Руководство по сборке RPM-пакетов для дистрибутивов Альт

Валентин Соколов и другие.

Оглавление

Введение в пакетные менеджеры

RPM — это семейство пакетных менеджеров, применяемых в различных дистрибутивах GNU/Linux, в том числе и в проекте Сизиф и в дистрибутивах Альт. Практически каждый крупный проект, использующий RPM, имеет свою версию пакетного менеджера, отличающуюся от остальных.

Различия между представителями семейства RPM выражаются в:

- наборе макросов, используемых в .spec-файлах,
- различном поведении RPM при сборке «по умолчанию» — при отсутствии каких-либо указаний в .spec-файлах,
- формате строк зависимостей,
- мелких отличиях в семантике операций (например, в операциях сравнения версий пакетов),
- мелких отличиях в формате файлов.

Для пользователя различия чаще всего заключаются в невозможности поставить «неродной» пакет из-за проблем с зависимостями или из-за формата пакета.

RPM в проекте Сизиф также не является исключением. Основные отличия RPM в Альт и Сизиф от RPM других крупных проектов заключаются в следующем:

- обширный набор макросов для сборки различных типов пакетов,
- отличающееся поведение «по умолчанию» для уменьшения количества шаблонного кода в .spec-файлах,
- наличие механизмов для автоматического поиска межпакетных зависимостей,
- наличие так называемых set-version зависимостей (начиная с `4.0.4-alt98.46`), обеспечивающих дополнительный контроль за изменением ABI библиотек,
- до `p8` и выпусков `8.x` включительно — очень древняя версия «базового» RPM (4.0.4), от которого началось развитие ветки RPM в Sisyphus (в Sisyphus и `p9` осуществлён частичный переход на `rpm 4.13`).

Вступление

Руководство по сборке RPM пакетов:

Как подготовить исходный код для сборкив RPM.

Ссылка на раздел для тех, кто не имеет опыта разработки ПО. [preparing-software-for-packaging].

Как собрать исходный код в RPM.

Ссылка на раздел для разработчиков ПО, которым необходимо собрать программное обеспечение в RPM пакеты. Программное обеспечение для упаковки.

Расширенные сценарии сборки.

Эта ссылка на справочный материал для сборщиков RPM, работающих с расширенными сценариями сборки RPM. [advanced-topics].

PDF Версия

Вы также можете скачать PDF версию данного документа.

Структура документации

Перед тем, как приступить к сборке, нужно создать структуру каталогов, необходимую RPM, находящуюся в Вашем «домашнем» каталоге:

- Отображение фаловой структуры будет представлено следующим образом:

```
$ tree ~/RPM/
/home/user/RPM/
|-- BUILD
|-- BUILDROOT
|-- RPMS
|   |-- i586
|   |-- x86_64
|   `-- noarch
|-- SOURCES
|-- SPECS
`-- SRPMS
```

- В дальнейшем вывод команд будет продемонстрирован следующим образом:

```
Name:           bello
Version:
Release:        alt1
Summary:
```

- Темы, представляющие интерес, или словарные термины упоминаются либо как ссылки на соответствующую документацию или веб-сайт выделены **жирным** шрифтом, либо *курсивом*. Первые упоминания некоторых терминов ссылаются на соответствующую документацию.
- Названия утилит, команд и других элементов, обычно встречающихся в коде, написаны `моноширинным` шрифтом.

Система управления пакетами. Знакомство с APT

Для установки, удаления и обновления программ и поддержания целостности системы в Linux в первую очередь стали использоваться *менеджеры пакетов* (такие, как RPM в дистрибутивах `RedHat` или dpkg в `Debian GNU/Linux`). С точки зрения менеджера пакетов программное обеспечение представляет собой набор компонентов — программных *пакетов*. Такие компоненты содержат в себе набор исполняемых программ и вспомогательных файлов, необходимых для корректной работы ПО. Менеджеры пакетов дают возможность унифицировать и автоматизировать сборку бинарных пакетов и облегчают установку программ, позволяя проверять наличие необходимых для работы устанавливаемой программы компонент подходящей версии непосредственно в момент установки, а также производя все необходимые процедуры для регистрации программы во всех операционных средах пользователя. Сразу после установки программа оказывается доступна пользователю из командной строки и появляется в меню всех графических оболочек.

