МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования Московский технический университет связи и информатики

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

«Инфраструктура разработки программных пакетов и сборки программного обеспечения»

для очного и дистанционного режимов обучения магистратуры и дополнительного профессионального образования

Базовая кафедра общественно-государственного объединения «Ассоциация документальной электросвязи» «Технологии электронного обмена данными» в Московском техническом университете связи и информатики

2023

Оглавление

Введение

Учебно-методическое пособие «Инфраструктура разработки программных пакетов и сборки программного обеспечения» предназначено для очного и дистанционного обучения студентов в рамках гуманитарно-технологической платформы по программам магистратуры и дополнительного профессионального образования «Информационная культура цифровой трансформации» базовой кафедры общественно-государственного объединения «Ассоциация документальной электросвязи» (АДЭ) «Технологии электронного обмена данными» (ТЭОД) в Московском техническом университете связи и информатики (МТУСИ).

Пособие состоит из введения, заключения, шести глав и содержит практикумы по следующим направлениям:

- Пакетный менеджер.
- Основные команды пакетного менеджера.
- Программное обеспечение, используемое для упаковки пакетов.
- Инструмент Gear.
- Инструмент hasher.
- Примеры использования инструментов ОС Альт для сборки пакетов.

Пособие включает сведения о программной платформе лабораторного практикума на базе отечественных операционных систем семейства «Альт», представленного компанией «Базальт СПО», единственного российского разработчика системного программного обеспечения, создавшего собственную технологическую среду распределённой коллективной разработки и обеспечения жизненного цикла программного обеспечения («Альт платформа»).

Разработки дистрибутивов операционных систем компании «Базальт СПО» основаны на отечественной инфраструктуре разработки «Сизиф» (Sisyphus), которая находится на территории РФ, принадлежит и поддерживается компанией «Базальт СПО». В основе Sisyphus лежат технологии сборки программ и учёта зависимостей между ними, а также отработанные процессы по взаимодействию разработчиков. На базе репозитория периодически формируется стабильная ветка (программная платформа), которая поддерживается в течение длительного

времени и используется в качестве базы для построения дистрибутивов линейки «Альт» и обеспечения их жизненного цикла.

ООО «Базальт СПО» выпускает линейку дистрибутивов разного назначения для различных аппаратных архитектур. В репозитории Sisyphus поддерживаются архитектуры: i586, x86_64, armh (armv7), aarch64 (armv8), Эльбрус (с третьего по шестое поколение), riscv64, mipsel, loongarch.

Часть дистрибутивов включена в Единый реестр российских программ для электронных вычислительных машин и баз данных — это «Альт СП», имеющий сертификаты ФСТЭК России, Минобороны России и ФСБ России, Альт Виртуализация, Альт Сервер, Альт Рабочая станция, Альт Образование, так и другие, бесплатные и свободные: Simply Linux, различные стартеркиты (Starterkits) и регулярные (regular) сборки. Дистрибутив — это составное произведение, в составе которого есть программа для дистрибуции (установки), называемая инсталлятор и набор системного и прикладного ПО. В основе всех дистрибутивов лежат пакеты свободного программного обеспечения.

Свободное программное обеспечение (СПО) — это программное обеспечение, распространяемое на условиях простой (неисключительной) лицензии, которая позволяет пользователю:

1. использовать программу для ЭВМ в любых, не запрещённых законом целях;

2. получать доступ к исходным текстам (кодам) программы как в целях изучения и адаптации, так и в целях переработки программы для ЭВМ; распространять программу (бесплатно или за плату, по своему усмотрению);

3. вносить изменения в программу для ЭВМ (перерабатывать) и распространять экземпляры изменённой (переработанной) программы с учётом возможных требований наследования лицензии;

4. в отдельных случаях (CopyLeft лицензия) распространять модифицированную компьютерную программу пользователем на условиях, идентичных тем, на которых ему предоставлена исходная программа.

Примерами свободных лицензий являются:

1. `GNU general public license`. Version 3, 29 June 2007 (Стандартная общественная лицензия GNU. Версия 3, от 29 июня 2007 г.)

2. `BSD license`, New Berkley Software Distribution license (Модифицированная программная лицензия университета Беркли)

СПО отлично подходит для целей обучения и для разработки собственных решений, потому, что весь код доступен для изучения и модификации, однако авторы настоятельно советуют всем, кто использует СПО для построения своих программных продуктов учитывать особенности лицензирования не только

самих пакетов, но и входящих в их состав библиотек, так как если вы используете copyleft библиотеку, это обязывает вас распространять свою программу под аналогичной лицензией — любой человек, который поучит вашу программу легальным способом может потребовать предъявить ему исходный код.

В работе над пособием принимали участие:
Авторы: А. А. Абрамов, М. А. Фоканова, Мельников.
Рецензенты: А. А. Калинин, А. А. Лимачко, Валерий Синельников.
Редактура: В. Л Черный, В. А. Соколов.

Глава 1

Пакетный менеджер

Операционная система состоит из разнообразных программ, библиотек, скриптов и приложений — число компонент может достигать тысячи единиц. В каждой из которых могут быть включены десятки файлов. Для удобства работы пользователя системные компоненты в Linux представлены в виде пакетов[1]. Пакет объединяет в общий архив все файлы, используемые программой. Пользователь подбирает программы — устанавливает, обновляет, проверяет, удаляет их — только по имени пакета, не вдаваясь в отдельные детали подбора всех необходимых файлов. Работа с пакетом, как единым целым, позволяет управлять всеми данными программы.

Пакет — это специально подготовленный архив, содержащий программы, конфигурационные файлы, данные и управляющую информацию. Метаданные пакета содержат полное имя, номер версии, описание пакета, имя разработчика, контрольную сумму, зависимости от других пакетов. Управляющая информация пакета содержит контрольные суммы устанавливаемых файлов, зависимости устанавливаемого пакета от других пакетов, краткое описание, сценарии установки и удаления пакета, и прочую информацию, которую использует менеджер пакетов. Пакеты хранятся в специальном хранилище — **репозитории пакетов**.

Для удобства работы с пакетами придумали собственные форматы архивов:

- RPM (.rpm). Разработан компанией Red Hat. Применяется в системе Альт, Ред ОС, RHEL и CentOS.
- DEB (.deb). Формат пакетов дистрибутива Debian, а также Ubuntu.
- TAR.XZ. Применяется в дистрибутивах ArchLinux и Manjaro.

[1] Курячий Г., Маслинский К. (2010). Операционная система Linux. Курс лекций. ДМК Пресс.

Каждый пакет определяется архитектурой системы под которую он собран, именем программы, номером её версии и номером релиза этой программы в дистрибутиве. Если пакет не зависит от архитектуры процессора, то в качестве архитектуры указывается «noarch».

Например, `admc-0.15.0-alt1.x86_64.rpm`:

Имя: `admc`
Номер версии: `0.15.0`
Номер релиза: `alt1`
Архитектура: `x86_64`

Пакетный менеджер — это программа для управления установкой, удалением, настройкой и обновлением пакетов с различным программным обеспечением. Пакетные менеджеры упрощают для пользователя процесс управления программами в системе. Пакетный менеджер также ведёт учёт пакетов, установленных в системе. Существует также менеджер зависимостей — специальная программа, подбирающая пакеты, зависимые друг от друга, и загружающая эти пакеты из хранилища[2]. Менеджер зависимостей подбирает правильные версии пакетов и определяет порядок их установки. При помощи менеджера зависимостей можно узнать, с каким пакетом поставляется тот или иной файл.

Задачи пакетного менеджера:

- **установка программ**. Позволяет устанавливать программы из центрального хранилища или из локальных источников;
- **обновление программ**. Позволяет обновлять установленные программы до последних версий, представленных в хранилище;
- **удаление программ**. Позволяет безопасно удалять программы и все связанные с ними файлы;
- **управление зависимостями**. Автоматически устанавливает и управляет зависимостями программ;
- **проверка целостности пакетов**. Предотвращает конфликты при установке новых программ, обеспечивая целостность. системы.

Пакетный менеджер позволяет:

- узнать информацию о пакете;
- определить пакет, которому принадлежит установленная программа;
- определить список компонент, установленных из указанного пакета.

Пакетные менеджеры делятся на две категории — низкоуровневые и высокоуровневые.

[2] Кетов Д. (2021). Внутреннее устройство Linux. 2-е изд,. перераб. и доп. БХВ-Петербург.

- **Низкоуровневые пакетные менеджеры**. Используются для установки локальных пакетов, загруженных вручную пользователем, или высокоуровневым пакетным менеджером.

- **Высокоуровневые менеджеры**. Применяются для поиска и скачивания пакетов из репозиториев. В процессе работы могут задействовать низкоуровневые менеджеры для установки загруженных программ.

В операционной системе Альт используется формат пакетов .rpm. Пакеты rpm хранятся в удалённом хранилище (Sisyphus и/или его подветках — бранчах). Для работы с такими пакетами применяется низкоуровневый пакетный менеджер RPM и консольная утилита APT (Advanced Packaging Tool).

1.1 Основной пакетный менеджер в Альт Платформа

В дистрибутивах Альт применяется пакетный менеджер RPM. RPM Package Manager — это семейство пакетных менеджеров, применяемых в различных дистрибутивах GNU/Linux. Практически каждый крупный проект, использующий RPM, имеет свою версию пакетного менеджера, отличающуюся от остальных.

Различия между представителями семейства RPM выражаются в:

- наборе макросов, используемых в .spec-файлах;

- различии сборки rpm-пакетов «по умолчанию» — при отсутствии каких-либо указаний в .spec-файлах, формате строк зависимостей;

- отличиях в семантике операций (например, в операциях сравнения версий пакетов);

- отличиях в формате файлов.

Для пользователя различия чаще всего заключаются в невозможности поставить пакет постороннего формата (например, .deb) из-за проблем с зависимостями другого формата пакета.

1.2 Система управления пакетами

Системой управления пакетами в дистрибутивах Альт является программа APT — Advanced Packaging Tool (усовершенствованный инструмент работы с пакетами). Программа APT способна автоматически устанавливать и настраивать программы в операционных системах Альт из предварительно откомпилированных пакетов и из исходных кодов. Утилита загружает пакеты из хранилища (репозитория), либо устанавливает с локального хранилища. Список источников пакетов хранится в файле /etc/apt/sources.list и в каталоге

/etc/apt/sources.list.d/. В системе Альт применяется графическая оболочка для apt — программа Synaptic[3]. Утилита apt значительно упрощает процесс установки программ в командном режиме.

