Установка программного обеспечения

Пакеты

Пригодная для работы пользователя система состоит из множества (сотен и тысяч) программ и утилит. В Linux каждый компонент системы представлен в виде **пакета**. Все операции, связанные с изменением состава системы: установка, удаление, проверка, обновление компонентов, -- производятся над пакетами. В целом, **пакет** -- это средство сделать так, чтобы пользователь-администратор, изменяя или обновляя программное наполнение системы, работал не с *файлами* (имена которых ему подчас неизвестны), а с определёнными *функциональностями* самой системы: например, добавлял в неё не "500 файлов", а "WWW-сервер `apache`".

Архив файлов

На первый взгляд, программа состоит из одного -- исполняемого -- файла: запускаем файл, получаем работающую программу. Однако во время работы даже самая простая программа использует другие файлы, содержащие различные ресурсы: **библиотеки**, конфигурационные файлы, файлы-дырки и даже запускает другие программы. Чтобы программа действительно заработала, необходимо помимо главного исполняемого файла обеспечить наличие в системе всех нужных файлов с ресурсами.

Понятно, что при установке или удалении программы нужно позаботиться и обо всех используемых ею файлах (которых может быть даже очень много). Это -- *первая задача пакетирования*: все файлы, используемые программой, объединяются в один файл -- **архив**. Это позволяет не копировать при установке программы все файлы по-отдельности, а потом не удалять их таким же способом, а работать со всеми данными программы как с одним целым -- устанавливать и удалять один пакет.

файловый архив	Дерево каталогов, представленное в виде единого файла, состоящего из содержимого всех файлов в этом дереве и информации об имени и атрибутах каждого файла.

Нет жёсткого требования, чтобы один пакет содержал только одну программу. В пакет естественно объединять такие ресурсы, с которыми удобно работать как с одним целым. Это может быть отдельная программа или набор утилит (например, `coreutils`, основные утилиты, унаследованные Linux от UNIX) или модуль с дополнительными возможностями программы, или общие для нескольких программ ресурсы. В процессе развития и/или устаревания программного обеспечения выделение некоторых задач в отдельный пакет может приобретать или терять смысл, поэтому способ объединения ресурсов в пакеты -- это не что-то раз и навсегда выбранное: пакеты могут разделяться и сливаться.

Самый простой и традиционный для Linux способ объединить несколько файлов в один -- использовать утилиту `tar`(1). Мефодий, написав несколько программ и сценариев, решил собрать их в одном файловом архиве, чтобы их удобнее было переносить на все системы, в которых ему случается работать. Для этого Мефодий создал архив со всем содержимым своего подкаталога `bin/`.

```
methody@localhost:~ $ tar -cf methody.progs.tar bin/
methody@localhost:~ $ tar -tf methody.progs.tar
 bin/
 bin/loop
 bin/script
```

```
 bin/to.sort
 bin/two
```

Создание файлового архива при помощи `tar`

Первый параметр `tar` состоит из двух частей: операция, которую следует произвести над архивом, в данном случае "`c`" (**c**reate, создать), и ключ "`f`", который указывает, что архив следует создать в файле, имя файла архива -- следующий параметр. Имя архива может быть любым, но Гуревич посоветовал снабдить его расширением "`.tar`", чтобы потом не путаться. После имени архива следуют имена файлов и каталогов, которые следует запаковать.

С каждым указанным каталогом `tar` работает рекурсивно, т. е. запаковывает все содержащиеся в нём файлы и подкаталоги.

Чтобы проверить, какие файлы попали в архив, Мефодий просмотрел содержимое получившегося архива командой "`tar -tf имя_архива`" ("`t`" -- просмотреть, "`f`" использовать файл, указанный следующим параметром). Тут Мефодий обратил тут внимание на два обстоятельства. Во-первых, в архиве имена файлов сохранились вместе с путём. Во-вторых, все пути, сохранённые в архиве -- относительные.

При распаковке архива `tar` файлы извлекаются вместе с путём, недостающие подкаталоги создаются по мере необходимости. Все пути `tar` считает относительными от своего рабочего каталога. Если теперь Мефодий распакует свой архив (командой "`tar xf имя_архива`"), то в рабочем каталоге будет создан подкаталог "`bin/`" и в него будут записаны все файлы из архива.

```
methody@localhost:~ $ tar -xvf methody.progs.tar
 bin/
 bin/loop
 bin/script
 bin/to.sort
 bin/two
```

Распаковка архива

Ключ "`v`" велит `tar` быть "разговорчивым" (**v**erbose), т. е. выводить больше диагностических сообщений, благодаря этому `tar` при распаковке выводит имена (с путём) всех записываемых файлов. Если в рабочем каталоге уже есть файл, который `tar` собирается создать при распаковке, то этот файл будет попросту *заменён* файлом из архива. Так, когда Мефодий распаковал свой архив, подкаталог "`bin/`" со всем его содержимым *заменился* на подкаталог из архива. В данной ситуации это не страшно, поскольку в архиве файлы такие же, но вот если бы Мефодий перед распаковкой изменил какие-то из своих файлов в "`bin/`", он лишился бы всех изменений.

Формат пакета

Помимо хранения архива файлов у пакета есть и другие задачи (они обсуждаются в двух следующих разделах), поэтому для пакета в Linux не очень подходит обычный файловый архив, наподобие `.tar`, а нужен специальный формат. Таких форматов в Linux бывает несколько (краткое перечисление и характеристика -- в разделе Установщики пакетов), в

системе Мефодия используется один из наиболее распространённых -- `rpm`, поэтому все примеры в данной лекции будут с его участием. Для работы с пакетами в специальном формате нужна специальная программа-установщик, которая так же и называется -- `rpm`: она занимается установкой, удалением, обновлением и проверкой пакетов.

Пакет в формате `rpm` -- это единый файл со всеми необходимыми данными. Существуют определённые (хотя и не очень жёсткие и последовательные) соглашения относительно названий файлов-пакетов. Например, `rpm`-файл, содержащий комплект стандартных утилит `coreutils`, в системе Мефодия называется `coreutils-5.2.1-some5.rpm`: сначала -- **имя пакета**, затем через дефис -- служебная информация о номерах версий программного обеспечения и самого пакета.

Регистрация в системе

Итак, пакет с компонентом системы -- это в первую очередь **файловый архив**, в котором хранятся все необходимые файлы вместе с путями к ним (т. е. каталогами). Когда компонентов много, нужно обеспечить, чтобы в разных пакетах не оказалось файлов с одинаковым именем и путём, чтобы файл, принадлежащий одному пакету, не мог быть заменён файлом другого пакета при установке. Отслеживать такого рода **конфликты** пакетов -- *вторая задача пакетирования*.

Чтобы предупреждать конфликты, в системе должна храниться информация обо всех уже установленных пакетах и принадлежащих им файлах. Когда точно известно, какие файлы принадлежат пакету, можно полностью удалить пакет, не оставив и не удалив при этом ничего лишнего. Такой подход препятствует образованию в системе "мусора" -- бесполезных файлов, оставшихся от удалённых программ -- и делает операцию установки/удаления пакета полностью обратимой.

`Rpm` хранит информацию обо всех установленных в системе пакетах и принадлежащих им файлах и может выдать эту информацию по запросу пользователя. Почитав руководство по `rpm`, Мефодий выяснил, что список всех установленных в системе пакетов (скорее всего, очень длинный, в несколько тысяч строк) можно получить командой "`rpm -qa`", список всех файлов, принадлежащих пакету, командой "`rpm -ql имя_пакета`". Он решил проверить, есть ли в его системе уже известный ему по предыдущим лекциям пакет `coreutils` и узнать, какие утилиты ему принадлежат:

```
methody@localhost:~ $ rpm -qa | grep coreutils
 coreutils-5.2.1-some5
methody@localhost:~ $ rpm -ql coreutils | grep bin
 /bin/basename
 /bin/cat
 /bin/chgrp
 /bin/chmod
  . . .
```

Какие файлы принадлежат пакету?

Мефодий получил довольно длинный список имён файлов, среди которых встретил многие из уже знакомых ему утилит и кое-какие незнакомые. Причём запрашивая список файлов, Мефодий использовал только *имя* пакета, без **версии** -- `rpm` позволяет обращаться так любому из установленных пакетов(2).

