная команда fakeroot которая при создании архива подменит владельца файло root-ом.

В нашем примере, скрипт должен иметь бит выполнимости.

Потом выходим на папку назад, чтоб было видно корневую папку пакета, и пакет создаётся лёгким ~~пинжом~~ сам:

```
$ fakeroot dpkg-deb --build supersh
```

Созданный пакет необходимо переименовать, чтобы он соответствовал порядку именования *.deb пакетов: <имя пакета>_<версия>_<архитектура>.deb

```
$ mv supersh.deb supersh_1.0-1_all.deb
```

Всё, пакет готов!

Автоматическая проверка пакета

Существует утилита lintian, позволяющая проверить пакет и выявить типичные ошибки в его структуре. Делается это так:

```
$ lintian supersh_1.0-1_all.deb
```

Установка пакета

```
$ sudo dpkg -i supersh_1.0-1_all.deb
```

Создаём собственный репозиторий пакетов

Теперь у нас есть собственный пакет. Когда их будет несколько, и тем более — с зависимостями, окажется, что намного удобнее быстренько поднять собственный локальный микро-репозиторий, и включить его в список источников менеджера пакетов :) Здесь я опишу быстрый HowTo «как создать свой репозиторий». Идею будет легко развить, почитывая соответствующую документацию :)

Сперва установим помощника:

```
$ sudo apt-get install reprepro
```

Описание будущего репозитория

Центр репозитория — его описание. Главное в нём — список компонент репозитория. Мы создадим компоненты «soft» и «games».

Выберите папку для будущего репозитория. Все действия производятся из её корня.

Создаём файл `conf/distributions` следующего содержания:

```
Description: my local repository
Origin: Ubuntu
Suite: testing
AlsoAcceptFor: unstable experimental
Codename: karmic
Version: 5.0
Architectures: i386 amd64 source
Components: soft games
UDebComponents: soft games
```

В нашем деле создания простого репозитория все поля не играют принципиальной роли, и используются лишь для визуального определения «что есть что» :)

Создание репозитория

Репозиторий описан! Теперь сгенерируем болванку на основе описания. Команды выполняются в корне репозитория:

```
$ reprepro export
$ reprepro createsymlinks
```

И добавим готовый репозиторий в `/etc/apt/sources.list`:

```
deb file:///path/to/repo/ karmic soft games
```

Этот репозиторий можно также расшарить при помощи веб-сервера.

Управление пакетами в репозитории

В корень репозитория кладем *.deb файлы для добавления, и добавляем их в компоненту soft дистрибутива karmic:

```
reprepro -C soft includedeb karmic *.deb
```

теперь пакеты доступны из менеджера пакетов :)

Удаление пакетов:

```
reprepro -C soft remove karmic supersh
```

Финиш

В статье рассмотрены материалы по созданию deb пакетов. Акцент сделан на моментах, для которых в сети нет достаточно наглядного описания. Надеюсь, что моя попытка изложить просто и понятно не провалилась :)

Домашнее задание :)) — вполне неплохо документированные вещи, которые легко найти в man'ax и статьях:

- Создание source пакетов, компилирующих исходники: на примере Zabbix об этом отлично рассказал хабраюзер [mahoro](#) в [своей статье](#)
- Debconf, debhelper в конфигурационных скриптах: читаем маны по debconf-devel и debhelper. Они также позволяют создать скелет пакета командой dh_make.
- Продвинутые способы создания документации в пакетах: файлы DEBIAN/docs, DEBIAN/manpage.*
- Создание init скриптов
- Управление заданиями cron
- Подписывание репозитория ключём gpg

UPD: @ICD2 [подсказывает](#), что есть GUIшная прога для создания пакетов: [GiftWrap](#).

Cheers! :)

P.S. В статье наверняка встречаются неточности и ошибки. Давайте причешем её вместе! :)