Команда $ apt-get выведет описание и возможности утилиты apt-get:

```
$ apt-get
apt 0.5.15lorg2 для linux x86_64 собран Jul 26 2023 18:10:41
Использование: apt-get [параметры] команда
apt-get [параметры] install|remove пакет1 [пакет2 ...]
apt-get [параметры] source пакет1 [пакет2 ...]

apt-get предоставляет простой командный интерфейс для получения и
установки пакетов. Чаще других используются команды update (обновить)
и install (установить).

Команды:
update - Получить обновлённые списки пакетов
upgrade - Произвести обновление
install - Установить новые пакеты
remove - Удалить пакеты
source - Скачать архивы исходников
build-dep - Установить всё необходимое для сборки исходных пакетов
dist-upgrade - Обновление системы в целом, см. apt-get(8)
clean - Удалить скачанные ранее архивные файлы
autoclean - Удалить давно скачанные архивные файлы
check - Удостовериться в отсутствии неудовлетворённых зависимостей
dedup - Remove unallowed duplicated pkgs

Параметры:
-h  Краткая справка
-q  Скрыть индикатор процесса
-qq Не показывать ничего кроме сообщений об ошибках
-d  Получить пакеты и выйти БЕЗ их установки или распаковки
-s  Симулировать упорядочение вместо реального исполнения
-y  Автоматически отвечать 'ДА' на все вопросы
-f  Пытаться исправить положение если найдены неудовлетворённые зависимости
-m  Пытаться продолжить если часть архивов недоступна
-u  Показать список обновляемых пакетов
-b  Собрать пакет после получения его исходника
-D  При удалении пакета стремиться удалить компоненты, от которых он зависит
-V  Подробно показывать номера версий
-c=? Использовать указанный файл конфигурации
-o=? Изменить любой из параметров настройки (например: -o dir::cache=/tmp)
```

[3] apt и synaptic развиваются ALT Linux Team, не нужно сравнивать версии с аналогичными утилитами в Debian

Более полное описание доступно на страницах руководства man: `apt-get(8)`, `sources.list(5)` и `apt.conf(5)`.

В ОС `Альт` утилита `apt-get` использует основной пакетный менеджер `RPM Package Manager` — `RPM` для установки, обновления, удаления пакетов (программ), управления зависимостями. Обе утилиты `RPM` и `APT` одинаково позволяют установить, обновить или удалить пакет.

Отличия `RPM` и `APT`:

- `APT` учитывает зависимости устанавливаемого пакета;
- `APT` умеет работать с репозиторием в целом:
  - искать пакеты;
  - вычислять список обновлений — находить разницу версий пакетов, установленных локально и хранящихся в репозитории;
- `APT` получает информацию из пакетов, используя `RPM`.

`RPM` подразумевает работу с конкретными пакетами. Удовлетворение их зависимостей остаётся на усмотрении пользователя. `APT` позволяет вычислять, какие пакеты из репозитория нужно установить, чтобы удовлетворить зависимости, которые указаны в каждом конкретном `RPM` пакете. `APT` сам не устанавливает пакеты, он использует для этого `RPM`.

Установка пакетов в АЛЬТ Платформа осуществляется с помощью утилиты `APT`

Репозиторий — актуализируемое хранилище электронных данных. В данном документе речь идёт о репозитории как об инфраструктуре разработки операционных систем, включающих, помимо системного ПО, любые программы пользовательского и серверного назначения («репозиторий пакетов»). Основная задача репозиториев этого рода — интеграция разных пакетов программ в единую систему. Объектом хранения в таких репозиториях выступают пакеты программ, где каждое наименование ПО (будь то ядро операционной системы, служебная библиотека, текстовый редактор, сервер для обслуживания электронных сообщений или медиа проигрыватель) представлено в виде отдельного пакета. В основе создания операционных систем `ALT` лежит экспериментальный репозиторий `Sisyphus` из которого формируются время от времени стабильные бранчи, на которых и создаются коммерческие операционные системы. На конец 2023 года стабильным бранчем является p10, называемый «Альт Платформа 10».

Репозиторий программных пакетов — это замкнутая совокупность программных пакетов (исходных и собранных из них бинарных, плюс метаинформации о них) с поддерживаемой целостностью, то есть программы с формализованными инструкциями по установке и разрешёнными зависимостями.

Наличие репозитория пакетов и APT автоматизирует процессы управления установкой, обновления и удаления программного обеспечения; исключают риск случайного повреждения целостности операционной системы и прикладных программ.

Программа APT по запросу пользователя получает метаданные из репозитория, рассчитывает зависимости, получает от пользователя информацию о том, какие именно пакеты он хочет обновить или установить. Утилита предложит пути решения — например, загрузит новые пакеты из репозитория, установит дополнительные или обновит имеющиеся пакеты. APT, в зависимости от настроек, может использовать удалённый репозиторий с помощью сетевого протокола (например, ftp), или использовать локальный репозиторий (например, на жёстком диске).

Для обновления практически всего программного обеспечения (за исключением ядра операционной системы) на локальном компьютере до новой версии необходимо выполнить команды:

```
apt-get update
apt-get dist-upgrade
```

При использовании APT и обновляемого стабильного репозитория операционная система может функционировать на компьютере годами, плавно обновляясь до новых версий без переустановки системы.

1.3 Установка необходимых пакетов для процесса сборки

Сборка — формирование пакета на основе сборочных инструкций.

Сборка программного обеспечения в RPM упрощает распространение, управление и обновление программного обеспечения. Упаковка программного обеспечения в пакеты RPM, а не обычный архив, имеет ряд преимуществ[4]:

- Пользователи могут использовать средства управления пакетами (например, Yum или PackageKit) для установки, переустановки, удаления, обновления и проверки пакетов RPM.
- Пакетный менеджер RPM предполагает наличие базы данных, которая с помощью специализированных утилит позволяет получать информацию о пакетах в системе.
- Каждый пакет RPM содержит метаданные, описывающие компоненты пакета, версию, выпуск, размер, URL проекта, установочные инструкции и т. д.
- RPM позволяет брать оригинальные источники программного обеспечения и упаковывать их в пакеты с исходным кодом(.src.rpm) и бинарные пакеты(.rpm). В пакетах с исходным кодом хранятся оригинальные исходные

[4] https://rpm-packaging-guide-ru.github.io/#Why-Package-Software-with-RPM

данные вместе со всеми изменениями(`*.patch`), а так же сборочные инструкции(`.spec`) и дополнительная информация. В бинарных пакетах вместо исходного кода упакованы подготовленные файлы и скрипты установки, но нет сборочных инструкций. Ещё существуют случаи распространения пакетов без исходного кода и бинарных данных, в таких пакетах присутствуют скрипты для скачивания и модификации файлов, необходимых для работы приложения.

- Для обеспечения верификации подлинности `RPM`-пакетов используется механизм электронных цифровых подписей `GPG`. Он позволяет подписать `RPM` пакет или обновить цифровую подпись: `rpm -addsign package.rpm` и `rpm -resign package.rpm`.
- Вы можете добавить свой пакет в `RPM` репозиторий, что позволит клиентам легко находить и устанавливать ваше программное обеспечение.

Задача сборки пакета начинается со сбора всех необходимых компонентов и завершается этапами сборки и тестирования.

Классическая сборка пакетов `rpm` состоит из следующих этапов:

- поиск исходных данных;
- написание инструкций сборки;
- сборка пакета.

Для сокращения команд, встречающихся в тексте, будет использоваться нотация:
— команды **без административных привилегий** начинаются с символа `"$"`;
— команды **с административными привилегиями** начинаются с символа "#".

Необходимые инструменты для сборки `rpm`-пакетов устанавливаются в системе через пакетный менеджер `apt` командой:

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

В наборе параметров команды `install` перечислены имена пакетов, необходимых для сборочной инфраструктуры[5]:

- `gcc` — набор компиляторов для различных языков программирования, разработанный в рамках проекта GNU;

[5] https://www.altlinux.org/Технология_сборки_пакетов_RPM

- `rpm-build` — содержит сценарии и исполняемые программы, которые используются для сборки пакетов с помощью `RPM`;
- `rpmlint` — инструмент для проверки распространённых ошибок в пакетах `rpm`. Можно проверить бинарные и исходные пакеты;
- `make` — инструмент GNU, упрощающий процесс сборки для пользователей;
- `python` — интерпретируемый интерактивный объектно-ориентированный язык программирования;
- `gear` — этот пакет содержит утилиты для сборки пакетов `RPM` из `GEAR`.репозитория и управления `GEAR`.репозиториями;
- `hasher` — современная технология создания независимых от сборочной системы пакетов;
- `patch` — программа исправлений применяет патчи к оригиналам;
- `rpmdevtools` — пакет содержит скрипты и файлы поддерживающие `(X)Emacs`, помогающие в разработке пакетов `RPM`.

1.4 Вопросы для самопроверки

1. Что такое программный пакет?
2. Какие форматы пакетов вы знаете?
3. Что такое репозиторий ПО?
4. Какие низкоуровневые пакетные менеджеры вы знаете?
5. Какой пакетный менеджер используется в Альт Платформа?
6. Вы обнаружили в системе пакет `nagios-domain-discovery-0.1.1-alt1.noarch.rpm`, определите его имя, версию, релиз и архитектуру.
7. Верно ли утверждение, что утилита `apt` при установке пакетв требует явного указания всех зависимых пакетов?
8. Для чего нужна команда `apt-get update`?
9. Выполнится ли успешно команда `$ apt-get dist-upgrade`
10. Какие инструменты (программы) нужны для сборки пакетов в «Альт Платформа»

Глава 2

Основные команды пакетного менеджера

Управлять пакетами можно из командной строки при помощи программы `rpm`, которая имеет следующий синтаксис:

```
rpm [options]
```

Пакетный менеджер `RPM` предоставляет базовые возможности для управления пакетами. Основной набор команд[1] позволяет установить, удалить, обновить пакеты, получить разнообразную информацию о самих пакетах и их содержимом:

- **Информация о пакете**: `rpm -qi ИМЯ_ПАКЕТА ...`
  `rpm -qi` выводит подробную информацию о конкретном установленном пакете.

  

  `rpm -qi` означает «query information» (запросить информацию). Но в случае `rpm -i <имя пакета>`, `-i` означает инструкцию `install` (установи).

- **Просмотр установленных пакетов**: `rpm -qa`
  Эта команда выводит список всех установленных пакетов в системе.

  

  `-a: All` (весь, все). `rpm -qa` означает «query all».

- **Проверка установки пакета в системе**: `rpm -q ИМЯ_ПАКЕТА ...`
  Эта команда проверяет установлен ли пакет в системе.

[1] https://wiki.altlinux.ru/Команды_RPM

- **Проверка зависимостей пакета**: `rpm -qR ИМЯ_ПАКЕТА ...`
  `rpm -qR` выводит список зависимостей (другие пакеты), необходимых для работы указанного пакета.

`-R: Requires` (нуждается). Например, `rpm -qR` означает «query requires» (запрос нуждается).

- **Проверка файла на принадлежность пакету**: `rpm -qf ФАЙЛ`
  Команда `rpm -qf` определяет, к какому пакету принадлежит указанный файл.

`-f: File` (файл). Например, `rpm -qf` означает «query file» (файл запроса).

- **Просмотр файлов пакета**: `rpm -ql ИМЯ_ПАКЕТА ...`
  `rpm -ql` выводит список всех файлов, содержащихся в установленном пакете.