Можно выполнить и обратную процедуру -- выяснить про любой файл, какому пакету он принадлежит:

```
methody@localhost:~ $ rpm -qf /etc/passwd
```

```
setup-2.2.5-some1
```

Какому пакету принадлежит файл?

Файлы, принадлежащие пакету, могут быть не только удалены, но и изменены. Это может быть сделано сознательно (например, отредактирован конфигурационный файл), в таком случае при обновлении или удалении пакета изменённый файл нужно сохранить, потому что в нём содержится результат работы, проделанной администратором. Для этого система должна уметь определять, что принадлежащий пакету файл изменился. Для этого в пакете должна храниться информация обо всех файлах архива: размер, атрибуты, **контрольная сумма** -- в этом случае процедура проверки сможет проверить соответствие атрибутов файла в пакете атрибутам установленного в системе файла. Мефодий может проверить, что изменилось в пакете командой "`rpm -V имя_пакета`".

```
methody@localhost:~ $ rpm -V setup
 S.5....T c /etc/X11/fs/config
 S.5....T c /etc/exports
 S.5....T c /etc/fstab
 S.5....T c /etc/printcap
 ..?..... c /etc/securetty
```

Что изменилось в пакете?

Мефодий получил список изменившихся с момента установки пакета файлов. Видимо, всё это -- конфигурационные файлы, отредактированные администратором системы. Мефодий догадался, что комбинация цифр, букв и точек -- это список атрибутов, по которым `rpm` сравнивал установленные файлы с данными пакета, однако чтобы разобраться, что именно означает каждая буква, ему придётся внимательнее читать руководство.

Система Linux раскладывается на компоненты без остатка: *каждый* файл в Linux принадлежит какому-нибудь (и только одному!) пакету(3). Это позволяет свести любые изменения в составе системы к операциям над пакетами -- от начальной установки до сложных комплексных обновлений. В этом случае для всех изменений будет использоваться одна и та же программа-установщик, например, `rpm`.

Изменение настроек системы

Для полноценной регистрации пакета в системе обычно недостаточно, чтобы система хранила список файлов, принадлежащих пакету. При установке, удалении или обновлении пакета часто требуется выполнить ряд операций, чтобы обновить сведения о пакете, адаптировать *настройки* -- как самого пакета к уже имеющимися в системе, так и наоборот. К тому же, некоторые изменения в системе, например, добавление и удаление **псевдопользователя**, не сводятся к добавлению и удалению файлов, и вдобавок зависят от *текущего* состояния системы. Получается, что регистрация в системе -- дело не только системы, но и самого пакета. Поэтому в каждом пакете должны храниться сведения о том, какие действия следует выполнить в момент различных операций с ним -- в этом состоит *третья задача пакетирования*.

Списки необходимых действий представлены в пакете в виде **сценариев**. Эти сценарии привязаны к процедурам установки и удаления пакета, причём могут быть вызваны по необходимости как до, так и после соответствующей процедуры. В результате процедуры установки и удаления пакета выглядят так:

- выполнение предшествующих установке/удалению сценариев;
- копирование файлов из архива в систему или удаление файлов из системы

- выполнение следующих за установкой/удалением сценариев.

```
methody@localhost:~ $ rpm -q --scripts coreutils
 preinstall scriptlet (through /bin/sh):
 # Remove info pages from fileutils, textutils and sh-utils.
 for f in /usr/share/info/{fileutils,textutils,sh-utils}.info*; do
         [ -f "$f" ] || continue
         RPM_INSTALL_ARG1=0 /usr/sbin/uninstall_info "$f" ||:
 done
 postinstall scriptlet (through /bin/sh):
 /usr/sbin/install_info coreutils.info
 preuninstall scriptlet (through /bin/sh):
 /usr/sbin/uninstall_info coreutils.info
```

Просмотр сценариев в пакете

Мефодий выяснил, что сценарии в пакете `coreutils` запускаются перед началом установки (preinstall), после установки (postinstall) и перед удалением (preuninstall), они выполняются стандартным командным интерпретатором (`/bin/sh`). Все эти сценарии нужны для того, чтобы зарегистрировать в системе `info` установленную пакетом документацию или удалить эту документацию из системы (командами `/usr/sbin/install_info` и `/usr/sbin/uninstall_info` соответственно). Поскольку `info` строит общее оглавление всей имеющейся в системе документации, простого копирования файлов было бы недостаточно.

В результате подобных операций по интеграции пакета в систему могут быть изменены или удалены файлы, не принадлежащие данному пакету, созданы новые файлы. Если программа, содержащаяся в пакете, пользуется услугами какой-нибудь уже установленной службы (например, `syslogd`), может понадобиться регистрация этой программы в конфигурационных файлах службы. Конечно, изменение "чужих" файлов в процессе установки пакета нежелательно: впоследствии, удаляя пакет, потребуется вернуть файл в исходное состояние, что не всегда возможно (например, после вдумчивого редактирования администратором). Избежать редактирования конфигурационных файлов позволяет схема ".d", описанная в лекции Этапы загрузки системы.

Цена удобства

Удобство, которое получает пользователь при работе с пакетами достигается не само собой, а человеческим трудом: пакеты должен создавать человек, его работа называется "сопровождающий" ("package maintainer" или "packager"). В обязанности сопровождающего пакет входит подготовка файлового архива, необходимых для установки и удаления сценариев и прочей информации о пакете и его содержимом, и объединение их в одном файле-пакете(4). Узнать, кто и когда создал пакет, получить краткую справку о программном обеспечении, которое в нём содержится, можно командой "`rpm -qi имя_пакета`".

```
methody@localhost:~ $ rpm -qi setup
 Name        : setup                        Relocations: (not relocateable)
 Version     : 2.2.5                             Vendor: Some Linux Team
 Release     : some1                          Build Date: Чтв 29 Янв 2004 18:08:05
 Install date: Пнд 23 Авг 2004 15:08:45   Build Host: shogun.somelinux.org
 Group       : Система/Настройка/Прочее   Source RPM: setup-2.2.5-some1.src.rpm
 Size        : 39969                             License: GPL
 Packager    : Leon B. Gourievitch <shogun@somelinux.org>
 Summary     : Initial set of configuration files
 Description :
 Initial set of configuration files to be placed into /etc.
```

Описание пакета

Нужно принимать во внимание, что любой пакет, содержащий программное обеспечение для Linux, не является универсальным. Если у вас есть такой пакет, это ещё не означает, что его можно установить в вашей системе. Дело в том, что разные дистрибутивы Linux различаются именно тем, каким образом программное обеспечение организовано в *систему* (о дистрибутивах речь пойдёт в лекции Политика свободного лицензирования. История Linux: от ядра к дистрибутивам). Дистрибутивы могут различаться размещением файлов и процедурами, предусмотренными для интеграции в систему программного обеспечения, не говоря уже о том, что в разных дистрибутивах используется разный формат пакетов. Это значит, что пакет, подготовленный с ориентацией на один дистрибутив, может оказаться несовместимым с другим. Чтобы в вашем дистрибутиве появилась некоторая программа, кто-то должен подготовить и сделать доступным соответствующий пакет.

пакет Ресурсы, необходимые для установки и интеграции в систему некоторого компонента (архив файлов, до- и послеустановочные сценарии, информация о пакете и его сопровождающем), объединённые в одном файле.

Несмотря на частные различия, дистрибутивы Linux представляют собой варианты одной и той же системы, поэтому в конечном итоге любую программу, работающую в одном дистрибутиве, можно "приспособить" к любому другому. Только для этого нужно располагать *исходными текстами* соответствующей программы. До сих пор речь шла только о так называемых **двоичных пакетах**, в которых программы содержатся в виде уже скомпилированных двоичных (исполняемых) файлов, в таком виде программа может зависеть от некоторых свойств системы и работать не везде. Чтобы получить работающую программу в системе, нужно установить именно двоичный пакет. Однако пакет может содержать и исходные тексты программ, такие пакеты называются **исходными**. Доступность исходных кодов -- обязательное условие распространения большей части программного обеспечения для Linux, см. лекцию Политика свободного лицензирования. История Linux: от ядра к дистрибутивам. Если получилось так, что никто не подготовил пакет с нужной вам программой для вашего дистрибутива, у вас есть возможность установить исходный пакет и скомпилировать программу самостоятельно(5). При успешной компиляции из исходного пакета получается соответствующий двоичный, который уже можно установить в системе.

