

Руководство по сборке RPM пакетов

Adam Miller, Maxim Svistunov, Marie Doleželová, и другие. Перевод - Valentin Sokolov

Оглавление

Вступление	1
PDF Версия	1
Файловая структура	1
Вклад в руководство	2
Необходимые пакеты	3
Зачем упаковывать программное обеспечение с помощью RPM?	4
Ваш Первый пакет RPM	5
Подготовка программного обеспечения для упаковки	7
Что такое Исходный код?	7
Как создаются программы	8
Изначально скомпилированный код	8
Интерпретируемый код	8
Сборка программного обеспечения из исходного кода	9
Изначально скомпилированный код	9
Интерпретируемый код	11
Программное обеспечение для исправления ошибок	12
Установка Произвольных Артефактов	14
Использование команды install	15
Использование команды make install	15
Подготовка исходного кода для упаковки	16
Создание Tarball с исходным кодом	17
bello	17
pello	18
cello	18
Программное обеспечение для упаковки	20
RPM Пакеты	20
Что такое RPM?	20
Инструменты для упаковки RPM	21
Рабочее пространство для упаковки RPM	21
Что такое SPEC файл?	21
BuildRoots	24
RPM Макросы	24
Работа со SPEC файлами	25
Сборка RPMS	45
Исходный RPMS	45
Бинарный RPMS	46
Проверка RPMS на корректность	47
Проверка SPEC файла bello	47

Проверка бинарного RPM bello	48
Проверка SPEC файла pello	49
Проверка бинарного RPM pello	50
Проверка SPEC файла cello	50
Проверка бинарного RPM cello	51
Дополнительные материалы	53
Подпись пакетов	53
Добавление подписи к пакету	53
Замена подписи пакета	54
Подпись во время сборки	55
Mock	56
Система контроля версий	61
tito	61
dist-git	63
Подробнее о макросах	64
Определение Ваших Собственных Макросов	64
%setup	65
%files	67
Встроенные макросы	67
RPM Макросы, предоставляемые дистрибутивом	68
Пользовательские макросы	69
Epoch, Scriptlets, and Triggers	69
Epoch	69
Scriptlets and Triggers	70
Условные обозначения RPM	73
Синтаксис условий RPM	73
Примеры условий RPM	74
Appendix A: Новые возможности RPM в RHEL 7	77
Appendix B: Ссылки на источники	78
Appendix C: Выражение благодарности	79

Вступление

Руководство по упаковке RPM пакетов:

Как подготовить исходный код для упаковки в RPM.

Для тех, кто не имеет опыта разработки программного обеспечения. [Подготовка программного обеспечения для упаковки.](#)

Как упаковать исходный код в RPM.

Это для разработчиков программного обеспечения, которым необходимо упаковать программное обеспечение в RPMs. [Программное обеспечение для упаковки.](#)

Расширенные сценарии упаковки.

Это справочный материал для упаковщиков RPM, работающих с расширенными сценариями упаковки RPM. [Дополнительные материалы.](#)

PDF Версия

Вы также можете скачать [PDF версию данного документа.](#)

Файловая структура

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

- Выходные данные команд и содержимое текстовых файлов, включая исходный код, размещаются в блоках:

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

[command output trimmed]
```

```
Name:          bello
Version:
Release:       1%{?dist}
Summary:
```

```
[file contents trimmed]
```

```
#!/usr/bin/env python  
  
print("Hello World")
```

- 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.

Вклад в руководство

Вы можете внести свой вклад в это руководство, отправив сообщение о проблеме или запрос на извлечение на [GitHub репозиторий](#).

Обе формы вклада высоко ценятся и приветствуются.

Не стесняйтесь подавать заявку на решение проблемы с обратной связью, отправлять запрос на извлечение [на GitHub](#) или и то, и другое!

Необходимые пакеты

Чтобы следовать этому руководству, Вам понадобятся пакеты, упомянутые ниже.

NOTE

Некоторые из этих пакетов устанавливаются по умолчанию в [Fedora](#), [CentOS](#) и [RHEL](#). Они подробно перечислены, чтобы показать, какие инструменты используются в этом руководстве.

В Fedora, CentOS 8, и RHEL 8:

```
$ dnf install gcc rpm-build rpm-devel rpmlint make python bash coreutils diffutils  
patch rpmdevtools
```

В CentOS 7 и RHEL 7:

```
$ yum install gcc rpm-build rpm-devel rpmlint make python bash coreutils diffutils  
patch rpmdevtools
```

Зачем упаковывать программное обеспечение с помощью RPM?

Менеджер пакетов RPM (RPM) - это система управления пакетами, которая работает на Red Hat Enterprise Linux, CentOS и Fedora. RPM упрощает распространение, управление и обновление программного обеспечения, создаваемого для Red Hat Enterprise Linux, CentOS и Fedora. Многие поставщики программного обеспечения распространяют своё программное обеспечение через обычный архив (например, tarball). Однако, упаковка программного обеспечения в пакеты RPM имеет ряд преимуществ. Эти преимущества описаны ниже.

С помощью RPM Вы можете:

Устанавливать, переустанавливать, удалять, обновлять и проверять пакеты

Пользователи могут использовать стандартные средства управления пакетами (например, Yum или PackageKit) для установки, переустановки, удаления, обновления и проверки ваших пакетов RPM.

Использовать базу данных установленных пакетов для запроса и проверки пакетов

Поскольку RPM поддерживает базу данных установленных пакетов и их файлов, пользователи могут легко запрашивать и проверять пакеты в своей системе.

Использовать метаданные для описания пакетов, инструкций по их установке и т. д.

Каждый пакет RPM содержит метаданные, описывающие компоненты пакета, версию, выпуск, размер, URL-адрес проекта, инструкции по установке и так далее

Упаковывать нетронутые исходные коды программного обеспечения в исходные и двоичные пакеты

RPM позволяет вам брать нетронутые исходные коды программного обеспечения и упаковывать их в исходные и двоичные пакеты для ваших пользователей. В исходных пакетах у Вас есть первозданные исходные тексты вместе с любыми использованными исправлениями, а также полные инструкции по сборке. Такая конструкция облегчает обслуживание пакетов по мере выпуска новых версий вашего программного обеспечения.

Добавлять пакеты в репозитории Yum

Вы можете добавить свой пакет в репозиторий Yum, который позволяет клиентам легко находить и развертывать ваше программное обеспечение.

Установить цифровую подпись Ваших упаковок

Используя ключ подписи GPG, Вы можете подписать посылку цифровой подписью, чтобы пользователи могли проверить подлинность упаковки.

Ваш Первый пакет RPM

Создание пакета RPM может быть сложным. Вот полный рабочий файл спецификации RPM, в котором несколько вещей пропущены и упрощены.

```
Name:      hello-world
Version:    1
Release:    1
Summary:    Most simple RPM package
License:    FIXME

%description
This is my first RPM package, which does nothing.

%prep
# we have no source, so nothing here

%build
cat > hello-world.sh <<EOF
#!/usr/bin/bash
echo Hello world
EOF

%install
mkdir -p %{buildroot}/usr/bin/
install -m 755 hello-world.sh %{buildroot}/usr/bin/hello-world.sh

%files
/usr/bin/hello-world.sh

%changelog
# let's skip this for now
```

Сохраните этот файл как **hello-world.спис.**

Теперь используйте эти команды:

```
$ rpmdev-setuptree
$ rpmbuild -ba hello-world.спис
```

Команда **rpmdev-setuptree** создает несколько рабочих каталогов. Поскольку эти каталоги постоянно хранятся в \$HOME, эту команду не нужно использовать снова.

Команда **rpmbuild** создает фактический пакет rpm. Вывод этой команды может быть похож на:

```
... [SNIP]
```



```
Wrote: /home/mirek/rpmbuild/SRPMS/hello-world-1-1.src.rpm
Wrote: /home/mirek/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.wgaJzv
+ umask 022
+ cd /home/mirek/rpmbuild/BUILD
+ /usr/bin/rm -rf /home/mirek/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
+ exit 0
```

Файл `/home/mirek/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm` является Вашим первым пакетом RPM. Его можно установить в систему и протестировать.

Подготовка программного обеспечения для упаковки

Эта глава посвящена исходному коду и созданию программного обеспечения, которые являются необходимой основой для RPM-упаковщика.

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](#).

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

NOTE

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 командной строке.

NOTE

Знание того, как программировать, не обязательно для упаковщика программного обеспечения, но полезно.

Как создаются программы

Существует множество методов, с помощью которых читаемый человеком исходный код становится машинным кодом - инструкциями, которым компьютер следует для фактического выполнения программы. Однако все методы можно свести к этим трем:

1. Программа изначально скомпилирована.
2. Программа интерпретируется с помощью необработанной интерпретации.
3. Программа интерпретируется путем байтовой компиляции.

Изначально скомпилированный код

Изначально скомпилированное программное обеспечение - это программное обеспечение, написанное на языке программирования, которое компилируется в машинный код с результирующим двоичным исполняемым файлом. Такое программное обеспечение можно запускать автономно.

Пакеты RPM, созданные таким образом, зависят от **архитектуры**. Это означает, что если вы скомпилируете такое программное обеспечение на компьютере, использующем 64-разрядный (x86_64) процессор AMD или Intel, оно не будет выполняться на 32-разрядном (x86) процессоре AMD или Intel. В названии результирующего пакета будет указана архитектура.

Интерпретируемый код

Некоторые языки программирования, такие как **bash** или **Python**, не компилируются в машинный код. Вместо этого исходный код их программ выполняется шаг за шагом, без предварительных преобразований, **языковым интерпретатором** или языковой виртуальной машиной.

Программное обеспечение, написанное полностью на интерпретируемых языках программирования, не зависит от **архитектуры**. Следовательно, результирующий пакет RPM будет иметь строку **noarch** в своем названии.

Существует два типа интерпретируемых языков: языки, код которых **исполняется напрямую** и языки, код которых предварительно компилируется в **байт-код**. Процесс сборки программ для этих двух типов отличается.

Программы с прямой интерпретацией

Программы на языке с прямой интерпретацией вообще не нужно компилировать, они выполняются непосредственно интерпретатором.

Программы, скомпилированные в байт-код

Программы, скомпилированные в байт-код, которые затем исполняются виртуальной машиной соответствующего языка.

NOTE

Некоторые языки предоставляют выбор: они могут быть напрямую интерпретированы или скомпилированы в байт-код.

Сборка программного обеспечения из исходного кода

В этом разделе объясняется сборка программного обеспечения на основе его исходного кода.

- Для программного обеспечения, написанного на компилируемых языках, исходный код проходит процесс **сборки**, создавая машинный код. Этот процесс, обычно называемый **компиляцией** или **переводом**, различается для разных языков. Полученное в результате сборки программное обеспечение может быть **запущено** или "**выполнено**", что заставит компьютер выполнять задачу, поставленную программистом.
- Для программного обеспечения, написанного на прямо интерпретируемых языках, исходный код не компилируется, а выполняется напрямую.
- Для программного обеспечения, написанного на интерпретируемых языках с компиляцией в байт-код, исходный код компилируется в байт-код, который затем выполняется виртуальной машиной соответствующего языка.

Изначально скомпилированный код

В этом примере вы создадите `cello.c` программу, написанную 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
```

Вот и все. Вы создали и запустили изначально скомпилированное программное обеспечение из исходного кода.

Автоматическая сборка

Вместо того, чтобы создавать исходный код вручную, вы можете автоматизировать сборку. Это обычная практика, используемая в крупномасштабном программном обеспечении. Автоматизация сборки осуществляется путем создания **Makefile** и последующим запуском утилиты **GNU make**.

Чтобы настроить автоматическую сборку, создайте файл с именем **Makefile** в том же каталоге, что и **cello.c**:

Makefile

```
cello:  
    gcc -g -o cello cello.c  
  
clean:  
    rm cello
```

Теперь, чтобы собрать программу, просто запустите **make**:

```
$ make  
make: 'cello' is up to date.
```

Поскольку сборка уже создана, **make clean** очистит её, а затем снова запустит **make**:

```
$ make clean  
rm cello  
  
$ make  
gcc -g -o cello cello.c
```

Опять же, попытка сборки после другой сборки ничего не даст:

```
$ make
```

```
make: 'cello' is up to date.
```

Наконец, программа выполнится:

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

Теперь Вы скомпилировали программу как вручную, так и с помощью инструмента сборки.

Интерпретируемый код

Следующие два примера демонстрируют компиляцию в байт-код программы, написанной на [Python](#) и прямую интерпретацию программы, написанной на [bash](#).

NOTE

В двух приведенных ниже примерах **#!** строка в верхней части файла называется [shebang](#) и не является частью исходного кода.

[shebang](#) позволяет использовать текстовый файл в качестве исполняемого файла: загрузчик системной программы анализирует строку, содержащую **shebang**, чтобы получить путь к бинарному исполняемому файлу, который затем используется в качестве интерпретатора языка программирования.

Компиляция байт-кода

В этом примере Вы скомпилируете в байт-код [pello.py](#) - программу, написанную на Python, который затем выполняется виртуальной машиной Python. Исходный код Python также может быть напрямую интерпретирован, но исполнение байт-кода быстрее. Следовательно, RPM упаковщики предпочитают упаковывать скомпилированный байт-код для распространения среди конечных пользователей.

[pello.py](#)

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

print("Hello World")
```

Процедура программ в байт-код отличается для разных языков. Это зависит от языка, виртуальной машины языка, а также инструментов и процессов, используемых с этим языком.

NOTE

[Python](#) часто компилируется в байт-код, но не так, как описано здесь. Следующая процедура направлена не на то, чтобы соответствовать стандартам сообщества, а на то, чтобы быть простой. Для получения практических рекомендаций по Python см. раздел [Упаковка и распространение программного обеспечения](#).

Компиляция [pello.py](#) в байт-код:

```
$ python -m compileall pello.py

$ file pello.pyc
pello.pyc: python 2.7 byte-compiled
```

Выполните байт-код в **pello.pyc**:

```
$ python pello.pyc
Hello World
```

Напрямую интерпретированный код

В этом примере Вы будете интерпретировать программу **bello** написанную на встроенном языке оболочки **bash**.

bello

```
#!/bin/bash

printf "Hello World\n"
```

Для программ, написанных на языках сценариев оболочки, таких как *bash*, используется прямая интерпретация. Следовательно, Вам нужно только сделать файл с исходным кодом исполняемым и запустить его:

```
$ chmod +x bello
$ ./bello
Hello World
```

Программное обеспечение для исправления ошибок

Patch - это исходный код, который исправляет другой исходный код. Он отформатирован как *diff*, потому что представляет разницу между двумя версиями текста. Разница создаётся с помощью утилиты **diff**, которая затем применяется к исходному коду с помощью утилиты **patch**.

NOTE

Разработчики программного обеспечения часто используют системы контроля версий, такие как **git**, для управления своей кодовой базой. Такие инструменты предоставляют свои собственные методы создания различий или исправления программного обеспечения.

В следующем примере мы создаем исправление из исходного кода с помощью **diff**, а затем используем **patch**. Исправление используется в следующих разделах при создании RPM и

работе со .спес-файлом. [Работа со СПЕС файлами.](#)

Как исправление связано с упаковкой RPM? В упаковке, вместо того, чтобы просто изменять исходный код, мы сохраняем его и используем на нем исправления.

Чтобы создать патч для **cello.c**:

1. Сохраним исходный код:

```
$ cp cello.c cello.c.orig
```

Это наиболее распространённый способ сохранить файл исходного кода.

2. Изменим **cello.c**:

```
#include <stdio.h>

int main(void) {
    printf("Hello World from my very first patch!\n");
    return 0;
}
```

3. Сгенерируем патч используя утилиту **diff**:

NOTE

Мы используем несколько общих аргументов для утилиты **diff**. Для получения дополнительной информации о них см. руководство по использованию **diff**.