- **Установка пакета**: `rpm -i ФАЙЛ_ПАКЕТА`
  Команда `rpm -i` используется для установки пакета из файла `.rpm`. Например, `rpm -i package.rpm` установит содержимое пакета в системе.

- **Удаление пакета**: `rpm -e ФАЙЛ_ПАКЕТА`
  `rpm -e` удаляет установленный пакет. Например, `rpm -e package` удалит пакет с именем `package`.

- **Обновление пакета**: `rpm -U ФАЙЛ_ПАКЕТА`
  Команда `rpm -U` обновляет пакет до новой версии, если он уже установлен.

- **Проверка целостности пакета**: `rpm -V ИМЯ_ПАКЕТА|ФАЙЛ_ПАКЕТА ...`
  `rpm -V` проверяет целостность файлов в пакете, сравнивая их с информацией в базе данных `rpm`. Дополнительные ключи:

  – `-v`: Verbose (подробно). Например, `rpm -qv` означает «query verbose» (подробный запрос) и используется для вывода более подробной информации о пакете. Подробный вывод существует не для всех ключей утилиты.

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

2.1 Установка RPM-пакета

В команде должен быть указан файл пакета или полный путь к нему

Для установки пакета из `rpm-файла` используйте команду:

```
# rpm -i ФАЙЛ_ПАКЕТА ...
```

Для работы с командой потребуются права суперпользователя. Их можно получить через команду `su-`, либо команду `sudo`

Пример выполнения команды [2]:

```
[root@test-alt]# rpm -i gpupdate-0.9.12.6-alt1.src.rpm
```

В конце команды возможно указать дополнительные опции:

- `-nodeps` — не проверять зависимости пакета;
- `-replacepkgs` или `-reinstall` — переустановить пакет.

Подробный вывод. Для отображения прогресса установки, используйте дополнительные параметры `-v` и `-h`.

- `-v` — вывести детальные сообщения;
- `-h` — вывести «#» строку индикатора прогресса по мере установки пакета (используется с `-v`).

Пример выполнения команды:

```
rpm -ivh gpupdate-0.9.12.6-alt1.src.rpm

Подготовка...
################################[100%]

Обновление / установка...

1: gpupdate-0.9.12.6-alt1.src.rpm
################################[100%]
```

[2] https://wiki.altlinux.ru/Команды_RPM

2.2 Проверка установки пакета в системе

Чтобы проверить, установлен ли пакет, введите следующую команду:

```
$ rpm -q ИМЯ_ПАКЕТА
```

Пример:

```
$ rpm -q gpupdate
gpupdate-0.9.12.2-alt2.noarch

$ rpm -q mediinfo
пакет mediinfo не установлен
```

2.3 Просмотр файлов пакета, установленного в системе

Чтобы получить список файлов пакета, введите следующую команду:

```
$ rpm -ql ИМЯ_ПАКЕТА ...
```

В этой команде используется ключ «`-l`» (list).

Для развернутой информации укажите ключ «`-i`».

Пример использования:

```
$ rpm -ql admc
/usr/bin/admc
/usr/lib64/libadldap.so
/usr/share/applications/admc.desktop
/usr/share/doc/admc-0.11.2
/usr/share/doc/admc-0.11.2/CHANGELOG.txt
/usr/share/doc/admc-0.11.2/CHANGELOG_ru.txt
/usr/share/doc/admc-0.11.2/README.md
/usr/share/icons/hicolor/scalable/apps/admc.svg
/usr/share/man/man1/admc.1.xz
```

Чтобы узнать содержимое неустановленного `rpm`-пакета, используйте команду:

```
$ rpm -qlp ИМЯ_ПАКЕТА ...
```

Пример выполнения команды:

```
$ rpm -qlp udisks2-2.9.4-alt1.1.src.rpm
udisks-2.9.4.tar.bz2 udisks2.control udisks2.spec
```

2.4 Просмотр недавно установленных пакетов

Чтобы получить список последних установленных пакетов, введите следующую команду:

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

Вывод:

```
smplayer-23.6.0-alt2.10169.x86_64             Пт 01 дек 2023 12:45:39
qt5-wayland-5.15.10-alt1.x86_64               Пт 01 дек 2023 12:35:44
qt5-tools-5.15.10-alt2.x86_64                 Пт 01 дек 2023 12:35:44
qt5-sql-mysql-5.15.10-alt1.x86_64             Пт 01 дек 2023 12:35:44
qt5-dbus-5.15.10-alt2.x86_64                  Пт 01 дек 2023 12:35:44
libqt5-xmlpatterns-5.15.10-alt1.x86_64        Пт 01 дек 2023 12:35:44
libqt5-x11extras-5.15.10-alt1.x86_64          Пт 01 дек 2023 12:35:44
libqt5-webenginewidgets-5.15.15-alt1.x86_64   Пт 01 дек 2023 12:35:44
libqt5-test-5.15.10-alt1.x86_64               Пт 01 дек 2023 12:35:44
libqt5-quickparticles-5.15.10-alt1.x86_64     Пт 01 дек 2023 12:35:44
```

Команда `rpm -qa -last` используется для вывода списка всех установленных пакетов, отсортированных по времени их установки. Пакеты будут отсортированы в порядке убывания времени установки — самые последние установленные пакеты отобразятся в верхней части списка.

Фильтрация вывода: утилита `grep` отфильтрует вывод и поможет найти искомый пакет. Например, следующая команда выведет информацию только о тех пакетах, название которых содержит «`kernel`»:

```
rpm -qa -last | grep kernel
```

2.5 Поиск пакета в системе

Чтобы найти в системе необходимый пакет среди уже установленных, используйте утилиту `grep`. Утилита `grep` находит строки по запросу.

```
$ rpm -qa | grep ИМЯ_ПАКЕТА
```

Например, запрос:

```
kde5-smplayer-common-22.7.0-alt1.noarch
smplayer-23.6.0-alt2.10169.x86_64
```

2.6 Проверка файла, относящегося к пакету

Чтобы определить, какому пакету принадлежит указанный файл, используйте команду:

```
$ rpm -qf ФАЙЛ
```

Например, запрос:

```
$ rpm -qf /usr/share/FBReader/help
```

Предоставит вывод:

```
fbreader-0.99.5-alt6.x86_64
```

2.7 Вывод информации о пакете

Чтобы получить вывод подробной информации о конкретном установленном пакете[3] — название, версию и прочее — используйте команду:

```
$ rpm -qi ИМЯ_ПАКЕТА ...
```

В запросе указывается имя пакета из списка уже установленных в системе, либо путь к загруженному пакету.

Пример выполнения команды для установленного пакета в системе:

```
$ rpm -qi gpupdate
```

```
Name        : gpupdate
Version     : 0.9.12.6
Release     : alt1
DistTag     : p10+323062.200.4.1
Architecture: noarch
Install Date: Ср 12 июл 2023 13:26:50
Group       : Other
Size        : 951193
License     : GPLv3+
Signature   : RSA/SHA512, Сб 08 июл 2023 22:59:22, Key ID 42f343a2c7eb80f9
Source RPM  : gpupdate-0.9.12.6-alt1.src.rpm
Build Date  : Сб 08 июл 2023 22:59:21
Build Host  : greh-p10.hasher.altlinux.org
Relocations : (not relocatable)
Packager    : Valery Sinelnikov (ALT) <greh@altlinux.org>
Vendor      : ALT Linux Team
URL         : https://github.com/altlinux/gpupdate
Summary     : GPT applier
Description :
gpupdate is the facility to apply various GPO/GPT settings retrieved
from Active Directory domain in UNIX environment.
```

Пример выполнения команды для rpm-пакета, расположенного на диске, но неустановленного в системе:

[3] https://www.inp.nsk.su/ bolkhov/teach/inpunix/make_rpm.ru.html

```
$ rpm -qip /путь/к/пакету/foo-1.0-i586.rpm
```

Пример выполнения:

```
$ rpm -qip kooha-2.2.4-alt1.x86_64.rpm

Name        : kooha
Epoch       : 1
Version     : 2.2.4
Release     : alt1
DistTag     : sisyphus+330279.100.2.2
Architecture: x86_64
Install Date: (not installed)
Group       : Video
Size        : 2261140
License     : GPL-3.0+
Signature   : RSA/SHA512, Пн 25 сен 2023 15:28:07, Key ID ff979dedda2773bb
Source RPM  : kooha-2.2.4-alt1.src.rpm
Build Date  : Пн 25 сен 2023 15:28:01
Build Host  : lvol-sisyphus.hasher.altlinux.org
Relocations : (not relocatable)
Packager    : Leontiy Volodin <lvol@altlinux.org>
Vendor      : ALT Linux Team
URL         : https://github.com/SeaDve/Kooha
Summary     : Simple screen recorder with a minimal interface
Description :
Simple screen recorder with a minimal interface.
```

2.8 Обновление пакета

Чтобы обновить пакет до новой версии, используйте команду:

```
rpm -U ФАЙЛ_ПАКЕТА ...
```

Для обновления пакетов в пакетном менеджере RPM используют запрос с двумя типами ключей[4]:

1. `rpm -U` — может устанавливать и обновлять пакеты;
2. `rpm -F` — только обновляет уже установленные пакеты.

```
rpm {-U|-upgrade} [install-options] ФАЙЛ_ПАКЕТА ...
```

Команда `rpm -U` (Upgrade) обновляет пакеты до новых версий. Если установлена старая версия пакета, `rpm -U` заменит старую версию на новую, обновив пакет. Если пакет не установлен, команда `rpm -U` установит его.

[4] https://wiki.altlinux.ru/QuickStart/PkgManagment

```
rpm {-F|-freshen} [install-options] ФАЙЛ_ПАКЕТА ...
```

Команда `rpm -F` (Freshen) обновляет только те пакеты, которые уже установлены в системе. Если установлена старая версия пакета, `rpm -F` заменит старую версию на новую, обновив пакет. Если пакет не установлен, команда `rpm -F` ничего не сделает с этим пакетом[5].

Пример установки пакета:

```
# rpm -U foo-1.0-i586.rpm
```

2.9 Вопросы для самопроверки

1. С помощью какого ключа к `rpm` можно получить информацию об установленном `rpm`-пакете?
2. Как получить информацию об `rpm`-пакете, если он еще не установлен в системе, но у вас есть файл.
3. Как получить список всех файлов `rpm`-пакета?
4. Верно ли что команда `rpm -Uvh` обновит пакет, а если его нет, то установит его? Какой параметр отвечает за обновление/установку? Что означают другие параметры?
5. Как проверить установлен ли пакет с именем `foo` в вашей системе? Напишите такую команду.
6. Что делает команда `rpm -qi mate-text-editor |grep License`?