Зависимости

Мефодий нашёл в Интернете пакет с заинтересовавшей его программой в подходящем формате `rpm` и решил попробовать его установить(6).

```
[root@localhost RPMS.local]# rpm -i xsltproc-1.0.32-some1.i586.rpm
 ошибка: неудовлетворённые зависимости:
        libxslt = 1.0.32-some1 нужен для xsltproc-1.0.32-some1
[root@localhost RPMS.local]#
```

Пакет не установлен из-за неудовлетворённых зависимостей

Однако `rpm` отказался выполнять установку, ссылаясь на **зависимость** от другого пакета. Здесь Мефодий впервые столкнулся с тем, что пакеты -- не всегда (точнее, почти никогда) бывают независимы от имеющейся системы. В разделе Архив файлов уже говорилось о том, что для работы программы нужны различные ресурсы, причём несколько программ могут нуждаться в одном и том же ресурсе. В последнем случае общий ресурс может оказаться в отдельном собственном пакете (чтобы не включать его сразу в несколько), и этот пакет

должен быть установлен в системе, чтобы заработали нуждающиеся в нём программы. Потребность пакета в ресурсах, находящихся в другом пакете, называют **зависимостью** этого пакета от другого. В процедуре установки `rpm` проверяет, все ли зависимости устанавливаемого пакета *удовлетворены* (т. е. все ли необходимые пакеты уже установлены в системе), и если чего-то не хватает -- прекращает установку. Именно с такой ситуацией и столкнулся Мефодий.

зависимость пакетов — Ситуация, при которой **пакет** не может быть установлен в систему, если в ней не установлен хотя бы один из некоторого множества пакетов. Аналогично, пакет не может быть удалён из системы до тех пор, пока в ней установлен хотя бы один зависящий от него пакет.

Библиотеки

Мефодию помешала установить пакет самая типичная зависимость -- на **библиотеку**. Библиотеки возникают оттого, что все программы, сколько бы они не отличались друг от друга, нуждаются в выполнении одних и тех же операций: вводе и выводе, получении доступа к ресурсам системы (памяти, процессорному времени, файлам), вычислениях, работе с сетью, рисовании окошек, кнопок, меню и т. п. Для выполнения таких операций используются небольшие подпрограммы -- **функции**. Любые функции, необходимые более чем одной программе, есть смысл не включать в текст каждой программы, а собирать в отдельных библиотеках. Тогда программа сможет использовать не собственную подпрограмму, а готовую функцию из библиотеки. Поскольку библиотеки нужны нескольким программам, они обычно оформляются в виде отдельного пакета. Если библиотека не будет установлена, использующая её программа просто не будет работать.

Библиотеки подвержены тем же изменениям с течением времени, что и все прочие программы: исправлению обнаруженных ошибок, модернизации, оптимизации и пр. Поэтому версии библиотек должны быть согласованы с версией программного обеспечения. Например, программа может отказаться работать даже при наличии библиотеки, если эта библиотека слишком старая либо слишком новая по сравнению с самой программой.

Цепочки зависимостей

Однако понятие зависимости включает не только зависимость программы от библиотек. Вообще говоря, зависимость возникает там, где программное обеспечение использует любой не поставляемый непосредственно с ним ресурс(7). Это могут быть и утилиты, которые запускаются при работе самой программы или во включённых в пакет сценариях, программа-интерпретатор для исполнения этих сценариев, и даже определённые файлы, которые должны присутствовать для правильной работы программы (например, утилита `passwd` предполагает, что существует файл `/etc/passwd`).

Зависимость может быть и небезусловной. Например, в некоторых случаях нужно обеспечить наличие ресурса не к моменту запуска программы, а прямо к моменту установки пакета, так, для выполнения доустановочного сценария нужна программа-интерпретатор. В некоторых случаях требуется ресурс строго определённой версии, ни больше, ни меньше. Бывают случаи, когда зависимость имеет обобщённую форму, например, почтовому клиенту (программе для чтения и написания электронной почты) может требоваться служба доставки электронной почты. В Linux такую услугу предоставляют несколько разных программ, и любая из них удовлетворит зависимость.

Разобравшись с понятием зависимости, Мефодий набрался твёрдой решимости установить-таки нужный ему пакет, установив всё, что он потребует. Но не тут-то было: взявшись

устанавливать библиотеки, Мефодий выяснил, что каждой из них требуются какие-то ещё пакеты, отсутствующие в системе, у каждого из них тоже есть зависимости и т. п. -- один единственный пакет повлёк за собой снежный ком других, вытягивая их по цепочкам зависимостей.

Конфликты и альтернативы

В противоположность зависимости, когда пакет не может быть установлен при отсутствии некоторого другого, **конфликт пакетов** -- это ситуация, когда пакет не может быть установлен при *наличии* некоторого другого, т. е. они несовместимы в рамках одной системы. Одна из причин возникновения конфликтов уже упоминалась выше -- в пакетах есть файлы с совпадающими именами. Самый распространённый источник конфликтов -- программы, которые предоставляют разные реализации одной и той же функциональности системы (например, службы доставки электронной почты или печати, программы проверки орфографии, компиляторы и т. п.). Можно было бы, конечно, просто *назвать* конфликтующие файлы по-разному, но и тогда путаница неизбежна: если, допустим, старый компилятор Си называется `gcc2.96`, а новый -- `gcc3.3`, то *что* запускается по стандартной команде `gcc`? В каждом пакете должна содержаться информация о том, с какими пакетами он конфликтует. Конфликт пакетов может быть разрешён очень просто: следует удалить один из конфликтных пакетов, после чего свободно устанавливать другой.

Каждый пакет помимо имени обозначен и номером версии, указывающим степень обновлённости содержащегося в пакете программного обеспечения и самого пакета. В системе одновременно может быть установлена только одна версия любого пакета, со всеми остальными версиями она конфликтует. Такой подход вполне понятен, поскольку файлы в пакете имеют строго определённый путь, по которому они должны быть размещены в файловой системе. Поэтому при использовании пакетов не должно (и не может) возникнуть ситуации, когда одна и та же программа установлена в разных местах файловой системы.

Однако не все функции в системе должны эксклюзивно выполняться одной программой. Например, в системе может быть установлено сколько угодно текстовых редакторов, и даже несколько разных реализаций одного редактора, например, Vim (Vi и **nvi**). Пакеты Vi и **nvi** не конфликтуют друг с другом, однако оба могут с равным правом быть вызваны по команде `vi`. Чтобы определить, какой именно из них запускать как `vi`, во многих дистрибутивах Linux (в частности, в том, который использует Мефодий) используется механизм **альтернатив**. Альтернативы -- это система символьных ссылок на принадлежащие пакетам файлы. Однотипные файлы из пакетов называются по-разному, а символьная ссылка, к которой обращается пользователь, указывает на один из них. Например, файл `/usr/bin/vi` может быть символьной ссылкой либо на `/usr/bin/vim`, либо на `/usr/bin/nvi` (то же самое относится и к руководствам по этим редакторам). При установке и удалении любого из пакетов с одной из альтернативных программ символьная ссылка автоматически обновляется. На какую из них будет указывать ссылка решается на основании **веса** каждого пакета. Вес -- это условное число, выбирается та альтернатива из установленных, у которой наибольший вес. Пользователь может вмешаться в выбор альтернатив и вручную. Все необходимые утилиты для работы с альтернативами предоставляет пакет `alternatives`.

Установщики пакетов

Для выполнения всех операций над пакетами требуется специальная программа -- **установщик пакетов**. В её задачи входит весь цикл работ с пакетом: от создания пакета (компиляции **исходного пакета** в **двоичный**), до его установки, удаления, обновления, а также хранение и вывод по запросу пользователя или системы информации об

установленных и неустановленных пакетах, принадлежащих им файлах и т. п.