Полное описание APT можно узнать, перейдя по ссылке: Установка и удаление программ (пакетов)

ПРИМЕЧАНИЕ	Установка пакетов в АЛЬТ Линукс осуществляется с помощью утилиты APT
ПРИМЕЧАНИЕ	Для сокращения команд, встречающихся в тексте, будет использоваться нотация:

- - команды без административных привелегий будут начинаться с символа “$”
- - команды с административными привелегиями будут начинаться с символа “#”

ПРИМЕЧАНИЕ	По умолчанию `sudo` может быть отключено. Для получения административных привелегий используется команда `su`. Для включения `sudo` в стандартном режиме можно использовать команду:

```
# control sudowheel enabled
```

Установка необходимых пакетов для процесса сборки

Чтобы следовать данному руководству, Вам потребуется установить следующие пакеты:

ПРИМЕЧАНИЕ	Некоторые из этих пакетов устанавливаются по умолчанию в Альт. Установка проводится с правами суперпользователя.

```
# apt-get update

# apt-get install gcc rpm-build rpmlint make python gear hasher patch
```

Основные команды RPM

Для ознакомления с данным разделом потребуется пакет. В качестве примера мы будем использовать пакет Yodl-docs.

Как узнать информацию о RPM-пакете без установки?

После скачивания пакета можно посмотреть данные о нём перед установкой. Для этого используется **-qip**, (Query|Install|Package)чтобы вывести информацию о пакете.

ПРИМЕЧАНИЕ	ключ `-p` (-package) работает не с базой RPM-пакетов, а с конкретным пакетом. Например: чтобы получить информацию о файлах, находящихся в пакете, который не установлен в систему, используют ключи `-qpl`(Query|Package|List).

```
$ rpm -qip yodl-docs-4.03.00-alt2.noarch.rpm
```

Вывод:

```
Name        : yodl-docs
Epoch       : 1
Version     : 4.03.00
Release     : alt2
DistTag     : sisyphus+271589.100.1.2
Architecture: noarch
Install Date: (not installed)
Group       : Documentation
Size        : 3701571
License     : GPL
Signature   : DSA/SHA1, Чт 13 мая 2021 05:44:49, Key ID 95c584d5ae4ae412
Source RPM  : yodl-4.03.00-alt2.src.rpm
Build Date  : Чт 13 мая 2021 05:44:44
Build Host  : darktemplar-sisyphus.hasher.altlinux.org
Relocations : (not relocatable)
Packager    : Aleksei Nikiforov <darktemplar@altlinux.org>
Vendor      : ALT Linux Team
URL         : https://gitlab.com/fbb-git/yodl
Summary     : Documentation for Yodl
Description :
Yodl is a package that implements a pre-document language and tools to
process it.  The idea of Yodl is that you write up a document in a
pre-language, then use the tools (eg. yodl2html) to convert it to some
final document language.  Current converters are for HTML, ms, man, LaTeX
SGML and texinfo, plus a poor-man's text converter.  Main document types
are "article", "report", "book" and "manpage".  The Yodl document
language is designed to be easy to use and extensible.

This package contais documentation for Yodl.
```

Как установить RPM-пакет?

Для установки используется параметр **-ivh** (Install | Verbose | Hash).

ПРИМЕЧАНИЕ	Ключи -v и -h не влияют на установку, а служат для вывода наглядного процесса сборки в консоль. Ключ -v (verbose) выводит детальные значения. Ключ -h (hash) выводит "#" по мере установки пакета.

```
$ rpm -ivh yodl-docs-4.03.00-alt2.noarch.rpm
```

Вывод:

```
Подготовка...
########################################################### [100%]
Обновление / установка...
1: yodl-docs-1:4.03.00-alt2
########################################################### [100%]
Running /usr/lib/rpm/posttrans-filetriggers
```

Проверка установки пакета в системе.

```
$ rpm -q () yodl-docs
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch
```