[5] https://access.redhat.com/solutions/1189

Глава 3

Программное обеспечение используемое для упаковки

В разделе 1.3 «Установка необходимых пакетов для процесса сборки» упоминался перечень пакетов для работы со сборкой:

`gcc, rpm-build, rpmlint, make, python, git, gear, hasher, patch, rpmdevtools.`

Опишем основные инструменты для управления пакетами и их сборки. Технологическую базу репозитория Sisyphus составляют адаптированные к нуждам команды разработчиков программы и специально разработанные решения[1]. К ним можно отнести[2]:

- **RPM** (менеджер пакетов). Используется для просмотра, сборки, установки, инспекции, проверки, обновления и удаления отдельных программных пакетов. Каждый такой пакет состоит из набора файлов и информации о пакете, включающей название, версию, описание пакета и т. д. Отличительными особенностями `RPM` в Sisyphus являются: удобное поведение «по умолчанию» для уменьшения количества шаблонного кода в `.spec`-файлах, обширный модульный набор макросов для упаковки различных типов пакетов, развитые механизмы автоматического вычисления межпакетных зависимостей при сборке пакетов, поддержка `set`-версий в зависимостях на разделяемые библиотеки, автоматическое создание пакетов с отладочной информацией с поддержкой зависимостей между такими пакетами. В контексте данной темы `RPM` рассматривается не только как менеджер пакетов, но и как набор инструментов для их сборки (`rpmbuild`).
- **APT** (усовершенствованная система управления программными пакетами). Умеет автоматически разрешать зависимости при установке, обеспечивает установку из нескольких источников и целый ряд других уникальных

[1] https://wiki.altlinux.ru/QuickStart/PkgManagment
[2] https://www.altlinux.org/Репозиторий_СПО#APT_и_репозиторий_пакетов.

возможностей, включая получение последней версии списка пакетов из репозитория и обновление системы.

- **Hasher** — инструмент для безопасной, воспроизводимой и высокопроизводительной сборки RPM-пакетов в контролируемой среде.
- **Gear** — набор инструментов для поддержки совместной разработки RPM-пакетов в системе контроля версий git. Gear интегрирован с hasher и rpmbuild. Gear подготавливает SRPM из git репозитория, распаковывает его и запускает hasher или rpmbuild.

3.1 Описание RPM-пакета

RPM-пакет — это специальный архив с файлами. Сам файл пакета состоит из четырёх секций — начального идентификатора, сигнатуры, бинарного заголовка и cpio-архива с файлами проекта и деревом каталога[3].

RPM-пакеты делятся на несколько категорий — пакеты с исходным кодом, бинарные пакеты и платформо-независимые бинарные пакеты.

- **RPM-пакет** (бинарные) — это архив с расширением .rpm. Такой пакет содержит скомпилированные под определённую процессорную архитектуру исполняемые файлы и библиотеки. На системах с разной процессорной архитектурой не получится использовать один и тот же скомпилированный бинарный rpm-пакет.
- **noarch-пакет** — платформо-независимый бинарный пакет.
- **SRPM-пакет** (source RPM, пакет с исходным кодом) — это архив с расширением .src.rpm. SRPM содержит исходный код, патчи, если необходимо, и SPEC-файл. Эти пакеты содержат всю информацию для сборки пакета.

По инструкциям из SPEC файла собирается бинарный RPM-пакет[4]. На основе бинарных пакетов строится база данных в /var/lib/rpm. Вся информация о пакетах хранится в базе данных Packages. Инструкции содержат также информацию о правах доступа и их применении в процессе установки, скрипты, запускаемые при установке или удалении пакета.

Принято называть пакеты RPM по типу:

```
имя-версия-релиз.процессорная_архитектура.rpm
```

При этом в системе имя установленного пакета будет отличаться: в командной строке можно обратиться за информацией по пакету, указывая только его имя, если установлена одна версия пакета, и указывая имя-версию-релиз, если установлено больше версий[5].

[3] https://www.opennet.ru/docs/RUS/rpm_guide/13.html
[4] https://uneex.ru/static/RedHatRPMGuideBook/rpm_guide-linux.html#16_html
[5] https://www.opennet.ru/docs/RUS/rpm_guide/13.html

3.2 Инструменты для сборки RPM-пакетов

Инструменты сборки rpm-пакета — это пакеты и программы, с помощью которых из набора исходных файлов можно получить специальный архив в формате rpm-пакета. Приведём список этих инструментов[6]: `rpmdevtools`, `rpmdev-setuptree`, `rpmbuild`, `rpmspec`, `rpmlint`.

- **rpmdevtools** — пакет с набором программ для сборки пакетов;
  - **rpmdev-setuptree** — утилита для создания структуры рабочих каталогов;
  - **rpmdev-newspec** — утилита для создания spec-файла;
- **rpmspec** — утилита работы с файлами спецификации — текстовыми файлами с расширением `.spec`. Служит для проверки подготовленного spec-файла;
- **rpmbuild** — утилита сборки rpm-пакета из набора подготовленных файлов;
- **rpmlint** — утилита для тестирования собранного rpm-пакета;
- **rpm-utils** — пакет с набором программ для работы с rpm-пакетами.

3.3 Рабочее пространство для сборки RPM-пакетов

Рабочая область упаковки RPM-пакета — это структура файлов и каталогов. Эту структуру можно создать двумя способами — вручную или через утилиту `rpmdev-setuptree`.

Для подготовки ручным способом структуры каталогов выполните команду:

```
mkdir -p ~/RPM/{BUILD,SRPMS,RPMS,SOURCES,SPECS}
```

Альтернативный способ подготовки рабочей среды — утилита `rpmdev-setuptree`. Утилита входит в состав пакета `rpmdevtools` (см. раздел 1.3). Для подготовки структуры каталогов через утилиту `rpmdev-setuptree` выполните команду:

```
$ rpmdev-setuptree
```

Утилита создаст базовую структуру каталогов и файл `/.rpmmacros`, если его не существовало.

Для системы ALT расположение структуры каталоговпо умолчанию определятся в файле `/.rpmmacros` и находится в каталоге `/RPM`:

[6] https://www.altlinux.org/Сборка_пакета_с_нуля

```
$ tree ~/RPM
/home/tefst/RPM
├── BUILD
├── RPMS
├── SOURCES
├── SPECS
└── SRPMS

5 directories, 0 files
```

1. **BUILD** — в каталог попадают распакованные исходные файлы из `SOURCES` с уже применёнными патчами — стадия `%prep`. В каталоге `BUILD` происходит сборка программного обеспечения.

2. **RPMS** — в каталог `RPMS` бинарные `RPM`-пакеты после сборки, в соответствии с подкаталогами для поддерживаемых архитектур.

3. **SOURCES** — в каталоге размещают архивы исходных данных и патчи.

4. **SPECS** — в каталоге размещают`spec`-файлы пакетов для сборки.

5. **SRPMS** — в каталог SRPMS попадают результаты сборки SRPM пакетов.

Созданная структура каталогов становится рабочей областью упаковки `RPM`-пакета.

В структуре сборочного окружения `RPM` существует понятие `buildroot`. Это каталог `/TMP`, в который попадают служебные файлы в ходе сборки пакета и уже подготовленные бинарные данные для упаковки в соответствии со структурой каталогов. По умолчанию создаётся во временном системном каталоге. Может быть переназначен.

3.4 Описание SPEC-файла

SPEC-файл — RPM Specification File — это текстовый файл, который описывает процесс сборки и конфигурацию пакета, служит инструкцией для утилиты `rpmbuild`. Он содержит метаданные, такие как имя пакета, версию, лицензию, а также разделы с инструкциями для сборки, установки и упаковки программного обеспечения, журнал изменений пакета[7]. `SPEC`-файл можно рассматривать как «инструкцию», которую утилита `rpmbuild` использует для сборки `RPM`-пакета.

`Spec`-файл состоит из трёх разделов: Header (Заголовок/Преамбула), Body (Тело) и Changelog (Журнал изменений).

[7] https://wiki.mageia.org.ru/index.php?title=Структура_RPM_SPEC-файла

1. **Header** (Заголовок) — этот раздел содержит метаданные о пакете, такие как его имя (Name), версия (Version), релиз (Release), краткое описание (Summary), лицензия (License) и другие параметры, которые идентифицируют и характеризуют пакет.

2. **Body** (Тело) — этот раздел содержит инструкции для процесса сборки пакета. В нём определяются различные секции, такие как `BuildRequires` (зависимости для сборки), `%build` (инструкции для сборки), `%install` (инструкции для установки), `%files` (список файлов, включённых в пакет), и другие.

3. **Changelog** (Журнал изменений) — этот раздел содержит историю изменений пакета. Он включает записи о внесённых изменениях, включая дату изменения, автора и краткое описание того, что было изменено.

Преамбула (Заголовок)

Заголовок SPEC-файла содержит информацию о пакете: версию, исходный код, патчи, зависимости.

Рекомендуемый порядок заголовочных тэгов:

- Name, Version, Release, Serial
- далее Summary, License, Group, Url, Packager, BuildArch
- потом Source*, Patch*
- далее PreReqs, Requires, Provides, Conflicts
- и, наконец, Prefix, BuildPreReqs, BuildRequires.

Ниже приведён пример части SPEC-файла `notepadqq`:

```
Summary: A Linux clone of Notepad++
Name: notepadqq
Version: 1.4.8
Release: alt2
License: GPLv3
Group: Editors
URL: http://notepadqq.altervista.org/wp/
Source0: %name-%version.tar
Source1: codemirror.tar
```

3.5 Пример .spec-файла

SPEC-файл — RPM Specification File — это текстовый файл, который описывает процесс сборки и конфигурацию пакета. SPEC-файл можно рассматривать как «инструкцию», которую утилита `rpmbuild` использует для сборки RPM-пакета.

Он содержит метаданные, такие как имя пакета, версию, лицензию, а также разделы с инструкциями для сборки, установки и упаковки программного обеспечения, журнал изменений пакета.

Представим образцы SPEC-файлов[8] — шаблонный образец, образец для программы на `autotools`, образец для программы на `cmake` и образец модуля для `Python 3`.

Примеры SPEC-файлов

Пример 1. Пустой SPEC

```
Name: <имя-пакета>
Version: <версия-пакета>
Release: alt<релиз-пакета>

Summary: <однострочное описание>
License: <лицензия>
Group: <группа>

Url: <URL>
Source: %name-%version.tar
Patch1:

PreReq:
Requires:
Provides:
Conflicts:

BuildPreReq:
BuildRequires:
%{?!_without_test:%{?!_disable_test:%{?!_without_check:%{?!_disable_check:BuildRequires: }}}}
BuildArch:

%description
<многострочное
описание>

%prep
%setup
%patch1 -p1

%build
%configure
%make_build

%install
%makeinstall_std

%check
%make_build check

%files
%_bindir/*
%_man1dir/*
%doc AUTHORS NEWS README

%changelog
* <дата> <ваше имя> <$login@altlinux.org> <версия_пакета>-<релиз_пакета>
- initial build for ALT Linux Sisyphus
```

[8] https://www.altlinux.org/SampleSpecs

Пример 2. Для программы (на autotools)

Название GNU Autotools обычно относится к программным пакетам Autoconf, Automake, Libtool и Gnulib. Вместе они составляют систему сборки GNU. Этот SPEC-файл является примером пакета с программой.