В системах Linux формат пакетов не унифицирован, распространено несколько различных форматов, и для каждого из них требуется собственный установщик пакетов. Наиболее известны уже описанный `rpm`, `dpkg`, используемый в Debian (см. подробнее лекцию Политика свободного лицензирования. История Linux: от ядра к дистрибутивам), а также пакеты в формате `tgz` (он же `tar.gz` -- файловый архив `tar`, сжатый упаковщиком `gzip`, GNU Zip), то есть обычные файловые архивы, где вся необходимая в пакете метаинформация упакована в виде файлов наряду с файлами программного обеспечения. Установщики пакетов различаются не только форматом пакетов, с которыми они работают, но и кругом возможностей, внутренним форматом хранения информации и т. д.

установщик пакетов	Программа, выполняющая основные операции с пакетами: установку, удаление, проверку, вывод информации о пакетах.

В рамках этой лекции мы ограничимся обсуждением только одного из установщиков пакетов -- `rpm` (**R**ed Hat **P**ackage **M**anager). Он первоначально возник в дистрибутиве RedHat, но в настоящее время используется и во многих других дистрибутивах. Пожалуй, сейчас его можно назвать самым распространённым форматом: авторы программ для Linux обычно выкладывают свои программы в Интернет в виде файловых архивов `tgz` и пакетов `rpm`.

Обратной стороной популярности `rpm` является его нестандартность. Под расширением `.rpm` довольно редко оказывается канонический формат, разрабатываемый RedHat. В формате `rpm` смогли усмотреть много недостатков и недоделок, поэтому распространено множество улучшенных и дополненных версий `rpm`, и, соответственно, пакетов, ориентированных на какую-то из этих версий, но носящих всё то же расширение. На практике это означает, что разные версии `rpm` не полностью совместимы между собой, поэтому даже если в вашей системе используется `rpm`, это совершенно не означает, что вы сможете установить любой найденный в Интернете пакет в этом формате.

Случай `rpm` -- только самая яркая демонстрация более общей проблемы: в общем случае ни в одном дистрибутиве нельзя без потерь, помех или ручного вмешательства установить пакет, не разработанный *специально* для данного дистрибутива. В следующем разделе (Менеджеры пакетов) изложены некоторые соображения, почему это нежелательно, и почему следует по возможности пользоваться именно "родными" пакетами, а если их нет -- делать их самостоятельно.

Другая проблема установщиков пактов в том, что они годятся только для установки/удаления отдельных пакетов, но не предназначены для доставки пакетов в системы (пользователь сам должен найти и скачать нужный пакет, и указать местоположение файла пакета установщику в командной строке). Кроме того, установщик работает с каждым пакетом по отдельности: он может указать, что не удовлетворены некоторые зависимости, или имеют место конфликты, но не может в ходе процедуры установки ни установить все необходимые пакеты по цепочке зависимостей, ни удалить конфликтующие -- пользователь должен делать это вручную. Установщики пакетов не предоставляют также никаких средств по автоматизации обновления системы.

Менеджеры пакетов

Установщики пакетов делают атомарными (одношаговыми) операции с отдельными пакетами: вместо копирования множества файлов и запуска нескольких сценариев пользователь вводит одну команду "установить/удалить пакет". Однако атомарная с точки зрения пользователя операция -- добавление в систему *одного* нового компонента может состоять из нескольких (и даже многих) операций над пакетами. Мефодий уже столкнулся с

подобным случаем, изучая на собственном опыте понятие "цепочка зависимостей". Здесь установщики пакетов никак не могут облегчить работу пользователя. Чтобы сделать процедуру установки, удаления и обновления *компонента системы* атомарной, были разработаны **менеджеры пакетов**. Менеджер пакетов -- это программа, которая вычисляет весь комплекс операций над отдельными пакетами, который нужно произвести для установки/удаления нового компонента (пакета), и сама запускает **установщик пакетов** сколько нужно раз с нужными параметрами. Кроме того, менеджер пакетов хранит информацию не только о пакетах, уже установленных в системе, но и обо всех, которые доступны для установки с какого-либо носителя или по Сети (подробнее об этом в разделе Доставка).

менеджер пакетов	Программа, выполняющая установку, удаление или обновление любого пакета или группы пакетов, и автоматически выполняющая все необходимые для этого процедуры (доставку пакетов из удалённых репозиториев, вычисление зависимостей и установку требуемых по ним пакетов, удаление замещаемых пакетов и т. п.).

Наиболее известный и популярный менеджер пакетов называется APT (**A**dvanced **P**ackage **T**ool). Первоначально он был разработан в рамках дистрибутива Debian и работал только с установщиком пакетов `dpkg`, впоследствии для других дистрибутивов была разработана версия, работающая с `rpm`. В дистрибутиве Мефодия также используется APT.

Чтобы установить пакет, прежде всего нужно узнать о его существовании. Пакетов для каждого дистрибутива Linux доступны тысячи и даже десятки тысяч, ориентироваться в них непросто. APT предоставляет возможность поиска нужного среди доступных пакетов, для этого используется утилита `apt-cache`. В каждом пакете обязательно содержится краткая аннотация (в одну строку) и небольшое описание содержащихся в пакете ресурсов (не длиннее нескольких абзацев). По команде "`apt-cache search подстрока`" APT найдёт и выведет список из имён и аннотаций пакетов, где в имени, аннотации или описании нашлась указанная подстрока.

```
[root@localhost shogun]# apt-cache search python | wc
    146    1158    8994
[root@localhost shogun]# apt-cache search python | grep "programming"
 python - An interpreted, interactive object-oriented programming language
```

Поиск пакетов в APT

Для установки и удаления пакетов предназначена утилита `apt-get`, а команда установки выглядит совсем просто: "`apt-get install имя_пакета`", причём не нужно указывать никаких сведений о версии и местонахождении пакета: APT сам найдёт и установит самую последнюю из доступных версий.

```
[root@localhost shogun]# apt-get install python
 Чтение списков пакетов... Завершено
 Построение дерева зависимостей... Завершено
 Следующие дополнительные пакеты будут установлены:
   libpython libgdbm libgmp python-base python-modules python-modules-bsddb
   python-modules-compiler python-modules-curses python-modules-email
   python-modules-encodings python-modules-hotshot python-modules-logging
   python-modules-xml python-strict
 Следующие НОВЫЕ пакеты будут установлены:
   libpython libgdbm libgmp python python-base python-modules
   python-modules-bsddb python-modules-compiler python-modules-curses
   python-modules-email python-modules-encodings python-modules-hotshot
   python-modules-logging python-modules-xml python-strict
 0 будет обновлено, 15 новых установлено, 0 пакетов будет удалено и 0 не будет
```

```
обновлено.
 Необходимо получить 0B/4466kB архивов.
 После распаковки потребуется дополнительно 16,9MB дискового пространства.
 Продолжить? [Y/n] y
 Получено: 1 cdrom://SomeLinux CD RPM/main libpython 2.3.3-some2 [17,4kB]
 Получено: 2 cdrom://SomeLinux CD RPM/main libgdbm 1.8.3-some3 [25,6kB]
 Получено: 3 cdrom://SomeLinux CD RPM/main libgmp 4.1.2-some3 [153kB]
  . . .
 Получено: 14 cdrom://SomeLinux CD RPM/main python-base 2.3.3-some12 [782kB]
 Получено: 15 cdrom://SomeLinux CD RPM/main python 2.3.3-some12 [11,5kB]
 Получено 4466kB за 0s (19,5MB/s).
 Совершаем изменения...
 Preparing...                ########################################## [100%]
  1: libpython               ##########################################  [  6%]
  2: libgdbm                 ##########################################  [ 13%]
  3: libgmp                  ##########################################  [ 20%]
  4: python-base             ##########################################  [ 26%]
  . . .
 13: python-modules-logging  ##########################################  [ 86%]
 Завершено.
```

Установка пакета с помощью APT

Процедуру установки APT выполняет в несколько этапов: сначала он ищет запрошенный пакет в списках доступных, найдя, рассчитывает, какие пакеты следует установить, чтобы удовлетворить его зависимости, после чего получает файлы всех нужных пакетов (в данном случае APT нашёл нужные пакеты на диске CD-ROM), и запускает установщик пакетов последовательно для установки всего необходимого. Аналогично, чтобы удалить пакет, достаточно выполнить команду "`apt-get remove имя_пакета`".