```
$ diff -Naur cello.c.orig cello.c
--- cello.c.orig      2016-05-26 17:21:30.478523360 -0500
+++ cello.c           2016-05-27 14:53:20.668588245 -0500
@@ -1,6 +1,6 @@
 #include<stdio.h>

 int main(void){
-    printf("Hello World!\n");
+    printf("Hello World from my very first patch!\n");
     return 0;
 }
```

Строки, начинающиеся с **-** удалятся из исходного кода и заменятся на строки, начинающихся с **+**.

4. Сохраним патч в файл:

```
$ diff -Naur cello.c.orig cello.c > cello-output-first-patch.patch
```


5. Восстановим исходный код `cello.c`:

```
$ cp cello.c.orig cello.c
```

Мы сохраняем исходный файл `cello.c`, потому что при создании RPM используется исходный файл, а не измененный. Дополнительные сведения см. в разделе [Работа со SPECS файлами](#).

Чтобы исправить `cello.c` с помощью `cello-output-first-patch.patch`, перенаправьте патч-файл `patch` командой:

```
$ patch < cello-output-first-patch.patch  
patching file cello.c
```

Содержимое `cello.c` теперь отражает изменения:

```
$ cat cello.c  
#include<stdio.h>  
  
int main(void){  
    printf("Hello World from my very first patch!\n");  
    return 0;  
}
```

Чтобы собрать и запустить отредактированную `cello.c`:

```
$ make clean  
rm cello  
  
$ make  
gcc -g -o cello cello.c  
  
$ ./cello  
Hello World from my very first patch!
```

Вы создали патч, отредактировали программу, собрали отредактированную программу и запустили её.

Установка Произвольных Артефактов

Большим преимуществом [Linux](#) и других Unix-подобных систем является [Стандарт иерархии файловой системы](#). Он указывает, в каком каталоге должны быть расположены файлы. Файлы, установленные из пакетов RPM, должны быть размещены в соответствии с ИФС. Например, исполняемый файл должен находиться в каталоге, который находится в переменной `PATH`.

В контексте этого руководства, *Произвольный артефакт* - это все, что устанавливается из RPM в систему. Для RPM и для системы это может быть скрипт, бинарный файл, скомпилированный из исходного кода пакета, предварительно скомпилированный бинарный файл или любой другой файл.

Мы рассмотрим два популярных способа размещения *произвольных артефактов* в системе: с помощью команды `install` и с помощью команды `make install`.

Использование команды `install`

Иногда с помощью инструментов автоматизации сборки, таких как `GNU make` не является оптимальным - например, если упакованная программа проста. В этих случаях упаковщики часто используют команду `install` (предоставляемая системе `coreutils`), которая помещает артефакт в указанный каталог в файловой системе с указанным набором разрешений.

В приведенном ниже примере будет использоваться файл `bello`, который мы ранее создали в качестве произвольного артефакта, зависящего от нашего метода установки. Обратите внимание, что Вам либо понадобятся разрешения `sudo`, либо запустите эту команду от имени `root`, исключая часть команды `sudo`.

В этом примере `install` помещает файл `bello` в `/usr/bin` с разрешениями, общими для исполняемых скриптов:

```
$ sudo install -m 0755 bello /usr/bin/bello
```

Теперь `bello` находится в каталоге, который указан в переменной `$PATH`. Таким образом, Вы можете запустить `bello` из любого каталога, не указывая его путь:

```
$ cd ~  
  
$ bello  
Hello World
```

Использование команды `make install`

Популярным автоматизированным способом установки программного обеспечения в систему является использование команды `make install`. Вы указываете, как установить произвольные артефакты в систему в файле `Makefile`.

NOTE Обычно `Makefile` пишется разработчиком, а не упаковщиком.

Добавьте секцию `install` в `Makefile`:

`Makefile`

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

```
clean:
    rm cello

install:
    mkdir -p $(DESTDIR)/usr/bin
    install -m 0755 cello $(DESTDIR)/usr/bin/cello
```

Переменная `$(DESTDIR)` является встроенной в [GNU make](#) и обычно используется для указания установки в каталог, отличный от корневого каталога.

Теперь вы можете использовать [Makefile](#) не только для сборки программного обеспечения, но и для его установки в систему.

Для сборки и установки программы `cello.c`:

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

$ sudo make install
install -m 0755 cello /usr/bin/cello
```

Теперь `cello` находится в каталоге, который указан в переменной `$PATH`. Таким образом, Вы можете запустить `cello` из любого каталога, не указывая его полный путь.

```
$ cd ~

$ cello
Hello World
```

Вы установили артефакт сборки в выбранное место в системе.

Подготовка исходного кода для упаковки

NOTE Код, созданный в этом разделе, можно найти [здесь](#).

Разработчики часто распространяют программное обеспечение в виде сжатых архивов исходного кода, которые затем используются для создания пакетов. В этом разделе Вы создадите такие архивы.

NOTE Создание архивов исходного кода обычно выполняется не RPM-упаковщиком, а разработчиком. Упаковщик работает с готовым архивом исходного кода.

Программное обеспечение должно распространяться с [лицензией](#). Для примера мы будем использовать лицензию [GPLv3](#). Текст лицензии помещается в файл `LICENSE` для каждой из примеров программ. Упаковщику RPM необходимо иметь дело с файлами лицензий при

упаковке.

Для использования со следующими примерами создайте файл **LICENSE**:

```
$ cat /tmp/LICENSE
This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Создание Tarball с исходным кодом

В приведенных ниже примерах мы помещаем каждую из трех программ **Hello World** в архив, сжатый с помощью **gzip**. Программное обеспечение часто выпускается таким образом, чтобы позже быть упакованным для распространения.

bello

Проект *bello* реализует **Hello World** в **bash**. Реализация содержит только сценарий оболочки **bello**, поэтому результирующий архив **tar.gz** будет содержать только один файл, кроме файла **LICENSE**. Давайте предположим, что это версия программы - **0.1**

Подготовьте проект *bello* для распространения:

1. Поместите файлы в один каталог:

```
$ mkdir /tmp/bello-0.1
$ mv ~/bello /tmp/bello-0.1/
$ cp /tmp/LICENSE /tmp/bello-0.1/
```

2. Создайте архив для распространения и переместите его в **~/rpmbuild/SOURCES/**:

```
$ cd /tmp/
$ tar -cvzf bello-0.1.tar.gz bello-0.1
bello-0.1/
bello-0.1/LICENSE
```

```
bello-0.1/bello
```

```
$ mv /tmp/bello-0.1.tar.gz ~/rpmbuild/SOURCES/
```

pello

Проект *pello* реализует **Hello World** на **Python**. Реализация содержит только программу **pello.py**, так что результирующий архив **tar.gz** будет содержать только один файл, кроме файла **LICENSE**. Предположим, что это версия программы - **0.1.1**

Подготовьте проект *pello* для распространения:

1. Поместите файлы в один каталог:

```
$ mkdir /tmp/pello-0.1.1  
  
$ mv ~/pello.py /tmp/pello-0.1.1/  
  
$ cp /tmp/LICENSE /tmp/pello-0.1.1/
```

2. Создайте архив для распространения и переместите его в **~/rpmbuild/SOURCES/**:

```
$ cd /tmp/  
  
$ tar -cvzf pello-0.1.1.tar.gz pello-0.1.1  
pello-0.1.1/  
pello-0.1.1/LICENSE  
pello-0.1.1/pello.py  
  
$ mv /tmp/pello-0.1.1.tar.gz ~/rpmbuild/SOURCES/
```

cello

Проект *cello* реализует **Hello World** на **C**. Реализация содержит только файлы **cello.c** и **Makefile**, поэтому результирующий архив **tar.gz** будет содержать только два файла, помимо файла **LICENSE**. Давайте предположим, что это версия программы - **1.0**

Обратите внимание, что **patch** не распространяется в архиве вместе с программой. Упаковщик RPM применяет исправление при создании RPM. Патч будет помещен в каталог **~/rpmbuild/SOURCES/** рядом с **.tar.gz** архивом.

Подготовьте проект *cello* для распространения:

1. Поместите файлы в один каталог:

```
$ mkdir /tmp/cello-1.0
```

```
$ mv ~/cello.c /tmp/cello-1.0/

$ mv ~/Makefile /tmp/cello-1.0/

$ cp /tmp/LICENSE /tmp/cello-1.0/
```

2. Создайте архив для распространения и переместите его в `~/rpmbuild/SOURCES/`:

```
$ cd /tmp/

$ tar -cvzf cello-1.0.tar.gz cello-1.0
cello-1.0/
cello-1.0/Makefile
cello-1.0/cello.c
cello-1.0/LICENSE

$ mv /tmp/cello-1.0.tar.gz ~/rpmbuild/SOURCES/
```

3. Добавьте патч:

```
$ mv ~/cello-output-first-patch.patch ~/rpmbuild/SOURCES/
```

Теперь исходный код готов к упаковке в RPM.

Программное обеспечение для упаковки

В этом руководстве объясняется, как упаковывать RPM для дистрибутивов Linux семейства Red Hat, в первую очередь:

- [Fedora](#)
- [CentOS](#)
- [Red Hat Enterprise Linux \(RHEL\)](#)

Эти дистрибутивы используют формат упаковки [RPM](#).

Хотя эти дистрибутивы являются целевой средой, данное руководство в основном применимо ко всем дистрибутивам, основанным на [RPM based](#). Однако инструкции должны быть адаптированы для функций, специфичных для дистрибутива, таких как обязательные элементы установки, рекомендации или макросы.

В этом руководстве не предполагается никаких предварительных знаний об упаковке программного обеспечения для операционных систем Linux или какой-либо другой.

NOTE

Если Вы не знаете, что такое программный пакет или дистрибутив GNU/Linux, рассмотрите возможность изучения некоторых статей на темы [Linux](#) и [Package Managers](#).

RPM Пакеты

В этом разделе рассматриваются основы формата упаковки RPM. Дополнительные сведения смотри в разделе [Дополнительные материалы](#).

Что такое RPM?

Пакет RPM - это просто файл, содержащий другие файлы и информацию о них, необходимую системе. В частности, пакет RPM состоит из архива [cpio](#), который содержит файлы, и заголовка RPM, который содержит метаданные о пакете. Диспетчер пакетов [rpm](#) использует эти метаданные для определения зависимостей, места установки файлов и другой информации.

Существует два типа пакетов RPM:

- исходник RPM (SRPM)
- бинарный RPM

SRPMs и бинарные RPMs имеют общий формат файла и инструментарий, но имеют разное содержимое и служат разным целям. SRPM содержит исходный код, при необходимости исправления к нему и файл спецификации, в котором описывается, как встроить исходный код в бинарный RPM. Бинарный RPM содержит двоичные файлы, созданные из исходных текстов и патчей.

Инструменты для упаковки RPM

Пакет `rpmdevtools`, установленный на этапе [Необходимые пакеты](#), предоставляет несколько утилит для упаковки RPM. Чтобы перечислить эти утилиты, запустите:

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

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

Рабочее пространство для упаковки RPM

Чтобы настроить макет каталога, который является рабочей областью упаковки RPM, используйте утилиту `rpmdev-setuptree`:

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/user/rpmbuild/
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS`

5 directories, 0 files
```

Созданные каталоги служат следующим целям:

Каталог	Назначение
BUILD	Содержит все файлы, которые появляются при создании пакета.
RPMS	Бинарные RPM создаются здесь, в подкаталогах для разных архитектур, например, в подкаталогах <code>x86_64</code> и <code>noarch</code> .
SOURCES	Здесь упаковщик помещает сжатые архивы исходного кода и патчи. Команда <code>rpmbuild</code> ищет их здесь.
SPECS	Упаковщик помещает сюда файлы спецификаций.
SRPMS	Когда <code>rpmbuild</code> используется для сборки SRPM вместо бинарного RPM, результирующий SRPM создается здесь.

Что такое SPEC файл?

Файл спецификации можно рассматривать как "рецепт", который утилита `rpmbuild` использует для фактической сборки RPM. 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.

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 файла спецификации.
Version	Версия upstream-кода.
Release	Релиз пакета используется для указания номера сборки пакета при данной версии upstream-кода. Как правило, установите начальное значение равным 1%{?dist} и увеличивайте его с каждым новым выпуском пакета. Сбросьте значение до 1 при создании новой версии программного обеспечения.
Summary	Краткое, в одну строку, описание пакета.
License	Лицензия на упаковываемое программное обеспечение. Для пакетов, распространяемых в дистрибутивах сообщества, таких как Fedora , это должна быть лицензия с открытым исходным кодом, соответствующая рекомендациям по лицензированию конкретного дистрибутива.
URL	Полный URL-адрес для получения дополнительной информации о программе. Чаще всего это веб-сайт upstream-проекта для упаковываемого 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 упаковщика. При необходимости можно добавить дополнительные исходные директивы, каждый раз увеличивая их количество, например: Source1, Source2, Source3 и так далее.
Patch0	Название первого исправления, которое при необходимости будет применено 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 построен, 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. 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 файла пакета RPM. Разработчики пакетов RPM и системные администраторы часто называют эти три директивы **N-V-R** или **NVR**, поскольку имена файлов пакетов RPM имеют формат **NAME-VERSION-RELEASE**.

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

```
$ rpm -q python
python-2.7.5-34.el7.x86_64
```

Здесь **python** - это имя пакета, **2.7.5** - версия, а **34.el7** - релиз. Последний маркер **x86_64** - сведения об архитектуре. В отличие от NVR, маркер архитектуры не находится под прямым управлением RPM упаковщика, а определяется средой сборки **rpmbuild**. Исключением из этого правила является архитектурно-независимый пакет **noarch**.

Составляющие основной части

В этой таблице перечислены элементы, используемые в разделе Body (Тело, основная часть) файла спецификации RPM:

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 в Source0. Эта директива может содержать сценарий оболочки.
%build	Команда или серия команд для фактической сборки программного обеспечения в машинный код (для скомпилированных языков) или байт-код (для некоторых интерпретируемых языков).

<code>%install</code>	Команда или серия команд для копирования требуемых артефактов сборки из <code>%builddir</code> (где происходит сборка) в <code>%buildroot</code> каталог (который содержит структуру каталогов с файлами, подлежащими упаковке). Обычно это означает копирование файлов из <code>~/rpmbuild/BUILD</code> в <code>~/rpmbuild/BUILDROOT</code> и создание необходимых каталогов <code>~/rpmbuild/BUILDROOT</code> . Это выполняется только при создании пакета, а не при установке пакета конечным пользователем. Подробности см. в разделе Работа со SPEC файлом .
<code>%check</code>	Команда или серия команд для тестирования программного обеспечения. Обычно включает в себя такие вещи, как модульные тесты.
<code>%files</code>	Список файлов, которые будут установлены в системе конечного пользователя.
<code>%changelog</code>	Запись изменений, произошедших с пакетом между различными <code>Version</code> или <code>Release</code> сборками.

Дополнительные элементы

Файл спецификации также может содержать дополнительные элементы. Например, файл спецификации может содержать *скриплеты* и триггеры. Они вступают в силу в разные моменты процесса установки в системе конечного пользователя (не в процессе сборки).

Дополнительную информацию см. [Триггеры и скриплеты](#).

BuildRoots

В контексте упаковки RPM "buildroot" - это среда [chroot](#). Это означает, что артефакты сборки размещаются здесь с использованием той же иерархии файловой системы, что и в системе конечного пользователя, при этом "buildroot" выступает в качестве корневого каталога. Размещение артефактов сборки должно соответствовать стандарту иерархии файловой системы конечного пользователя.

Файлы в "buildroot" позже помещаются в архив [cpio](#), который становится основной частью RPM. Когда RPM устанавливается в системе конечного пользователя, эти файлы извлекаются в корневой каталог, сохраняя правильную иерархию.

NOTE

Начиная с выпуска Red Hat Enterprise Linux 6, программа `rpmbuild` имеет свои собственные значения макросов по умолчанию. Поскольку переопределение этих значений по умолчанию приводит к ряду проблем, Red Hat не рекомендует определять собственное значение этого макроса. Вы можете использовать макрос `%{buildroot}` с параметрами по умолчанию из каталога `rpmbuild`.