Просмотр файлов пакета, установленного в системе.

```
$ rpm -ql yodl-docs
```

Вывод:

```
/usr/share/doc/yodl
/usr/share/doc/yodl-doc
/usr/share/doc/yodl-doc/AUTHORS.txt
/usr/share/doc/yodl-doc/CHANGES
/usr/share/doc/yodl-doc/changelog
/usr/share/doc/yodl-doc/yodl.dvi
/usr/share/doc/yodl-doc/yodl.html
/usr/share/doc/yodl-doc/yodl.latex
/usr/share/doc/yodl-doc/yodl.pdf
/usr/share/doc/yodl-doc/yodl.ps
```

```
/usr/share/doc/yodl-doc/yodl.txt
/usr/share/doc/yodl-doc/yodl01.html
/usr/share/doc/yodl-doc/yodl02.html
/usr/share/doc/yodl-doc/yodl03.html
/usr/share/doc/yodl-doc/yodl04.html
/usr/share/doc/yodl-doc/yodl05.html
/usr/share/doc/yodl-doc/yodl06.html
/usr/share/doc/yodl/AUTHORS.txt
/usr/share/doc/yodl/CHANGES
/usr/share/doc/yodl/changelog
```

Просмотр недавно установленных пакетов.

```
rpm -qa --last|head
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch                Чт 22 дек 2022 18:09:10
source-highlight-3.1.9-alt1.git.904949c.x86_64 Вт 20 дек 2022 18:38:29
libsource-highlight-3.1.9-alt1.git.904949c.x86_64 Вт 20 дек 2022 18:38:29
gem-asciidoctor-doc-2.0.10-alt1.noarch        Вт 20 дек 2022 18:34:04
w3m-0.5.3-alt4.git20200502.x86_64             Вт 20 дек 2022 18:23:05
sgml-common-0.6.3-alt15.noarch                Вт 20 дек 2022 18:23:05
libmaa-1.4.7-alt4.x86_64                      Вт 20 дек 2022 18:23:05
docbook-style-xsl-1.79.1-alt4.noarch          Вт 20 дек 2022 18:23:05
docbook-dtds-4.5-alt1.noarch                  Вт 20 дек 2022 18:23:05
dict-1.12.1-alt4.1.x86_64                     Вт 20 дек 2022 18:23:05
```

Поиск пакета в системе.

Команда **grep** поможет определить, установлен пакет в системе или нет:

```
$ rpm -qa | grep yodl-docs
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch
```

Проверка файла, относящегося к пакету.

Предположим, что нужно узнать, к какому конкретному пакету относится файл. Для этого используют команду:

```
$ rpm -qf /usr/share/doc/yodl-doc
```

Вывод:

```
yodl-docs-4.03.00-alt2.noarch
```

Вывод информации о пакете.

Чтобы получить информацию о пакете, установленном в систему, используем команду:

```
$ rpm -qi yodl-docs
```

Вывод:

```
Name        : yodl-docs
Epoch       : 1
Version     : 4.03.00
Release     : alt2
DistTag     : sisyphus+271589.100.1.2
Architecture: noarch
Install Date: Чт 22 дек 2022 18:09:10
Group       : Documentation
Size        : 3701571
License     : GPL
Signature   : DSA/SHA1, Чт 13 мая 2021 05:44:49, Key ID 95c584d5ae4ae412
Source RPM  : yodl-4.03.00-alt2.src.rpm
Build Date  : Чт 13 мая 2021 05:44:44
Build Host  : darktemplar-sisyphus.hasher.altlinux.org
Relocations : (not relocatable)
Packager    : Aleksei Nikiforov <darktemplar@altlinux.org>
Vendor      : ALT Linux Team
URL         : https://gitlab.com/fbb-git/yodl
Summary     : Documentation for Yodl
Description :
Yodl is a package that implements a pre-document language and tools to
process it.  The idea of Yodl is that you write up a document in a
pre-language, then use the tools (eg. yodl2html) to convert it to some
final document language.  Current converters are for HTML, ms, man, LaTeX
SGML and texinfo, plus a poor-man's text converter.  Main document types
are "article", "report", "book" and "manpage".  The Yodl document
language is designed to be easy to use and extensible.
```

Обновление пакета.