Кроме APT, есть ещё несколько менеджеров пакетов. Большинство из них специфичны для определённого дистрибутива, как, например, `emerge` для Gentoo или `yast` для SUSE. Их задачи и возможности примерно совпадают с APT.

Контроль целостности

Поскольку менеджер пакетов умеет строить цепочки зависимостей пакетов друг от друга, с его помощью всегда можно определить, все ли зависимости удовлетворены у пакетов, установленных в системе. Система, в которой нет пакетов с неудовлетворёнными зависимостями, называется **целостной**. Если целостность нарушена, это означает, что часть установленного в системе программного обеспечения попросту неработоспособна или работает некорректно.

Целостность системы может нарушиться в момент каких-то изменений в её составе: при установке, удалении или обновлении части пакетов или всей системы. Если для всех этих операций использовать менеджер пакетов, то целостность системы не должна нарушиться. Хотя иногда даже менеджеру пакетов бывает сложно найти правильное решение, чтобы удовлетворить все зависимости и устранить конфликты.

При наличии менеджера пакетов механизм зависимостей можно обернуть и на пользу человеку. Так, можно создать пакет, в котором есть только зависимости и нет никаких ресурсов -- такой пакет называется **виртуальным**. Это бывает полезно в том случае, когда нужно упростить пользователю установку полной среды для выполнения какой-либо задачи. Необходимые для этого пакеты могут напрямую не зависеть друг от друга, но чтобы установить их все за один шаг, пользователю будет достаточно установить один -- виртуальный -- пакет. Таким виртуальным пакетом оказался сам пакет `python` в примере, и

ещё один -- `python-strict`:

```
[root@localhost shogun]# rpm -ql python
 (не содержит файлов)
[root@localhost shogun]# rpm -ql python-strict
 (не содержит файлов)
```

Виртуальные пакеты не содержат файлов

Именно поэтому `apt` "получил" 15 пакетов (включая два виртуальных), а "совершил изменения" только для 13-ти.

Доставка

Важная задача, которую не решает установщик пакетов -- доставка файла пакета в систему для последующей установки. Архивы пакетов обычно не хранятся в самой системе: они слишком велики (тысячи пакетов) и должны регулярно обновляться (выход обновлений программ, т. е. новых версий пакетов). Поэтому для установки обычно требуется сначала скопировать необходимые файлы с того носителя, где они хранятся (это либо установочные диски дистрибутива, либо хранилища в сети Интернет).

Чтобы APT мог работать с пакетами, они должны содержаться в организованном по специальным правилам хранилище -- **репозитории**. Список доступных APT репозиториев хранится в файле `/etc/apt/sources.list`, для каждого репозитория указан способ доступа (например, "`cdrom:`", "`ftp:`", "`file:`" и др.) и адрес.

```
rpm cdrom:[SomeLinux CD]/ RPM contrib main
rpm [sme] ftp://updates.somelinux.com 2.4/i586 updates
```

Файл `sources.list`

После каждого изменения файла `/etc/apt/sources.list` нужно обновлять кеш APT, в котором хранятся сведения о доступных пакетах, командой `apt-get update`. Для того, чтобы добавить в кеш информацию о пакетах, доступных на CD, следует использовать команду "`apt-cdrom add`", а не редактировать `sources.list` вручную.

APT позволяет и просто доставить пакет в систему, не устанавливая его. Так, например, всегда происходит с **исходными пакетами**, которые просто копируются из репозитория в определённый каталог системы по команде "`apt-get source имя_пакета`".

Обновление

Программное обеспечение в мире Linux (и не только) постоянно обновляется: исправляются ошибки, расширяются возможности. Разработчики каждого дистрибутива по мере выхода новых версий программ готовят новые версии соответствующих пакетов и делают их доступными в своём **репозитории** (репозитории, отражающие наиболее современное состояние программного обеспечения, доступны через Интернет). Пользователю имеет смысл не отставать от обновлений программного обеспечения, потому что новые версии программ -- это и бОльшая надёжность работы системы, и новые возможности.

Менеджеры пакетов позволяют делать **комплексные обновления** всей системы. В APT эту процедуру можно выполнить одной командой: "`apt-get dist-upgrade`". Эта процедура сначала исследует содержимое всех доступных репозиториев и находит там все пакеты более поздних версий, чем соответствующие пакеты, установленные в системе. После этого

вычисляется объём обновления: должна быть удалена связная область зависящих друг от друга устаревших пакетов и заменена соответствующей областью более новых версий. Сложные ситуации могут возникать в том случае, если изменилось распределение ресурсов по пакетам: пакеты были разделены или объединены -- здесь может потребоваться ручное вмешательство пользователя. Тот род обновлений системы, который нужно делать регулярно и *обязательно* -- это обновления, связанные с безопасностью (security updates). Когда в программе обнаруживают и исправляют серьёзные ошибки, угрожающие безопасности всей системы, разработчики дистрибутивов обычно заботятся о том, чтобы соответствующие обновления достигли пользователя. Обычно присутствует отдельный репозиторий с обновлениями, существенными для безопасности.

Цена удобства

Удобство менеджеров пакетов оплачивается тем, что они могут успешно работать только со специальными целостными областями источников (**репозиториями пактов**). Хотя для большинства пользователей это ограничение не так существенно: те дистрибутивы, в которых используются менеджеры пакетов, обычно имеют огромные репозитории пакетов, где можно найти любое мыслимое и немыслимое программное обеспечение. Если же нужной программы всё-таки нет в официальном репозитории дистрибутива, обычно находятся "частные" репозитории, доступные по сети Интернет, включающие не вошедшие в официальный репозиторий пакеты.

Если всё-таки нужный вам пакет нигде не найти собранным именно для вашего дистрибутива, можно установить и сторонний пакет, но это может быть выполнено только при помощи установщика пакетов, менеджер пакетов в этой ситуации будет бесполезен. Можно установить программу и самостоятельно скомпилировав её из исходных кодов, однако здесь стоит иметь в виду следующее.

Автор *программы* совершенно не обязан учитывать в ней все тонкости всех дистрибутивов, поэтому возможны, с одной стороны, прямые конфликты с файлами в системе (которые никто уже не отследит), а с другой стороны -- конфликты и противоречия скрытые (например, программа устанавливается в подкаталог каталога `/usr/local`, и ожидает, что *все остальные* программы тоже находятся в этом каталоге). Это значит, что придётся самостоятельно разобраться и с тем, как и с какими параметрами компилировать программу, как устанавливать её в систему, и как избежать при этом конфликтов. А если так, если вы и в самом деле в состоянии правильно собрать и установить в систему нужную вам, а значит и ещё кому-нибудь, программу, которой пока нет в дистрибутиве, то самое правильное -- сделать из неё *пакет*, по крайней мере **исходный пакет**, а если получится, то и **двоичный**. Это здорово облегчит жизнь и вам, когда вы будете компилировать и устанавливать эту программу ещё раз (на другом компьютере или обновляя версию самой программы), и, главное, *всему сообществу пользователей* вашего дистрибутива!

Наконец, во многие современные дистрибутивы включаются средства, помогающие сборке двоичных пакетов. Такие средства (например, пакет `hasher` из ALT Linux) позволяют не только скомпилировать программу в "универсальной среде", содержащей лишь заданный набор пакетов, но и автоматически выстраивают зависимости, проверяют правильность установки, отслеживают конфликты. Короче говоря, собрав пакет с помощью такого средства, вы можете серьёзно претендовать на роль сопровождающего этот пакет в дистрибутиве. Напротив, скомпилировав программу втихомолку, с помощью шаманства и ручной работы, вы проявите себя как лентяй и эгоист, которому нет дела до роста и улучшения собственной операционной системы.

---

(1) `Tar` появился намного раньше Linux и изначально служил для создания файловых

архивов на магнитной ленте. Отсюда и его название -- **t**ape **ar**chiver, "архиватор для (магнитных) лент". В настоящее время `tar` присутствует в любой UNIX-подобной системе и позволяет работать с файловыми архивами на любых носителях.

(2) Что логично, поскольку в системе может быть установлена только *одна* версия данного пакета. См. подробнее раздел Конфликты и альтернативы.