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

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

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

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

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

%description -l ru_RU.UTF-8

%prep
%setup
%patch0 -p1

%build
%configure
%make_build

%install
%makeinstall_std
%find_lang %name

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

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

Пример 3. Для программы на cmake

CMake — это кроссплатформенный инструмент с открытым исходным кодом, который использует независимые от компилятора и платформы файлы конфигурации для создания собственных файлов инструментов сборки, специфичных для вашего компилятора и платформы. Является стандартом де-факто для сборки кода на C++.

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

Summary: Sample program specfile
License: GPLv2+
Group: Development/Other
```

```
Url: http://www.altlinux.org/SampleSpecs/cmakeprogram
Source: %name-%version.tar.bz2

BuildRequires(pre): cmake rpm-macros-cmake

%description
This specfile is provided as a sample specfile
for a package built with cmake.

%prep
%setup

%build
%cmake_insource
%make_build # VERBOSE=1

%install
%makeinstall_std
%find_lang %name

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

%changelog
* Sat Jan 33 3001 Example Packager <example@altlinux.org> 1.0-alt1
- initial build
```

Пример 4. Модуль для `Python 3`

Макросы для сборки модулей `python3` содержатся в пакете `rpm-build-python3` и аналогичны тем, что используются в ALT для `python`[9].

```
%define pypi_name @NAME@
%define mod_name %pypi_name

%def_with check

Name: python3-module-%pypi_name
Version: 0.0.0
Release: alt1
Summary: @TEMPLATE@
License: MIT
Group: Development/Python3
Url: https://pypi.org/project/@NAME@
Vcs: @SOURCE_GIT@
BuildArch: noarch
Source: %name-%version.tar
Source1: %pyproject_deps_config_name
Patch: %name-%version-alt.patch

# mapping of PyPI name to distro name
Provides: python3-module-%{pep503_name %pypi_name} = %EVR

%pyproject_runtimedeps_metadata
BuildRequires(pre): rpm-build-pyproject
%pyproject_builddeps_build
%if_with check
%pyproject_builddeps_metadata
%endif
```

[9] https://www.altlinux.org/Python3

```
%description
@DESCR@

%prep
%setup
%autopatch -p1
%pyproject_deps_resync_build
%pyproject_deps_resync_metadata

%build
%pyproject_build

%install
%pyproject_install

%check
%pyproject_run_pytest

%files
%doc README.*
%python3_sitelibdir/%mod_name/
%python3_sitelibdir/%{pyproject_distinfo %pypi_name}

%changelog
```

3.6 Составляющие основной части

SPEC-файл — RPM Specification File — это текстовый файл, который описывает процесс сборки и конфигурацию пакета, служит инструкцией для утилиты `rpmbuild`. Он содержит метаданные, такие как имя пакета, версию, лицензию, а также разделы с инструкциями для сборки, установки и упаковки программного обеспечения, журнал изменений пакета. SPEC-файл можно рассматривать как «инструкцию», которую утилита `rpmbuild` использует для сборки RPM-пакета.

SPEC-файл состоит из трёх разделов:

1. Header (Заголовок/Преамбула);
2. Body (Тело);
3. Changelog(Журнал изменений).

Тело SPEC-файла отвечает за выполнение сборки, установки или очистки пакета.

Описание структуры

- В секции `%prep` производится распаковка архивов с исходными кодами и формируется директория с исходниками[10].
  - Макрос `%setup` этой секции выполняет смену рабочего каталога на каталог сборки и распаковку в него архивов с исходными кодами.
  - Макрос `%patch1` будет описывать применение патча.

[10] https://www.opennet.ru/docs/RUS/rpm_guide/48.html

- В секции `%build` внутри ранее подготовленной директории производится сборка программы. Если это компилируемый язык, то исходники компилируются в бинарные файлы. Если это интерпретируемый язык, то процесс может не подразумевать компиляцию. Обычно за процесс сборки отвечают системы сборки, отличающиеся для разных языков программирования. Для `C/C++` обычно используется `automake/autoconf` и макросы `%configure` и `%make_build`. Есть и другие системы сборки с другими макросами — `CMake`, `meson`, `pyproject` и т. д.

- В секции `%install` подготавливается новая директория с теми файлами, которые будут помещены в `RPM` пакет в конце процесса сборки. Эта директория обозначается макросом `%buildroot`. Из текущей директории подготовленные на предыдущем этапе файлы (бинарные файлы, файлы документации, конфигурационные файлы и т. д.) нужно перенести в `%buildroot`. Например файл `build/application.bin` нужно перенести в `%buildroot/usr/bin/application.bin`. За это в некоторых случаях может также отвечать система сборки. Например, `automake/autoconf` так умеет. Запускается через макрос `%makeinstall_std`. Для других систем сборки есть другие макросы.

- Возможно добавление секции `%clean`. Её задача — очистить дерево сборки и каталог установки.

- Если разработчик добавляет в `SPEC`-файл собственные скрипты, их следует распределять в секции:

  - `%pre` (выполнение перед установкой)
  - `%post` (выполнение после установки)
  - `%preun` (перед удалением пакета)
  - `%postun` (после удаления пакета)

- Секция `%files` содержит список путей и файлов, которые будут упакованы в `RPM`-пакет и в дальнейшем установлены в систему.

  - В этой секции можно создать каталог (`%dir`), отметить, что файл является документацией (`%doc`) или файлом конфигурации (`%config`), или файл не относится к пакету, но необходим в начале работы приложения (`%gost`).

Сборка `rpm`-пакета выполняет все инструкции, указанные в `SPEC`-файле. В процессе сборки утилита `rpmbuild` выводит на экран информацию о процессе сборки. С помощью этой информации можно отследить возможные ошибки описания `SPEC`-файла.

3.7 RPM макросы

Макрос RPM — это именованная переменная, которая напрямую подставляет текст в SPEC-файл во время сборки rpm-пакета. Имена макросов начинаются с символа %. Имена макросов — это сокращённые псевдонимы для часто используемых фрагментов текста.

Ниже приведены примеры макросов[11]:

1. **Пример макроса, содержащего значение.** Если во время сборки некоторым командам необходимо передать имя собираемого пакета, то можно передавать им макрос %name. Во время сборки этот макрос подставляет имя пакета, объявленное в начале SPEC-файла:

```
%define some_macro foo
....
Name: bar-%some_macro
.....
%build
%dune_build -p %some_macro
```

2. **Пример макроса с набором команд.** %cmake_build — макрос необходимый для сборки пакетов с помощью cmake. Он подставляет следующую последовательность команд:

```
mkdir -p x86_64-alt-linux-gnu;
cmake \
-DCMAKE_SKIP_INSTALL_RPATH:BOOL=yes \
-DCMAKE_C_FLAGS:STRING='-O2 -g' \
-DCMAKE_CXX_FLAGS:STRING='-O2 -g' \
-DCMAKE_Fortran_FLAGS:STRING='-O2 -g' \
-DCMAKE_INSTALL_PREFIX=/usr \
-DINCLUDE_INSTALL_DIR:PATH=/usr/include \
-DLIB_INSTALL_DIR:PATH=/usr/lib64 \
-DSYSCONF_INSTALL_DIR:PATH=/etc \
-DSHARE_INSTALL_PREFIX:PATH=/usr/share \
-DLIB_DESTINATION=lib64 \
-DLIB_SUFFIX="64" \
-S . -B "x86_64-alt-linux-gnu"
```

Преимущества использования макросов:

- упрощение сборки;
- унификация SPEC-файлов;
- подбор шаблонов для создания SPEC-файлов;

[11] https://www.altlinux.org/Spec/Предопределённые_макросы

- сокращение размера SPEC-файлов позволяет упростить отладку;
- использование макросов обеспечивает гибкость в настройке и конфигурации пакетов, позволяя быстро изменять параметры сборки.

Где объявлены RPM-макросы:

- Стандартные макросы объявляет установка пакета `rpmbuild (librpm)`. Информацию о них можно получить из файла `/usr/lib/rpm/macros` или выполнив команду: `rpm -showrc`;
- Макросы можно объявить самостоятельно, добавив в SPEC-файл;
- Макросы можно объявить в отдельных файлах. Команда `%include` позволяет загрузить специальные файлы с объявленными макросами;
- Макросы, объявленные в файлах, поставляемые с другими пакетами.

  Файлы с объявленными RPM-макросами хранятся в каталоге

  `/usr/lib/rpm/macros.d`.

  Например, пакет `rpm-build-ruby` содержит готовые макросы для сборки пакетов с программами, написанными на языке Ruby. Для того чтобы использовать эти макросы, необходимо этот пакет добавить в зависимости: `BuildRequires(pre): rpm-build-ruby`.

Команда получения значения макроса: `rpm -eval <имя_макроса>`

Некоторые макросы могут быть вложенными.

3.8 Вопросы для самопроверки

1. Какие пакеты используются для сборки rpm-пакетов?
2. Какая утилита используется для непосредственной сборки бинарного пакета?
3. Верно ли что `hasher` обеспечивает воспроизводимую сборку в контроллируемой среде?
4. Каково внутреннее устройство RPM-пакета?
5. У вас есть пакет `admc-0.15.0-alt1.x86_64.rpm`.

   a) можно ли его установить на компьтер с процессором Байкал-М?
   b) А на компьютер с процессором Эльбрус 16С?
   c) Какой пакет нужно взять, чтобы собрать тот же пакет под другую целевую архитектуру?

6. Какая структура каталогов необходима для сборки `rpm`-пакета?
7. В чём смысл понятия `buildroot`?
8. Для чего служит `SPEC`-файл?
9. Из каких обязательных частей состоит `SPEC`-файл?
   a) В какой части `SPEC`-файла указывается имя пакета?
   b) В какой части `SPEC`-файла указывается лицензия под которой распространяется пакет?
   c) В какой части `SPEC`-файла указывается история изменений пакета?
10. Что такое `RPM`-макрос?
    a) Какие преимущества даёт использование макросов?
    b) Как найти стандартные макросы?
    c) Где в системе находтся макросы, поставляемые сторонними пакетами?

Глава 4

Инструмент Gear

GEAR (Get Every Archive from git package Repository) — инструмент для поддержки и сборки пакетов из `git`-репозиториев[1,2].

Для работы с текущим разделом необходимо изучить документацию по работе с распределённой системой управления версиями `Git`.

Git — это технология контроля версий, которая следит за изменениями в файлах. `Git`-репозиторий хранит исходный код и данные для сборки пакетов, включая историю изменений, сведения о версиях, авторах изменений и прочую информацию.

При сборке пакетов `gear` предлагает работать в том же `git`-репозитории, в котором хранятся исходные тексты пакета. `Gear` позволяет целиком импортировать историю разработки, предоставляет различные средства импорта исходных текстов и различные варианты организации репозитория.

Идея `gear` в доступности всего необходимого для сборки пакета в `git`-репозитории, либо в репозитории пакетов, на основе которого ведётся сборка. Система контроля версий `Git` предоставляет встроенные механизмы обеспечения целостности (контрольные суммы, криптографически подписанные теги) для задач управления пакетами. Появляется возможность воспроизводимой сборки — собрать «такой же» пакет ещё раз, опираясь на логически законченную версию изменений зафиксированную на момент времени — коммит (commit) в терминологии git.

Сборка пакета ведётся из конкретного коммита, который мы в дальнейшем будем называть главным. Именно в нём должна находится нужная версия `SPEC`-файла и правил экспорта. `GEAR` позволяет экспортировать из `git`-репозитория каталоги и подкаталоги в виде архивов; экспортировать отдельные файлы; вычислять разницу (`diff`) и сохранять её в виде патча. При этом может использо-

[1] https://www.altlinux.org/Gear
[2] https://www.altlinux.org/Gear/Справочник

ваться как состояние в главном коммите, так и в любом из его предков, прямых и не прямых. Это позволяет гибко и удобно организовать работу по поддержке пакета: все нужные исходные тексты можно хранить в `git` так, чтобы с ними было удобно работать, а затем экспортировать так, чтобы ими было удобно воспользоваться из `SPEC`-файла.

Правила экспорта для `gear` описываются в текстовом файле, который также хранится в репозитории. По умолчанию `gear` ищет этот файл по пути `.gear/rules` или `.gear-rules`. Подробнее про синтаксис этого файла можно прочитать ниже, в разделе «Правила экспорта».

Для удобства сборки пакетов `gear` интегрирован с инструментами `rpm-build` и `hasher`: из `gear`-репозитория можно одной командой собрать пакет при помощи `rpmbuild` или `hsh`. Подробнее об этом можно прочитать далее в разделе «Быстрый старт».

4.1 Структура репозитория

Обобщим опыт, упрощающий работу с `git`-репозиториями, предназначенными для хранения исходного кода пакетов.

- **Одна кодовая база — один репозиторий.**

  Не имеет смысла хранить в одном репозитории исходные тексты, не связанных между собой пакетов. В этом правиле бывают исключения в случае, если культура разработки предполагает хранение разных составных частей проекта в одном репозитории.

- **Храните в `git`-репозитории распакованный исходный код.**

  Исходные тексты, поставляемые в виде архивов (`tar`, `zip`) и сжатые различными компрессорами (`gz`, `bzip2`, `xz`) удобнее распаковать. Это упрощает использование средств `git` для работы с этими данными: так легче отслеживать изменения, ниже трафик при обновлениях и т. д.

- **Отделяйте свои изменения от изменений исходной кодовой базы.**

  Если вы не являетесь разработчиком пакета, который собираете, лучше хранить свои изменения отдельно от кода разработчиков, например в отдельной ветке (и формировать `diff` средствами `GEAR`) или в виде патчей. Это заметно упрощает совместную работу над пакетом, обновления и аудит.

4.2 Правила экспорта

Правила экспорта для `GEAR` описываются в текстовом файле, который также хранится в репозитории. По умолчанию `GEAR` ищет этот файл по пути `.gear/rules` или `.gear-rules`, от корня репозитория, в главном коммите.

Этот файл состоит из одной или нескольких строк, каждая из которых имеет следующий формат: `<директива>: <параметры>`

Параметры разделяются пробельными символами. Пробелы между директивой и «:» и «:» и параметрами не обязательны.

Пустые строки и строки, начинающиеся с «#», игнорируются.

В значениях многих параметров и опций директив могут применяться ключевые слова:

- `@name@` — будет заменено на имя пакета (извлекается из SPEC-файла);
- `@version@` — будет заменено на версию пакета (извлекается из SPEC-файла);
- `@release@` — будет заменено на релиз пакета (извлекается из SPEC-файла).

Теги и пути

По умолчанию все пути в аргументах директив считаются от корня репозитория главного коммита. Однако большинство директив позволяет указать другой коммит в качестве основы. Для этого путь должен быть передан в формате:

```
base_tree:path_to_file.
```

В качестве `base_tree` может выступать:

- полный идентификатор коммита (SHA-1, 40 шестнадцатиричных цифр);
- имя тега GEAR;
- символ «.», обозначающий главный коммит.

В любом случае, коммит, на который так ссылаются, должен быть предком главного коммита.

Основные директивы

Ниже приведены основные директивы GEAR, и их аргументы. Подробнее с ними можно ознакомиться в `man gear-rules(5)`.

- `spec: <путь>`

  Задаёт путь к SPEC-файлу. По умолчанию, GEAR использует файл с расширением `.spec` из корня репозитория в главном коммите, если такой файл там только один. Единственный аргумент — путь к SPEC-файлу.

- `copy: <glob>...` Скопировать файл, соответствующий указанному шаблону поиска (glob pattern). Может принимать несколько аргументов, для каждого из которых должны быть найдены соответствующие файлы.

  Также существуют директивы `gzip`, `bzip2`, `lzma`, `lzma`, `zstd`, аналогичные copy, но сжимающие экспортированный файл подходящим алгоритмом сжатия.

- `tar: <tree_path>`

  Экспортировать каталог из репозитория в виде `tar`-архива. Допустимые опции:

  - `name=<archive_name>` имя архива (без суффикса `.tar`);
  - `base=<base_name>` внутри архива будет создан каталог с указанным именем, и все файлы будут помещены в него;
  - `suffix=<suffix>` расширение создаваемого архива (по умолчанию — `.tar`);
  - `exclude=<glob_patter>` не включать в архив файлы, соответствующие указанному шаблону поиска.

  Помимо стандартных ключевых слов, в опциях `name` и `base` может применяться ключевое слово `@dir@`, которое будет заменено на имя каталога из параметра `tree_path`.

- `zip: <tree_path>`

  Экспортировать каталог из репозитория в виде `zip`-архива. Принимает те же аргументы, что и директива `tar`, использует `.zip` в качестве расширения по умолчанию.

  Также существуют директивы `tar.gz, tar.bz2, tar.lzma, tar.xz, tar.zst`, аналогичные `tar`, но сжимающие созданный архив подходящим алгоритмом сжатия. Чаще всего используются несжатые архивы, так как сжатия, используемого при сборке `SRPM`, обычно достаточно.

- `diff: <old_tree_path> <new_tree_path>`

  Создать `unified diff` между указанными каталогами и сохранить его в виде патча. Допустимые опции:

  - `name=<diff_name>` имя создаваемого файла;
  - `exclude=<glob_patter>` игнорировать файлы, соответствующие указанному шаблону поиска.

  Помимо стандартных ключевых слов, в опции `name` могут применяться ключевые слова `@old_dir@` и `@new_dir@`, которые будут заменены на имя каталога из параметра `old_tree_path` и `new_tree_path` соответственно.

Все приведённые директивы требуют, чтобы все указанные в них файлы и каталоги существовали; если какого-то файла или каталога не будет существовать, экспорт завершится ошибкой. Однако существуют аналогичные им директивы, заканчивающиеся знаком вопроса (например, `tar?:` или `copy?:`), которые игнорируют отсутствующие файлы. Это может быть удобно, чтобы не приходилось менять правила экспорта при добавлении и удалении патчей.

4.3 Основные типы устройства gear-репозитория

Гибкость GEAR[3] означает, что каждый пользователь может настроить его по своему усмотрению. Существует несколько распространённых способов организации gear-репозитория в качестве основы для более сложных конфигураций. Знакомство с ними поможет понять организацию репозитория при совместной работе над пакетами.

Базовые виды ведения GEAR репозиториев:

- **Архив с исходными текстами и патчи.**

  Исходные тексты хранятся в каталоге package_name; дополнительные изменения хранятся в виде патчей. Подобные репозитории создаёт команда gear-srpmimport, и также выглядят импортированные пакеты в git.altlinux.org/srpms.

  В каталоге package_name принято хранить не изменённые исходники; для их обновления удобно использовать команду gear-update.

  Такой формат будет больше всего знаком пользователям без опыта поддержки пакетов source RPM, и при работе с проектами, не имеющими публичного git-репозитория или не использующими git.

  Пример .gear/rules:

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

- **Репозиторий с историей исходного репозитория и модифицированными исходными текстами.**

  Создаётся копия исходного git-репозитория. Находится коммит, из которого нужно взять исходные тексты для сборки. На основе этого коммита добавляется каталог .gear и SPEC-файл. При обновлении новые исходные тексты сливаются (merge) в эту ветку. Создаётся gear-тег, соответствующий собираемой версии. Изменения могут храниться в виде патчей. Но чаще вносятся в текущую ветку со spec-файлом, затем изменения экспортируются в виде одного большого патча.

  Пример .gear/rules с генерацией патча:

  ```
  tar: v@version@:.
  diff: v@version@:. . exclude=.gear/** exclude=*.sp
  ```

- **Пустая ветка со SPEC-файлом и отдельная история разработки.**

  Изначально SPEC-файл и .gear ведутся в отдельной ветке, содержащей главный коммит. Исходные тексты, необходимые для сборки пакета, сливаются (merge) при необходимости с git стратегией ours — таким образом,

[3] https://www.altlinux.org/Руководство_по_gear

нужные коммиты оказываются в истории главного, но сами исходные тексты в ветку не попадают. Если нужно внести какие-то специфичные изменения, они вносятся в отдельную ветку, например `alt-fixes`. При каждом обновлении кода ветка `alt-fixes` и соответствующий ей `GEAR`-тег должны обновляться. В нашем примере `GEAR`-тег совпадает с именем ветки.

Пример `.gear/rules`:

```
tar: v@version@:.
diff: v@version@:. alt-fixes:.
```

4.4 Быстрый старт Gear

Установка gear

В разделе 1.3 «Установка необходимых пакетов для процесса сборки» также упоминается инструмент `gear`. Для отдельной установки используйте команду:

```
#apt-get install gear
```

Импорт .src.rpm

Пакет в формате `source RPM` можно импортировать в `gear`-репозиторий командой `gear-srpmimport`:

```
mkdir package_name
cd package_name
git init -b sisyphus
gear-srpmimport /путь/к/package_name.src.rpm
```

Получение готового репозитория с внешнего git-сервера

Достаточно клонировать репозиторий:

```
git clone <repository url> package_name
cd package_name
```

Никаких дополнительных настроек не требуется.

Сборка пакета

Основным инструментом экспорта и сборки пакетов является команда `gear`. На практике удобно использовать предоставляемые команды-обёртки, а к самой команде `gear` прибегать только в самых сложных случаях. Командной обёртке можно передавать опции как утилиты `gear`, так и вложенной утилиты.

Командная обертка `gear` для `rpmbuild` — `gear-rpm`. Для сборки пакета используйте команду:

```
gear-rpm -verbose -ba
```

Собрать пакет при помощи `hsh` (установка и настройка `hasher` описана в следующих главах):