RPM Макросы

[Макрос RPM](#) - это прямая замена текста, которая может быть условно назначена на основе необязательной оценки оператора при использовании определенной встроенной

функциональности. Это означает, что Вы можете заставить RPM выполнять замены текста за Вас.

Это полезно, например, при многократной ссылке на *Version* упакованного программного обеспечения в файле спецификации. Вы определяете *Version* только один раз - в макросе `%{version}`. Затем используйте `%{version}` во всем файле спецификации. Каждый раз, во время определения строки `%{version}`, в конечном итоге будет подставлена *Version* программы, которую вы определили ранее.

Если Вы видите незнакомый макрос, Вы можете узнать о нём с помощью:

```
$ rpm --eval %{_MACRO}
```

Например:

NOTE

```
$ rpm --eval %{_bindir}
/usr/bin
```

```
$ rpm --eval %{_libexecdir}
/usr/libexec
```

Распространённым макросом является `%{?dist}`, который обозначает “тег распространения”. Он сигнализирует, какой дистрибутив используется для сборки.

Например:

```
# On a RHEL 7.x machine
$ rpm --eval %{?dist}
.el7
```

```
# On a Fedora 23 machine
$ rpm --eval %{?dist}
.fc23
```

Больше информации о макросах см. в разделе [Подробнее о макросах](#).

Работа со SPEC файлами

Большая часть упаковки программного обеспечения в RPMs - это редактирование файла спецификации. В этом разделе мы обсудим, как создать и изменить SPEC файл.

Чтобы упаковать новое программное обеспечение, Вам необходимо создать новый файл спецификации. Вместо того, чтобы писать его вручную с нуля, используйте утилиту `rpmdev-newspec`. Она создаёт незаполненный файл спецификации, и Вы заполняете необходимые директивы и поля.

В этом руководстве мы используем три примера реализации программы 'Hello World!', созданной при подготовке [программного обеспечения для упаковки](#):

- [bello-0.1.tar.gz](#)
- [pello-0.1.1.tar.gz](#)
- [cello-1.0.tar.gz](#)
 - [cello-output-first-patch.patch](#)

Переместите их в `~/rpmbuild/SOURCES`.

Создайте SPEC файл для каждой из трёх программ:

NOTE

Некоторые текстовые редакторы, ориентированные на программистов, предварительно заполняют новый `.spec` файл с их собственным шаблоном спецификации. `rpmdev-newspec` предоставляет независимый от редактора метод, именно поэтому он используется в этом руководстве.

```
$ cd ~/rpmbuild/SPECS

$ rpmdev-newspec bello
bello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec cello
cello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec pello
pello.spec created; type minimal, rpm version >= 4.11.
```

`~/rpmbuild/SPECS/` каталог теперь имеет три SPEC файла с именами `bello.spec`, `cello.spec`, и `pello.spec`.

Изучите файлы. Директивы в них представляют собой директивы, описанные в разделе [Что такое SPEC файл](#). В следующих разделах Вы заполните эти файлы спецификаций.

NOTE

Утилита `rpmdev-newspec` не использует рекомендации или соглашения, характерные для какого-либо конкретного дистрибутива Linux. Однако этот документ предназначен для Fedora, CentOS и RHEL, поэтому Вы заметите, что:

- Используйте `rpm $RPM_BUILD_ROOT` при сборке на *CentOS* (версии, предшествующие версии 7.0) или на *Fedora* (версии, предшествующие версии 18).
- Мы предпочитаем использовать обозначение `%{buildroot}` вместо `$RPM_BUILD_ROOT` при обращении к Buildroot RPM для обеспечения согласованности со всеми другими определенными или предоставленными макросами во всем файле спецификации..

Ниже приведены три примера. Каждый из них полностью описан, так что вы можете

перейти к конкретному, если он соответствует вашим потребностям в упаковке. Или прочтите их все, чтобы полностью изучить упаковку различных видов программного обеспечения.

Имя программы	Объяснение примера
bello	Программа, написанная на необработанном интерпретируемом языке программирования. Пример демонстрирует, когда исходный код не нужно собирать, а нужно только установить. Если необходимо упаковать предварительно скомпилированный бинарный файл, Вы также можете использовать этот метод.
pello	Программа, написанная на интерпретируемом языке программирования с последующей байт-компиляцией. Пример демонстрирует байт-компиляцию исходного кода и установку байт-кода - результирующих, предварительно оптимизированных файлов.
cello	Программа, написанная на изначально скомпилированном языке программирования. Пример демонстрирует общий процесс компиляции исходного кода в машинный код и установки результирующих исполняемых файлов.

bello

Первый SPEC файл создан для bash скрипта **bello** из раздела [Подготовка программного обеспечения для упаковки](#).

Убедитесь, что у вас есть:

1. Переместите исходный код **bello** в `~/rpmbuild/SOURCES/`. См. [Работа со SPEC файлом](#).
2. Теперь создайте пустой SPEC файл `~/rpmbuild/SPECS/bello.spec`. Файл будет иметь следующее содержание:

```
Name:          bello
Version:
Release:       1%{?dist}
Summary:

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q
```

```

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-

```

Теперь измените `~/rpmbuild/SPECS/bello.spec` для создания RPMs пакета **bello**:

1. Заполните поля **Name**, **Version**, **Release**, и **Summary** :

- Поле **Name** уже было указано в качестве аргумента для `rpmdev-newspec`.
- Установите **Version** в соответствии с “upstream” версией исходного кода **bello, 0.1**.
- **Release** автоматически установит `1%{?dist}`, что изначально равно **1**. Увеличивайте это значение при каждом обновлении пакета без изменения **Version**, например, при добавлении патча. Сбросьте **Release** до **1**, когда произойдёт новый выпуск новой версии программы. Например, если будет выпущена bello версии **0.2**. Макрос `disttag` более подробно описан в части про [RPM Макросы](#).
- **Summary** - это краткое, однострочное объяснение того, что представляет собой это программное обеспечение.

После Ваших изменений первый раздел SPEC файла примет следующий вид:

```

Name:      bello
Version:   0.1
Release:   1%{?dist}
Summary:   Hello World example implemented in bash script

```

2. Заполните поля **License**, **URL**, и **Source0**:

- Поле **License** это [Лицензия на программное обеспечение](#) связанная с исходным кодом из upstream-выпуска.

Для корректного заполнения поля **License**, обратитесь к: [Fedora Руководство по лицензированию](#)

Например, используйте **GPLv3+**.

+

- Поле `URL` - это URL-адрес страницы upstream-программного обеспечения. Для примера, используем <https://example.com/bello>. В данном поле рекомендуется использовать макрос `%{name}`, тогда адрес примет следующий вид: <https://example.com/%{name}>.
- Поле `Source0` содержит URL-адрес upstream-исходного кода программного обеспечения. Он должен быть напрямую связан с версией программного обеспечения, которое упаковывается. В этом примере мы можем использовать <https://example.com/bello/releases/bello-0.1.tar.gz>. Используйте макросы `%{name}` и `%{version}` для учета изменений в версии. В результате адрес примет вид: <https://example.com/%{name}/releases/%{name}-%{version}.tar.gz>.

После Ваших изменений первая секция SPEC файла примет вид:

```
Name:      bello
Version:   0.1
Release:   1%{?dist}
Summary:   Hello World example implemented in bash script

License:   GPLv3+
URL:       https://example.com/%{name}
Source0:   https://example.com/%{name}/release/%{name}-%{version}.tar.gz
```

3. Заполните директивы `BuildRequires` и `Requires` и подключите директиву `BuildArch`:

- `BuildRequires`- определяет зависимости для пакета во время сборки. Для `bello` нет этапа сборки, потому что `bash` - это интерпретируемый язык программирования, и файлы просто устанавливаются в их расположение в системе. Просто удалите эту директиву.
- `Requires` задает зависимости для пакета во время выполнения, то-есть, необходимые пакеты для работы программы. Для выполнения скрипта `bello` требуется только оболочка `bash`, поэтому укажите `bash` в этой директиве.
- Поскольку это программное обеспечение, написанное на интерпретируемом языке программирования без скомпилированных расширений, добавьте директиву `BuildArch` со значением `noarch`. Это говорит RPM о том, что этот пакет не нужно привязывать к архитектуре процессора, на которой он построен.

После Ваших изменений первая секция SPEC файла примет вид:

```
Name:      bello
Version:   0.1
Release:   1%{?dist}
Summary:   Hello World example implemented in bash script

License:   GPLv3+
URL:       https://example.com/%{name}
Source0:   https://example.com/%{name}/release/%{name}-%{version}.tar.gz
```


Requires:	bash
BuildArch:	noarch

4. Заполните поля `%description`, `%prep`, `%build`, `%install`, `%files`, and `%license`. Эти директивы являются заголовками секций, поскольку они определяют многостроковые, скриптовые или состоящие из нескольких инструкций задачи.

- `%description` - это более длинное и полное описание программного обеспечения, чем `Summary`, содержащее один или несколько абзацев. В нашем примере мы будем использовать только краткое описание.
- В разделе `%prep` указывается, как подготовить среду сборки. Обычно это включает в себя расширение сжатых архивов исходного кода, применение исправлений и, возможно, анализ информации, предоставленной в исходном коде, для использования в следующей части SPEC файла. В этом разделе мы просто используем встроенный макрос `%setup -q`.
- Секция `%build` определяет, как на самом деле создавать программное обеспечение, которое мы упаковываем. Однако, поскольку `bash` не нужно создавать, просто удалите то, что было предоставлено шаблоном, и оставьте этот раздел пустым.
- Секция `%install` содержит инструкции для `rpmbuild` о том, как установить программное обеспечение после его сборки в каталог `BUILDROOT`. Этот каталог представляет собой пустой базовый каталог `chroot`, который напоминает корневой каталог конечного пользователя. Здесь мы должны создать любые каталоги, которые будут содержать установленные файлы.

Поскольку для установки `bello` нам нужно только создать каталог назначения и установить туда исполняемый `bash` скрипт, мы будем использовать команду `install`. Макросы RPM позволяют нам делать это без жесткого кодирования путей.

Секция `%install` после Ваших изменений должен выглядеть следующим образом:

```
%install

mkdir -p %{buildroot}/%{_bindir}

install -m 0755 %{name} %{buildroot}/%{_bindir}/%{name}
```

- В секции `%files` указывается список файлов, предоставляемых этим RPM, и их полный путь в системе конечного пользователя. Следовательно, путь устанавливаемого файла `bello` - это `/usr/bin/bello`, или, с помощью макросов RPM, `%{_bindir}/%{name}`.

В этом разделе Вы можете указать роль различных файлов с помощью встроенных макросов. Это полезно для запроса метаданных с помощью команд `rpm`. Например, чтобы указать, что файл `LICENSE` является файлом лицензии на программное обеспечение, мы используем макрос `%license`.

После изменения, секция `%files` примет следующий вид:

```
%files
%license LICENSE
%{_bindir}/%{name}
```

5. Последняя секция, **%changelog**, представляет собой список записей с отметкой даты для каждой версии выпуска пакета. Они регистрируют изменения упаковки, а не изменения программного обеспечения. Примеры изменений упаковки: добавление исправления, изменение процедуры сборки в **%build**.

Следуйте следующему формату для первой строки:

*** Day-of-Week Month Day Year Name Surname <email> - Version-Release**

Следуйте данным правилам для фактической записи изменений:

- Каждая запись об изменении может содержать несколько элементов - по одному для каждого изменения
- Каждый элемент начинается с новой строки.
- Каждый элемент начинается с символа **-**.

Пример записи с отметкой даты

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First bello package
- Example second item in the changelog for version-release 0.1-1
```

Вы написали целый файл спецификации **bello**. Полный SPEC файл **bello** теперь выглядит так:

```
Name:      bello
Version:    0.1
Release:    1%{?dist}
Summary:    Hello World example implemented in bash script

License:    GPLv3+
URL:        https://www.example.com/%{name}
Source0:    https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Requires:   bash

BuildArch:  noarch

%description
The long-tail description for our Hello World Example implemented in
bash script.
```

```

%prep
%setup -q

%build

%install

mkdir -p %{buildroot}/%{_bindir}

install -m 0755 %{name} %{buildroot}/%{_bindir}/%{name}

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First bello package
- Example second item in the changelog for version-release 0.1-1

```

В следующем разделе рассказывается о том, как собрать RPM.

pello

Наш второй SPEC будет для примера, написанного на языке программирования [Python](#), который Вы скачали (или создали имитированный upstream- выпуск в разделе [Подготовка программного обеспечения](#)) и разместили его исходный код в `~/rpmbuild/SOURCES/`. Давайте продолжим и откроем файл `~/rpmbuild/SPECS/pello.spec`, и начнём заполнять некоторые поля.

Прежде чем мы начнем идти по этому пути, нам нужно рассмотреть кое-что несколько уникальное в интерпретируемом программном обеспечении с последующей компиляцией в байт-код. Поскольку мы будем использовать компиляцию в байт-код, [shebang](#) больше не применим, поскольку результирующий файл не будет содержать эту запись. Общепринятой практикой является либо использование сценария оболочки без компиляции в байт-код, который будет вызывать исполняемый файл, либо наличие небольшого фрагмента кода [Python](#), который не скомпилирован в байт-код, в качестве “точки входа” в выполнение программы. Это может показаться глупым для нашего небольшого примера, но для больших программных проектов со многими тысячами строк кода увеличение производительности при предварительной компиляции в байт-код является значительным.

NOTE

Создание скрипта для вызова байт-скомпилированного кода или наличие небайт-скомпилированной точки входа в программное обеспечение - это то, к чему разработчики upstream программного обеспечения чаще всего обращаются перед выпуском своего программного обеспечения в мир, однако это не всегда так, и это упражнение призвано помочь решить, что делать в таких ситуациях. Для получения дополнительной информации о том, как обычно выпускается и распространяется код [Python](#), пожалуйста, обратитесь

к следующей документации: [Упаковка и распространение программного обеспечения](#).

Мы создадим небольшой сценарий оболочки для вызова нашего байт-скомпилированного кода, который станет точкой входа в наше программное обеспечение. Мы сделаем это как часть самого нашего файла спецификации, чтобы продемонстрировать, как вы можете создавать сценарии действий внутри SPEC файла. Мы рассмотрим эти особенности позже в разделе `%install`.

Давайте продолжим и откроем файл `~/rpmbuild/SPECS/pello.spec` и начнем заполнять некоторые поля.

Ниже приведен шаблон вывода, который мы получили из `rpmdev-newspec`.

```
Name:          pello
Version:
Release:       1%{?dist}
Summary:

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Как и в первом примере, давайте начнем с первого набора директив, которые `rpmdev-newspec` сгруппировал в верхней части файла: `Name`, `Version`, `Release`, `Summary`. Поле `Name` уже заполнено, так как мы передали его в командной строке при использовании команды

Давайте установим **Version** соответствующую версии “upstream” релиза исходного кода *pello*, которая, как мы видим, равна **0.1.1**, как указано в примире кода, который мы загрузили (или создали в разделе [Подготовка программного обеспечения](#) section).

В поле **Release** уже установлено значение **1%{?dist}** которое изначально равно **1**, и должно увеличиваться каждый раз, когда пакет обновляется по какой-либо причине, например, включает новый патч для устранения проблемы, но не имеет новой версии upstream-выпуска. Когда происходит новый upstream-выпуск (например, была выпущена версия *pello* **0.1.2**) тогда **Release** должен быть сброшен до значения **1**. **disttag%{?dist}** выглядит знакомо по описанию макросов из [RPM Макросы](#) в предыдущем разделе.

Поле **Summary** должно представлять собой краткое, в одну строку, объяснение того, что представляет собой это программное обеспечение.

После Ваших изменений первый раздел SPEC файла примет следующий вид:

```
Name:      pello
Version:    0.1.1
Release:    1%{?dist}
Summary:    Hello World example implemented in Python
```

Теперь давайте перейдем ко второму набору директив, которые **rpmdev-newspec** сгруппировал вместе в нашем SPEC файле: **License**, **URL**, **Source0**.