(3) Естественно, кроме тех файлов, которые созданы пользователями.

(4) Функции по созданию пакета в формате `rpm` также выполняет программа `rpm`.

(5) Слухи о том, что для сборки программы из исходных текстов не обязательно даже знать, что такое "компилятор", далеки от действительности.

(6) Для установки и удаления пакетов нужны права администратора -- это серьёзные изменения в системе.

(7) Имеет смысл исключать из понятия зависимости использование наиболее стандартных ресурсов, без которых немыслима система Linux как таковая. К таким ресурсам можно отнести **системные вызовы** и некоторые стандартные файлы, вроде `/dev/null`.

Сообщество вокруг дистрибутива

Чуть ранее был блок, посвященный сообществу, и было сказано о сообществе не вокруг одного конкретной свободной программы, а вокруг дистрибутива. Аналогия достаточно прямая, есть разделение на ядро, разработчиков и пользователей и в том, и в другом случае. В сообществе вокруг дистрибутива в роли разработчиков выступают сопровождающие (мейнтейнеры, от англ. maintainer), роль разработчика остается, потому что они обеспечивают этой структуре стабильность. В случае организации бизнеса на основании свободного ПО, у нас есть ресурс (средний слой -- разработчики), который выполняет роль "бесплатных разработчиков", в нашем проекте участвуют все, кому это интересно и кто может себе помочь в развитии этого проекта. Пользователи у нас — это такие "бесплатные тестеры" (опять же, это неправильно сказано, так обычно говорят наши оппоненты: вот у вас есть бесплатные разработчики, которые кое-как все делают, и тестируете вы на пользователях. На самом деле картина другая, она становится более очевидной, когда мы говорим, что есть информационная связность, и главное — это не только наличие этих самых людей, но и активное участие в информационном пространстве. Тогда пользователи уже не подопытные кролики, а в первую очередь заказчики. К тому же, совершенно не понятно, почему разработчики должны работать хуже, чем ядро, они работают иногда даже лучше, например, человек может быть слишком высоким профессионалом в своей узкой области, чтобы все своё время посвящать нашему проекту, но иногда принимает в нем участие, за что ему большое спасибо.)

В случае дистрибутивов роль core team — определять стратегическое направление, в котором развивать дистрибутив, и не давать проявляться субъективным факторам, например ленивым или просто ушедшим на другую деятельность разработчикам. Необходимо, чтобы и веб-браузер был на уровне, и чтобы ядро ОС было достаточно защищённое и современное, а если какой-то сторонний человек, не входящий в core, и вообще изредка участвующий в этом сообществе, занимается ключевой частью дистрибутива, то может получиться такая ситуация: в ядре ОС давно обнаружены и исправлены ошибки, а в дистрибутиве все ещё находится старое.

Так вот, по отношению к дистрибутиву у core team сохраняются все те же функции, что и по отношению к обычной свободной программе, а понятие разработчика слегка размывается, т.к. основной фигурой на этом месте становится мейнтейнер, он же сопровождающий. Это человек, который имеет некоторую необходимость использовать программный продукт в рамках нашего дистрибутива, и эта необходимость настолько сильная, что он готов сам заниматься сборкой этого продукта, т.е. превращением из исходного кода в бинарный, и адаптацией его под конкретные условия, под дисциплину сборки программных продуктов, принятую в нашем дистрибутиве. Более того. в действительности дистрибутив — это конечный продукт некоей технологии, которая в качестве базового элемента имеет не дистрибутив, а то что называется хранилищем (англ. repository).

Хранилище

С точки зрения человека, не знакомого с этой технологией, хранилище выглядит просто как свалка, набор программных продуктов, адаптированных для работы в конкретном дистрибутиве. Набор достаточно внушительный, в хранилище ALTLinux их порядка 9 тысяч. Если приглядеться, в хранилище есть несколько дополнительных свойств, которые делают его не просто FTP-сайтом со складом файлов. Во-первых, файлы, которые помещаются в хранилище, помещаются туда мейнтейнерами пакетов.

Мейнтейнеры

Человек, который вдруг решил, что ему необходимо использовать некий программный продукт, который до этого не был адаптирован к некоторому дистрибутиву, может этот продукт скачать с сайта производителя, собрать из исходников бинарный пакет, и поместить результат работы в хранилище, пользуясь правами члена сообщества. Не всякий человек может туда положить пакет, а только тот, который, условно говоря, сдал экзамен на право называться разработчиком. Ничего такого в этом экзамене нет, человек просто должен показать, что он может в принципе такие вещи делать. Вот, допустим, в Debian GNU/Linux правила приёма сложнее, человек иногда не может годами туда попасть. Дело в том, что Debian --- это сообщество, которое полностью базируется на дисциплине. Там нет оплачиваемых людей, зарплату получают только администраторы сборочных серверов, больше никто, поэтому такие вещи там очень важны, если какой-то член core team против включения кого-то в состав разработчиков, значит есть тому резон.

Дисциплина оформления пакетов

В хранилище мейнтейнеры помещают исходный и собранный варианты программного продукта, адаптированного к местной дисциплине сборки. Пакет попадает в хранилище не сразу. Первым делом происходит проверка его на пригодность, на то, насколько правильно в нем соблюдена дисциплина сборки программных продуктов для хранилища (в случае Альт Линукс -- пакет помещается в хранилище под названием Сизиф, и проверка называется sisyphus-check). Дисциплина оформления пакетов в Сизиф -- это подробное описание, достаточно строгие рекомендации, что можно делать, а что нельзя, и фактически мейнтейнер закачивает в это хранилище не бинарный пакет, который он собрал руками, а адаптированный пакет с исходными текстами (исходный пакет), и специальный робот берет этот пакет, и собирает его по тем правилам, которые внутри пакета заключены. Если по этим правилам пакет собирается, значит, он соответствует дисциплине сборки пакетов. Если он не собирается, т.е. автоматом в изолированном окружении исходные тексты не удаётся превратить в бинарный продукт, значит мейнтейнер у себя использовал не такое окружение или забыл что-то ещё, и такой пакет в хранилище не помещается.

То есть

- проверяется формальное соответствие правилам оформления программных продуктов для Сизифа;
- фактическая его собираемость в изолированном окружении.

Было бы неплохо, если бы где-то были какие-то правила по тестированию, чтобы вместе с программным продуктов подготавливался некий набор тестов, на которых бы он запускался и работал, но такого не происходит, просто потому что это не всегда легко организовать. Если кто работал в промышленном программировании, он знает, что система тестирования занимает процентов 50 от общего времени разработки, и если авторы этого не сделали, то это очень сложно. Допустим, автор три года разрабатывал программный продукт, значит, чтобы написать полную систему тестов, нужно ещё три года.

Далее, предполагается, что если мейнтейнер поместил пакет в хранилище, то он

- сам им пользуется;
- тот факт, что он им пользуется и не видит никаких ошибок, как раз и есть если не лучший, то весьма эффективный способ тестирования. В конце концов, мало ли какие роботы какие программы как тестируют, а тут человек с помощью программы что-то делает, работает, она у него работает, и он на этом основании помещает её в хранилище.

Это и есть те преимущества, которые сообщество вокруг дистрибутива предоставляет мейнтейнеру. Когда мейнтейнер пакета, условно говоря, просто помещает во входную

очередь для сборки некий программный продукт, предварительно его оформив как надо, он соберется, проверяется, проверяются также всякие хитрости, связанные с интеграцией его в общую среду хранилища. Это дает гарантию, что, если ошибок не произошло, то этот пакет никому не навредит и не войдет в конфликт с чем-то ещё.

Тут стоит заметить, что процедура сборки — превращения исходного текста в бинарный, результирующий программный продукт в изолированном окружении, в нашем хранилище Сизиф из известных во всем мире наиболее продвинутая и технологичная. В Debian все устроено примерно так же, но есть некие тонкие различия. Что касается других дистрибутивов, то там пожиже. Это достаточно непростой процесс, могу отослать к статьям Димы Левина про технологию сборки Сизиф и про hasher — инструмент, который позволяет организовывать изолированное окружение и сборку.

ftp://ftp.altlinux.org/pub/people/ldv/hasher/index.html

http://freesource.info/wiki/AltLinux/Dokumentacija/Hasher?v=vsm&search=hasher

Зависимости между пакетами. Обновления и стабильность

Мы в прошлый раз говорили про разделяемые библиотеки. Поскольку программные продукты в хранилище взаимодействуют друг с другом, например, программы, написанные для графической подсистемы, пользуются большим множеством соответствующих программных инструментов, интерфейсной графической библиотекой, математической библиотекой, различными утилитами, которые вызываются при работе программы. Каждый из этих элементов на самом деле является отдельным программным продуктом и лежит в хранилище отдельным пакетом.