Для обновления пакета используется параметр **-Uvh**.

```
$ rpm -Uvh yodl-docs-4.03.00-alt2.noarch.rpm
```

Вывод:

```
 Подготовка...
```

```
########################################################### [100%]
    пакет yodl-docs-1:4.03.00-alt2.noarch уже установлен
```

ПРИМЕЧАНИЕ

Справку по ключам можно получить, набрав в консоли команду `rpm --help`

Программное обеспечение для упаковки

RPM-пакеты

В этом разделе рассматриваются основы сборки RPM-пакетов. Дополнительные сведения смотри в разделе Дополнительные материалы.

Что такое RPM-пакет?

RPM-пакет - это архив, содержащий в себе архив .cpio с файлами (скомпилированные исполняемые файлы, библиотеки и данные), а также метаданные - имя пакета, его описание, зависимости и т.д. Менеджер пакетов RPM использует эти метаданные для проверки наличия необходимых пакетов из списка зависимостей, исполнения иструкций по установке файлов и сохранения общей иформации о пакете у себя в базе.

Существует два типа RPM-пакетов:

- SRPM-пакеты (исходники) - архив с расширением `.src.rpm`
- RPM-пакеты (бинарники) - архив с расширением `.rpm`

SRPM и RPM-пакеты имеют общий формат и инструментарий, но имеют разное содержимое и служат разным целям. SRPM содержит исходный код, при необходимости патчи к нему и spec-файл, в котором описывается, как собрать исходный код в бинарный RPM-пакет. Бинарный RPM-пакет содержит бинарные файлы, созданные из исходных текстов и патчей, если таковые имелись.

Инструменты для сборки RPM-пакетов

Пакет`rpmdevtools`, установленный на этапе Необходимые пакеты, предоставляет несколько утилит для сборки RPM-пакетов. Чтобы перечислить эти утилиты, выполните в консоли следующую команду:

```
$ rpm -ql rpmdevtools | grep bin
```

Для получения дополнительной информации о вышеуказанных утилитах см. их страницы руководства или диалоговые окна справки. /

Рабочее пространство для сборки RPM-пакетов

Чтобы создать дерево каталогов, которое является рабочей областью сборки RPM-пакетов, используйте утилиту `rpmdev-setuptree`:

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/user/rpmbuild/
```

```
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
 -- SRPMS
```

Созданные каталоги служат следующим целям:

Каталог	Назначение
BUILD	Содержит все файлы, которые появляются при сборке пакета.
RPMS	Здесь появляются готовые бинарные RPM-пакеты (`.rpm`), в подкаталогах для разных архитектур, например, в подкаталогах`x86_64` и `noarch`.
SOURCES	Здесь находятся архивы исходного кода и патчи. Утилита `rpmbuild` ищет их здесь.
SPECS	Здесь хранятся spec-файлы.
SRPMS	Здесь находятся пакеты с исходниками (`.src.rpm`).

Что такое SPEC-файл?

Spec-файл можно рассматривать как "инструкцию", который утилита `rpmbuild` использует для фактической сборки RPM-пакет. Он сообщает системе сборки, что делать, определяя инструкции в серии разделов. Разделы определены в *Преамбуле* и в *Основной части*. *Преамбула* содержит ряд элементов метаданных, которые используются в *Основной части*. Тело содержит основную часть инструкций.

Пример .spec-файла

Данный пример взять из ALT Linux Wiki.

```
Name: sampleprog
Version: 1.0
Release: alt1

Summary: Sample program specfile
Summary(ru_RU.UTF-8): Пример спек-файла для программы

License: GPLv2+
Group: Development/Other
Url: http://www.altlinux.org/SampleSpecs/program

Packager: Sample Packager <sample@altlinux.org>

Source: %name-%version.tar
Patch0: %name-1.0-alt-makefile-fixes.patch

%description
```

```
This specfile is provided as sample specfile for packages with programs.
It contains most of usual tags and constructions used in such specfiles.

%description -l ru_RU.UTF-8
Этот спек-файл является примером спек-файла для пакета с программой. Он содержит
основные теги и конструкции, используемые в подобных спек-файлах.