Поле **License** - это [Лицензия на программное обеспечение](#), связанная с исходным кодом из upstream выпуска. Точный формат обозначения лицензии в вашем файле SPEC будет варьироваться в зависимости от того, каким конкретным рекомендациям по дистрибутиву [Linux](#), использующему RPM, Вы следуете. Мы будем использовать стандарты обозначения из [Fedora Руководство по лицензированию](#), поэтому это поле будет содержать лицензию **GPLv3+**

Поле **URL** - это веб-сайт upstream программного обеспечения. Это не ссылка на скачивание исходного кода, а фактический веб-сайт проекта, продукта или компании, где кто-то может найти больше информации о конкретной части программного обеспечения. Поскольку это просто пример, мы будем использовать адрес <https://example.com/pello>. Однако, мы применим макрос RPM **%{name}** для корректности оформления.

Поле **Source0** - это место, откуда должен быть загружен upstream исходный код программного обеспечения. Этот URL-адрес должен содержать прямую ссылку на конкретную версию выпуска исходного кода, которую мы упаковываем. Еще раз, поскольку это пример, мы будем использовать ссылку на следующий архив: <https://example.com/pello/releases/pello-0.1.1.tar.gz>.

Мы должны отметить, что в этом примере URL-адреса есть жёстко закодированные значения, которые можно изменить в будущем, и потенциально они даже могут измениться, например, версия выпуска **0.1.1**. Мы можем упростить это, если потребуется обновить только одно поле в SPEC файле и разрешить его повторное использование. Мы

будем использовать макросы <https://example.com/{name}/releases/{name}-{version}.tar.gz> вместо ссылок из примеров ранее.

После ваших изменений верхняя часть Вашего SPEC файла должна выглядеть следующим образом:

```
Name:          pello
Version:       0.1.1
Release:       1%{?dist}
Summary:       Hello World example implemented in Python

License:       GPLv3+
URL:           https://example.com/{name}
Source0:       https://example.com/{name}/release/{name}-{version}.tar.gz
```

У нас есть секции **BuildRequires** и **Requires**, каждая из которых определяет что-то, что требуется для пакета. Однако, **BuildRequires** должен сообщать **rpmbuild** о том, что необходимо Вашему пакету во время **сборки**, а **Requires** - это то, что необходимо Вашему пакету во время **установки**.

В этом примере нам понадобится пакет **python** для выполнения процесса сборки с компиляцией в байт-код. Этот пакет понадобится во время выполнения скомпилированного байт-кода, поэтому нам необходимо определить **python** как требуемый пакет в директиве **Requires**. Нам также понадобится пакет **bash** для выполнения небольшого сценария точки входа, который мы будем использовать здесь.

Поскольку эта программа написана на интерпретируемом языке программирования без изначально скомпилированных расширений, нужно добавить секцию `BuildArch`. В ней задано значение `noarch`, чтобы сообщить RPM, что этот пакет не нужно привязывать к архитектуре процессора, на которой он построен.

После Ваших изменений верхняя часть Вашего SPEC файла должна выглядеть следующим образом:

```
Name:          pello
Version:       0.1.1
Release:       1%{?dist}
Summary:       Hello World example implemented in Python

License:       GPLv3+
URL:           https://example.com/{name}
Source0:       https://example.com/{name}/release/{name}-{version}.tar.gz

BuildRequires: python
Requires:      python
Requires:      bash
```

BuildArch: noarch

Следующие директивы можно рассматривать как “заголовки разделов”, поскольку они являются директивами, которые могут определять многостроковые, скриптовые или состоящие из нескольких инструкций задачи. Мы пройдемся по ним одна за другой, как и по предыдущим пунктам.

Секция `%description` - это более длинное и полное описание программного обеспечения, чем `Summary`, содержащее один или несколько абзацев. В нашем примере мы будем использовать только краткое описание. Эта секция не будет содержать глубокое описание, но при желании раздел может быть целым абзацем или более.

Секция `%prep` - это место, где мы *подготавливаем* нашу среду сборки или рабочее пространство для сборки. Чаще всего здесь происходит расширение сжатых архивов исходного кода, применение исправлений и, возможно, анализ информации, предоставленной в исходном коде, которая необходима в следующей части SPEC файла. В этом разделе мы просто будем использовать предоставленный макрос `%setup -q`.

Секция `%build` - это раздел, где мы рассказываем системе, как на самом деле собирать программное обеспечение, которое мы упаковываем. Здесь мы выполним компиляцию нашего программного обеспечения в байт-код. Для тех, кто читал раздел [Подготовка программного обеспечения](#), эта часть примера должна показаться знакомой.

Секция `%build` нашего SPEC файла должна выглядеть следующим образом:

```
%build  
  
python -m compileall pello.py
```

Секция `%install` - это раздел, отвечающий за инструктирование `rpmbuild`, устанавливающее наше ранее созданное программное обеспечение в `BUILDROOT`, который фактически является базовым каталогом `chroot`, в котором ничего нет, и нам нужно будет создать любые пути или иерархии каталогов, которые нам понадобятся, чтобы установить наше программное обеспечение в определенных местах. Однако наши макросы RPM помогают нам выполнить эту задачу без необходимости жестко кодировать пути.

Ранее мы обсуждали, что, поскольку мы потеряем контекст файла со строкой `shebang` в нём при компиляции в байт-код, нам нужно будет создать простой сценарий-оболочку для выполнения этой задачи. Есть много вариантов того, как это сделать, включая, но не ограничиваясь этим, создание отдельного скрипта и использование его в качестве отдельной директивы `SourceX`, а также вариант, который мы собираемся показать в этом примере, который заключается в сборке файла в строке в SPEC файле. Причина, по которой мы показываем примерный вариант, заключается в том, чтобы просто продемонстрировать, что сам файл спецификации доступен для сценариев. Мы собираемся создать небольшой “сценарий-оболочку”, который будет выполнять скомпилированный байт-код `Python`, используя [here document](#). Нам также нужно будет установить скомпилированный байт-код в каталог библиотеки в системе, чтобы к нему можно было получить доступ.

NOTE

Ниже Вы заметите, что мы жестко кодируем путь к библиотеке. Существуют различные методы, позволяющие избежать необходимости делать это, многие из которых рассматриваются в [\[дополнительных разделах\]](#), в разделе [Подробнее о макросах](#), и специфичны для языка программирования, на котором было написано упаковываемое программное обеспечение. В этом примере мы жестко закодировали путь для простоты, чтобы не охватывать слишком много тем одновременно.

Секция **%install** после Ваших изменений должна выглядеть следующим образом:

```
%install

mkdir -p %{buildroot}/%{_bindir}
mkdir -p %{buildroot}/usr/lib/%{name}

cat > %{buildroot}/%{_bindir}/%{name} <<-EOF
#!/bin/bash
/usr/bin/python /usr/lib/%{name}/%{name}.pyc
EOF

chmod 0755 %{buildroot}/%{_bindir}/%{name}

install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/
```

Секция **%files** - это место, где мы предоставляем список файлов, которые предоставляет этот RPM и где они должны находиться в системе, на которую установлен RPM. Обратите внимание, что это относится не к **%{buildroot}**, а к полному пути к файлам, поскольку ожидается, что они будут существовать в конечной системе после установки. Таким образом, список устанавливаемого файла **pello** будет: **%{_bindir}/pello**. Нам также нужно будет предоставить список **%dir**, чтобы определить, что этот пакет “владеет” каталогом библиотеки, который мы создали, а также всеми файлами, которые мы разместили в нём.

Кроме того, в этом разделе Вам иногда понадобится встроенный макрос для предоставления контекста файла. Это может быть полезно для системных администраторов и конечных пользователей, которые могут захотеть запросить систему о конечном пакете с помощью **rpm**. Встроенный макрос, который мы будем использовать здесь, - это **%license**, который сообщит **rpm**, что это файл лицензии на программное обеспечение в метаданных манифеста файла пакета.

Секция **%files** после Ваших изменений должен выглядеть следующим образом:

```
%files
%license LICENSE
%dir /usr/lib/%{name}/
%{_bindir}/%{name}
/usr/lib/%{name}/%{name}.py*
```


Последняя секция, `%changelog`, представляет собой список записей с отметками о дате, которые соотносятся с конкретной версией-выпуском пакета. Это не журнал изменений в программном обеспечении от выпуска к выпуску, а конкретно изменения в упаковке. Например, если программное обеспечение в пакете нуждалось в исправлении или было необходимо внести изменения в процедуру сборки, указанную в секции `%build`, эта информация будет размещена здесь. Каждая запись изменения может содержать несколько элементов, и каждый элемент должен начинаться с новой строки и символа `-`. Ниже приведен наш пример записи:

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
- First pello package
- Example second item in the changelog for version-release 0.1.1-1
```

Обратите внимание на приведенный выше формат: отметка даты будет начинаться с символа `*`, за которым следует календарный день недели, месяц, день месяца, год, затем контактная информация для упаковщика RPM. Оттуда у нас есть символ `-` перед выпуском версии, что является часто используемым, но не строго регламентированным. Затем, наконец, Версия-Релиз.

Вот и все! Мы написали целый файл спецификаций для **pello**! В следующем разделе мы расскажем, как создать RPM!

Полный файл спецификации теперь должен выглядеть следующим образом:

```
Name:          pello
Version:       0.1.1
Release:       1%{?dist}
Summary:       Hello World example implemented in python

License:       GPLv3+
URL:           https://www.example.com/%{name}
Source0:       https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

BuildRequires: python
Requires:      python
Requires:      bash

BuildArch:     noarch

%description
The long-tail description for our Hello World Example implemented in
Python.

%prep
%setup -q

%build
```

```
python -m compileall %{name}.py

%install

mkdir -p %{buildroot}/%{_bindir}
mkdir -p %{buildroot}/usr/lib/%{name}

cat > %{buildroot}/%{_bindir}/%{name} <<-EOF
#!/bin/bash
/usr/bin/python /usr/lib/%{name}/%{name}.pyc
EOF

chmod 0755 %{buildroot}/%{_bindir}/%{name}

install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/

%files
%license LICENSE
%dir /usr/lib/%{name}/
%{_bindir}/%{name}
/usr/lib/%{name}/%{name}.py*

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
- First pello package
```

cello

Наш третий SPEC файл будет для нашего примера на языке [C](#) , для которого мы ранее создали имитированную версию upstream (или вы скачали) и разместили его исходный код в [~/rpmbuild/SOURCES/](#).

Давайте откроем файл [~/rpmbuild/SPECS/cello.spec](#) и начнём заполнять некоторые поля.

Ниже приведен шаблон вывода, который мы получили от [rpmdev-newspec](#).

```
Name:          cello
Version:
Release:       1%{?dist}
Summary:

License:
URL:
Source0:

BuildRequires:
Requires:

%description
```

```

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-

```

Как и в предыдущих примерах, давайте начнем с первого набора директив, которые `rpmdev-newspec` сгруппировал в верхней части файла: `Name`, `Version`, `Release`, `Summary`. The `Name` уже указано, потому что мы предоставили эту информацию в командной строке для `rpmdev-newspec`.

Давайте установим в поле `Version` значение, соответствующее “upstream” версии исходного кода `cello`, которая, как мы видим, равна `1.0`, как указано в примере кода, который мы загрузили (или создали в секции [Подготовка программного обеспечения](#)).

В `Release` уже установлено значение `1%{?dist}` числовое значение, которое изначально равно `1`, должно увеличиваться каждый раз, когда пакет обновляется по какой-либо причине, например, включает новый патч для устранения проблемы, но не имеет новой версии upstream выпуска. Когда происходит новый upstream выпуск (например, была выпущена версия `cello 2.0`), тогда значение `Release` должно быть сброшено до `1`. `disttag%{?dist}` выглядит знакоммо по описанию макросов из [RPM Макросы](#) в предыдущем разделе.

`Summary` должно представлять собой краткое, в одну строку, объяснение того, что представляет собой это программное обеспечение.

После ваших изменений первый раздел SPEC файла должен выглядеть следующим образом:

```

Name:          cello
Version:       1.0
Release:       1%{?dist}
Summary:       Hello World example implemented in C

```

Теперь давайте перейдем ко второму набору директив, которые `rpmdev-newspec` сгруппировал вместе в нашем SPEC файле: `License`, `URL`, `Source0`. Однако, мы добавим одну директиву в эту группу, поскольку она тесно связана с `Source0`, и это наш `Patch0` в котором будет указан первый патч, который нам нужен для нашего программного обеспечения.

Поле **License** - это [Лицензия на программное обеспечение](#), связанная с исходным кодом из upstream выпуска. Точный формат обозначения лицензии в вашем SPEC файле будет варьироваться в зависимости от того, каким конкретным рекомендациям по дистрибутиву [Linux](#), использующим RPM, Вы следуете. Мы будем использовать стандарты обозначения из [Fedora. Руководство по лицензированию](#), поэтому это поле будет содержать лицензию **GPLv3+**

Поле **URL** - это веб-сайт upstream программного обеспечения. Это не ссылка на скачивание исходного кода, а фактический веб-сайт проекта, продукта или компании, где кто-то может найти больше информации об этой конкретной части программного обеспечения. Поскольку это просто пример, мы будем использовать адрес. <https://example.com/cello>. Однако, мы применим макрос RPM **%{name}** для корректности оформления.

Поле **Source0** - это место, откуда должен быть загружен upstream исходный код программного обеспечения. Этот URL-адрес должен содержать прямую ссылку на конкретную версию выпуска исходного кода, которую мы упаковываем. Еще раз, поскольку это пример, мы будем использовать ссылку на следующий архив: <https://example.com/cello/releases/cello-1.0.tar.gz>

Мы должны отметить, что в этом примере URL-адреса есть жестко закодированные значения, которые можно изменить в будущем и потенциально они даже могут измениться, например, версия выпуска **1.0**. Мы можем упростить это, если потребуется обновить только одно поле в SPEC файле и разрешить его повторное использование. Мы будем использовать макросы <https://example.com/%{name}/releases/%{name}-%{version}.tar.gz>, вместо ссылок из примеров ранее.

Следующий пункт - предоставить список для файла **.patch** который мы создали ранее, чтобы мы могли применить его к коду позже в секции **%prep**. Нам понадобится список **Patch0: cello-output-first-patch.patch**.

После Ваших изменений верхняя часть SPEC файла должна выглядеть следующим образом:

```
Name:          cello
Version:       1.0
Release:       1%{?dist}
Summary:       Hello World example implemented in C

License:       GPLv3+
URL:           https://example.com/%{name}
Source0:       https://example.com/%{name}/release/%{name}-%{version}.tar.gz

Patch0:        cello-output-first-patch.patch
```

У нас есть секции **BuildRequires** и **Requires**, каждая из которых определяет что-то, что требуется для пакета. Однако, **BuildRequires** должен сообщать **rpmbuild**, что необходимо Вашему пакету во время **сборки**, а **Requires** - это то, что необходимо пакету во время **установки**.

В этом примере нам понадобятся пакеты **gcc** и **make** для выполнения процесса сборки и

компиляции. Требования времени выполнения, к счастью, обрабатываются `rpmbuild`, потому что эта программа не требует ничего за пределами основных стандартных библиотек `C`, и поэтому нам не нужно будет определять что-либо вручную в качестве `Requires`, и мы можем опустить эту директиву.

После Ваших изменений верхняя часть SPEC Вашего файла должна выглядеть следующим образом:

```
Name:          cello
Version:       0.1
Release:       1%{?dist}
Summary:       Hello World example implemented in C

License:       GPLv3+
URL:           https://example.com/%{name}
Source0:       https://example.com/%{name}/release/%{name}-%{version}.tar.gz

BuildRequires: gcc
BuildRequires: make
```

Следующие директивы являются заголовками секций, поскольку они определяют многостроковые, скриптовые или состоящие из нескольких инструкций задачи. Мы пройдемся по ним один за другим, как и по предыдущим пунктам.

Секция `%description` - это более длинное и полное описание программного обеспечения, чем `Summary`, содержащее один или несколько абзацев. В нашем примере мы будем использовать только краткое описание. В нашем примере это секция не будет содержать глубокое описание, но при желании этот раздел может быть целым абзацем и более.

Секция `%prep` - это место, где мы *подготавливаем* нашу среду сборки или рабочее пространство для сборки. Чаще всего здесь происходит расширение сжатых архивов исходного кода, применение исправлений и, возможно, анализ информации, предоставленной в исходном коде, которая необходима в следующей части SPEC файла. В этом разделе мы просто будем использовать предоставленный макрос `%setup -q`.

Секция `%build` это то, где мы рассказываем системе, как на самом деле собирать программное обеспечение, которое мы упаковываем. Поскольку мы написали простой `Makefile` для нашей реализации на `C`, мы можем просто использовать команду `GNU make: rpmdev-newspec`. Однако нам нужно удалить вызов, `%configure`, поскольку мы не предоставили `configure script`. Секция `%build` нашего SPEC файла должна выглядеть следующим образом.