Хранилище Сизиф именно в связи с этим небезопасно для повседневного использования. Допустим, сопровождающий пакета, gtk (популярная интерфейсная библиотека графической среды, используемая многими программными продуктами), принимает решение, что новая версия gtk, со всякими изменениями внутри, очень хороша, и кладет ее в хранилище. Что при этом происходит? Все программные продукты, которые требуют библиотеку gtk, становятся нерабочими. Потому что библиотека новая, программы старые, собраны с предыдущей версией библиотеки, которой уже нет, вместо неё есть новая версия, а старая удалилась. Попытка воспользоваться этим множеством пакетов при таком состоянии хранилища, приведет к тому, что библиотека эта обновится, а все другие пакеты, от неё зависящие, удалятся на вашей машине, потому что у них есть неудовлетворённая зависимость. Это немедленно заметит сообщество, особенно те люди, которые используют Сизиф в качестве повседневного программного окружения, и за мейнтейнером gtk потянутся мейнтейнеры других программных продуктов, которые используют gtk, пересобирать свои программные продукты с поддержкой новой версии gtk. В какой-то момент ситуация вокруг gtk стабилизируется, потому что те мейнтейнеры, которым это интересно, поднимут свои версии, а те, которым это неинтересно и которых мы давно не встречали на просторах интернета, про это дело забудут, а программы останутся неработоспособными. Но если мейнтейнера у программы нет, практически это значит, что её никто не использует. Сизиф — это такое вечно нестабильно хранилище всего, и название как нельзя лучше отражает суть работы: вот ты значит старался-старался, компилировал-компилировал, исправлял причудливую мысль автора этой программы, наконец, все ошибки исправил, выложил в Сизиф, получил обратную связь от пользователей, ещё раз какие-то ошибки исправил, а через два дня вышла новая версия этой программы. Значит, камень скатывается обратно и все начинается сначала.

Заморозка. Стабильные ветки

Для того, чтобы этот процесс как-то упорядочить, некое подмножество значимых программных продуктов в этом самом хранилище регулярно, скажем, два раза в год (так

принято в Сизифе) подвергается процедуре под названием заморозка. Замораживается весь Сизиф со словами: вот сейчас, в течение месяца-двух происходит заморозка, в течение которой никакие обновления версий этих программных продуктов недопустимы, кроме случаев, если программа обновляется вместе с устранением критических ошибок. Единственное, что разрешается делать, это устранять критические и вообще любые ошибки в ваших программных продуктах, если такие устранения появились, заниматься только этим, не смотреть в сторону новых версий, а смотреть на имеющиеся, из них удалять ошибки. В какой-то момент сообществу это дело надоедает, т.к. сообщество хочет все время новое все, и снимок достаточно стабильного состояния Сизифа, когда ситуация с новой библиотекой, которая все завалила — исправлена, откладывается в сторону под названием ветка (branch), после чего в этой ветке некоторое время по инициативе дистростроителей, то есть людей, которые будут делать дистрибутив, ещё продолжаются какие-то изменения, но по инерции. А сообщество работает дальше над Сизифом, который размораживается, через неделю приходит в абсолютно нерабочее состояние, потому что все туда накидают обновлений и экспериментируют вовсю, а что касается ветки, то в идеале её стабильность может только увеличиваться с течением времени, потому что туда будут помещать исправления ошибок, если кто-то в этом заинтересован.

Дистрибутивы

Кто заинтересован в том, чтобы исправлять ошибки в наборе ПО, который уже не развивается, потому что это ветка? Заинтересованы производители дистрибутивов, т.е. продуктов, которые распространяются и за распространение которых фирма хочет получить какие-то бонусы, начиная с продажи коробок, заканчивая внедрением в российские школы. Дистрибутивы делаются не на базе Сизифа. Берется ветка, в которой все уже более или менее устаканено, нет никаких строгих противоречий между программными продуктами, берется некое подмножество, поскольку дистрибутив отличается от хранилища тем, что в него входят только нужные программные продукты, и называется дистрибутивом.

Есть некоторая доля содержимого, входящего в дистрибутив, но не в хранилище -- это картинки, специальные версии программных продуктов, в которых что-то перенастроено, инсталлятор (хотя у нас он весь в Сизифе лежит, как в Debian, а у других дистрибутивов инсталлятор, как правило, пишется самостоятельно). Есть какая-то часть этих программных продуктов, которые даже ветке не принадлежат, но остальная часть берется из ветки. Дистростроитель понимает, что от исправления ошибок зависят потребительские свойства дистрибутива, и в ветку продолжают вносить какие-то изменения. Вот, собственно, технологический цикл: Постоянно нестабильное хранилище Сизиф, в которые постоянно помещают самые свежие версии программ, два раза в год осуществляется процесс заморозки, который приводит хранилище в относительно стабильное состояние, потом Сизиф отпускается, а сфотографированный стабильный срез продолжает жить своей жизнью, и его жизнь имеет две цели

- организация дистрибутивов на его основе
- ресурс для тех людей, которые взяли готовый дистрибутив, но которым потребовалось что-то ещё из стабильной ветки хранилища.

backports

Есть еще одно ответвление в этом технологическом процессе. Пусть есть хранилище, которое объявлено стабильным — ветка. В нем запрещается делать обновления, за исключением обновлений по безопасности и исправлений грубых ошибок. Что если мы хотим, пользуясь программным окружением этого стабильного хранилища, воспользоваться суперновой программой, которой либо вообще во времена стабилизации ветки в ней не было, либо со времён стабилизации она серьёзно обновилась? Болезнь новых версий у всех есть, даже у тех

осторожных людей, которые не пользуются Сизифом, а пользуются дистрибутивами. Для них есть отдельное небольшое хранилище, которые поддерживается таким же сообществом больных новыми версиями, которое называется backports. Берутся пакеты из Сизифа и пересобираются в окружении ветки. Предполагается, что такого рода пересборка проходит (а бывает, что нет, допустим нужна более свежая версия компилятора), предполагается что использование пакетов из backports не портит стабильности ветки, хотя это никто не гарантирует. В любом случае, это намного лучше, чем пытаться одновременно брать пакеты со стабильной ветки и из Сизифа.

Относительно объёмов — хранилище содержит около 9000 пакетов. Типичный дистрибутив на DVD — порядка 4000 пакетов, на CD — порядка 800. Двухслойный DVD — 8000-9000 пакетов.

Посмотрим репозиторий

`lftp ftp.altlinux.org`

```
lftp ftp.altlinux.org:/pub> ls
lrwxrwxrwx      1 ftp         ftp                  30 Jun 15   2007 Mozilla ->
distributions/ALTLinux/Mozilla
lrwxrwxrwx      1 ftp        ftp                 33 Jun 15   2007 OpenOffice ->
distributions/ALTLinux/OpenOffice
drwxr-sr-x   10 ftp      ftp          4096 Jul 24 12:49 beta
drwxr-xr-x    6 ftp      ftp          4096 Aug 01  2006 distributions
drwxr-xr-x    5 ftp      ftp          4096 Apr 02  2006 docs
drwxrwsr-t    3 ftp      ftp          4096 Jun 10  2006 mirrors
drwxr-sr-x    5 ftp      ftp          4096 Mar 11  2003 partners
drwxr-xr-x   72 ftp      ftp          4096 Jul 09 14:51 people
drwxr-xr-x    3 ftp      ftp          4096 Jun 25  2004 security
```