```
gear-hsh -verbose
```

Команде `gear-hsh` можно передать как аргументы `gear`, так и аргументы `hsh`.

Если не указана опция `-tree-ish`, `gear` в качестве главного коммита использует текущий коммит (HEAD). Если хочется проверить свежие изменения без создания коммита, можно использовать опцию `-commit`, доступную и для `gear`, и для `gear-rpm`, и для `gear-hsh`. В таком случае будет создан временный коммит (аналогично `git commit -a`), и пакет будет собран уже из него. Стоит отметить, что, аналогично `git commit -a`, в таком коммите не будет новых файлов, если они ещё не были добавлены в `git` командой `git add`; если они нужны для сборки, то их стоит добавить.

Сборка пакета из репозитория разработчиков

Для примера возьмём конкретный пакет, который уже есть в репозитории Сизиф, например `pixz`, и попробуем собрать его с нуля.

Для начала клонируем репозиторий. Сразу зададим имя удалённого репозитория:

```
git clone https://github.com/vasi/pixz -o upstream
cd pixz
```

Перейдём в ветку, из которой будем собирать пакет:

```
git checkout -b sisyphus upstream/master
```

Переместимся на тег (тег — это ссылка, указывающие на определённый `git`-коммит в репозитории), соответствующий версии, которую мы будем собирать. На момент написания пособия последний тег `v1.0.7`:

```
git reset -hard v1.0.7
```

Определим правила экспорта:

```
mkdir .gear
vim .gear/rules
```

Правила зададим следующие:

- `SPEC`-файл перенесём в каталог `.gear`, чтобы он не путался с основными исходными текстами;
- исходники будем забирать из тега `GEAR`, соответствующего апстримному;
- сразу создадим патч, включающий все наши изменения, каталог `.gear` в этот патч мы включать не будем.

Получаем следующий файл `.gear/rules`:

```
spec: .gear/pixz.spec
tar: v@version@:.
diff: v@version@:. . exclude=.gear/**
```

Создадим соответствующий версии тег GEAR:

```
gear-store-tags v1.0.7
```

Напишем SPEC-файл:

```
vim .gear/pixz.spec
```

Важно, что версия в спеке — 1.0.7, поэтому конструкция v@version@, которую мы использовали в .gear/rules, раскроется в строку v1.0.7 — именно такой тег GEAR мы создали.

Добавим каталог .gear в git:

```
git add .gear
```

Можно попробовать собрать пакет:

```
gear-rpm -verbose -commit -ba
```

При необходимости, можно поправить spec, добавить нужные зависимости. Когда пакет собирается и всем устраивает, можно зафиксировать все изменения:

```
gear-commit -a
```

4.5 Вопросы для самопроверки

- Что такое инструмент git?
- Что такое инструмент gear?
- Где хранятся правила экспорта для gear?
- Перечислете правила, упрощающие работу с git-репозиториями?
- Какой формат у файла с правилами экспорта для gear?
- Что означает параметр @name@ в файле .gear-rules?
- Как формируется путь в аргументах директив в файле .gear-rules?
- Перечислите основные дерективы в файле .gear-rules?
- Какой командой собирается пакет с помощью gear?
- Как создать тег соответствующий версии пакета с помощью gear?

Глава 5

Инструмент Hasher

Hasher — инструмент для сборки пакетов с использованием минимальной контролируемой среды в `chroot`. `Hasher` опирается на системный вызов `chroot` и создаёт изолированную среду для сборки отдельного пакета утилитой `rpm-build`.

`Hasher` облегчает поддержание сборочных зависимостей и позволяет собирать пакеты для разных дистрибутивов. `Hasher` проверяет пакеты с помощью утилиты `sisyphus_check` и создаёт локальный `APT`-репозиторий с результатами сборки.

Для подготовки сборочного окружения `hasher` берёт пакеты, как из удалённых репозиториев, настроенных в основной системе, так и из локального репозитория (`~/hasher/repo/`), в который попадают ранее собранные пакеты.

Так же `hasher` удобен для отладки процесса сборки. Если сборка пакета прервалась, выполните команду `hsh-shell`, чтобы попасть в терминал `chroot`, исправьте ошибку и продолжайте сборку с прерванного этапа.

5.1 Описание системы Hasher

Опишем структуру каталогов в `hasher`.

- `hasher`
  - `chroot` — само минимальное окружение. В этом каталоге находится корневое дерево содержащее минимальный набор пакетов, `rpmbuild` и сборочные зависимости.
  - `aptbox` — набор утилит для установки, обновления и удаления пакетов `chroot`. Например, тут лежит модифицированный `apt-get`, с помощью которого происходит установка пакетов в `chroot`.
  - `cache` — в этом каталоге кэшируются файлы, необходимые для создания `chroot`.
    - каталог `~/hasher/chroot/usr/src/RPM`, который содержит подкаталоги для `rpmbuild`:

        - BUILD
        - RPMS
        - SOURCES
        - SPECS
        - SRPMS
    - repo, который содержит подкаталоги:
        * SRPMS.hasher — исходники (sources) пакета.
        * <платформа>/RPMS.hasher/ — каталог с пакетами, собранными под конкретную архитектуру. В этот каталог можно добавить необходимые сборочные зависимости, когда пакета ещё нет в репозитории и оценить сборку.

- ~/.hasher
    - apt.config — конфигурация для apt-get из ~/hasher/aptbox/
    - config — конфигурация самого hasher

- /etc/hasher-priv/ каталог с конфигурацией для вспомогательной утилиты hasher-priv
    - ./user.d
    - fstab — информация о точках монтирования вспомогательной программы hasher-priv system — конфигурация вспомогательной программы hasher-priv

5.2 Принцип действия

В каталоге ~/hasher/chroot создаётся сборочная среда. В неё устанавливается минимальный набор пакетов, rpmbuild и сборочные зависимости. Вся работа, связанная непосредственно со сборкой пакетов, происходит в этом chroot. Hasher не использует одну и ту же созданную среду. У каждого пакета свои зависимости и поэтому для каждого пакета его изолированное сборочное окружение будет уникальным. Для этого в Hasher заложено создание новой сборочной среды под каждый собираемый пакет. При успешном завершении сборки каталог chroot очищается

Сборочное окружение можно сохранить. Для этого используют флаг: -lazy-cleanup.

Сборочное окружение каждый раз удаляется при инициализации нового chroot.

Из принципа действия Hasher следует:

- сборка не пройдёт корректно, если в пакете не указаны все зависимости;

- сборку в `hasher` можно воспроизвести на другом компьютере, поскольку `hasher` не использует индивидуальные пользовательские настройки;
- в настройках можно определить, для какого репозитория будет проходить сборка. Поэтому не меняя пользовательское окружение, можно собрать пакет под разные дистрибутивы.

5.3 Настройка Hasher

В разделе 1.3 «Установка необходимых пакетов для процесса сборки» упоминалась установка пакета `hasher`. Для отдельной установки используйте команду:

```
# apt-get install hasher
```

Добавьте пользователя в группу `rpm`:

- для локального пользователя:

```
# usermod -aG rpm <user>
```

- для доменного пользователя:

```
# gpasswd -a <username> rpm
```

Запустите демон `hasher-priv`:

```
# systemctl enable -now hasher-privd.service
```

Укажите учётную запись для работы с `hasher`. Добавьте учётные записи для работы с `hasher` в группу `hashman`:

```
# hasher-useradd <user>
```

Заново авторизуйтесь, чтобы изменения вступили в силу.

Укажите расположение сборочной среды (каталоги настроек и их инициализации):

```
$ mkdir ~/hasher
$ mkdir ~/.hasher
```

При создании каталога `.hasher` следует учитывать два правила:

1. права доступа соответствуют `drwxr-xr-x`, то есть каталог доступен на запись;
2. на файловой системе, смонтированной с `noexec` или `nodev`, каталог располагать нельзя[1];

[1] https://serverfault.com/questions/547237/explanation-of-nodev-and-nosuid-in-fstab

- `noexec` устанавливается, если в системе есть файлы с правами `rwsr-xr-x` (запустить исполняемые файлы с правами владельца или группы) и владельцем `root`. Запустить файл `rwxr-xr-x` на такой файловой системе невозможно. а следовательно, и создать корректное сборочное окружение. `Hasher` не зависит от пользовательского окружения.
- `nodev` говорит о том, что на файловой системе не будут созданы файлы устройств. Это не соответствует функциональности `hasher` (см. раздел 5.7 «Монтирование файловых систем внутри `Hasher`»).

Создайте конфигурационный файл:

```
packager="‘rpm -eval %packager‘"
no_sisyphus_check="packager,buildhost,gpg"
allowed_mountpoints=/dev/pts,/proc,/dev/shm
lazy_cleanup=1
# share_ipc=1
# Необходимо в случае, когда требуется разрешить доступ в сеть
# из Hasher chroot.
# Данная опция не должна применяться без особой необходимости.
# share_network=1
# нужен для сборки с репозиториями отличающимися от системного:
# apt_config="$HOME/.hasher/apt.conf"
```

Создадим сборочное окружение:

```
$ hsh -initroot-only
# путь ~/hasher является параметром по умолчанию.
# Полностью команда выглядит так:
$ hsh -initroot-only ~/hasher
$ ls -al ~/hasher
```

Если сборочное окружение не создано, оно построится при первой сборке пакета.

Чем больше пакет, тем больше времени понадобится на его сборку. В конфигурационном файле `hasher-priv` прописываются ограничения на ресурсы, выделяемые на сборку (CPU, память, общее время исполнения и другие). Чтобы увеличить время бездействия утилиты `hasher-priv`, требуется отредактировать файл `/etc/hasher-priv/system/`:

```
wlimit_time_elapsed=360000
wlimit_time_idle=360000
```

Сборочная среда строится по умолчанию на источниках для основной системы (пользователя/хост-системы).

5.4 Сборка в Hasher

Сборка в `hasher` предполагает два возможных сценария[2]:

1. **Сборка из `.src.rpm`.**

   Этот сценарий подходит, если у нас уже есть `.src.rpm` (см. раздел 3.1 «Описание RPM-пакета»).

   Команда сборки выполняется от обычного пользователя, так как он добавлен в группу `hashman` (см. раздел 5.3 «Настройка Hasher»):

   ```
   hsh /home/work/rpm/package.src.rpm
   ```

   Собранный пакет появляется в директории:
   ~/.hasher/repo/<платформа>/RPMS.hasher/.

   Если сборка завершилась неудачей, информация об ошибках будет выведена в терминале. Следует ознакомиться с выводом в терминале для понимания, на каком этапе сборки возникают ошибки, и на основе этого вносить необходимые исправления в SPEC-файл.

2. **Сборка в связке с `GEAR`.**

   Этот сценарий подходит, если у нас есть готовый `gear`-репозиторий (см. глава 4 «Инструмент Gear»).

   Команда выполняется внутри `gear`-репозитория и выглядит следующим образом:

   ```
   gear-hsh
   ```

   Эта команда часто используется с параметрами:

   - `-v` — сделать подробный вывод;
   - `-no-sisyphus-check` отключить проверки.

   Так же в строке запуска `gear-hsh` можно передавать параметры для самого `hasher`.

В обоих сценариях при успешном выполнении сборки новые пакеты будут находиться в каталоге ~/hasher/repo/<архитектура>.

[2] https://www.basealt.ru/fileadmin/user_upload/manual/Alt_Server_V_life_cicle_p10.pdf

5.5 Сборочные зависимости

Сборочные зависимости (build dependencies) — необходимые для сборки пакета другие пакеты. Также выделяют зависимости исполнения (runtime dependencies). При этом некоторые сборочные зависимости могут быть необходимы и во время выполнения программы.

Зависимость — это ресурс, необходимый программному обеспечению, но не поставляемый вместе с ним. Например, программы, библиотеки, утилиты, необходимые во время работы программы, скрипты и программа-интерпретатор для их запуска, файлы, необходимые для корректной работы программы.

При сборке `rpm`-пакета в пакетной базе `Sisyphus` могут не оказаться зависимые для него пакеты. Такие пакеты придётся собрать и только после этого заняться сборкой интересующего пакета.

Как искать зависимости

Чаще всего зависимости уже прописаны в спецификации к пакету.

Если пакет новый, попробуйте составить начальный SPEC-файл и собрать пакет. Проанализируйте вывод процесса сборки и отметьте, чего не хватает. Часто сборочные зависимости указываются в документации к собираемому приложению — добавляйте их по необходимости.

Виды зависимостей можно обозначить по названию тегов SPEC-файла для пакета и порядку их употребления:

- `BuildRequires(pre)` — пакеты, которые содержат определения макросов, указанных в SPEC-файле. Они помогают создать исходный `src.rpm` из SPEC-файла; тег не содержит макросы;
- `BuildRequires` — пакеты, необходимые для сборки.

Установите в системе пакеты-зависимости первого вида (`BuildRequires (pre)`), так как `hasher` собирает пакеты из `src.rpm`. Название большинства из таких пакетов начинается с `rpm-build-*`.

Установка зависимостей

- локально (`gear-rpm -ba`) — необходимо вручную установить пакеты;
- в `hasher'e` (`gear-hsh`) — `hasher` сам установит в контейнер все необходимое на основе SPEC-файла из тегов `BuildRequires (pre)` и `BuildRequires`.

Когда в базе данных менеджера пакетов отсутствуют необходимые зависимости, протестировать сборку пакета можно с помощью флага `-nodeps`. Флаг `-nodeps` даёт команду пропустить проверку зависимостей. Использовать эту опцию нужно крайне аккуратно[3]. Флаг `-nodeps` отключает всю систему отслеживания RPM — нарушается порядок установки зависимостей и назначение владельца и группы.

```
$ rpm -bs -nodeps package.spec
```

[3] https://www.endpointdev.com/blog/2008/08/rpm-nodeps-really-disables-all/

5.6 Справочная страница Hasher

Возможности `Hasher` задокументированы в инструкциях[4] к пакетам `hasher` и `hasher-priv`. Для вызова справочной информации по `hasher` наберите в консоли `man <package>`:

```
$ man hsh

HASHER(7)                          ALT Linux                          HASHER(7)

NAME
hasher - modern safe package building technology

SYNOPSIS
hsh [options] <path-to-workdir> <package>...
...

$ man hasher-priv

HASHER-PRIV(8)          System Administration Utilities         HASHER-PRIV(8)

NAME
hasher-priv - privileged helper for the hasher project

SYNOPSIS
hasher-priv [options] <args>
...
```

Это достаточно подробная инструкция по использованию утилиты `hasher`. Здесь вы найдёте полное описание, опции и флаги, содержимое рабочего каталога.

5.7 Монтирование файловых систем внутри Hasher

`Hasher` умеет монтировать внутрь `hasher`-контейнера точки монтирования и виртуальные файловые системы из основной машины[5]. Этот механизм применяется в тех случаях, когда собираемому приложению для сборки требуется доступ к ресурсам основной машины, которые `Hasher` не предоставляет по умолчанию. Например, виртуальная файловая система `/proc` или `/dev/pts`, которых по умолчанию нет в `hasher`-контейнере. Файловая система `/proc` получает информацию о состоянии и конфигурации ядра и системы.

Для монтирования файловой системы следует:

1. В файле `/etc/hasher-priv/fstab` описать файловую систему.

[4] http://uneex.ru/static/AltlinuxOrg_Hasher/
[5] https://www.altlinux.org/Hasher/Руководство

2. В файле `/etc/hasher-priv/system` указать файловую систему с помощью опции `allowed_mointpoints`.

3. Указать файловую систему либо при запуске `Hasher` в опции `-mountpoints`, либо в конфигурационном файле `~/.hasher/config` в ключе `known_mountpoints`.

4. Прописать необходимую файловую систему в `SPEC`-файле в теге `BuildReq`, либо в списке зависимостей.

5.8 Вопросы для самопроверки

1. Что такое `hasher` и для чего он предназначен?

2. Какоыа структура каталогов `hasher`?

3. Почему ранее собранные в `hasher` пакеты не оказывают влияния на новую сборку?

4. Можно ли указать из какого репозитория будет происходить сборка в `hasher`?

5. Какие шаги, помимо установки, необходимо сделать для настройки `hasher`?

6. Какие два сценария сборки возможны при использовании `hasher`?

7. Что такое зависимость программного пакета?

8. Какие зависимости называют «сборочными», а какие «времени исполнения»?

9. Как посмотреть справочную информацию по пакету `hasher` и `hasher-priv`?

10. Зачем монтировать внутрь `hasher` файловые системы?

Глава 6

Примеры использования инструментов ОС Альт для сборки пакетов

В текущих разделах представлен материал с практическими примерами по сборке пакетов. Простыми шагами разберём подготовку системы к работе с инструментами, описанными в предыдущих главах пособия. Ниже следуют примеры подготовки и сборки `rpm`-пакета инструментами `rpmbild`, `gear` и `hasher`.

- **Пример №1. Сборка пакета с помощью** `rpmbuild`.

  Подготовьте дерево каталогов для сборки пакета с помощью `rpmbuild`. Сформируйте `.tar` архив с исходным кодом программы и загрузите в каталог `SOURCES`. Назовите архив в соответствии с именем `SPEC`-файла. В каталог `SPECS` загрузите `SPEC`-файл.

```
$ cd ~/RPM
$ tree
.
|- BUILD
|   ‘- HelloUniverse-1.0
|- RPMS
|   |- noarch
|   ‘- x86_64
|- SOURCES
|   ‘- HelloUniverse-1.0.tar
|- SPECS
|   ‘- HelloUniverse.spec
‘- SRPMS
```

  Для сборки `RPM`-пакета выполните команду:

```
$ rpmbuild -bb ./SPECS/HelloUniverse.spec
```

```
...
$ ls ./RPMS/x86_64/
HelloUniverse-1.0-alt1.x86_64.rpm
HelloUniverse-debuginfo-1.0-alt1.x86_64.rpm
```

- **Пример №2. Сборка из `.src.rpm` с помощью `hasher`.**

  Возьмите готовый файл `.src.rpm` и выполните команду:

  ```
  $ hsh -v HelloUniverse-1.0-alt1.src.rpm  -no-sisyphus-check
  ```

  Результат успешного завершения сборки:

  ```
  $ ls ~/hasher/repo/x86_64/RPMS.hasher/
  HelloUniverse-1.0-alt1.x86_64.rpm
  HelloUniverse-debuginfo-1.0-alt1.x86_64.rpm
  ```

- **Пример №3. Сборка с помощью `gear-hsh`.**

  Выполните команду:

  ```
  $ gear-hsh -v -no-sisyphus-check
  ```

  Результат успешного завершения сборки — собранный RPM-пакет и `.src.rpm`:

  ```
  $ ls ~/hasher/repo/x86_64/RPMS.hasher/
  HelloUniverse-1.0-alt1.x86_64.rpm
  HelloUniverse-debuginfo-1.0-alt1.x86_64.rpm

  $ ls ~/hasher/repo/SRPMS.hasher/
  HelloUniverse-1.0-alt1.src.rpm
  ```

  Установите собранный пакет командой:

  ```
  $ sudo apt-get install ~/hasher/repo/x86_64/RPMS.hasher/\
  HelloUniverse-1.0-alt1.x86_64.rpm
  ```

6.1 Подготовка пространства

Соберем в `rpm`-пакет собственную программу. В каталоге сохраним файлы с исходным кодом — `HelloUniverse.cpp` и `Makefile`.

```
$ ls -la
HelloUniverse.cpp
Makefile
```

Подготовка окружения

Инструменты, необходимые для сборки:

- система контроля версий Git;
- текстовый редактор (Vim, Emacs, MC);
- `rpmbuild`;
- `Hasher`;
- `Gear`.

Для работы с `rpmbuild` установите также сборочные зависимости:

- make;
- gcc-c++.

После установки программ:

1. Настройте Git;
2. Создайте `gpg`-ключ подписи для работы с `gear`;
3. Настройте окружение `RPM`;
4. Настройте окружение `Hasher`.

6.2 Написание SPEC-файла и правил Gear

Подготовка `gear`-репозитория.
Перейдите в каталог с исходным кодом и проинициализируйте репозиторий:

```
$ cd HelloUniverse/
$ git init
```

Создайте каталог `.gear`, перейдите в него и создайте файлы `HelloUniverse.spec` и `rules`:

```
$ mkdir .gear
$ cd .gear
$ touch HelloUniverse.spec
$ touch rules
```

Заполните SPEC-файл `HelloUniverse.spec`:

```
%define _unpackaged_files_terminate_build 1

Name:       HelloUniverse
Version:    1.0
Release:    alt1
Summary:    Most simple RPM package
License:    no
```

```
Group: Development/Other
Source: %name-%version.tar
BuildRequires: gcc-c++
%description
This is my first RPM

%prep
%setup

%build
%make_build HelloUniverse

%install
mkdir -p %{buildroot}%_sbindir
install -m 755 HelloUniverse %{buildroot}%_sbindir

%files
%_sbindir/%name

%changelog
```

В файл `rules` пропишите:

```
tar: . name=@name@-@version@ base=@name@-@version@
spec: .gear/HelloUniverse.spec
```

Перейдите на уровень выше, в корневой каталог проекта, и сделайте первый коммит:

```
$ cd ../
$ git add HelloUniverse.cpp
$ git add Makefile
$ git add .gear/HelloUniverse.spec
$ git add .gear/rules
$ git commit -m "first commit"
```

Теперь у нас есть готовая к сборке в RPM-пакет программа.

После того, как мы убедимся, что программа собирается в пакет и работает, рекомендуется написать `changelog`, сделать новый коммит с релизом и поставить тег.

6.3 Описание пакета с исходными текстами на C++

Содержимое файла `HelloUniverse.cpp`.

```
#include <iostream>
int main()
{
```

```
std::cout « "Hello Universe\n";
return 0;
}
```

Содержимое файла `Makefile`.

```
HelloUniverse: HelloUniverse.cpp
g++ ./HelloUniverse.cpp -o HelloUniverse
clear:
rm ./HelloUniverse
```