```
%build
make %{_smp_mflags}
```

Секция `%install` - это то, где мы инструктируем `rpmbuild` как установить наше программное обеспечение в `BUILDROOT`, который фактически является базовым каталогом `chroot`, в котором ничего нет, и нам нужно будет создать любые пути или иерархии каталогов, которые нам понадобятся, чтобы установить наше программное обеспечение. Однако наши

макросы RPM помогают нам выполнить эту задачу без необходимости жестко кодировать пути.

Еще раз, поскольку у нас есть простой `Makefile`, шаг установки можно легко выполнить, оставив на месте макрос `%make_install`, который снова был предоставлен нам командой `rpmdev-newspec`.

Секция `%install` после Ваших изменений должна принять следующий вид:

```
%install
%make_install
```

Секция `%files` - это место, где мы предоставляем список файлов, которые предоставляет этот RPM, и где они должны находиться в системе. Обратите внимание, что это относится не к `%{buildroot}`, а к полному пути к файлам, поскольку ожидается, что они будут существовать в конечной системе после установки. Таким образом, путь устанавливаемого файла `cello` будет: `%{_bindir}/cello`.

Кроме того, в этом разделе Вам иногда понадобится встроенный макрос для предоставления контекста для файла. Это может быть полезно для системных администраторов и конечных пользователей, которые могут захотеть запросить систему с помощью `rpm` о конечном пакете. Встроенный макрос, который мы будем использовать здесь, это `%license`, который сообщит `rpmbuild`, что это файл лицензии на программное обеспечение в метаданных.

Секция `%files` после Ваших изменений должна выглядеть следующим образом:

```
%files
%license LICENSE
%{_bindir}/%{name}
```

Последняя секция, `%changelog`, представляет собой список записей с отметками о дате, которые соотносятся с конкретной версией-выпуском пакета. Это не журнал изменений в программном обеспечении от выпуска к выпуску, а конкретно изменения в упаковке. Например, если программное обеспечение в пакете нуждалось в исправлении или было необходимо внести изменения в процедуру сборки, указанную в секции `%build`, эта информация будет размещена здесь. Каждая запись изменения может содержать несколько элементов, и каждый элемент должен начинаться с новой строки и символа `-`. Ниже приведен наш пример записи:

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First cello package
```

Обратите внимание на приведенный выше формат, отметка даты будет начинаться с символа `*`, за которым следует календарный день недели, месяц, день месяца, год, затем контактная информация для упаковщика RPM. Оттуда у нас есть символ `-` перед выпуском

версии, что является часто используемым, но не строго регламентированным. Затем, наконец, Версия-Релиз.

Вот и все! Мы написали целый файл спецификаций для **cello**!

Полный файл спецификации теперь должен выглядеть следующим образом:

```
Name:          cello
Version:       1.0
Release:       1%{?dist}
Summary:       Hello World example implemented in C

License:       GPLv3+
URL:           https://www.example.com/%{name}
Source0:       https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Patch0:        cello-output-first-patch.patch

BuildRequires: gcc
BuildRequires: make

%description
The long-tail description for our Hello World Example implemented in
C.

%prep
%setup -q

%patch0

%build
make %{?_smp_mflags}

%install
%make_install

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 1.0-1
- First cello package
```

Пакет **rpmdevtools** предоставляет набор шаблонов файлов спецификаций для нескольких популярных языков в каталоге **/etc/rpmdevtools/**.

Сборка RPMS

RPMS собираются с помощью команды `rpmbuild`. Различные сценарии и желаемые результаты требуют различных комбинаций аргументов для `rpmbuild`. В этом разделе описываются два основных сценария:

1. сборка исходного RPM
2. сборка бинарного RPM

Команда `rpmbuild` ожидает определенную структуру каталогов и файлов. Это та же структура, что и в утилите `rpmdev-setuptree`. Предыдущие инструкции также подтвердили требуемую структуру.

Исходный RPMS

Зачем создавать исходный RPM (SRPM)?

1. Чтобы сохранить точный источник определенного Name-Version-Release RPM, который был развернут в среде. Это включает в себя точный SPEC файл, исходный код и все соответствующие исправления. Это полезно для просмотра истории и для отладки.
2. Чтобы иметь возможность создавать бинарный RPM на другой аппаратной платформе или [архитектуре](#).

Для сборки SRPM:

```
$ rpmbuild -bs _SPECFILE_
```

Замените *SPECFILE* именем SPEC файла. Параметр `-bs` "исходный код сборки".

Здесь мы собираем SRPMs для `bello`, `pello` и `cello`:

```
$ cd ~/rpmbuild/SPECS/

$ rpmbuild -bs bello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
```

Обратите внимание, что SRPMs были помещены в каталог `rpmbuild/SRPMS`, который является частью структуры, ожидаемой `rpmbuild`.

Это все, что нужно для сборки SRPM.

Бинарный RPMs

Существует два метода сборки бинарных RPMs:

1. Восстановление его из SRPM с использованием команды `rpmbuild --rebuild`.
2. Собираем его из файла спецификации с помощью команды `rpmbuild -bb`. Опция `-bb` означает "собрать бинарный файл" (`build binary`).

Восстановление из исходного RPM

Чтобы перестроить `bello`, `pello` и `cello` из исходных RPM (SRPMs), запустите:

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
[output truncated]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
[output truncated]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
[output truncated]
```

Теперь Вы собрали RPM. Несколько заметок:

- Выходные данные, генерируемые при сборке бинарного RPM, являются подробными, что полезно для отладки. Выходные данные различаются для разных примеров и соответствуют их SPEC файлам.
- Конечные бинарные RPM находятся в `~/rpmbuild/RPMS/YOURARCH`, где `YOURARCH` - это Ваша [архитектура](#), или в `~/rpmbuild/RPMS/noarch/`, если пакет не зависит от архитектуры.
- Вызов `rpmbuild --rebuild` включает в себя:
 1. Установку содержимого RPM - файла спецификации и исходного кода - в каталог `~/rpmbuild/`.
 2. Сборка с использованием установленного содержимого.
 3. Удаление файла спецификации и исходного кода.

Вы можете сохранить файл спецификации и исходный код после сборки. Для этого у Вас есть два варианта:

- При сборке используйте опцию `--recompile` вместо `--rebuild`.
- Установите SRPMS с помощью следующих команд:

```
$ rpm -Uvh ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
Updating / installing...
  1:bello-0.1-1.el7                ##### [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
Updating / installing...
```

```
1:pello-0.1.1-1.el7 ##### [100%]  
  
$ rpm -Uvh ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm  
Updating / installing...  
1:cello-1.0-1.el7 ##### [100%]
```

В этом руководстве выполните приведенные выше команды `rpm -Uvh` чтобы продолжить взаимодействие с файлами спецификаций и исходными кодами.

Создание бинарного файла из SPEC файла

Чтобы собрать `bello`, `pello`, и `cello` из их SPEC файлов, запустите:

```
$ rpmbuild -bb ~/rpmbuild/SPECS/bello.spec  
  
$ rpmbuild -bb ~/rpmbuild/SPECS/pello.spec  
  
$ rpmbuild -bb ~/rpmbuild/SPECS/cello.spec
```

Теперь Вы собрали RPM из SPEC файлов.

Большая часть информации, содержащейся в разделе [Восстановление из исходного RPM](#) применима здесь.

Проверка RPMs на корректность

После создания упаковки хорошо бы проверить её качество. Качество пакета, а не программного обеспечения, поставляемого в нём. Основным инструментом для этого является `rpmlint`. Это улучшает редактируемость RPM и обеспечивает проверку работоспособности и ошибок путем выполнения статического анализа RPM. Эта утилита может проверять бинарные RPM, исходные RPM (SRPMS) и spec файлы, поэтому она полезна на всех этапах упаковки, как показано в следующих примерах.

Обратите внимание, что `rpmlint` имеет очень строгие правила, и иногда допустимо и необходимо пропустить некоторые из его ошибок и предупреждений, как показано в следующих примерах.

NOTE

В примерах мы запускаем `rpmlint` без каких-либо опций, что приводит к невербальному выводу. Для получения подробных объяснений каждой ошибки или предупреждения вместо этого запустите `rpmlint -i` instead.

Проверка SPEC файла bello

Это результат выполнения `rpmlint` в SPEC файле `bello`:

```
$ rpmlint bello.spec  
bello.spec: W: invalid-url Source0: https://www.example.com/bello/releases/bello-
```

```
0.1.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

Наблюдения:

- Для `bello.spec` есть только одно предупреждение. В нем говорится, что URL-адрес, указанный в директиве `Source0` недоступен. Это ожидаемо, поскольку указанный `example.com` URL-адрес не существует. Предполагая, что мы ожидаем, что этот URL-адрес будет работать в будущем, мы можем проигнорировать это предупреждение

Это результат выполнения `rpmlint` на SRPM для `bello`:

```
$ rpmlint ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
bello.src: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.src: W: invalid-url Source0: https://www.example.com/bello/releases/bello-
0.1.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

Наблюдения:

- Для `bello` SRPM появилось новое предупреждение, в котором говорится, что URL-адрес, указанный в директиве `URL`, недоступен. Предполагая, что ссылка будет работать в будущем, мы можем проигнорировать это предупреждение.

Проверка бинарного RPM `bello`

При проверке бинарных RPMs, `rpmlint` проверяет дополнительные параметры, в том числе:

1. документацию
2. [страницы руководства](#)
3. корректность [Иерархии файловой системы](#)

Это результат выполнения `rpmlint` на бинарном RPM для `bello`:

```
$ rpmlint ~/rpmbuild/RPMS/noarch/bello-0.1-1.el7.noarch.rpm
bello.noarch: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.noarch: W: no-documentation
bello.noarch: W: no-manual-page-for-binary bello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

Наблюдения:

- `no-documentation` и `no-manual-page-for-binary` говорят о том, что в RPM нет документации или страниц руководства, потому что мы их не предоставили.

Помимо вышеприведенных предупреждений, наш RPM проходит проверку `rpmlint`.

Проверка SPEC файла pello

Это результат выполнения `rpmlint` на SPEC файле `pello`:

```
$ rpmlint pello.spec
pello.spec:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.spec:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.spec:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.spec:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.spec:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.spec: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.1.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 5 errors, 1 warnings.
```

Наблюдения:

- Предупреждение `invalid-url Source0` говорит о том, что URL-адрес, указанный в директиве `Source0` - недоступен. Это ожидаемо, поскольку указанный `example.com` URL-адрес не существует. Предполагая, что мы ожидаем, что этот URL-адрес будет работать в будущем, мы можем проигнорировать это предупреждение.
- Ошибок много, потому что мы намеренно написали этот файл спецификации, чтобы он был простым и показывал, о каких ошибках может сообщать `rpmlint`.
- Ошибки `hardcoded-library-path` предполагают использование макроса `%{_libdir}` вместо жесткого кодирования пути к библиотеке. Ради этого примера мы игнорируем эти ошибки, но для пакетов, запущенных в производство, Вам нужна веская причина для игнорирования этой ошибки.

Это результат выполнения `rpmlint` на SRPM `pello`:

```
$ rpmlint ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
pello.src: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.src:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.src:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.src:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.src:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.src:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.src: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.1.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 5 errors, 2 warnings.
```

Наблюдения:

- Новая ошибка `invalid-url URL` здесь связана с директивой `URL`, которая недоступна. Предполагая, что мы ожидаем, что URL-адрес станет действительным в будущем, мы можем игнорировать эту ошибку.

Проверка бинарного RPM pello

При проверке бинарного RPMs, `rpmlint` проверяет дополнительные параметры, в том числе:

1. документацию
2. [страницы руководства](#)
3. последовательное использование
4. корректность [Иерархии файловой системы](#)

Это результат выполнения `rpmlint` на бинарном RPM для `pello`:

```
$ rpmlint ~/rpmbuild/RPMS/noarch/pello-0.1.1-1.el7.noarch.rpm
pello.noarch: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.noarch: W: only-non-binary-in-usr-lib
pello.noarch: W: no-documentation
pello.noarch: E: non-executable-script /usr/lib/pello/pello.py 0644L /usr/bin/env
pello.noarch: W: no-manual-page-for-binary pello
1 packages and 0 specfiles checked; 1 errors, 4 warnings.
```

Наблюдения:

- Предупреждения `no-documentation` и `no-manual-page-for-binary` говорят о том, что в RPM нет документации или страниц руководства, потому что мы их не предоставили.
- Предупреждение `only-non-binary-in-usr-lib` гласит, что Вы предоставили только бинарные артефакты `/usr/lib/`. Этот каталог обычно зарезервирован для общих объектных файлов, которые являются бинарными файлами. Следовательно, `rpmlint` ожидает, что по крайней мере один или несколько файлов в `/usr/lib/` будут бинарными.

Это пример проверки `rpmlint` на соответствие [Иерархии Файловой Системы](#) .

Обычно для обеспечения правильного размещения файлов используются макросы RPM. Ради этого примера мы можем проигнорировать это предупреждение.

- Ошибка `non-executable-script` предупреждает о том, что `/usr/lib/pello/pello.py` файл не имеет прав на выполнение. Поскольку этот файл содержит `shebang`, `rpmlint` ожидает, что файл будет исполняемым. Для целей примера оставьте этот файл без разрешений на выполнение и проигнорируйте эту ошибку.

Помимо вышеприведенных предупреждений и ошибок, наш RPM проходит проверку `rpmlint`.

Проверка SPEC файла cello

Это результат выполнения `rpmlint` на SPEC файле `cello`:

```
$ rpmlint ~/rpmbuild/SPECS/cello.spec
/home/admiller/rpmbuild/SPECS/cello.spec: W: invalid-url Source0:
https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

Наблюдения:

- Единственное предупреждение для **cello.spec** гласит, что URL-адрес, указанный в директиве **Source0**, недоступен. Это ожидаемо, поскольку указанный **example.com** URL-адрес не существует. Предполагая, что мы ожидаем, что этот URL-адрес будет работать в будущем, мы можем проигнорировать это предупреждение.

Это результат выполнения **rpmlint** в файле SRPM для **cello**:

```
$ rpmlint ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
cello.src: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.src: W: invalid-url Source0: https://www.example.com/cello/releases/cello-
1.0.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

Наблюдения:

- Для **cello** SRPM появилось новое предупреждение, в котором говорится, что URL-адрес, указанный в директиве **URL**, недоступен. Предполагая, что ссылка будет работать в будущем, мы можем проигнорировать это предупреждение.

Проверка бинарного RPM cello

При проверке бинарных RPMs, **rpmlint** проверяет дополнительные параметры, в том числе:

1. документацию
2. [страницы руководства](#)
3. корректность [Иерархии файловой системы](#).