%prep
%setup
%patch0 -p1

%build
%configure
%make_build

%install
%makeinstall_std
%find_lang %name

%files -f %name.lang
%doc AUTHORS ChangeLog NEWS README THANKS TODO contrib/ manual/
%_bindir/*
%_man1dir/*

%changelog
* Sat Sep 33 3001 Sample Packager <sample@altlinux.org> 1.0-alt1
- initial build
```

Пункты преамбулы

В этой таблице перечислены элементы, используемые в разделе преамбулы файла спецификации RPM:

SPEC Директива	Определение
`Name`	Базовое имя пакета, которое должно совпадать с именем spec-файла.
`Version`	Версия upstream-кода.
`Release`	Релиз пакета используется для указания номера сборки пакета при данной версии upstream-кода. Как правило, установите начальное `alt1` и увеличивайте его с каждым новым выпуском пакета, например: alt1, alt2, alt3 и т.д. Сбросьте значение до alt1 при создании новой версии программного обеспечения.
`Summary`	Краткое, в одну строку, описание пакета.
`License`	Лицензия на собираемое программное обеспечение.
`URL`	Полный URL-адрес для получения дополнительной информации о программе. Чаще всего это ссылка на **GitHub** upstream-проекта для собираемого программного обеспечения.

`Source0`	Путь или URL-адрес к сжатому архиву исходного кода (не исправленный, исправления обрабатываются в другом месте). Этот раздел должен указывать на доступное и надежное хранилище архива, например, на upstream-страницу, а не на локальное хранилище сборщика. При необходимости можно добавить дополнительные исходные директивы, каждый раз увеличивая их количество, например: Source1, Source2, Source3 и так далее.
`Patch0`	Название первого патча, который при необходимости будет применен к исходному коду. При необходимости можно добавить дополнительные директивы PatchX, увеличивая их количество каждый раз, например: Patch1, Patch2, Patch3 и так далее.
`BuildArch`	Если пакет не зависит от архитектуры, например, если он полностью написан на интерпретируемом языке программирования, установите для этого значение `BuildArch: noarch`. Если этот параметр не задан, пакет автоматически наследует архитектуру компьютера, на котором он собран, например `x86_64`.
`BuildRequires`	Разделённый запятыми или пробелами список пакетов, необходимых для сборки программы, написанной на скомпилированном языке. Может быть несколько записей `BuildRequires`, каждая в отдельной строке в SPEC файле.
`Requires`	Разделённый запятыми или пробелами список пакетов, необходимых программному обеспечению для запуска после установки. Это его **зависимости** Может быть несколько записей `Requires`, каждая в отдельной строке в SPEC файле.
`ExcludeArch`	Если часть программного обеспечения не может работать на определенной архитектуре процессора, Вы можете исключить эту архитектуру здесь.

Директивы `Name`, `Version` и `Release` содержат имя RPM-пакета. Эти три директивы часто называют **N-V-R** или **NVR**, поскольку имена RPM-пакета имеют формат `NAME-VERSION-RELEASE`.

Вы можете получить пример `NAME-VERSION-RELEASE`, выполнив запрос с использованием `rpm` для конкретного пакета:

```
$ rpm -q rpmdevtools
rpmdevtools-8.10-alt2.noarch
```

Здесь `rpmdevtools` - это имя пакета, `8.10` - версия, а `alt2` - релиз. Последний маркер `noarch` - сведения об архитектуре. В отличие от NVR, маркер архитектуры не находится под прямым управлением сборщика, а определяется средой сборки `rpmbuild`. Исключением из этого правила является архитектурно-независимый пакет `noarch`.

Составляющие основной части

В этой таблице перечислены элементы, используемые в теде файла спецификации RPM-пакета:

SPEC Директива	Определение
`%description`	Полное описание программного обеспечения, входящего в комплект поставки RPM. Это описание может занимать несколько строк и может быть разбито на абзацы.
`%prep`	Команда или серия команд для подготовки программного обеспечения к сборке, например, распаковка архива из Source0. Эта директива может содержать сценарий оболочки (shell скрипт).
`%build`	Команда или серия команд для фактической сборки программного обеспечения в машинный код (для скомпилированных языков) или байт-код (для некоторых интерпретируемых языков).
`%install`	Раздел, который во время сборки пакета эмулирует конечные пути установки файлов в систему. Команда или серия команд для копирования требуемых артефактов сборки из `%builddir` (где происходит сборка) в`%buildroot` каталог (который содержит структуру каталогов с файлами, подлежащими сборке). Обычно это означает копирование файлов из `~/rpmbuild/BUILD` в `~/rpmbuild/BUILDROOT` и создание необходимых каталогов `~/rpmbuild/BUILDROOT`. Это выполняется только при создании пакета, а не при установке пакета конечным пользователем. Подробности см. в разделе Работа со SPEC файлом.
`%check`	Команда или серия команд для тестирования программного обеспечения. Обычно включает в себя такие вещи, как модульные тесты.
`%files`	Список файлов, которые будут установлены в системе конечного пользователя.
`%changelog`	Запись изменений, произошедших с пакетом между различными `Version` или `Release` сборками.

Инструмент Gear

Вступление

Gear (Get Every Archive from git package Repository) - система для работы с произвольными архивами программ. В качестве хранилища данных gear использует `git`, что позволяет работать с полной историей проекта.

Основной смысл хранения исходного кода пакетов в `git-репозитории` заключается в более эффективной и удобной совместной разработке, а также в минимизации используемого дискового пространства для хранения архива репозитория за длительный срок и минимизации трафика при обновлении исходного кода.

Идея gear заключается в том, чтобы с помощью одного файла с простыми правилами (для обработки которых достаточно sed и git) можно было бы собирать пакеты из произвольно устроенного git-репозитория, по аналогии с hasher, который был задуман как средство для сборки пакетов из произвольных `srpm-пакетов`.

Структура репозитория

Хотя `gear` и не накладывает ограничений на внутреннюю организацию git-репозитория (не считая требования наличия файла с правилами), есть несколько соображений о том, как более эффективно и удобно организовывать git-репозитории, предназначенные для хранения исходного кода пакетов.

Одна сущность — один репозиторий

Не стоит помещать в один репозиторий несколько разных пакетов, за исключением случаев, когда у этих пакетов есть общий пакет-предок.

- Плюсы: Соблюдение этого правила облегчает совместную работу над пакетом, поскольку неперегруженный репозиторий легче клонировать и в целом инструментарий git больше подходит для работы с такими репозиториями.
- Минусы: Несколько сложнее выполнять операции **fetch** и **push** в случае, когда репозиториев, которые надо обработать, много. Впрочем, **fetch**/**push** в цикле выручает.

Несжатый исходный код

Сжатый разными средствами (`gzip`, `bzip2` и т.п.) исходный код лучше хранить в git-репозитории в несжатом виде.

- Плюсы: Изменение файлов, которые помещены в репозиторий в сжатом виде, менее удобно отслеживать штатными средствами (**git diff**). Поскольку git хранит объекты в сжатом виде, двойное сжатие редко приводит к экономии дискового пространства. Наконец, алгоритм, применяемый для минимизации трафика при обновлении репозитория по протоколу git, более эффективен на несжатых данных.
- Минусы: Поскольку некоторые виды сжатия одних и тех же данных могут приводить к разным результатам, может уменьшиться степень первозданности (нативности)

исходного кода.

Распакованный исходный код

Исходный код, запакованный архиваторами (`tar`, `cpio`, `zip` и т.п.), лучше хранить в `git` -репозитории в распакованном виде.

- Плюсы: Существенно удобнее вносить изменения в конечные файлы и отслеживать изменения в них, заметно меньше трафик при обновлении.
- Минусы: Поскольку `git` из информации о владельце, правах доступа и дате модификации файлов хранит только исполняемость файлов, любой архив, созданный из репозитория, будет по этим параметрам отличаться от первозданного. Помимо потери нативности, изменение прав доступа и даты модификации может теоретически повлиять на результат сборки пакета. Впрочем, сборку таких пакетов, если они будут обнаружены, всё равно придётся исправить.

Форматированный changelog

В changelog релизного `commit`а имеет смысл включать соответствующий текст из `changelog`а пакета, как это делают утилиты `gear-commit` (обёртка к `git commit`, специально предназначенная для этих целей) и `gear-srpmimport`. В результате можно будет получить представление об изменениях в очередном релизе пакета, не заглядывая в spec-файл самого пакета.

Правила экспорта

С одной стороны, для того, чтобы srpm-пакет мог быть импортирован в git-репозиторий наиболее удобным для пользователя способом, язык правил, согласно которым производится экспорт из коммита репозитория (в форму, из которой можно однозначно изготовить srpm-пакет или запустить сборку), должен быть достаточно выразительным.

С другой стороны, для того, чтобы можно было относительно безбоязненно собирать пакеты из чужих gear-репозиториев, этот язык правил должен быть достаточно простым.

Файл правил экспорта (по умолчанию в `.gear/rules`) состоит из строк формата:

```
директива: параметры
```

Параметры разделяются пробельными символами.

Директивы позволяют экспортировать:

1. Любой файл из дерева, соответствующего коммиту;
2. Любой каталог из дерева, соответствующего коммиту в виде `tar-` или `zip-архива`;
3. `nified diff` между любыми каталогами, соответствующими коммитам.

Файлы на выходе могут быть сжаты разными средствами (`gzip`, `bzip2` и т.п.). В качестве коммита может быть указан как целевой коммит (значение параметра -t утилиты gear), так и любой из его предков при соблюдении условий, гарантирующих однозначное вычисление полного имени коммита-предка по целевому коммиту.

(Правила экспорта из gear-репозитория описаны детально в `gear-rules`.)(ссылка под редактуру)

Основные типы устройства gear-репозитория

Правила экспорта реализуют основные типы устройства gear-репозитория следующим образом:

Архив с модифицированным исходным кодом

С помощью простого правила

```
tar: .
```

Всё дерево исходного кода экспортируется в один tar-архив. Если у проекта есть upstream, публикующий tar-архивы, то добавление релиза в имя tar-архива, например, с помощью правила:

```
tar: . name=@name@-@version@-@release@
```

позволяет избежать коллизий.

Архив с немодифицированным исходным кодом и патчем, содержащем локальные изменения

Если дерево с немодифицированным исходным кодом хранится в отдельном подкаталоге, а локальные изменения хранятся в gear-репозитории в виде отдельных патч-файлов, то правила экспорта могут выглядеть следующим образом:

```
tar: package_name
copy: *.patch
```

Такое устройство репозитория получается при использовании утилиты `gear-srpmimport`, предназначенной для быстрой миграции от srpm-файла к `gear`-репозиторию.

Смешанные типы

Вышеперечисленные типы устройства gear-репозитория являются основными, но не исчерпывающими. Правила экспорта достаточно выразительны для того, чтобы реализовать всевозможные сочетания основных типов и создать полнофункциональный gear-репозиторий на любой вкус.

Быстрый старт Gear

Создание gear-репозитория путём импорта созданного ранее srpm-пакета.

Пусть у нас есть srpm-пакет `foobar-1.0-alt1.src.rpm`, и, к примеру, в нём находится

следующее:

```
$ rpm -qpl foobar-1.0-alt1.src.rpm
foobar-1-fix.patch
foobar-2-fix.patch
foobar.icon.png
foobar-1.0.tar.bz2
foobar-plugins.tar.gz
```

Для того чтобы сделать из него gear-репозиторий, нам нужно:

1. Создать каталог, в котором будет располагаться наш архив:

   ```
   $ mkdir foobar
   $ cd foobar
   ```

2. Создать новый git-репозиторий:

   ```
   $ git init
   Initialized empty Git repository in .git/
   ```

   Получившийся пустой git-репозиторий будет выглядеть примерно следующим образом:

   ```
   $ ls -dlog .*
   drwxr-xr-x 4 4096 Aug 12 34:56 .
   drwxr-xr-x 6 4096 Aug 12 34:56 ..
   drwxr-xr-x 8 4096 Aug 12 34:56 .git
   ```

   Таким образом, git-репозиторий готов для импорта srpm-пакета.

3. В проекте `gear` есть утилита gear-srpmimport, предназначенная для автоматизации импортирования srpm-пакета в git-репозиторий:

   ```
   $ gear-srpmimport foobar-1.0-alt1.src.rpm
   Committing initial tree deadbeefdeadbeefdeadbeefdeadbeefdeadbeef
   gear-srpmimport: Imported foobar-1.0-alt1.src.rpm
   gear-srpmimport: Created master branch
   ```

   После выполнения импорта git-репозиторий будет выглядить следующим образом:

   ```
   $ ls -Alog
   drwxr-xr-x 1 4096 Aug 12 34:56 .gear
   drwxr-xr-x 1 4096 Aug 12 34:56 .git
   -rw-r--r-- 1 6637 Aug 12 34:56 foobar.spec
   drwxr-xr-x 3 4096 Aug 12 34:56 foobar
   ```