Это все разные каталоги, тут нет строгой дисциплины, за исключением желаний отдельных людей или групп что-то поместить.

```
lftp ftp.altlinux.org:/pub/distributions> ls
drwxr-xr-x   13 ftp      ftp          4096 Jul 30 21:49 ALTLinux
drwxr-xr-x    3 ftp      ftp          4096 Feb 21  2006 FreeMusic
drwxr-xr-x    3 ftp      ftp          4096 Feb 21  2006 OpenMusic
drwxr-xr-x    3 ftp      ftp          4096 Jun 15  2007 archive
```

Это по именам дистрибутивов.

```
lftp ftp.altlinux.org:/pub/distributions/ALTLinux> ls
drwxr-xr-x    3 ftp      ftp          4096 Feb 13  2003 2.2
drwxr-xr-x    4 ftp      ftp          4096 Jul 01  2004 2.3
drwxr-xr-x    3 ftp      ftp          4096 Nov 03  2004 2.4
drwxr-xr-x   10 ftp      ftp          4096 Mar 03  2006 3.0
drwxr-xr-x    7 ftp      ftp          4096 Jun 05 20:14 4.0
drwxr-xr-x    3 ftp      ftp          4096 May 07 00:12 4.1
drwxrwsr-x    4 ftp      ftp          4096 May 31  2007 Daedalus
drwxr-xr-x   14 ftp      ftp          4096 Aug 01 07:09 Sisyphus
drwxrwsr-x    8 ftp      ftp          4096 May 12 09:57 backports
drwxr-xr-x    5 ftp      ftp          4096 Jul 30 21:37 old
drwxr-xr-x    9 ftp      ftp          4096 Dec 09  2007 updates
```

- `Sisyphus` — это хранилище Сизиф,
- `2.2, 2.3, ..., 4.1` — это стабильные ветки, сделанные в своё время.
- `backports` — это backports (внутри — по веткам)
- `updates` — обновления к веткам и дистрибутивам (по отдельности, т.к. одно и то же

может обновляться по-разному из-за особенностей допилки в дистрибутивах)

Диспетчер пакетов

В разделе "Система" главного меню выберем "Диспетчер пакетов". Для установки и удаления программ, так же как для управления системой и настройки сети, необходимы права суперпользователя. Программа Synaptic (Диспетчер пакетов) --- это по сути не более чем графический интерфейс к семейству утилит apt. Однако, в некоторых случаях использование Synaptic может оказаться предпочтительней, чем работа с apt в командой строке. Перечислим основные преимущества графического диспетчера задач.

- Наличие каталогизатора.
- Возможность легко и быстро получить информацию о пакете (просто выделив его).

- Возможность ускорить и облегчить установку большого количества программных продуктов. Вы можете пометить (двойным щелчком левой кнопкой мыши) сразу несколько пакетов для установки, и затем установить их одним нажатием кнопки "Применить". Обратите внимание, что просто пометить пакеты недостаточно.

Другие варианты установки программ

В некоторых случаях приходится иметь дело с программами распространяемыми в виде бинарных сборок -- например, если собственноручная компиляция и сборка необходимого приложения вызывают затруднения или же программа не распространяется ни в каком ином виде. Рассмотрим несколько типов подобных сборок.

- Архивы. Наиболее предпочтительный для использования вид бинарных сборок. Как правило, для корректной работы достаточно распаковать такой архив. Обычно его распаковывают в каталог /opt.
- rpm. Весьма часто программы распространяются в виде rpm-пакетов. Однако, в этом случае при установке могут возникнуть трудности. Существуют некоторые различия в именах пакетов в ALTLinux и других дистрибутивах, использующих rpm, в следствие чего, при установке rpm могут появиться зависимости, удовлетворить которые возможно лишь восстановив соответствие между именами в ALTLinux и rpm-based дистрибутивах. В общем случае это нетривиальная задача.
- Бинарные установщики. Наименее удобным вариантом бинарной сборки является самостоятельная программа-установщик. С одной стороны, при использовании этого варианта, пользователю не задается вопросов о зависимостях. Но, с другой стороны, весьма сложно отследить, что именно делает подобный установщик. Успешный результат в этом случае отнюдь не гарантирован. В отличие от rpm, не гарантировано и восстановление исходного состояния системы при возникновении проблем. Отчасти, проблему использования таких установщиков можно решить, проводя установку в изолированном окружении.

Если вы хотите сами распространять некоторое приложение, то рекомендуется оформить его в виде пакета. Это облегчает поддержание приложения в актуальном состоянии и позволяет некоторым образом синхронизировать изменения, вносимые различными разработчиками с обновлениями программы. Кроме того, существующий инструмент сборки пакетов --- Hasher --- формирует хранилище, которое можно затем просто указать как локальный источник для диспетчера пакетов.

Запуск Win32-приложений

Несмотря на то, что:

- хранилища содержат очень много различного программного обеспечения;
- пакеты ставить в любом случае проще и комфортнее для пользователя, чем какие-либо приложения с собственным инсталлятором;

иногда всё-таки возникает необходимость воспользоваться каким-то приложением, которое распространяется исключительно под Windows.

Для того, чтобы осуществить подобное, в Linux используется программа WINE (WINE = Wine Is Not an Emulator).

Таким образом, название сообщает нам, что эта программа не является эмулятором. И верно, на деле принцип работы Wine проще: так как windows-приложение состоит из тех же процессорных инструкций, что и любое другое приложение под данный процессор, то для его исполнения достаточно сделать вид, что оно исполняется в привычной для себя Windows-среде --- т.е. вызывает из нужных библиотек нужные функции, распознает правильную абстракцию файловой системы и т.д.

Параметры данной "искуственной" среды задаются в конфигурационных файлах wine (их удобно редактировать с помощью утилиты winecfg): к примеру, там задаётся соответствие "дисков" Windows, которые сможет "увидеть" запускаемая Windows-программа, некоторым каталогам вашей файловой системы в Linux. Стоит быть осторожным при задании этих каталогов: если исполняемое при помощи wine приложение находится вне них, работать оно не будет.

Могут возникнуть и другие неожиданности --- например, при запуске консольных Windows-приложений:

```
[user@demo ~]$ wine Far.exe
err:winedevice:ServiceMain driver L"eusk3usb" failed to load
err:winedevice:ServiceMain driver L"SNTNLUSB" failed to load
err:seh:setup_exception_record stack overflow 188 bytes in thread 0009 eip
cdcdcdcd esp 00231274 stack 0x230000-0x231000-0x330000
```

Far Commander не запустится просто так, потому что здесь он должен работать в консоли Linux. Wine сам этого отличить не может, и надо это явно указать: `wine-console --backend=user Far.exe`

Также при работе с консольными приложениями, скорее всего, стоит указать в конфигураторе winecfg режим эмуляции Windows 98 из-за некоторых проблем с вызовом программ в консоли Windows.

Но, увы, все эти и многие другие ухищрения могут ни к чему не привести - большое количество Windows-приложений ни при каких условиях не сможет заработать в среде wine по тем или иным причинам. Сообщество поддерживает различные базы приложений, работающих в wine, одна из них находится на самом сайте wine --- Wine HQ App DB.

Чтобы завершить наш разговор на тему пакета wine, определим его общее предназначение: wine --- свободная программа, предназначенная для запуска Windows-программ. Помимо этого, на базе wine в данный момент разрабатываются как минимум три известных несвободных версии, которые предназначены для более специфичных прикладных задач:

- Cedega --- расширенный wine с встроенной поддержкой DirectX для запуска Windows-игр. Разработкой и поддержкой программы занимается компания Transgaming software;

- Crossover office --- расширение wine, созданное для запуска и комфортной работы с офисными Windows-приложениями;
- В Санкт-Петербурге функционирует команда wine@etersoft, которая занимается приспособлением wine для запуска программных продуктов, специфичных для российского рынка --- 1С, генераторы отчётов и документов и т.д. В отличие от предыдущих программных продуктов, wine@etersoft local --- свободный программный продукт, и в такой версии он входит в дистрибутив ALTLinux. Помимо версии local существуют также версии network и sql: первая позволяет запускать продукты 1С для многопользовательской работы с клиентами, а вторая умеет транслировать sql-запросы по образу и подобию Microsoft SQL. Кстати, документация для wine, созданная wine@etersoft, рекомендуется к прочтению как лучшая из существующих.

Стоит отметить, что приложения на Java, скорее всего, будут работать.