Это результат выполнения **rpmlint** на бинарном RPM для **cello**:

```
$ rpmlint ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el7.x86_64.rpm
cello.x86_64: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.x86_64: W: no-documentation
cello.x86_64: W: no-manual-page-for-binary cello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

Наблюдения:

- Предупреждения **no-documentation** и **no-manual-page-for-binary** говорят о том, что в RPM нет

документации или страниц руководства, потому что мы их не предоставили.

Помимо вышеприведенных предупреждений и ошибок, наш RPM проходит проверку `rpmlint`.

Наши RPM теперь готовы и проверены с помощью `rpmlint`. На этом учебное пособие заканчивается. Для получения дополнительной информации о RPM упаковке перейдите к главе [Дополнительные материалы](#).

Дополнительные материалы

В этой главе рассматриваются темы, которые выходят за рамки вводного руководства, но часто полезны в реальной упаковке RPM.

Подпись пакетов

Подпись пакета - это способ защитить пакет для конечного пользователя. Безопасная транспортировка может быть достигнута с помощью реализации протокола HTTPS. Такой метод используют, когда пакет загружается непосредственно перед установкой. Однако пакеты часто загружаются заранее и хранятся в локальных репозиториях перед их использованием. Пакеты подписываются, чтобы гарантировать, что никакая третья сторона не сможет изменить содержимое пакета.

Существует три способа подписи пакета:

- [Добавление подписи к уже существующему пакету.](#)
- [Замена подписи на уже существующем пакете.](#)
- [Подпись пакета во время сборки.](#)

Добавление подписи к пакету

В большинстве случаев пакеты создаются без подписи. Подпись добавляется непосредственно перед выпуском пакета.

Чтобы добавить другую подпись к пакету, используйте опцию `--addsign`. Наличие более чем одной подписи позволяет зафиксировать путь владения пакетом от разработчика пакета до конечного пользователя.

В качестве примера подразделение компании создает пакет и подписывает его ключом подразделения. Затем штаб-квартира компании проверяет подпись пакета и добавляет корпоративную подпись к пакету, заявляя, что подписанный пакет является подлинным.

С двумя подписями пакет попадает к продавцу. Продавец проверяет подписи и, если они проверяются, также добавляет свою подпись.

Теперь пакет отправляется в компанию, которая желает его развернуть. Проверив каждую подпись на упаковке, они знают, что это подлинная копия, не изменившаяся с момента её первого создания. В зависимости от внутреннего контроля внедряющей компании, они могут добавить свою собственную подпись, чтобы заверить своих сотрудников в том, что пакет получил их корпоративное одобрение.

Вывод из команды `--addsign`:

```
$ rpm --addsign blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
```



```
blather-7.9-1.i386.rpm:
```

Для проверки подписей пакета с несколькими подписями:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp md5 OK
```

Два обозначения **pgp** в выходных данных команды **rpm --checksig** показывают, что пакет был подписан дважды.

RPM позволяет добавлять одну и ту же подпись несколько раз. Параметр **--addsign** не проверяет наличие нескольких идентичных подписей.

```
$ rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp pgp md5 OK
```

На выходе команды **rpm --checksig** отображается четыре подписи.

Замена подписи пакета

Чтобы изменить открытый ключ без необходимости пересобирать каждый пакет, используйте опцию **--resign**.

```
$ rpm --resign blather-7.9-1.i386.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
```

Использование опции **--resign** с несколькими пакетами:

```
$ rpm --resign b*.rpm
Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
bother-3.5-1.i386.rpm:
```

Подпись во время сборки

Чтобы подписать пакет во время сборки, используйте команду `rpmbuild` с параметром `--sign`. Для этого необходимо ввести кодовую фразу PGP.

Для примера:

```
$ rpmbuild -ba --sign blather-7.9.spec
Enter pass phrase:

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
Finding dependencies...
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
```

Сообщение "Generating signature" появляется как в бинарном, так и в исходном разделах упаковки. Число, следующее за сообщением, указывает на то, что добавленная подпись была создана с использованием PGP.

NOTE

При использовании опции `--sign` в `rpmbuild`, используйте только аргументы `-bb` или `-ba` для сборки пакета. Аргумент `-ba` обозначает сборку бинарных и исходных пакетов.

Чтобы проверить подпись пакета, используйте команду `--checksig`. Для примера:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp md5 OK
```

Сборка нескольких пакетов

При сборке нескольких пакетов используйте следующий синтаксис, чтобы избежать многократного ввода кодовой фразы PGP. Например, при сборке пакетов **blather** и **bother**, подпишите их, следуя примеру ниже:

```
$ rpmbuild -ba --sign b*.spec
    Enter pass phrase:

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
...
* Package: bother
...
Binary Packaging: bother-3.5-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/bother-3.5-1.i386.rpm
...
Source Packaging: bother-3.5-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/bother-3.5-1.src.rpm
```

Mock

Mock - это инструмент для сборки пакетов. Он может создавать пакеты для разных архитектур и разных версий Fedora или RHEL. Mock создает chroots и собирает в них пакеты. Его единственная задача - надежно заполнить chroot и попытаться собрать пакет в этом chroot.

Mock также предлагает многопакетный инструмент **mockchain**, который может собирать цепочки пакетов, зависящих друг от друга.

Mock способен создавать RPM из управления конфигурацией исходного кода, если присутствует пакет **mock-scm**, а затем встраивать SRPM в RPMs. Смотрите **scm-enable** в документации. (Из upstream документации)

NOTE

Чтобы использовать **Mock** в системе RHEL или CentOS, Вам необходимо

включить репозиторий “Extra Packages for Enterprise Linux” (EPEL) . Это репозиторий, предоставляемый сообществом Fedora, содержащий множество полезных инструментов для пакетов RPM, системных администраторов и разработчиков.

Одним из наиболее распространенных вариантов для RPM-упаковщиков, использующих Mock , является создание так называемой “нетронутой среды сборки”. При использовании mock в качестве “нетронутой среды сборки”, ничто в текущем состоянии вашей системы не влияет на сам пакет RPM. Mock использует различные конфигурации, чтобы указать, какова “цель” сборки, они находятся в Вашей системе в каталоге `/etc/mock/` (после установки пакета `mock`). Вы можете выполнить сборку для разных дистрибутивов или выпусков, просто указав это в командной строке. Следует иметь в виду, что файлы конфигурации, поставляемые с макетом, предназначены для упаковщиков Fedora RPM, и поэтому выпускаемые версии RHEL и CentOS помечены как “epel” , потому что это «целевой» репозиторий, для которого эти RPM пакеты будут созданы. Вы просто указываете конфигурацию, которую хотите использовать (без расширения файла `.cfg`). Например, вы можете создать наш пример `cello` как для RHEL 7, так и для Fedora 23, используя следующие команды, даже не используя разные машины.

```
$ mock -r epel-7-x86_64 ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm

$ mock -r fedora-23-x86_64 ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
```

Один из примеров того, почему Вы можете захотеть использовать `mock` - это если вы упаковывали RPMs на своем ноутбуке, и у Вас был установлен пакет (в этом примере мы назовем его `foo`), который был указан в секции `BuildRequires` того пакета, который Вы собирали, но забыли фактически сделать запись `BuildRequires: foo`. Сборка завершится успешно, когда вы запустите `rpmbuild`, потому что `foo` был необходим для сборки, и он был найден в системе во время сборки. Однако, если Вы перенесёте SRPM в другую систему, в которой отсутствовал `foo`, он выйдет из строя, что вызовет неожиданный побочный эффект. `Mock` решает эту проблему, сначала анализируя содержимое SRPM и устанавливая `BuildRequires` в его `chroot`, что означает, что если бы Вам не хватало записи `BuildRequires` , сборка завершилась бы с ошибкой, потому что `mock` не знал бы, как её установить, и поэтому она не присутствовала бы в `buildroot`.

Другой пример - противоположный сценарий, допустим, Вам нужен `gcc` для сборки пакета, но он не установлен в вашей системе (что маловероятно для RPM-упаковщика, но просто ради примера давайте притворимся, что это правда). С `Mock`, Вам не нужно устанавливать `gcc` в Вашей системе, потому что он будет установлен в `chroot` как часть процесса `mock`.

Ниже приведен пример попытки пересобрать пакет, у которого есть зависимость, которой мне не хватает в моей системе. Главное, что следует отметить, это то, что `gcc` обычно используется в большинстве систем RPM упаковщиками, некоторые пакеты RPM могут содержать более дюжины сборочных запросов, и это позволяет Вам не загромождать свою рабочую станцию ненужными пакетами.

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
```

```

Installing /home/admillier/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
error: Failed build dependencies: gcc is needed by cello-1.0-1.el7.x86_64

$ mock -r epel-7-x86_64 ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
INFO: mock.py version 1.2.17 starting (python version = 2.7.5)...
Start: init plugins
INFO: selinux enabled
Finish: init plugins
Start: run
INFO: Start(/home/admillier/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm) Config(epel-7-
x86_64)
Start: clean chroot
Finish: clean chroot
Start: chroot init
INFO: calling preinit hooks
INFO: enabled root cache
Start: unpacking root cache
Finish: unpacking root cache
INFO: enabled yum cache
Start: cleaning yum metadata
Finish: cleaning yum metadata
Mock Version: 1.2.17
INFO: Mock Version: 1.2.17
Start: yum update
base | 3.6 kB
00:00:00
epel | 4.3 kB
00:00:00
extras | 3.4 kB
00:00:00
updates | 3.4 kB
00:00:00
No packages marked for update
Finish: yum update
Finish: chroot init
Start: build phase for cello-1.0-1.el7.src.rpm
Start: build setup for cello-1.0-1.el7.src.rpm
warning: Could not canonicalize hostname: rhel7
Building target platforms: x86_64
Building for target x86_64
Wrote: /builddir/build/SRPMS/cello-1.0-1.el7.centos.src.rpm
Getting requirements for cello-1.0-1.el7.centos.src
--> Already installed : gcc-4.8.5-4.el7.x86_64
--> Already installed : 1:make-3.82-21.el7.x86_64
No uninstalled build requires
Finish: build setup for cello-1.0-1.el7.src.rpm
Start: rpmbuild cello-1.0-1.el7.src.rpm
Building target platforms: x86_64
Building for target x86_64
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.v9rPOF
+ umask 022

```

```

+ cd /builddir/build/BUILD
+ cd /builddir/build/BUILD
+ rm -rf cello-1.0
+ /usr/bin/gzip -dc /builddir/build/SOURCES/cello-1.0.tar.gz
+ /usr/bin/tar -xf -
+ STATUS=0
+ '[' 0 -ne 0 ']'
+ cd cello-1.0
+ /usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
Patch #0 (cello-output-first-patch.patch):
+ echo 'Patch #0 (cello-output-first-patch.patch):'
+ /usr/bin/cat /builddir/build/SOURCES/cello-output-first-patch.patch
patching file cello.c
+ /usr/bin/patch -p0 --fuzz=0
+ exit 0
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.UxRVtI
+ umask 022
+ cd /builddir/build/BUILD
+ cd cello-1.0
+ make -j2
gcc -g -o cello cello.c
+ exit 0
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.K3i2dL
+ umask 022
+ cd /builddir/build/BUILD
+ '[' /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64 '!=' / ']'
+ rm -rf /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
++ dirname /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
+ mkdir -p /builddir/build/BUILDROOT
+ mkdir /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
+ cd cello-1.0
+ /usr/bin/make install DESTDIR=/builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
mkdir -p /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64/usr/bin
install -m 0755 cello /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64/usr/bin/cello
+ /usr/lib/rpm/find-debuginfo.sh --strict-build-id -m --run-dwz --dwz-low-mem-die -limit 10000000 --dwz-max-die-limit 110000000 /builddir/build/BUILD/cello-1.0
extracting debug info from /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64/usr/bin/cello
dwz: Too few files for multifile optimization
/usr/lib/rpm/sepdebugcrcfix: Updated 0 CRC32s, 1 CRC32s did match.
+ /usr/lib/rpm/check-buildroot
+ /usr/lib/rpm/redhat/brp-compress
+ /usr/lib/rpm/redhat/brp-strip-static-archive /usr/bin/strip
+ /usr/lib/rpm/brp-python-bytecompile /usr/bin/python 1
+ /usr/lib/rpm/redhat/brp-python-hardlink
+ /usr/lib/rpm/redhat/brp-java-repack-jars
Processing files: cello-1.0-1.el7.centos.x86_64
Executing(%license): /bin/sh -e /var/tmp/rpm-tmp.vxtAu0
+ umask 022

```

```

+ cd /builddir/build/BUILD
+ cd cello-1.0
+ LICENSEDIR=/builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/share/licenses/cello-1.0
+ export LICENSEDIR
+ /usr/bin/mkdir -p /builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/share/licenses/cello-1.0
+ cp -pr LICENSE /builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/share/licenses/cello-1.0
+ exit 0
Provides: cello = 1.0-1.el7.centos cello(x86-64) = 1.0-1.el7.centos
Requires(rpmlib): rpmlib(CompressedFileNames) <= 3.0.4-1 rpmlib(FileDigests) <= 4.6.0-
1 rpmlib(PayloadFilesHavePrefix) <= 4.0-1
Requires: libc.so.6()(64bit) libc.so.6(GLIBC_2.2.5)(64bit) rtld(GNU_HASH)
Processing files: cello-debuginfo-1.0-1.el7.centos.x86_64
Provides: cello-debuginfo = 1.0-1.el7.centos cello-debuginfo(x86-64) = 1.0-
1.el7.centos
Requires(rpmlib): rpmlib(FileDigests) <= 4.6.0-1 rpmlib(PayloadFilesHavePrefix) <=
4.0-1 rpmlib(CompressedFileNames) <= 3.0.4-1
Checking for unpackaged file(s): /usr/lib/rpm/check-files
/builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
Wrote: /builddir/build/RPMS/cello-1.0-1.el7.centos.x86_64.rpm
warning: Could not canonicalize hostname: rhel7
Wrote: /builddir/build/RPMS/cello-debuginfo-1.0-1.el7.centos.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.JuPOtY
+ umask 022
+ cd /builddir/build/BUILD
+ cd cello-1.0
+ /usr/bin/rm -rf /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
+ exit 0
Finish: rpmbuild cello-1.0-1.el7.src.rpm
Finish: build phase for cello-1.0-1.el7.src.rpm
INFO: Done(/home/admillier/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm) Config(epel-7-
x86_64) 0 minutes 16 seconds
INFO: Results and/or logs in: /var/lib/mock/epel-7-x86_64/result
Finish: run

```

Как Вы можете видеть, **mock** - довольно подробный инструмент. Вы также заметите много выходных данных **yum** или **dnf** (в зависимости от фиктивной цели RHEL7, CentOS7 или Fedora), которых нет в этом выводе, который был опущен для краткости и часто опускается после того, как Вы выполнили **--init** для mock target. Например **mock -r epel-7-x86_64 --init**, который предварительно загрузит все необходимые пакеты, закеширует их и запустит предварительный этап сборки chroot.

Для получения дополнительной информации, пожалуйста, обратитесь к [Mock upstream](#) документации.

Система контроля версий

При работе с RPMs желательно использовать [Системы контроля версий](#) (VCS), такую как [git](#), для управления компонентами программного обеспечения, которое мы упаковываем. Следует отметить, что хранение бинарных файлов в системе контроля версий нецелесообразно, поскольку это резко увеличивает размер исходного репозитория, поскольку эти инструменты разработаны для обработки различий в файлах (часто оптимизированных для текстовых файлов), чему не поддаются бинарные файлы, поэтому обычно сохраняется весь бинарный файл целиком. Существуют некоторые утилиты, популярные среди upstream проектов с открытым исходным кодом, которые решают эту проблему, либо сохраняя SPEC файл, где исходный код находится в VCS (т. е. - он не находится в сжатом архиве для распространения), либо помещая в VCS только SPEC-файл и патчи и загружается сжатый архив upstream исходного кода в так называемый «кэш просмотра».

В этом разделе мы рассмотрим два различных варианта использования системы контроля версий [git](#) для управления содержимым, которое в конечном итоге будет преобразовано в пакет RPM. Первый называется [tito](#), второй - [dist-git](#).