```
drwxr-xr-x 3 4096 Aug 12 34:56 foobar-plugins
-rw-r--r-- 1  791 Aug 12 34:56 foobar-1-fix.patch
-rw-r--r-- 1 3115 Aug 12 34:56 foobar-2-fix.patch
-rw-r--r-- 1  842 Aug 12 34:56 foobar.icon.png
```

4. При необходимости в файл правил можно вносить изменения. Например, можно убрать сжатие исходников (соответствующие изменения следует вносить и в spec-файл).

Создание gear-репозитория на основе готового git-репозитория

1. Создать и добавить в git-репозиторий spec-файл.
2. Создать и добавить в git-репозиторий файл с правилами .gear/rules.

Сборка пакета из gear-репозитория

1. Сборка пакета при помощи hasher осуществляется командой gear-hsh:

```
$ gear-hsh
```

2. Чтобы собрать старый пакет, который не содержит определения тэга Packager в spec-файле, следует отключить соответствующую проверку:

```
$ gear-hsh --no-sisyphus-check=gpg,packager
```

3. Сборка пакета при помощи rpmbuild(8) осуществляется командой gear-rpm:

```
$ gear-rpm -ba
```

Фиксация изменений в репозитории

1. Для того, чтобы сделать commit очередной сборки пакета, имеет смысл воспользоваться утилитой `gear-commit`, которая помогает сформировать список изменений на основе записи в spec-файле:

```
$ gear-commit -a
```

2. Прежде чем сделать первый commit, не забудьте сконфигурировать ваш адрес. Это можно сделать глобально несколькими способами, например, прописав соответствующие значения в `~/.gitconfig`:

```
$ git config --global user.name 'Your Name'
$ git config --global user.email '<login>@altlinux.org'
```

Для отдельно взятого git-репозитория сконфигурировать адрес можно, прописав соответствующие значения в `.git/config` этого git-репозитория:

```
$ git config user.name 'Your Name'
$ git config user.email '<login>@altlinux.org'
```

Подготовка программ

В этом руководстве представлены три версии программы `Hello World` , каждая из которых написана на разных языках программирования. Программы, написанные на этих трех разных языках собираются по разному и охватывают три основных варианта использования RPM сборщиком ПО.

Что такое Исходный код?

Исходный код - это понятные для человека инструкции к компьютеру, в которых описывается, как выполнить вычисления. Исходный код выражается с помощью [языка программирования].

ПРИМЕЧАНИЕ	Существуют тысячи языков программирования. В этом документе представлены только три из них, но их достаточно для концептуального обзора.

Программа `Hello World`, написанная на bash:

`bello`

```
#!/bin/bash

printf "Hello World\n"
```

`Hello World`, написанная на Python:

`pello.py`

```
#!/usr/bin/env python

print("Hello World")
```

`Hello World`, написанная на C:

`cello.c`

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

Целью каждой из трех программ является вывод строки `Hello World` в консоли.

Сборка из исходников

В этом разделе объясняется процесс сборки программ из исходного кода.

- Для исходного кода программ, написанных на компилируемых языках, происходит процесс **компиляции**. Этот процесс различается для разных языков. Полученный в результате сборки код становится **исполняемым**.
- Программы, написанные на языке с прямой интерпретацией вообще не нужно компилировать, они выполняются непосредственно интерпретатором.
- Для программ, написанных на интерпретируемых языках с компиляцией в байт-код, исходный код компилируется в байт-код, который затем выполняется виртуальной машиной соответствующего языка.

В этом примере демонстрируется процесс преобразования программы `cello.c`, написанной на языке C в исполняемый файл.

`cello.c`

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

Ручная сборка

Вызовите компилятор [C] из коллекции компиляторов GNU (GCC), чтобы скомпилировать исходный код в бинарный файл:

```
gcc -g -o cello cello.c
```

Запустите бинарный файл `cello`.

```
$ ./cello
Hello World
```

В данном примере был продемонстрирован процесс компиляци программы в исполняемый файл.