NOTE

Вам нужно будет установить пакет [git](#) в Вашу систему, он понадобится нам для изучения данного раздела.

tito

- это утилита, которая предполагает, что весь исходный код программного обеспечения, которое будет упаковано, уже находится в репозитории [git](#). Это хорошо для тех, кто практикует рабочий процесс DevOps, поскольку позволяет команде, пишущей программное обеспечение, поддерживать свой нормальный [рабочий процесс ветвления](#). Затем Tito позволит поэтапно упаковывать программное обеспечение, собирать его в автоматическом режиме и по-прежнему обеспечивать собственный процесс установки для системы на основе RPM [RPM](#).

NOTE

Пакет [tito](#) доступен в [Fedora](#), а также в репозитории [EPEL](#) для использования на RHEL 7 и CentOS 7.

Tito работает на основе тегов [git tags](#) и будет управлять тегами для Вас, если Вы решите разрешить это, но при желании может работать по любой схеме тегов, которую Вы предпочитаете, поскольку эта функциональность настраивается.

Давайте немного познакомимся с tito, взглянув на исходный проект, который уже использует его. На самом деле мы будем использовать репозиторий [git](#) проекта, который является предметом нашего следующего раздела, [dist-git](#). Поскольку этот проект публично размещен на [GitHub](#), давайте клонируем репозиторий [git](#).

```
$ git clone https://github.com/release-engineering/dist-git.git
Cloning into 'dist-git'...
remote: Counting objects: 425, done.
remote: Total 425 (delta 0), reused 0 (delta 0), pack-reused 425
Receiving objects: 100% (425/425), 268.76 KiB | 0 bytes/s, done.
```



```
Resolving deltas: 100% (184/184), done.  
Checking connectivity... done.
```

```
$ cd dist-git/  
  
$ ls *.spec  
dist-git.spec  
  
$ tree rel-eng/  
rel-eng/  
├── packages  
│   └── dist-git  
└── tito.props  
  
1 directory, 2 files
```

Как мы видим, SPEC файл находится в корне репозитория git, и в репозитории есть каталог **rel-eng**, который используется tito для общего учета, настройки и различных дополнительных тем, таких как пользовательские модули tito. В макете каталога мы видим, что есть подкаталог с названием **packages**, в котором будет храниться файл для каждого пакета, которым tito управляет в репозитории, поскольку у Вас может быть много RPM в одном репозитории git, и tito справится с этим просто отлично. Однако, в этом сценарии мы видим только один список пакетов, и следует отметить, что он соответствует имени нашего SPEC файла. Все это настраивается командой **tito init**, когда разработчики **dist-git** впервые инициализировали свой репозиторий git для управления tito.

Если бы мы следовали обычному рабочему процессу DevOps Practitioner, мы, вероятно, хотели бы использовать его как часть процесса **Непрерывной интеграции** (CI) или **Непрерывной доставки** (CD). Что мы можем сделать в этом сценарии, так это выполнить “test build” для tito, мы даже можем использовать mock. Затем мы могли бы использовать выходные данные в качестве точки установки для какого-либо другого компонента в конвейере. Ниже приведен простой пример команд, которые могут это сделать, и их можно адаптировать к другим средам.

```
$ tito build --test --srpm  
Building package [dist-git-0.13-1]  
Wrote: /tmp/tito/dist-git-git-0.efa5ab8.tar.gz  
  
Wrote: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm  
  
$ tito build --builder=mock --arg mock=epel-7-x86_64 --test --rpm  
Building package [dist-git-0.13-1]  
Creating rpms for dist-git-git-0.efa5ab8 in mock: epel-7-x86_64  
Wrote: /tmp/tito/dist-git-git-0.efa5ab8.tar.gz  
  
Wrote: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm  
  
Using srpm: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm  
Initializing mock...
```

```
Installing deps in mock...
Building RPMs in mock...
Wrote:
  /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm
  /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm

$ sudo yum localinstall /tmp/tito/dist-git-*.noarch.rpm
Loaded plugins: product-id, search-disabled-repos, subscription-manager
Examining /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm: dist-git-
0.13-1.git.0.efa5ab8.el7.centos.noarch
Marking /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm to be installed
Examining /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm: dist-
git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch
Marking /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm to be
installed
Resolving Dependencies
--> Running transaction check
---> Package dist-git.noarch 0:0.13-1.git.0.efa5ab8.el7.centos will be installed
```

Обратите внимание, что последняя команда должна быть запущена либо с правами `sudo`, либо с правами `root`, и что большая часть выходных данных была опущена для краткости, поскольку список зависимостей довольно длинный.

На этом наш простой пример использования `tito` заканчивается, но в нём есть много удивительных функций для системных администраторов, разработчиков RPM-пакетов и практиков DevOps. Я бы настоятельно рекомендовал ознакомиться с upstream документацией, найденной на сайте `tito` GitHub, для получения дополнительной информации о том, как быстро начать использовать его для вашего проекта, а также о различных дополнительных функциях, которые он предлагает.

dist-git

Утилита `dist-git` использует несколько иной подход, чем у `tito`, так что вместо того, чтобы хранить исходный код в `git`, она вместо этого будет хранить файлы спецификаций и патчи в репозитории `git` и загружать сжатый архив исходного кода в так называемый “look-aside cache”. “Look-aside-cache” - это термин, который был придуман при использовании систем сборки RPM, хранящих большие файлы, подобные этим, “на стороне”. Подобная система, как правило, привязана к правильной системе сборки RPM, такой как `Koji`. Затем система сборки настраивается на извлечение элементов, которые перечислены в качестве записей `SourceX` в файлах спецификаций из этого внешнего кэша, в то время как SPEC файл и исправления остаются в системе контроля версий. Существует также вспомогательный инструмент командной строки, который поможет в этом.

Чтобы не дублировать документацию для получения дополнительной информации о том, как настроить такую систему, пожалуйста, обратитесь к документации `dist-git`.

Подробнее о макросах

Существует множество встроенных макросов RPM, и мы рассмотрим некоторые из них в следующем разделе, однако исчерпывающий список можно найти на странице [RPM Official Documentation](#).

Существуют также макросы, предоставляемые Вашим дистрибутивом [Linux](#), в этом разделе мы рассмотрим некоторые из них, предоставляемые [Fedora](#), [CentOS](#) и [RHEL](#), а также предоставим информацию о том, как проверить Вашу систему, чтобы узнать о других, которые мы не рассматриваем, или для их обнаружения в других дистрибутивах Linux на основе RPM

Определение Ваших Собственных Макросов

Вы можете определить свои собственные макросы. Ниже приводится выдержка из [RPM Official Documentation](#), в которой содержится исчерпывающая информация о возможностях макросов.

Чтобы определить макрос, используйте:

```
%global <name>[(opts)] <body>
```

Все пробелы, окружающие `\`, удаляются. Имя может состоять из буквенно-цифровых символов и символа `_`, и должно иметь длину не менее 3 символов. Макрос без поля `(opts)` является “простым” в том смысле, что выполняется только рекурсивное расширение макроса. Параметризованный макрос содержит поле `(opts)` field. The `opts` - (строка в круглых скобках) передается точно так же, как и в `getopt(3)` для обработки `argc/argv` в начале вызова макроса.

NOTE

Более старый SPEC файлы RPM могут использовать шаблон макроса `%define <name> <body>`. Различия между макросами `%define` и `%global` заключаются в следующем:

- `%define` имеет локальную область действия, что означает, что он применяется только к указанной части SPEC файла. Кроме того, тело макроса `%define` расширяется при использовании.
- `%global` имеет глобальную область действия, что означает, что он применяется ко всему SPEC файлу. Кроме того, тело макроса `%global` асшируется во время определения.

Пример:

```
%global githash 0ec4e58
%global python_sitelib %(%{__python} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())")
```

NOTE

Макросы всегда оцениваются, даже в комментариях. Иногда это безобидно. Но во втором примере мы выполняем команду `python`, чтобы получить содержимое макроса. Эта команда будет выполняться даже тогда, когда Вы закоментируете макрос, или когда Вы вводите имя макроса в `%changelog`. Чтобы закоментировать макрос, используйте `%%`. Например: `%%global`.

%setup

Макрос `%setup` можно использовать для сборки пакета с помощью `tarball`. Стандартное поведение макроса `%setup` можно увидеть в выходных данных `rpmbuild`. В начале каждой фазы макрос выводит `Executing(%something)`. Например:

```
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.DhddsG
```

Выходные данные оболочки устанавливаются с включенным `set -x`. Чтобы просмотреть содержимое `/var/tmp/rpm-tmp.DhddsG`, используйте опцию `--debug`, поскольку `rpmbuild` удаляет временные файлы после успешной сборки. Здесь отображается настройка переменных среды, например:

```
cd '/builddir/build/BUILD'
rm -rf 'cello-1.0'
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xof -
STATUS=$?
if [ $STATUS -ne 0 ]; then
    exit $STATUS
fi
cd 'cello-1.0'
/usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
```

Макрос `%setup` гарантирует, что мы работаем в правильном каталоге, удаляет остатки предыдущих сборок, распаковывает исходный архив и устанавливает некоторые привилегии по умолчанию. Существует несколько вариантов настройки поведения макроса `%setup`.

%setup -q

Параметр `-q` ограничивает детализацию макроса `%setup`. Вместо `tar -xof` выполняется только `tar -xvovf`. Этот параметр должен быть использован в качестве первого.

%setup -n

В некоторых случаях каталог из расширенного архива имеет другое имя, чем ожидалось `%{name}-%{version}`. Это может привести к ошибке макроса `%setup`. Имя каталога должно быть указано параметром `-n directory_name`.

Например, если имя пакета `cello`, но исходный код заархивирован в `hello-1.0.tgz` и содержит каталог `hello/`, содержимое SPEC файла должно быть следующим:

```
Name: cello
Source0: https://example.com/{name}/release/hello-{version}.tar.gz
...
%prep
%setup -n hello
```

%setup -c

Параметр **-c** можно использовать, если архив исходного кода не содержит никаких подкаталогов и после распаковки файлы из архива заполняют текущий каталог. Опция **-c** создает каталог и переходит к расширению архива. Наглядный пример:

```
/usr/bin/mkdir -p cello-1.0
cd 'cello-1.0'
```

Каталог не изменяется после расширения архива.

%setup -D и -T

Параметр **-D** отключает удаление каталога исходного кода. Этот параметр полезен, если макрос **%setup** используется несколько раз. По сути, параметр **-D** означает, что следующие строки не используются:

```
rm -rf 'cello-1.0'
```

Параметр **-T** отключает расширение хранилища исходного кода, удаляя следующую строку из скрипта:

```
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xvof -
```

%setup -a и -b

Параметры **-a** и **-b** расширяют определённые источники.

- Параметр **-b** (расшифровывается как **before**) расширяет определенные источники перед входом в рабочий каталог.
- Параметр **-a** (расшифровывается как **after**) расширяет эти источники после входа. Их аргументами являются исходные номера из преамбулы файла спецификации.

Например, допустим, что архив **cello-1.0.tar.gz** содержит пустой каталог **examples**, и примеры поставляются в отдельных **examples.tar.gz** tarball архивах, и они разархивируются в каталог с тем же именем. В этом случае используйте **-a 1**, так как мы хотим разархивировать **Source1** после входа в рабочий каталог:

```
Source0: https://example.com/{name}/release/{name}-{version}.tar.gz
```

```
Source1: examples.tar.gz
...
%prep
%setup -a 1
```

Но если бы примеры были в отдельном `cello-1.0-examples.tar.gz` tarball архиве, который расширяется до `cello-1.0/examples`, используйте параметры `-b 1`, поскольку `Source1` должен быть разархивирован перед входом в рабочий каталог:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: %{name}-%{version}-examples.tar.gz
...
%prep
%setup -b 1
```

Вы также можете использовать комбинацию всех этих опций.

%files

Общие “расширенные” макросы RPM, необходимые в разделе `%files`:

Макрос	Описание
<code>%license</code>	Макрос идентифицирует файл, указанный в списке, как файл ЛИЦЕНЗИИ, и он будет установлен и помечен как таковой RPM. Пример: <code>%license LICENSE</code>
<code>%doc</code>	Этот макрос идентифицирует файл, указанный как документация, и он будет установлен и помечен RPM как таковой. Это часто используется не только для документации об упаковываемом программном обеспечении, но и для примеров кода и различных элементов, которые должны сопровождать документацию. Пример: <code>%doc README</code>
<code>%dir</code>	Макрос указывает, что путь является каталогом, которым должен владеть этот RPM. Это важно, чтобы манифест RPM-файла точно знал, какие каталоги очищать при удалении. Пример: <code>%dir %{_libdir}/%{name}</code>
<code>%config(noreplace)</code>	Указывает, что следующий файл является файлом конфигурации и поэтому не должен перезаписываться (или заменяться) при установке или обновлении пакета, если файл был изменен по сравнению с исходной контрольной установкой. В случае внесения изменений файл будет создан с добавлением <code>.rpmnew</code> в конец имени файла при обновлении или установке, чтобы ранее существующий или измененный файл в целевой системе не был изменен. Пример: <code>%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf</code>

Встроенные макросы

В Вашей системе есть много встроенных макросов RPM, и самый быстрый способ просмотреть их все - это просто выполнить команду `rpm --showrc`. Обратите внимание, что

это будет содержать много выходных данных, поэтому его часто используют в сочетании с каналом для `grep`.

Вы также можете найти информацию о макросах RPM, которые поставляются непосредственно с версией RPM Вашей системы, просмотрев выходные данные `rpm -ql rpm`, обратив внимание на файлы с названием `macros` в структуре каталогов.

RPM Макросы, предоставляемые дистрибутивом

Различные дистрибутивы будут предоставлять разные наборы рекомендуемых макросов RPM в зависимости от языковой реализации упаковываемого программного обеспечения или конкретных рекомендаций рассматриваемого дистрибутива.

Они часто предоставляются в виде самих пакетов RPM и могут быть установлены с помощью пакетного менеджера, такого как `yum` или `dnf`. Сами файлы макросов после установки можно найти в `/usr/lib/rpm/macros.d/`, и они будут включены в вывод `rpm --showrc` по умолчанию после установки.

Одним из основных примеров этого является раздел [Fedora Packaging Guidelines](#), относящийся конкретно к [Application Specific Guidelines](#), который на момент написания этой статьи содержит более 60 различных наборов руководств вместе с соответствующими наборами макросов RPM для упаковки.

Одним из примеров такого рода RPM может быть `Python` версии 2.x, и если у нас установлен пакет `python2-rpm-macros` (доступный в EPEL для RHEL 7 и CentOS 7), у нас есть ряд доступных, специфичных для python2 макросов.

```
$ rpm -ql python2-rpm-macros
/usr/lib/rpm/macros.d/macros.python2

$ rpm --showrc | grep python2
-14: __python2 /usr/bin/python2
CFLAGS="%{optflags}" %{__python2} %{py_setup} %{?py_setup_args} build
--executable="%{__python2} %{py2_shbang_opts}" %{?1}
CFLAGS="%{optflags}" %{__python2} %{py_setup} %{?py_setup_args} install -O1 --skip
-build --root %{buildroot} %{?1}
-14: python2_sitearch   %{__python2} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib(1))"
-14: python2_sitelib   %{__python2} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())"
-14: python2_version   %{__python2} -c "import sys;
sys.stdout.write('{0.major}.{0.minor}'.format(sys.version_info))"
-14: python2_version_nodots   %{__python2} -c "import sys;
sys.stdout.write('{0.major}{0.minor}'.format(sys.version_info))"
```

В приведенном выше выводе отображаются необработанные определения макросов RPM, но нас, вероятно, больше интересует, что мы можем сделать с помощью `rpm --eval`, чтобы определить, что они делают, а также как они могут быть полезны для нас при упаковке RPMs.

```
$ rpm --eval %{__python2}
/usr/bin/python2

$ rpm --eval %{python2_sitedir}
/usr/lib64/python2.7/site-packages

$ rpm --eval %{python2_sitelib}
/usr/lib/python2.7/site-packages

$ rpm --eval %{python2_version}
2.7

$ rpm --eval %{python2_version_nodots}
27
```

Пользовательские макросы

Вы можете переопределить макросы в файле `~/.rpmmacros`. Любые внесенные вами изменения повлияют на каждую сборку на Вашем компьютере.

Существует несколько макросов, которые Вы можете использовать для переопределения

`%_topdir /opt/some/working/directory/rpmbuild`

Вы можете создать этот каталог, включая все подкаталоги, с помощью утилиты `rpmdev-setuptree`. Значение этого макроса по умолчанию равно `~/rpmbuild`.

`%_smp_mflags -l3`

Этот макрос часто используется для передачи в Makefile, например: `make %{?_smp_mflags}`, и для задания количества одновременных процессов на этапе сборки. По умолчанию для него задано значение `-jX`, где X - количество ядер. Если Вы измените количество ядер, Вы можете ускорить или замедлить сборку пакетов.

Хотя Вы можете определить любые новые макросы в файле `~/.rpmmacros` это не рекомендуется, поскольку эти макросы не будут присутствовать на других компьютерах, где пользователи могут захотеть попытаться пересобрать Ваш пакет.

Epoch, Scriptlets, and Triggers

В мире SPEC файлов RPM существуют различные разделы, которые считаются продвинутыми, поскольку они влияют не только на файл спецификации, способ сборки пакета, но и на конечный компьютер, на который устанавливается результирующий RPM. В этом разделе мы рассмотрим наиболее распространенные из них, такие как Epoch, Скриптлеты и триггеры.

Epoch

Первым в списке стоит **Epoch**, epoch - это способ определения взвешенных зависимостей на

основе номеров версий. Его значение по умолчанию равно 0, если директива `Epoch` не указана в SPEC файле. Это не рассматривалось в разделе "SPEC файл" этого руководства, потому что почти всегда вводить значения Epoch - плохая идея, поскольку это искажает то, что вы обычно ожидаете от RPM при сравнении версий пакетов.

Например, если был установлен пакет `foobar` с `Epoch: 1` и `Version: 1.0`, а кто-то другой упаковал `foobar` с `Version: 2.0`, но просто опустил директиву `Epoch` либо потому, что они не знали о её необходимости, либо просто забыли, эта новая версия никогда не будет считаться обновлением, потому что версия Epoch превалирует над традиционным маркером Name-Version-Release, который означает управление версиями для RPM-пакетов.

Этот подход обычно используется только в случае крайней необходимости для решения проблемы с порядком обновления, которая может возникнуть как побочный эффект upstream программного обеспечения, изменяющего схемы нумерации версий или версии, включающие буквенные символы, которые не всегда можно надёжно сравнить на основе кодирования.

Scriptlets and Triggers

В пакетах RPM существует ряд директив, которые можно использовать для внесения необходимых или желаемых изменений в систему во время установки RPM. Они называются **scriptlets**.

Один из основных примеров того, когда и почему Вы хотели бы это сделать, это когда установлена системная служба RPM и она предоставляет `systemd` файл. Во время установки нам нужно будет уведомить `systemd` о появлении нового модуля, чтобы системный администратор мог выполнить команду, аналогичную `systemctl start foo.service` после установки вымышленного `foo` (который в этом примере предоставляет демон). Аналогично, нам нужно было бы отменить это действие при деинсталляции, чтобы администратор не получал ошибок из-за того, что бинарный файл демона больше не установлен, но файл модуля все еще существует в запущенной конфигурации `systemd`.

Существует небольшая горстка распространенных скриптлетов, они похожи на “заголовки разделов”, такие как `%build` or `%install`, в том смысле, что они определяются многострочными сегментами кода, часто написанными, как стандартный сценарий оболочки `POSIX`, но могут быть на нескольких разных языках программирования. Исчерпывающий список этих доступных языков можно найти в *Официальной документации RPM*.

Следующие скриптлеты:

Директива	Описание
<code>%pre</code>	Скриптлет, который выполняется непосредственно перед установкой пакета в целевую систему.
<code>%post</code>	Скриптлет, который выполняется сразу после установки пакета в целевой системе.
<code>%preun</code>	Скриптлет, который выполняется непосредственно перед удалением пакета из целевой системы.

<code>%postun</code>	Скриптлет, который выполняется сразу после удаления пакета из целевой системы.
----------------------	--

Также часто для этой функции существуют макросы RPM. В нашем предыдущем примере мы обсуждали необходимость получения [systemd](#) уведомления о новом [unit file](#), , это легко обрабатывается макросами скриптлетов systemd, как Мы можем видеть из приведенного ниже примера вывода. Более подробную информацию об этом можно найти в [Fedora systemd Packaging Guidelines](#).

```
$ rpm --showrc | grep systemd
-14: __transaction_systemd_inhibit      %{__plugindir}/systemd_inhibit.so
-14: _journalcatalogdir /usr/lib/systemd/catalog
-14: _presetdir /usr/lib/systemd/system-preset
-14: _unitdir /usr/lib/systemd/system
-14: _userunitdir /usr/lib/systemd/user
/usr/lib/systemd/systemd-binfmt %{?*} >/dev/null 2>&1 || :
/usr/lib/systemd/systemd-sysctl %{?*} >/dev/null 2>&1 || :
-14: systemd_post
-14: systemd_postun
-14: systemd_postun_with_restart
-14: systemd_preun
-14: systemd_requires
Requires(post): systemd
Requires(preun): systemd
Requires(postun): systemd
-14: systemd_user_post %systemd_post --user --global %{?*}
-14: systemd_user_postun      %{nil}
-14: systemd_user_postun_with_restart  %{nil}
-14: systemd_user_preun
systemd-sysusers %{?*} >/dev/null 2>&1 || :
echo %{?*} | systemd-sysusers - >/dev/null 2>&1 || :
systemd-tmpfiles --create %{?*} >/dev/null 2>&1 || :

$ rpm --eval %{systemd_post}

if [ $1 -eq 1 ] ; then
    # Initial installation
    systemctl preset >/dev/null 2>&1 || :
fi

$ rpm --eval %{systemd_postun}

systemctl daemon-reload >/dev/null 2>&1 || :

$ rpm --eval %{systemd_preun}

if [ $1 -eq 0 ] ; then
    # Package removal, not upgrade
    systemctl --no-reload disable > /dev/null 2>&1 || :
    systemctl stop > /dev/null 2>&1 || :
```

Еще один элемент, который обеспечивает еще более детальный контроль над транзакцией RPM в целом, - это то, что известно как **триггеры**. По сути, это то же самое, что и скриптлет, но выполняется в очень определенном порядке операций во время транзакции установки или обновления RPM, что позволяет более точно контролировать весь процесс.

Порядок, в котором выполняется каждый из них, и подробная информация о котором - приведена ниже.

```
all-%pretrans
...
any-%triggerprein (%triggerprein from other packages set off by new install)
new-%triggerprein
new-%pre          for new version of package being installed
...              (all new files are installed)
new-%post         for new version of package being installed

any-%triggerin (%triggerin from other packages set off by new install)
new-%triggerin
old-%triggerun
any-%triggerun (%triggerun from other packages set off by old uninstall)

old-%preun       for old version of package being removed
...              (all old files are removed)
old-%postun      for old version of package being removed

old-%triggerpostun
any-%triggerpostun (%triggerpostun from other packages set off by old un
                    install)

...
all-%posttrans
```

Вышеуказанные элементы взяты из прилагаемой документации RPM, найденной в [/usr/share/doc/rpm/triggers](#) на системах Fedora и [/usr/share/doc/rpm-4.*/triggers](#) в системах RHEL 7 и CentOS 7.

Использование скриптов без оболочки в SPEC файле

Параметр скриптлета **-p**, в SPEC файле позволяет вызывать определенный интерпретатор вместо стандартного **-p /bin/sh**. Наглядным примером является скрипт, который выводит сообщение после установки **pello.py**.

1. Откройте файл **pello.spec**.
2. Найдите следующую строку:

```
install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/
```

Под этой строкой вставьте следующий код:

```
%post -p /usr/bin/python3
print("This is {}".format("python"))
```

3. Создайте свой пакет в соответствии с главой [Сборка RPMS](#).

4. Установите Ваш пакет:

```
# dnf install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.1-1.fc27.noarch.rpm
```

Результатом выполнения этой команды является следующее сообщение после установки:

```
Installing      : pello-0.1.1-1.fc27.noarch                1/1
Running scriptlet: pello-0.1.1-1.fc27.noarch                1/1
This is python code
```

NOTE

- Чтобы использовать скрипт Python 3: Напишите строку `%post -p /usr/bin/python3` под строкой `install -m` in a SPEC file.
- Чтобы использовать скрипт Lua: Напишите строку `%post -p <lua>` под строкой `install -m` в SPEC файле.
- Таким образом, в SPEC файле может быть указан любой интерпретатор.

Условные обозначения RPM

Условные обозначения RPM позволяют условно включать различные разделы SPEC файла.

Чаще всего условные обозначения имеют дело с:

- разделами, относящимися к конкретной архитектуре
- разделами, относящимися к конкретной операционной системе
- проблемами совместимости между различными версиями операционных систем
- существованием и определением макросов

Синтаксис условий RPM

Если *expression* истинно, то выполните какое-нибудь действие:

```
%if expression
...
%endif
```

Если *expression* истинно, то выполните какое-нибудь действие, в другом случае выполните другое действие:

```
%if expression
...
%else
...
%endif
```

Примеры условий RPM

Обозначение `%if`

```
%if 0%{?rhel} == 6
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

Это условие обрабатывает совместимость между RHEL6 и другими операционными системами с точки зрения поддержки макроса `AS_FUNCTION_DESCRIBE`. Когда пакет создается для RHEL, определяется макрос `%rhel`, и он расширяется до версии RHEL. Если его значение равно 6, что означает, что пакет создан для RHEL 6, то ссылки на `AS_FUNCTION_DESCRIBE`, который не поддерживается RHEL6, удаляются из сценариев автоконфигурации.

```
%if 0%{?el6}
%global ruby_sitearch %(ruby -rrbconfig -e 'puts Config::CONFIG["sitearchdir"]')
%endif
```

Это условие регулирует совместимость между Fedora версии 17 и новее и RHEL 6 с точки зрения поддержки макроса `%ruby_sitearch` макро. Fedora версии 17 и никогда не определяет `%ruby_sitearch` по умолчанию, но RHEL6 не поддерживает этот макрос. Условие проверяет, является ли операционная система RHEL 6. Если это так, `%ruby_sitearch` определяется явно. Обратите внимание, что `0%{?el6}` имеет то же значение, что и `0%{?rhel} == 6` из предыдущего примера, и он проверяет, построен ли пакет на RHEL 6.

```
%if 0%{?fedora} >= 19
%global with_rubypick 1
%endif
```

Это условие обрабатывает поддержку инструмента выбора `ruby`. Если операционная система Fedora версии 19 или новее, поддерживается `rubypick`.

```
%define ruby_archive %{name}-%{ruby_version}
```

```
%if 0{%?milestone:1}%{%?revision:1} != 0
%define ruby_archive %{ruby_archive}-
%{%?milestone}%{%?!milestone:%{%?revision:r{%revision}}}%
%endif
```

Это условие обрабатывает определение макросов. Если заданы макросы `%milestone` или `%revision`, переопределяется макрос `%ruby_archive`, который определяет имя вышестоящего файла архива.

Специальные варианты обозначения `%if`

Условные обозначения `%ifarch`, `%ifnarch` и `%ifos` являются специализированными вариантами условных обозначений `%if`. Эти варианты обычно используются, поэтому у них есть свои собственные макросы.

Обозначение `%ifarch``

Условие `%ifarch` спользуется для начала блока SPEC файла, который зависит от архитектуры. За ним следует один или несколько спецификаторов архитектуры, каждый из которых разделен запятыми или пробелами.

```
%ifarch i386 sparc
...
%endif
```

Все содержимое SPEC файла между `%ifarch` и `%endif` обрабатывается только на 32-разрядных архитектурах AMD и Intel или системах на базе Sun SPARC.

Условное обозначение `%ifnarch`

Условие `%ifnarch` имеет обратную логику, чем условие `%ifarch`.

```
%ifnarch alpha
...
%endif
```

Все содержимое SPEC файла между `%ifnarch` и `%endif` обрабатывается только в том случае, если оно не выполняется в системе на основе Digital Alpha/AXP.

Условие `%ifos`

Условие `%ifos` используется для управления обработкой на основе операционной системы сборки. За ним может следовать одно или несколько имен операционной системы.

```
%ifos linux
...
%endif
```

Все содержимое SPEC файла между `%ifos` и `%endif` обрабатывается только в том случае, если сборка была выполнена в системе Linux.

Appendix A: Новые возможности RPM в RHEL 7

Этот список документирует наиболее заметные изменения в упаковке RPM между Red Hat Enterprise Linux 6 и 7.

- Добавлена новая команда `rpmkeys`, используемая для импорта связки ключей и проверки подписи.
- Добавлена новая команда `rpmspec`, используемая для запросов спецификаций и анализируемых выходных данных.
- Добавлена новая команда `rpmsign`, используемая для подписи пакета.
- Расширения `posix.exec()` и `os.exit()` встроенные в `%{lua:...}` скрипты, завершают работу скрипта с ошибкой, если они не вызываются из дочернего процесса, созданного с помощью скриптлета `posix.fork()`.
- Сбой скриптлета `%pretrans` приводит к пропуску установки пакета
- Скриптлеты могут быть развернуты макросом и развернуты в формате запроса во время выполнения
- Зависимости скриптелетов до транзакции и после транзакции теперь могут быть правильно выражены с помощью `Requires(pretrans)`, `Requires(posttrans)` и скриптелетов.
- Добавлен тег `OrderWithRequires` для предоставления дополнительных подсказок. Тег следует синтаксису тега `Requires`, но не создает фактических зависимостей. Подсказки по порядку обрабатываются так, как если бы они были `Requires` при расчете порядка транзакции, только если задействованные пакеты присутствуют в одной и той же транзакции.
- The `%license`. Этот флаг можно использовать в разделе `%files``. Можно использовать аналогично флагу `%doc` для пометки файлов как лицензий, которые необходимо установить, несмотря на параметр `--nodocs``.
- Добавлен макрос `%autosetup` для автоматизации применения исправлений с дополнительной интеграцией с распределенной системой контроля версий.
- Автоматический генератор зависимостей был переписан в расширяемую и настраиваемую систему на основе правил со встроенной фильтрацией.
- Открытые ключи OpenPGP V3 больше не поддерживаются.

Appendix B: Ссылки на источники

Ниже приведены ссылки на различные темы, представляющие интерес для RPM, упаковки RPM и сборки RPM. Некоторые из них являются продвинутыми и выходят далеко за рамки вводного материала, включённого в данное руководство.

[Software Collections](#) - SoftwareCollections.org это проект с открытым исходным кодом для создания и распространения поддерживаемых сообществом коллекций программного обеспечения (SCL) для Red Hat Enterprise Linux, Fedora, CentOS, и Scientific Linux.

[Creating RPM package](#) - Пошаговое руководство по изучению основ упаковки RPM.

[Packaging software with RPM, Part 1, Part 2, Part 3](#) - Руководство по упаковке IBM RPM.

[RPM Documentation](#) - Официальная документация RPM.

[Fedora Packaging Guidelines](#) - Официальные рекомендации по упаковке для Fedora, полезные для всех дистрибутивов на основе RPM.

[rpmfluff](#) - библиотека python для создания пакетов RPM и их саботажа, чтобы они были взломаны контролируруемыми способами.

Appendix C: Выражение благодарности

Некоторые фрагменты этого текста впервые появились в <https://rpm-packaging-guide.github.io/> RPM Packaging Guide. Copyright © 2020 Adam Miller and others. Лицензировано в соответствии с <http://creativecommons.org/licenses/by-sa/3.0/> Creative Commons Attribution-ShareAlike 3.0 Unported License.